OPLE: A Heuristic Custom Instruction Selection Algorithm Based on Partitioning and Local Exploration of Application Dataflow Graphs

MEHDI KAMAL, ALI AFZALI-KUSHA, and SAEED SAFARI, University of Tehran
MASSOUD PEDRAM, University of Southern California

In this article, a heuristic custom instruction (CI) selection algorithm is presented. The proposed algorithm, which is called OPLE for "Optimization based on Partitioning and Local Exploration," uses a combination of greedy and optimal optimization methods. It searches for the near-optimal solution by reducing the search space based on partitioning the identified CI set. The partitioning of the identified set guarantees the success of the algorithm independent of the size of the identified set. First, the algorithm finds the near-optimal CIs from the candidate CIs for each part. Next, the suggested CIs from different parts are combined to determine the final selected CI set. To improve the set of the selected CIs, the solution is evolved by calling the algorithm iteratively. The efficacy of the algorithm is assessed by comparing its performance to those of optimal and nonoptimal methods. A comparative study is performed for a number of benchmarks under different area budgets and I/O constraints. The results reveal higher speedups for the OPLE algorithm, especially for larger identified candidate sets and/or small area budgets compared to those of the nonoptimal solutions. Compared to the nonoptimal techniques, the proposed algorithm provides 30% higher speedup improvement on average. The maximum improvement is 117%. The results also demonstrate that in many cases OPLE is able to find the optimal solution.

CCS Concepts: • **Hardware → Application specific instruction set processors** • **Hardware → Electronic design automation**

Additional Key Words and Phrases: ASIP, custom instruction selection, heuristic algorithm, speedup

ACM Reference Format:
Mehdi Kamal, Ali Afzali-Kusha, Saeed Safari, and Massoud Pedram. 2015. OPLE: A heuristic custom instruction selection algorithm based on partitioning and local exploration of application dataflow graphs. ACM Trans. Embedd. Comput. Syst. 14, 4, Article 72 (September 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2764458

1. INTRODUCTION

Embedded applications generally require high speed, low-power consumption, high flexibility, and low cost systems. Extensible processors have emerged in the field of embedded computing as a promising approach to remedy many shortcomings of ASICs and general-purpose processors [Keutzer et al. 2002; Gonzalez 2000].

Application-specific instruction set extension is another effective strategy to enhance the efficiency of embedded processors in terms of performance and energy consumption. By creating application-specific extensions to the Instruction Set Architecture (ISA) of

M. Kamal and A. Afzali-Kusha acknowledge financial support from the Iranian National Science Foundation (INSF).
Authors' addresses: M. Kamal, A. Afzali-Kusha (corresponding author), and S. Safari, School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran; emails: {mehdikamal, afzali, saeed}@ut.ac.ir; M. Pedram, Department of EE-Systems, University of Southern California, Los Angeles, USA; email: pedram@usc.edu.

DOI: http://dx.doi.org/10.1145/2764458

a base processor, the critical portions of an application Dataflow Graph (DFG) can be accelerated. The extension includes Custom Functional Units (CFUs) that implement the Custom Instructions (CIs). The approach provides more instruction level parallelism for reducing the latency of the critical paths as well as the number of intermediate results to be written in the register file [Pozzi et al. 2006; Galuzzi and Bertels 2011]. It also enhances the energy efficiency by reducing the access to different components of the base processor such as cache memories, register file, and ALU [Galuzzi and Bertels 2011; Biswas et al. 2007; Kamal et al. 2010]. In addition, the method could provide us with the required flexibility.

In the case of the extensible processor approach, the success depends on providing a highly automated design flow. There are two main steps in the design flow of customized processors, which are identification and selection of CIs. These steps are costly and time consuming. Many commercial and academic approaches have been introduced to automate these phases (see, e.g., Galuzzi and Bertels [2011]). The tools identify and select the set of CIs under different microarchitectural constraints such as power and area budget.

The aim of the selection phase is to find the best CIs from the CI candidate set that was generated in the identification phase. Hence, an optimization problem should be solved in this phase. Due to the existence of possible conflicts between the CIs, the selection optimization problem is NP-complete [Pozzi et al. 2006; Bonzini and Pozzi 2008]. Therefore, there are two general approaches for the CI selection: optimal and nonoptimal (see, e.g., Galuzzi and Bertels [2011] and Bonzini and Pozzi [2008]). The former is an exact method, which attempts to find the optimal CI set by searching the whole search space. The method becomes intractable for many real applications. The remedy is to prune the search space by removing the candidate solutions that are not feasible [Lu et al. 2009; Clark et al. 2005], giving rise to nonoptimal approaches, which are much faster than the optimal technique, but may not be able to find the optimal set.

In this article, we present a custom instruction selection algorithm for choosing near-optimal CIs for a given application. First, the algorithm divides the search space into small parts based on the DFG of the input application. Next, based on the nodes inside a part, a Greedy technique suggests a candidate CI set for the part. Since there are CIs for other parts that are similar to the CIs of this part (recurrent CIs), members of the candidate set for each part are extended by using the recurrent CIs. The optimal solution of the candidate set for the part is extracted quickly by using an optimal algorithm. Next, solutions for different parts are collected to form the complete candidate set for the application. As the last step in this iteration, an optimal algorithm is used to find the best CI set from this integrated set. This flow is iteratively called to improve the selected CI set. In the next iteration, it is guaranteed that some of the best selected CIs in the current iteration are augmented to the candidate CI set of the corresponding part.

The main advantage of the proposed method is that its efficacy is not a strong function of the size of the identified CI set. In addition, it supports realistic constraints such as the maximum number of the selected CIs and the area constraint. Moreover, to achieve higher speedup with smaller area usage, the recurrency of CIs is considered. The remainder of this article is organized as follows. In Section 2, the related works are briefly reviewed, while the ISA extension flow is described in Section 3. Section 4 explains the proposed selection algorithm. Experimental results are discussed in Section 5. Finally, Section 6 concludes the article.

2. RELATED WORKS

As mentioned previously, the ASIP design flow is divided into two CI identification and selection phases. Since the focus of this work is in the area of the selection phase, first,

some of the works in the area of the identification phase are reviewed, while most of the relevant works in the field of selection phase are then covered.

2.1. Identification

There are many works in the field of CI identification (see, e.g., Pozzi et al. [2006], Kamal et al. [2010], Atasu et al. [2012], Verma et al. [2007, 2010], and Pothineni et al. [2008]). It should be noted that while these works include both the identification and selection phases, their novelties are on the identification phase and they have either borrowed the selection technique from other works or used a simple iterative approach. In Atasu et al. [2012], two approaches to enumerate the maximum convex subgraphs of an input Direct Acyclic Graph (DAG) were proposed. For the first approach, the Integer Linear Programming (ILP) technique was used to formulate the enumeration, while in the second approach, some techniques were proposed to increase the speed of the CI enumeration. The techniques include the compaction of the graph representation and building a search tree tat is pruned through applying constraint propagation. In Verma et al. [2007] and Verma et al. [2010], a fast identification CI method has been proposed that is based on the clique enumeration. The proposed method performs some pruning before and after the clique enumeration to reduce the runtime.

An algorithm for identifying all the legal patterns under different microarchitectural constraints was proposed in Pothineni et al. [2008]. The method reduced the runtime of the identification by enumerating the patterns in the increasing order of sizes and also by making a relation between the characteristics of a $(k + 1)$-node pattern with the characteristics of its k-node subgraphs. The runtime complexity of different approaches for the CI identification has been studied in Reddington and Atasu [2012]. The study shows that the problem of the maximal convex subgraph enumeration has an exponential time complexity in general, while by defining some constraints such as an I/O constraint, the runtime of the enumeration becomes polynomial.

In Biswas et al. [2007], an architectural solution to mitigate the overhead of the memory access of Application-specific Functional Unit (AFU) was proposed. Hence, in the cases where this architectural technique is used, the memory access nodes will be removed from the list of the forbidden nodes obtained in the identification phase. The work described in Karuri et al. [2007] deals with proposing another architectural solution to mitigate the I/O problem during the CI identification phase by relaxing the I/O constraint. Obviously, the relaxation results in increasing the number of candidate CIs.

2.2. Selection

A key part of the ISA extension is the selection phase where the best CIs are selected from a pool of candidate CIs. The objective of the selection phase in most cases is to select the CIs that provide the highest speedup. This selection may be performed with or without considering a predefined area budget. Note that there are many works in the field of CI identification (for a survey of these works, see Galuzzi and Bertels [2011]). Because these works are out of the scope of this work, they are not reviewed in this part.

Several research efforts have been devoted to the problem of the CI selection. In Bonzini and Pozzi [2008] exact and approximate algorithms for solving the coverage and recurrence problems of candidate extensions were presented. The authors described an exact search technique that used a branch-and-bound algorithm in conjunction with a Greedy (approximate) method. The use of the branch-and-bound technique, which is an exhaustive approach, is not practical in real applications with large DFG sizes. Hence, the authors also presented a method that uses branch-and-bound in combination with a Greedy approach to speed up the process of finding the solution

while not being trapped in a local optimum. However, the applicability of the proposed near-optimal method depends on the specified maximum number of selected CIs. This means that, by increasing this maximum number, the runtime of the algorithm is increased exponentially.

In Clark et al. [2006], the design flow of the ISA extension starts by partitioning the DFG of the application into several subgraphs where each is considered as a candidate CI. Next, the set of candidate CIs is pruned through subgraph isomorphism check, which reduces the CI selection search space. In the final phase, the CIs are selected by solving a unate covering problem. The unate covering algorithm is, however, a branch-and-bound technique that it is too expensive for real applications. To make the branch-and-bound technique practical, the authors prune search space by using two techniques. One technique is based on a predefined maximum number of selected CIs, while the other one is based on the speedup achieved by the CIs (i.e., the CI speedup is used as the merit function). It should be noted that the second pruning method is only applicable in a design flow where the objective functions are based on the cycle saving.

In Lu et al. [2009], a technique for selecting CIs for multi-issue processors was presented. The method, which relied on a branch-and-bound algorithm, made use of a few pruning techniques to ensure that the proposed solution approach remained tractable. The pruning methods are useful only for multi-issue processors. In Scharwaechter et al. [2011], a complete design flow of the ISA extension was described. In the selection phase, a quadric optimization problem was used to select the best CIs. The authors mapped the CI selection problem to the Partitioned Boolean Quadratic Problem (PBQP). The proposed method is not applicable in cases of large applications.

An automatic framework for designing a customized processor, which deals with the instruction set identification and CI selection, is described in Clark et al. [2005]. In this work, the CI selection is performed based on the dynamic programming. A CI selection method for realizing on FPGA devices was presented in Lam et al. [2009]. The advantage of this approach was in fast and accurate estimation of the area and cycle saving of the CIs. In Pan and Mitra [2004] the authors suggested ILP and Greedy algorithms for the selection phase. As ILP is an exact technique; its usage becomes inapplicable for large problems. Similar to Pan and Mitra [2004], an ISA extension flow called CHIPS framework [Atasu et al. 2008] has used the ILP technique to select the optimum CIs.

In Liao and Devadas [1997], a linear programming relaxation technique for solving the binate covering was presented. The goal of this technique was to select the subgraphs that cover most parts of the input DFG. This method is not able to detect the recurrence of any templates of CIs in the input DFGs. In the heuristic method proposed in Li et al. [2009], the CIs are selected based on a merit function, which is a tuple of two ratios. The ratios are the cycle saving to the area usage and the cycle saving to the amount of conflicts with other CIs. The proposed method selects CIs with a higher cycle saving, smaller area, and fewer conflicts. In Peymandoust et al. [2003] an algebraic technique to select the CIs has been proposed. For selecting a CI, the method uses the Maple tool to do algebraic operations. The proposed selection method is an exact method based on the branch-and-bound algorithm providing optimal solutions. This approach may not be used for large problem sizes. To make the approach applicable for larger problems, the authors reduce the search space by using a bounding function that may yield nonoptimal solution. In Arora et al. [2010], a functional matching technique has been proposed that leads to finding more isomorphic CIs. In the technique presented in Arora et al. [2010], the custom instruction selection is performed using graph covering by finding a large set of equivalent pattern graphs. It means that, after determining the largest set of matched (isomorphism) subgraphs, a subset of patterns of this set that are mutually disjoint, is selected.

In this article, we present a heuristic selection method that attempts to divide the set of the identified CIs into smaller subsets. Next, an optimal approach is used to select the optimal CIs from these subsets. Optimal CIs are subsequently combined to form an integrated CI set. An optimal algorithm is used to find the best CI set for the ISA extension from this integrated set. This process is repeated to improve the speedup of the selected CIs. In contrast to optimal approaches, which cannot be used for large problems, the proposed method can be applied to solve large problem instances. In addition, compared to Greedy (nonoptimal approaches), the proposed technique provides solutions with higher cycle saving (with or without considering the area budget). The features of our proposed Optimization based on Partitioning and Local Exploration (OPLE) algorithm compared to the previous CI selection algorithms include

(1) scalability;
(2) closer solution to the optimal one leading to a higher speedup;
(3) considering recurrent CIs; and
(4) considering area constraint.

Also, it should be noted that the computational complexity of the CI identification is also high, which is a barrier to reduce the total runtime of the ISA extension flow. There have been some works proposing solutions to reduce the runtime of the CI identification phase as well (e.g., see Kamal et al. [2010], Bonzini and Pozzi [2007], and Xiao and Casseau [2011]). These techniques may be used in conjunction with the selection method proposed in this article. Finally, the differences between the reviewed works with those of our work are highlighted in Table I.

3. ISA EXTENSION DESIGN FLOW

Figure 1 shows the ISA extension design flow. The flow starts by the CI identification phase. In this phase, all subgraphs of the input application DFG that meet the specified constraints (e.g., I/O, propagation delay, convexity) are identified. Note that a subgraph is architecturally feasible if its inputs are available at the time of executing that operation, which is only possible if the subgraph is convex. (A subgraph S is called convex when there does not exist any path from a node $u \in S$ to another node $v \in S$ that includes a node $w \notin S$.) Moreover, there are some node types (e.g., Load, Store) in the DFG of the application that are excluded from the identified CIs. These nodes are called forbidden nodes [Pozzi et al. 2006]. In this article, all other nodes in the DFG of the application are called acceptable nodes. Note that the I/O constraint defines the maximum input and output ports of the selected CIs.

Among the identified subgraphs (CIs), there may be similar subgraphs based on the functional and structural isomorphism [Arora et al. 2010; Schmidt and Druffel 1976]. These CIs can be executed on one CFU, which results in less area overhead and better area utilization. Such similar subgraphs are identified and placed into CI groups [Bonzini and Pozzi 2008; Clark et al. 2006].

If a node of the DFG is included in two CIs, these two CIs are considered to have overlap with one another. Because any node in the DFG of the input application must be uniquely executed by one CI, all CIs that have overlaps with a selected CI are subsequently removed from the list of the identified CIs in the selection phase. Based on the overlap between the CIs, a *conflict graph* is constructed [Galuzzi and Bertels 2011]. The nodes in this graph denote the CIs, whereas an edge between two nodes shows that these CIs have overlap. Removing conflicting CIs causes changes in the members of CI groups, which consequently results in changes in the overall speedup of each CI group.

Finally, in the selection phase, the best CIs are extracted from the candidate set. The CIs are selected by considering an objective function specified by the designer.

Table I. Comparison of the Features of the Previous Works with Those of the OPLE

Reference	Focuses on*	Scalability	Chance of finding the optimal solution	Support Recurrent CIs	Area Constraint
Pozzi et al. [2006]	I	✗	Low	✗	✓
Biswas et al. [2007]	I	✗	Low	✗	✓
Kamal et al. [2010]	I	✓	Low	✗	✓
Atasu et al. [2012]	I	✗	High	✗	✓
Verma et al. [2007]	I	✓	Low	✗	✗
Vermaet al. [2010]	I	✓	Low	✗	✗
Pothineni et al. [2008]	1	✓	Low	✗	✗
Karuri et al. [2007]	I	–	–	–	–
Bonzini and Pozzi [2008] (Optimal)	S	✗	High	✓	✗
Bonzini and Pozzi [2008] (Greedy)	S	✓	Low	✓	✓
Clark et al. [2006]	S	✗	High	✗	✓
Lu et al. [2009]	S	✗	High	✗	✗
Scharwaechter et al. [2011]	S	✗	High	✗	✓
Clark et al. [2005]	I/S	✓	Low	✗	✓
Siew-Kei et al. [2009]	S	✓	Low	✗	✓
Pan and Mitra [2004]	S	✗	Htgh	✓	✓
Liao and Devadas [1997]	S	✗	High	✗	✗
Atasu et al. [2008]	S	✗	High	✓	✓
Peymandoust et al. [2003]	S	✗	High	✗	✗
Li et al. [2009]	S	✓	Low	✓	✓
Arora et al. [2010]	M	✓	Low	✓	✗
OPLE	S	✓	Medium	✓	✓

*I: Identification phase. S: Selection phase. M: Matching (generating similar groups) phase.

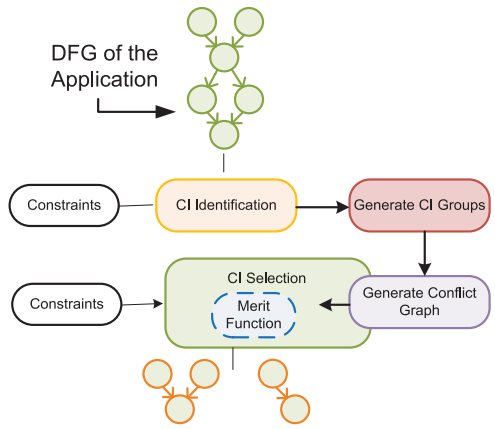


Fig. 1. The ISA extension flow.

Normally, *Cycle Saving* (*CS*) is the main objective in the ISA extension. The *CS* may be computed as follows [Kamal et al. 2011]:

$$CS_i = freq \times \left(\#CI_i.SW - CI_i.\text{IO\_}Penalty - \left\lceil \frac{CI_i.CriticalPathDelay}{Clock\ Period} \right\rceil \right), \quad (1)$$

where $CS\text{-}_i$ is the *cycle saving* of the ith CI (CI_i), *freq* is the execution frequency of the basic block to which CI_i belongs, and $\#CI_i.SW$ is the number of clock cycles of the base processor that the CI needs in order to be executed. $CI_i.IO\_Penalty$ denotes the number of extra accesses for reading/writing data to/from the register file (when the number of I/O ports of the CI is larger than the number of read/write ports of the register file) and the last fractional term calculates the number of clock cycles needed to execute the CI_i on the CFU (note that CIs can have multicycle as well as single cycle operations). In the fractional term, $CI_i.CriticalPathDelay$ denotes the propagation delay of the critical path of the CI, whereas *Clock Period* is the desired clock period for the extended processor.

Finally, we formulate the instruction selection problem as

$$Maximize \sum_{i=1}^{|SelectedCIGroups|} CS_{CIGroup_i}, \quad (2)$$

while

$$\forall CI_i, CI_j \in \{Selected\,CI\,Groups\}, CI_i \cap CI_j = \emptyset \quad (3)$$

$$\sum_{i=1}^{|Selected\ CI\ Groups|} Area_{CI\ Group_i} < Area_{Constraint}, \quad (4)$$

where (3) captures the no-overlap constraint. Note that the area constraint is an optional constraint, which may or may not be included in the problem. Also, we have assumed that constraints such as I/O and convexity were considered during the identification phase.

4. PROPOSED SELECTION ALGORITHM

The method proposed in this work, which finds the near-optimal solution, is based on the concept of divide and conquer. The problem is partitioned based on the input DFG of the application. Each part covers some acceptable (unforbidden) nodes. We can define some set of interest for each node. The set, which is called *ParentCI* set, contains all of the identified CIs that have this node in common. Figure 2 shows an example of a DFG with some corresponding identified CIs. For this example, the *ParentCI* set of node A is {CI1, CI2, CI4}, while members of the *ParentCI* set of node B are CI2, CI4, and CI5.

As the first step of generating the candidate set, the *ParentCI* sets for all acceptable nodes are extracted. Next, for each part, a candidate CI set is generated from the *ParentCI* sets of the part. From each *ParentCI* set, only one CI is considered to be added to the candidate set. The reason is that due to the overlap between the CIs, only one of them has the chance of being selected. Also, the recurrence count of the CIs is considered to expand the candidate CI set. The details of generating the candidate CI set along with an example will be provided in Section 4.3.

Now, the best set of CIs for each proposed candidate set is extracted by using the ILP technique. Note that since, in our approach, we select the candidate set to be small, any other exact technique may be invoked for this optimization problem. Then, the optimal CIs obtained for all the parts are combined to form an integrated candidate

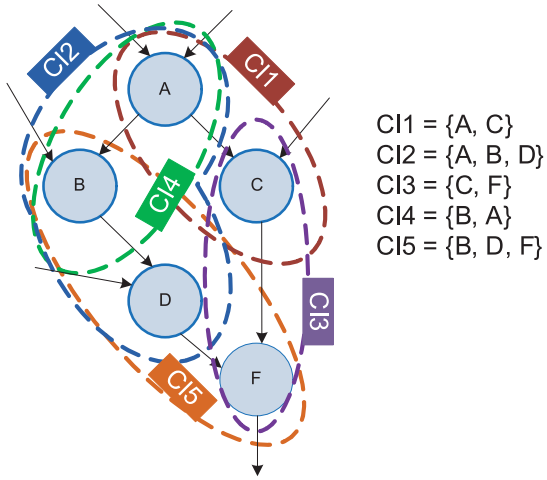


Fig. 2. A DFG of some identified CIs from the DFG.

set. By using an optimal technique, the best CIs from all parts are extracted, considering the predefined area budget. At this point, a single iteration of the proposed algorithm is completed. This process is repeated to improve the speed enhancement of the extensible processor. For this purpose, in the second iteration, the candidate CI set of all parts are updated based on the selected CIs from the integrated set of the previous iteration (see Section 4.1). This iterative procedure will be terminated when the improvement in the speedup does not increase after a given number of iterations. We call this algorithm the OPLE CI Selection algorithm. In the following, the details of the algorithm implementation are described.

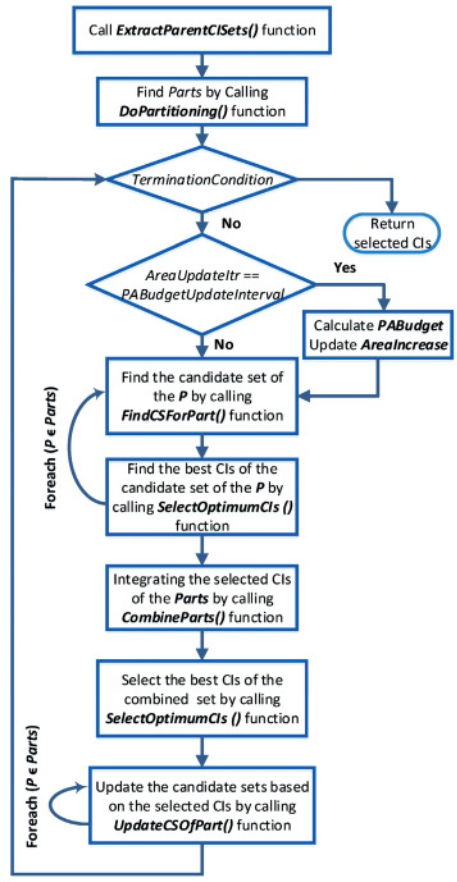
4.1. OPLE Method

A Pseudocode and flowchart of the proposed algorithm are given in Figure 3. The input arguments of the algorithm are the DFG of the input application, the maximum number of acceptable nodes in each part (denoted by *MaxPartSize*), the maximum number of CIs in the candidate set of each part (denoted by *MaxCSSize*), and the area budget (denoted by *Area*). The algorithm starts by extracting the *ParentCI* sets by calling ***ExtractParentCISets()*** (line 4). In this function, for each unforbidden node of a basic block, a *ParentCI* set is formed. Each set is formed by including the CIs that have this node as one of their nodes. The members of the *ParentCI* sets are sorted based on their cycle saving values. Next, the input DFG is partitioned by calling ***DoPartitioning()*** function (line 5), which divides the input DFG into parts with almost the same size. The details of this function will be described in the next subsection.

After partitioning, the main loop of the proposed method (lines 9–29) starts. First, if the area constraint is defined, the area budget will be determined for each part. Let us denote the number of part by *NumberOfParts*. Initially, the area budget for each part (denoted by *PABudget*) is set to *Area/NumberOfParts* (line 11). Since it is not known a priori that the optimal CIs selected from the integrated set at the end of each iteration belong to which part and how much of the area of that part is occupied, we can increase the area budget to a value more than *Area/NumberOfParts*. While this could have been done for initial iterations, to force the algorithm to select small CIs with high speedups, the area budgets of the parts are increased gradually. This increase is performed after a given number of cycles (denoted by *PABudgetUpdateInterval*) and with a ratio of *AreaIncrement*. Our simulations show that an *AreaIncrement* value

```
1:  Function OPLE (D: DFG of Application, MaxPartSize:
    integer, MaxCSSize: integer, Area: integer)
2:    Parts: List of Parts produced by the partitioning
    algorithm
3:    SelectedCIs: List of CIs
4:    ExtractParentCISets()
5:    Parts = DoPartitioning(D, MaxPartSize)
6:    AreaIncrease = 1;
7:    AreaUpdateItr = 0;
8:    CandidateCIsForSelection : List of CIs
9:    While (TerminationCondition) DO
10:     IF (AreaUpdateItr == PABudgetUpdateInterval)
11:       PABudget = AreaIncrease * (Area /
    NumberOfParts);
12:       AreaIncrease += AreaIncrement;
13:       AreaUpdateItr = 0;
14:       IF (PABudget > Area)
15:         PABudget = Area;
16:       END IF
17:     END IF
18:     Foreach (P ∈ Parts) DO
19:       FindCSForPart(D, P, IdentifiedCIs, MaxCSSize)
20:       P.SelectedCIs = SelectOptimumCIs (P,
    PABudget)
21:     END DO
22:     CP: Part
23:     CP = CombineParts({∀P , P ∈ Parts })
24:     SelectedCIs = SelectOptimumCIs (CP, Area)
25:     Foreach (P ∈ Parts) DO
26:       UpdateCSOfPart(P, SelectedCIs)
27:     END DO
28:     AreaUpdateItr++;
29:   END DO
30:   Return SelectedCIs
31: END
```

(a)



(b)

Fig. 3. (a) Pseudocode and (b) flowchart of the OPLE algorithm.

equal to 0.5 provides excellent results. This iterative procedure is continued until the same maximum speedup is achieved for a prespecified number of consecutive iterations (line 9). This is the termination condition of the algorithm. We denote this number by *IterCountSpeedupFixed*. Note that the algorithm keeps the same *PABudget* for each part for a number of iterations given by *PABudgetUpdateInterval*. If the condition of having the same maximum speedup occurs at a large number of iterations, then the increase in the *PABudget* may exceed the total area constraint (*Area*). To avoid this situation, areas of the parts are clamped to *Area* (lines 14–16). Note that *PABudgetUpdateInterval* is defined as

$$PABudgetUpdateInterval = \frac{AreaIncrement \times InterCountSpeedupFixed}{NumberOfparts} \quad (5)$$

This equation guarantees that under the case that the speedup of the selected CIs is not changed during all the iterations, the area budget for each part does not exceed the maximum area budget constraint.

```
1: Function SelectOptimumCIs (CL : List of CIs, Area : integer)
2:    SelectedCIs: List of CIs
3:    ConflictGraph = FindConflict(CL);
4:    WriteILPFormulation(CL, ConflictGraph, Area);
5:    SelectedCIs = CallILPSolver();
6:    Return SelectedCIs
7: END
```

Fig. 4. Pseudocode of the ***SelectOptimumCIs()*** function.

```
1: Function UpdateCSOfPart (P : Partition, SelectedCIs : List of CIs)
2:    ForEach (CI ∈ P.CS) DO
3:       IF (CI ∉ SelectedCIs)
4:          Remove(P.CS, CI)
5:       END IF
6:    END DO
7: END
```

Fig. 5. Pseudocode of the ***UpdateCSOfPart()*** function.

For a given *PABudget*, the candidate set for each part is determined by calling the function ***FindCSForPart()*** (see Section 4.3). The output of this function, which is the candidate set of the part along with its *PABudget,* is sent to the function ***SelectOptimumCIs()*** (line 20). In this function, by using an optimum algorithm, the best CIs of the candidate set of the part are extracted. Figure 4 provides the pseudocode of the ***SelectOptimumCIs()*** where ILP is used as the optimum selection algorithm. Note that before calling the algorithm, overlaps among the CIs must be determined. If the conflict graph of the identified CIs was extracted before the selection phase, calling the function ***FindConflict()*** (line 3) would not be needed. It should be noted that when the number of the identified CIs is large, storing the whole conflict graph in the memory is not practical. Hence, we should call ***FindConflict()*** for each candidate set in each iteration of the proposed algorithm.

After finding the best CIs of the candidate sets, the selected CIs are integrated into another set, which is used to find the best CIs considering *Area* as the area budget (when it is provided). In the last step, if a CI in the candidate CI set of a part is not selected (i.e., it does not belong to *SelectedCIs*), it will be removed from the candidate CI set of the corresponding part by calling ***UpdateCSOfPart()*** (see Figure 5).

4.2. Partitioning Method

The proposed selection method divides the problem into a number of subproblems. Figure 6 provides the pseudocode of the proposed partitioning scheme where the partition points (partitioning boundaries) are selected based on the basic blocks. Based on *MaxPartSize*, we group the basic blocks into different parts. In the case of basic blocks with a number of acceptable nodes higher than *MaxPartSize*, we use the Fiduccia-Mattheyses (F-M) partitioning method [Fiduccia and Mattheyses 1982], which is a well-known, linear time heuristic technique for solving the partitioning problem. In this method, the input graph is divided into two parts such that the numbers of edges crossing the parts is minimized. Minimizing the number of the adjacent nodes, which are connected by an edge, reduces the probability of overlapping between the selected CIs.

Hence, our DFG partitioning algorithm inspects sizes of the basic blocks one by one. If the number of the acceptable nodes of the basic block is more than the *MaxPartSize*,

```
1:  Function DOPartitioning(D : DFG of Application, MaxPartSize: integer)
2:    P : List of Parts
3:    ForEach (BB ∈ Basic Block belongs to D) DO
4:      IF(BB.NodeCount > MaxPartSize)
5:        FM_PartNodeCount = ⌈log2(BB.NodeCount / MaxPartSize)⌉
6:        P.ADD(DoFMPartitioning(BB, FM_PartNodeCount))
7:      END IF
8:      IF ( P[Last Member].NodeCount + BB.NodeCount ≤ MaxPartSize)
9:        P. [Last Member].Add(BB);
10:     ELSE
11:       P.ADD(BB)
12:     END IF
13:   END DO
14:   Return P
15: END
```

Fig. 6. Pseudocode of the ***DoPartitioning()*** function.

the F-M partitioning method is invoked (lines 4–7). In this case, the basic block is partitioned into $\lceil \log_2(\frac{BB.NodeCount}{MaxPartSize}) \rceil$ partitions. Note that the F-M technique divides a graph into two subgraphs and, if needed, continues to successively divide each of these two subgraphs into smaller subgraphs. Therefore, the number of the parts must be a value, which is a power of 2. Additionally, if the number of the acceptable nodes of the basic block is smaller than the *MaxPartSize* and the current part has sufficient free space for nodes of the current basic block, the basic block is added to the part (lines 8 and 9); otherwise, a new part is created to hold the basic block (line 11).

4.3. Generating Candidate Set for Parts

One of the important parts of the proposed selection algorithm is generating the candidate set for each part. The task is performed by function ***FindCSForPart()***. For each acceptable node of the part, this function determines a CI with the highest cycle saving from the *ParentCI* set of the node. Note that a CI may be the member of more than one *ParentCI* set. Hence, before adding a CI to the candidate set, the set must be checked to ensure that the CI is not a member of the candidate set. Additionally, if the CI corresponding to a node was selected as one of the best selected CIs from the integrated set in the previous iteration, then it would be included in the candidate set of this part. Moreover, we are not allowed to present a new candidate CI for this node. In fact, this method provides us with a combination of the best selected CIs obtained at the end of previous algorithm iteration plus some new CIs. This combination prevents us from missing the best solution found so far (by keeping the previous best ones) while not being trapped in local optimal solutions (by adding new CIs).

Also, considering the recurrent CIs in the selection phase leads to increasing the speedup of the selected CFU under a predefined area budget. Therefore, in the last part of this function, some additional CIs that are in the CI groups with the candidate CI set are added to the final candidate set for the part.

The pseudocode and flowchart for the function ***FindCSForPart()*** are given in Figure 7 and Figure 8, respectively. The selection process starts (line 3 of Figure 7) by generating the list of the CI groups of the CIs selected in the previous iteration (see line 23 of Figure 3). This list is denoted by *CIGIndex*. Next, for each acceptable node of the part, one CI is selected from the beginning of *ParentCI* set (lines 4–14 of Figure 7).

```
1:  Function FindCSForPart (D : DFG of Application, P : Part, SCIs :
    List of Identified CIs, PABudget : integer, MaxCSSize: integer)
2:    CIGIndex : List of integer
3:    CIGIndex.Add(CIGroupIndex of ∀CI ∈ P.CS)
4:    Foreach({N, ∀N ∈ P.Nodes} ) DO
5:      IF ( !ItsCandidateCIbelongstoSelectedCISet(N) )
6:        While ((N.ParentCISet[0] ∉ P.CS) &&
    (N.ParentCISet[0].Area > PABudget)) DO
7:          RotateParentCISet(P.Nodes.ParentCISet)
8:        END DO
9:        P.CS.Add(N.ParentCISet[0])
10:       IF (N.ParentCISet[0].CIGroupIndex ∉ CIGIndex)
11:         CIGIndex.Add(N.CIGroupIndex)
12:       END IF
13:     END IF
14:   END DO
15:   While(P.CS.Count <= MaxCSSize)
16:     CIGI = CIGIndex [Index]
17:     Index = Index++ % CIGIndex.Count
18:     P.CS.Add(BestCI (CIGI))
19:   END DO
20: END
```

Fig. 7. Pseudocode of the ***FindCSForPart()*** function.

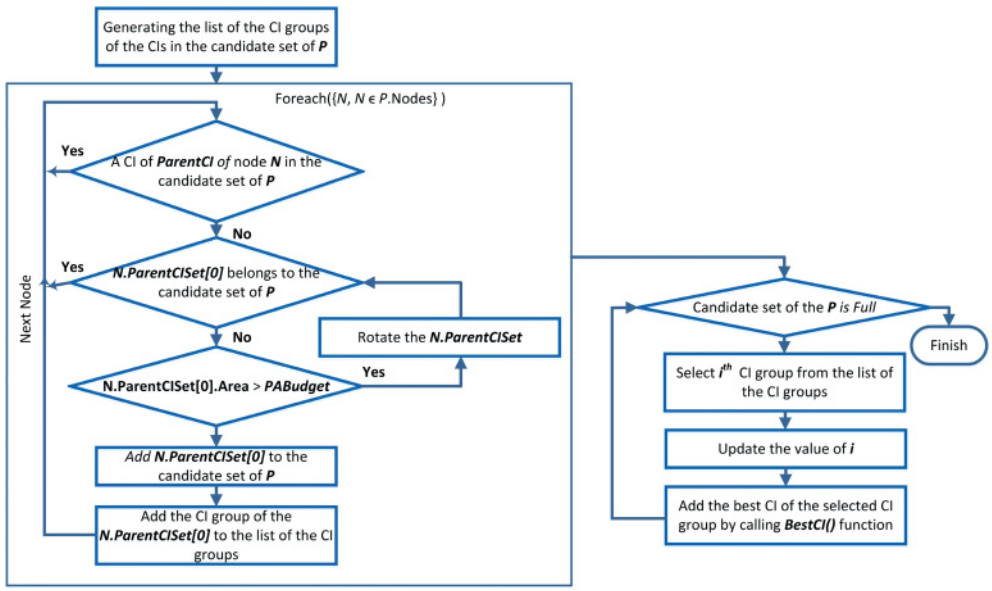


Fig. 8. Flowchart of the ***FindCSForPart()*** function.

As mentioned previously, before the iteration starts, the *ParentCI* set is sorted based on the CI cycle savings. If a suggested CI exists in the candidate set, then this CI will be moved to the end of the set and another CI from the head of the set will be considered (line 7 of Figure 7). This changes the order in which the CIs are sorted in the *ParentCI* set of this node. Additionally, we move the added CI to the end of the *ParentCI* set to provide the opportunity for the other CIs to be selected in subsequent iterations. This also reduces the chance of being trapped in a local optimum point. Finally, if a suggested CI from a node in the previous round is selected as the final selected CI list, then this node will not be allowed to suggest a CI from its *ParentCI* set (line 5 of Figure 7).

Having added the CIs based on the *ParentCI* set of nodes, we also augment some other CIs considering the CI groups in the candidate set (lines 15–19 of Figure 7). The CI groups included in the *CIGIndex* set are considered one by one. From each CI group, the best CI that is not a member of the candidate set and also has fewer conflicts with the CIs in the candidate set is added to the candidate set (line 18 of Figure 7). For this purpose, we select the CI that has higher value of $CS/|Conflicts|$. The process of adding CIs from the CI groups continues until the candidate set becomes full. Since we use an optimal selection algorithm to find the best CIs for each part, the size of the candidate set must not be too large. The size of the candidate set is larger than the number of the acceptable nodes of the part. The upper limit of the size is determined based on the maximum number of CIs that we wish to include from the CI groups.

We illustrate the process of finding the candidate set for the part by using the example given in Figure 9. We assume that the part has five acceptable nodes. The members of the *ParentCI* sets are shown in the figure. The size of the candidate CI set is denoted by MaxCSSize, which is equal to seven in this example. We assume that CI1 proposed by node B has been selected as the final selected CI in a previous iteration, and hence, no CI will be proposed for node B.

First, from each *ParentCI* set, a CI is suggested to be added to the candidate set and the orders of the members in the *ParentCI* sets are updated. For example, in the case of node A, CI2 is suggested to be added to the candidate set. Because the candidate set does not have CI2 as a member, CI2 is added to the set, and the CI2 is moved to the end of the *ParentCI* set for this node. For the case of node C, CI2 from the head of the *ParentCI* set already exists in the candidate set. Hence, this CI is moved to the end of the set for the node C. The next CI in this set is CI4, which is added to the candidate set and is moved to the end of the *ParentCI* set. Note that the process of adding CIs is done consecutively from node A to node E. After adding the CIs, members of the sets for the selected CI groups (denoted by *CIGIndex*) are updated. Finally, the CIs with higher cycle savings and fewer conflicts with the CIs in the candidate sets are added. In this example, CI13 and CI19, which belong to CI groups 1 and 2, are suggested to be added to the candidate set. Adding these two CIs to the set makes the candidate set full, completing the process of adding CIs to the candidate set.

5. RESULTS AND DISCUSSION

To assess the efficacy of the proposed selection algorithm, we have chosen eight benchmarks from mibench [Guthaus et al. 2001], MediaBench [Lee et al. 1997], Packetbench [Ramaswamy and Wolf 2003], and SNU-RT [SNU 2015] benchmark suits. The size of each benchmark and also the number of the identified CIs under three different I/O constraints are reported in Table II. Each benchmark was profiled by running on the instruction set simulator of an in-order MIPS processor, and the DFGs of these benchmarks were generated based on the operators of this machine model. Note that the CIs of each benchmark were enumerated by using the multicut identification algorithm proposed in Pozzi et al. [2006], and the CI groups were extracted based on the method

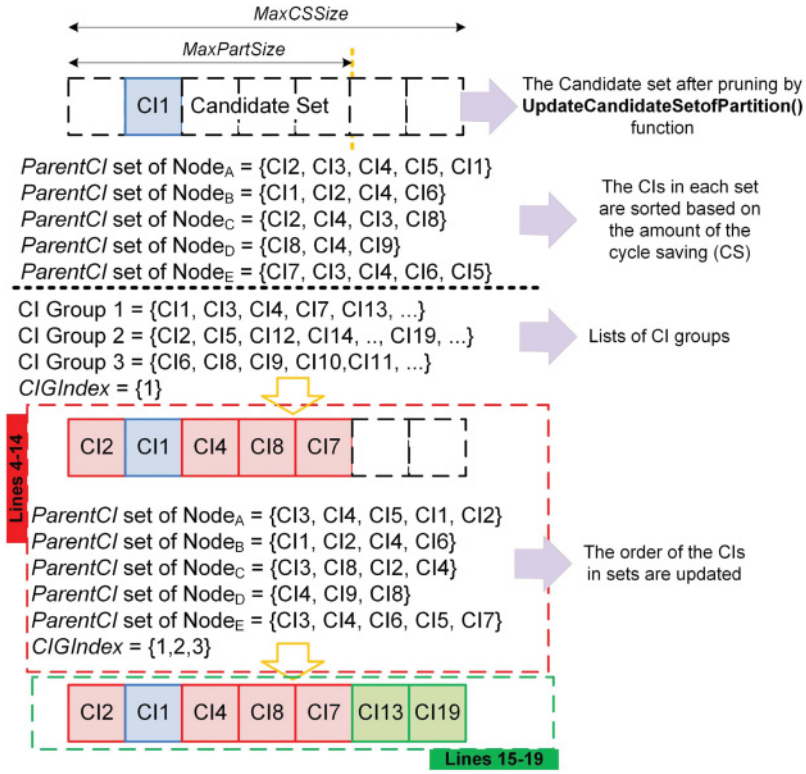


Fig. 9. An example of ***FindCSForPart()*** function process.

Table II. Feature of the Benchmarks

			I/O Constraint								
			4/4			3/3			2/2		
Benchmark	|N|*	|UN|	|CI|	|CIG|	RA	|CI|	|CIG|	|RA|	|CI	|CIG|	RA
1ms	119	104	42	33	5202	29	24	5469	12	10	2193
adpcm	83	72	62	44	5385	56	38	5385	32	20	3180
sha	173	111	1519	777	13384	650	265	12038	149	52	10090
G721eneode	368	292	1042	686	18045	722	466	17336	292	174	15421
G721decode	373	295	1077	602	17104	747	443	16806	302	172	17561
IP-sec	294	206	19927	14953	17559	6611	4097	15426	1246	517	9385
bircounter	163	124	39832	34153	10626	7467	5754	10696	776	514	6010
MD5	523	406	48676	29366	22350	9883	4096	33130	1348	310	15524

*N: Node counts of the application, UN: Acceptable Nodes count of the application, CI: Identified CIs, CIG: CI Group, RA: Reference Area.

proposed in Clark et al. [2006]. To estimate the delay and area usage of each CI, we used the delays and areas of its operations obtained by synthesizing their Register-Transfer Level (RTL) implementation to the gate-level netlists in a 45nm technology [FreePDK 2010]. In Table III, we provide the details of the components considered as the basic operators for the DFG of the benchmarks. The area of the CIs, which was normalized to the size of a NAND2 in the 45nm technology, was considered as the summation of the areas of their operations. The delays of the CIs were calculated based on the delay of the operators in the critical path.

Table III. The Delays and Areas of the Basic Operations

Primitive Name	Area	Delay (ns)
SUB	225	0.5
ADD	200	0.5
SHR/SHL	326	0.19
EQT/NEQ	87	0.16
GRT/LKS	115	0.21
AND	41	0.04
OR	42	0.05
XOR	64	0.05

The proposed OPLE selection algorithm has been implemented in C# while Gurobi [Gurobi 2015] was used as the ILP solver. The performance of the algorithm is compared to those of the Greedy selection method proposed in Bonzini and Pozzi [2008] (denoted by Greedy), the heuristic method proposed in Li et al. [2009] (denoted by Heuristic), and pure ILP technique. The Greedy and Heuristic methods were implemented using C# language, and Gurobi was used for the pure ILP method. Note that the proposed methods in Bonzini and Pozzi [2008] and Li et al. [2009] are applicable for large problem sizes, which can support recurrent CI selection and area constraint.

The selection algorithms were run on a machine with the i7 Intel processor and the clock speed of 2.66GHz. This processor was able to run eight threads simultaneously. In the cases of the ILP and Heuristic method, before the selection, the conflict graph had to be generated, which required a huge runtime. Hence, for the ILP, Heuristic, and OPLE (for each part), the process of finding the conflict graphs was implemented by a multithreaded programming method. Note that in the case of the Greedy method, during the selection phase, the CIs that had overlaps with the selected CIs were identified and subsequently eliminated from the candidate set. Therefore, there was no need to generate the conflict graph. Also, note that the Gurobi ILP solver made use of the multithreading technique. In the OPLE selection algorithm, the function ***SelectOptimumCIs*()** was also called by the multithreading approach.

A comparative study for the selection algorithms was performed under five different area constraints. The area constraints were 10%, 20%, 30%, 40%, and 50% of a reference area. The reference area was determined based on the area usage of the CFU when the CIs were selected by using the Greedy method without considering any area budget. In addition, we considered the case of no area constraint. The reference areas in units of equivalent two-input NAND gates for different cases are reported in Table II. Note that by decreasing the I/O constraint, the size of the identified CIs may be reduced, which leads to reducing the opportunity for selecting large CIs. Hence, by decreasing the I/O constraint, the area usage of the selected CFU may be reduced. On the other hand, the chance of selecting large CIs is reduced. However, in some cases, the number of the selected CIs that are small may increase due to less conflict between candidate CIs. The increase in the number of selected CIs may lead to some area increase for these cases.

Finally, notice that in the OPLE algorithm, we assumed the value of *IterCountSpeedupFixed* to be 80 (recall that this parameter is used to determine the termination condition). The speedups of the extensible processors under different selection algorithms when the I/O constraint was 4/4 for benchmarks are reported in Table IV. Note that **WNAB** corresponds to the case of selecting CIs With No Area Budget. Also, in the case of OPLE, the CIs were extracted under four tuples of (*PS*, *CSS*) where *PS* stands for the *Part Size* and *CSS* stands for the *Candidate Set Size*. A comparison between the speedup of the OPLE and Greedy shows that, in all cases, the proposed method provides higher speedups. The maximum speedup improvement is ∼117%,

Table IV. Speedup of the Extensible Processors under Different Area Budgets and I/O Constraint of 4/4 while the *CIS* are Extracted by Four Different CI Selection Methods

Benchmark	Selection Method	Area Constraint						Selection Method	Benchmark	Area Constraint					
		10%	20%	30%	40%	50%	WNAB			10%	20%	30%	40%	50%	WNAB
adpcm	PS-CCS	1.43	1.72	1.94	2.11	2.33	2.64	PS-CCS	G721decode	1.82	2.00	2.07	2.12	2.14	2.18
	OPLE 25–100	1.43	1.72	1.94	2.11	2.33	2.64	OPLE 25–100		1.82	2.00	2.07	2.12	2.14	2.18
	OPLE 25–200	1.39	1.72	1.88	2.08	2.33	2.64	OPLE 25–200		1.82	2.00	2.07	2.11	2.14	2.18
	OPLE 50–200	1.39	1.72	1.88	2.08	2.33	2.64	OPLE 50–200		1.82	2.00	2.07	2.11	2.14	2.18
	OPLE 50–400	1.07	1.30	1.50	1.58	1.72	2.28	OPLE 50–400		1.54	1.78	1.54	1.78	1.93	1.97
	Greedy	1.43	1.67	1.88	2.08	2.20	2.28	Greedy		1.55	1.68	1.55	1.68	1.73	1.75
	Heuristic	1.43	1.72	1.94	2.11	2.33	2.64	Heuristic		1.82	2.05	1.82	2.00	2.07	2.12
	ILP	1.13	1.23	1.31	1.35	1.40	1.49	ILP		3.59	3.78	3.98	4.17	4.30	4.39
lms	PS-CCS	1.13	1.23	1.31	1.35	1.40	1.49	PS-CCS	IPsec	3.77	4.19	4.23	4.36	4.37	4.39
	OPLE 25–100	1.13	1.23	1.31	1.35	1.40	1.49	OPLE 25–100		2.98	3.87	4.25	4.26	4.38	3.39
	OPLE 25–200	1.11	1.23	1.31	1.35	1.43	1.49	OPLE 25–200		4.22	4.26	4.35	5.17	5.51	4.39
	OPLE 50–200	1.11	1.23	1.31	1.35	1.43	1.49	OPLE 50–200		2.09	2.91	2.09	2.91	3.12	3.45
	OPLE 50–400	1.05	1.13	1.18	1.25	1.28	1.43	OPLE 50–400		2.49	2.83	2.49	2.83	2.83	2.83
	Greedy	1.13	1.20	1.31	1.37	1.43	1.45	Greedy		NaN	NaN	NaN	NaN	NaN	NaN
	Heuristic	1.13	1.23	1.31	1.37	1.43	1.49	Heuristic		2.76	1.61	1.99	2.20	2.40	2.55
	ILP	1.74	1.93	2.02	2.04	2.05	2.05	ILP		1.61	2.14	2.23	2.47	2.96	2.76
sha	PS-CCS	1.74	1.93	2.02	2.04	2.05	2.05	PS-CCS	bitcounter	1.65	2.09	2.26	2.59	2.51	2.76
	OPLE 25–100	1.74	1.93	2.02	2.04	2.05	2.05	OPLE 25–100		1.72	2.04	2.63	2.43	2.55	2.76
	OPLE 25–200	1.74	1.93	2.02	2.04	2.05	2.05	OPLE 25–200		1.27	1.39	2.63	1.39	1.42	1.51
	OPLE 50–200	1.74	1.93	2.02	2.10	2.05	2.05	OPLE 50–200		1.52	1.83	1.27	1.83	2.01	2.14
	OPLE 50–400	1.34	1.44	1.49	1.52	1.53	1.54	OPLE 50–400		NaN	NaN	NaN	NaN	NaN	NaN
	Greedy	1.34	1.40	1.43	1.43	1.43	1.84	Greedy		1.27	1.39	2.63	1.39	1.42	1.51
	Heuristic	1.74	1.93	2.02	2.10	2.05	2.05	Heuristic		1.52	1.83	1.27	1.83	2.01	2.14
	ILP	1.79	1.96	2.02	2.06	2.09	2.13	ILP		NaN	NaN	NaN	NaN	NaN	NaN
G721encode	PS-CCS	1.79	1.96	2.02	2.06	2.09	2.13	PS-CCS	MD5	3.90	4.22	5.02	5.12	4.49	4.64
	OPLE 25–100	1.79	1.96	2.02	2.06	2.09	2.13	OPLE 25–100		4.24	4.45	4.52	5.05	4.60	4.64
	OPLE 25–200	1.78	1.96	2.02	2.07	2.09	2.13	OPLE 25–200		3.86	4.43	4.54	4.61	4.62	4.64
	OPLE 50–200	1.78	1.96	2.02	2.07	2.09	2.13	OPLE 50–200		4.27	4.55	4.54	4.93	4.62	4.64
	OPLE 50–400	1.54	1.76	1.89	1.91	1.94	2.04	OPLE 50–400		2.08	2.68	2.08	2.68	2.86	2.96
	Greedy	1.52	1.64	1.67	1.69	1.71	1.99	Greedy		2.37	2.94	2.37	2.94	3.27	3.48
	Heuristic	1.79	1.96	2.03	2.07	2.10	2.13	Heuristic		NaN	NaN	NaN	NaN	NaN	NaN
	ILP							ILP							

*WNAB: With No Area Budget.

which belongs to the case of the *bitcounter* benchmark under an area budget of 10%. On average, the speedup improvement of OPLE compared to the Greedy is about 30%.

Compared to the Greedy case, the Heuristic method provides higher speedups for small area budgets. The reason is that the area usage is considered in the merit function. Compared to OPLE, for small benchmarks under the smallest area budget (i.e., 10%), the Heuristic method selects CIs with higher speedups. However, in most cases, the OPLE outperforms the Heuristic method (by an average of 22% higher) in terms of speedups. For small benchmarks (i.e., *adpcm* and *lms*), the OPLE selects CIs resulting in ∼2.2% (∼18.6%) higher speedups compared to those of the Heuristic (Greedy) method. However, for large benchmarks (i.e., *IPsec*, *bitcounter*, and *MD5*) OPLE provides ∼34% (50%) higher speedup compared to the Heuristic (Greedy) method.

In the case of the ILP selection, for the three largest benchmarks (i.e., *IPsec*, *bitcounter*, and *MD5*), the Gurobi solver was not able to solve the problem due to the memory size explosion (indicated by NaN in the table). For the other benchmarks, the results reveal that, in most cases, the speedups achieved through the proposed method were the same as those of the optimal solution (i.e., ILP). In the worst case, the speedup of the proposed algorithm was 2.35% less than that of the optimal solution. Also, on average, the speedup of the ILP is about 0.28% higher than that of the OPLE method, while the speedup of the ILP, on average, is ∼19% (∼17.2%) larger than that of the Greedy (Heuristic) method.

It should be noted that the efficacies of the proposed method for small area budgets (e.g., 10%) are higher compared to the cases of large area budgets (e.g., 50% or WNAB). Also, compared to the Greedy and Heuristic methods, the effectiveness of the proposed algorithm improves as the candidate set size increases. Additionally, increasing the *PS* and/or the ratio of *CSS*/*PS* resulted in higher speedups for the proposed selection method (efficiency improvement). As mentioned before, the CIs selected for a part are determined based on the *ParentCI* set. The *PS* value sets the initial number of the CIs selected for the part. Since these CIs are selected from neighboring nodes in the part, the chance of conflict (overlap) is high. The use of the ILP technique enables one to select the best nonconflicting CIs with the highest speedups in each subproblem (part). Increasing the *PS* value enlarges the subproblem size, improving the chance of selecting better CIs (at the expense of increasing the runtime). Also, the *CCS*/*PS* ratio shows the number of recurrent CIs considered for each subproblem. When the ratio increases, more recurrent CIs are included in the CI candidate set and considered in the optimization. When there is a large number of recurrent CIs (e.g., large problems), increasing the ratio helps select better CIs. Note that for small area budgets, the selection of more recurrent CIs improves the speedup gain. Therefore, increasing the size of *CSS* provides us with a lager exploration space for selecting these CIs. Note that there are some cases in the reported results in Table IV that by increasing the area budget the speedups of the inexact approaches are reduced. For the Greedy, Heuristic, and OPLE techniques, by which finding the exact solution is not guaranteed, some cases may occur that the techniques may be trapped in local optimum solutions. These cases, however, occur less frequently.

Next, we compare the runtimes of OPLE and other selection methods as reported in Table V. The runtimes are for three different problem sizes. In all the cases, the runtime of the Greedy selection method was below 1s. For the case of the Heuristic technique, the runtime is higher because of the time overhead of finding the conflict graph. The runtime, however, is less than those of the OPLE and ILP. As observed from the table, ILP is fast for small size problems, while its runtime scales exponentially with the problem size. It should be noted that the problem size in the CI selection problem is the number of the CIs. Note that the ILP solver was not able to solve the *MD5* benchmark problem. Results also reveal that the runtime complexity of the OPLE

Table V. Runtimes of the Selection Methods in Seconds while the I/O Constraint was 4/4

Benchmark	Selection Method PS-CCS		Area Constraint 10%	20%	30%	40%	50%	WNAB
adpcm	OPLE	25–100	5	6	7	7	7	4
		25–200	6	7	7	7	7	4
		50–200	3	7	5	8	6	4
		50–400	3	7	5	8	6	2
	Greedy		<1	<1	<1	<1	<1	<1
	Heuristic		<1	<1	<1	<1	<1	<1
	ILP		<1	<1	<1	<1	<1	<1
sha	OPLE	25–100	22	33	32	30	36	18
		25–200	69	87	85	79	87	63
		50–200	51	78	78	78	86	28
		50–400	59	98	97	101	114	63
	Greedy		<1	<1	<1	<1	<1	<1
	Heuristic		<1	<1	<1	<1	<1	<1
	ILP		77	79	5564	658	31	2640
MD5	OPLE	25–100	88	143	110	444	209	204
		25–200	474	1709	1591	2299	2309	356
		50–200	508	3094	4124	3235	2693	354
		50–400	3945	5795	2195	4786	3589	782
	Greedy		<1	<1	<1	<1	<1	<1
	Heuristic		98	98	98	100	99	98
	ILP		NaN	NaN	NaN	NaN	NaN	NaN

selection algorithm is polynomial. Note that, as mentioned before, increasing the *CSS* value and/or the ratio of *CSS*/*PS* makes the runtime of OPLE larger. Since ILP cannot solve larger problems, OPLE may be used for these problems. In the worst case (i.e., *MD5* benchmark under 20% area budget), the OPLE selection algorithm finds a CFU that provides us 70% (36%) higher speedup than that of the Greedy (Heuristic) selection method (with a runtime of 1h and 36min).

The ILP-based solution provides the optimal solution with the highest speedup. The approach, which is computationally intensive ($O(2^n)$), may not be invoked for large optimization problem sizes. The OPLE algorithm, however, breaks down the problem to k smaller ones to reduce the computational intensity of the problem ($O(2^m)$ where $m = n/k$ and k depends on PS and CCS). Therefore, in general, OPLE does not provide higher speedup compared to that of the ILP. In other words, OPLE only reduces the runtime of the selection algorithm. The amount of the reduction depends on n and k. Since there are some overheads associated with the OPLE, the use of OPLE is not justified for small problems (see Table IV). As the results show, for the candidate CI set with the size of about 1000, the use of OPLE may be justified.

To study the effect of the problem size on the speedups achieved when these techniques are used, Figure 10 shows the speedup of four benchmarks with the sizes of 42, 1,519, 19,927, and 48,676. For smallest size (*adpcm*), ILP, Heuristic, and OPLE lead to about the same speedup, while the Greedy approach provides the lowest speedup. For the second smallest size (*sha*), both the ILP and OLPE techniques give rise to the same speedups, which are 30% higher than those of the Greedy and Heuristic. In the case of the two larger problem sizes, the exact method of ILP cannot provide the optimal answer using reasonable computer resources, while the Greedy and Heuristic techniques yield 50% (51%) and 41% (44%) lower speedup for the *IPsec* (*MD5*) benchmark, respectively. The results indicate the ability of the OPLE to provide close to optimal solutions

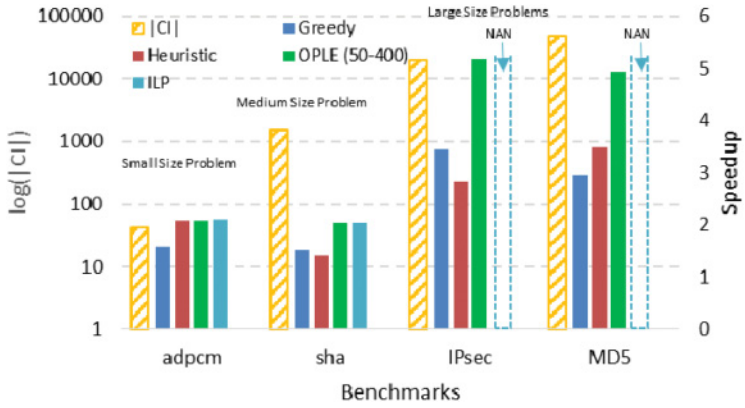


Fig. 10. The trend of OPLE speedup compared to those of the other approaches versus the problem size.

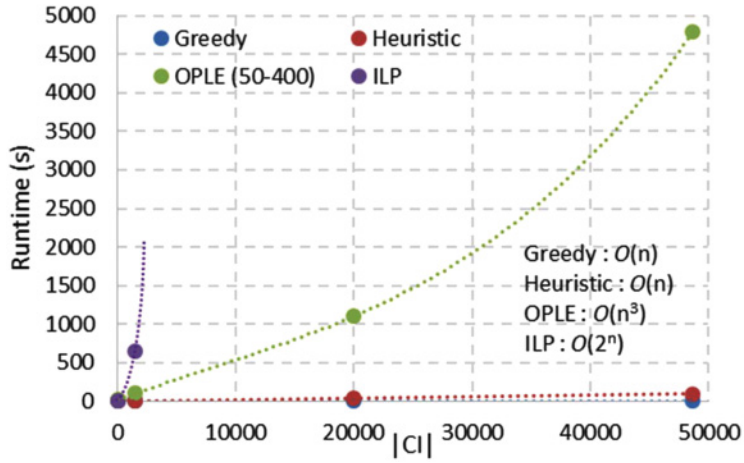


Fig. 11. The trend of OPLE runtime compared to those of the other approaches versus the problem size.

at least for the benchmarks that we have found the optimal solutions. To compare the computational complexities of these techniques, the runtimes of the techniques versus the problem size have been plotted in Figure 11. Since the runtime complexities of the Greedy and Heuristic methods are $O(n)$, their runtimes increase linearly with the problem size. In the case of the OPLE, the complexity follows $O(n^3)$, which is much lower than $O(2^n)$ in the case of the ILP method. It should be noted that, since the design is performed once, the higher speedup obtained using OPLE through the higher runtime is preferred over Greedy and Heuristic, which have smaller runtimes but provide lower speedups.

As discussed previously, in seeking higher speedups, we increase the area budget of the parts after a predetermined number of iterations in the OPLE algorithm. This process along with the achieved speedup of the extensible processor for the benchmark *MD5* is shown in Figure 12. The area budget was defined to be 30%. The results indicate that the speedup improvement has a strong dependence on the area increase only for iteration counts below 100. We have observed the same behavior for other benchmarks as well.

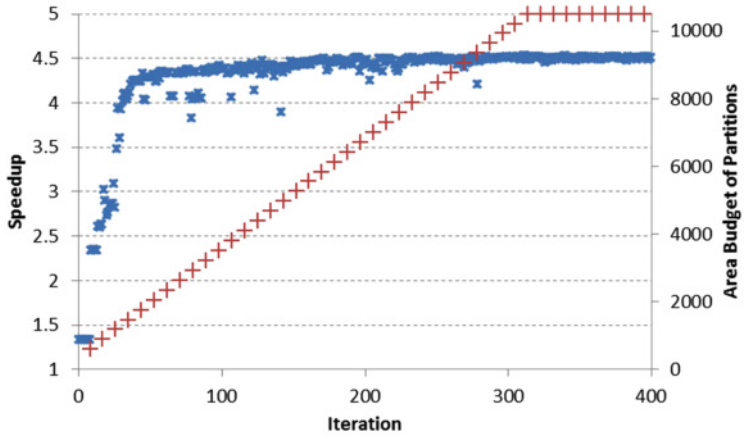


Fig. 12. Speedup improvement and the area budget of the parts during the OPLE algorithm exploration for the benchmark MD5 (Area = 30%).

Table VI. Speedup of the Extensible Processors under Different Area Budgets

I/O	Benchmark	Selection Method	Area Constraint					
			10%	20%	30%	40%	50%	WNAB
3/3	IPsec	OPLE	2.53	3.75	4.25	4.28	4.34	4.37
		Greedy	2.09	2.91	3.12	3.45	3.50	3.53
		Heuristic	1.93	2.74	2.92	2.96	2.96	2.99
		[LP	NaN	NaN	NaN	NaN	NaN	NaN
	bitcounter	OPLE	1.48	1.70	1.81	2.04	2.14	2.72
		Greedy	1.27	1.39	1.42	1.51	1.63	2.47
		Heuristic	1.33	1.52	1.65	1.83	1.92	2.36
		ILP	NaN	NaN	NaN	NaN	NaN	NaN
	MD5	OPLE	3.34	4.32	4.44	4.57	4.55	4.63
		Greedy	2.08	2.68	2.86	2.96	3.07	3.37
		Heuristic	2.18	2.70	2.90	3.02	3.09	3.29
		ILP	NaN	NaN	NaN	NaN	NaN	NaN
2/2	IPsec	OPLE	1.63	2.13	2.47	2.60	2.64	2.66
		Greedy	1.56	1.94	1.99	2.13	2.15	2.16
		Heuristic	1.55	1.70	1.96	1.99	1.99	2.00
		ILP	1.63	2.13	2.47	2.60	2.64	2.66
	bitcounter	OPLE	1.24	1.42	1.52	1.61	1.72	2.20
		Greedy	1.24	1.33	1.37	1.39	1.52	1.92
		Heuristic	1.19	1.37	1.47	1.55	1.61	2.04
		ILP	1.24	1.42	1.52	1.65	1.72	2.20
	MD5	OPLE	1.64	2.03	2.20	2.23	2.25	2.36
		Greedy	1.43	1.67	1.71	1.73	1.74	1.76
		Heuristic	1.57	1.80	1.85	1.87	1.88	1.89
		ILP	1.64	2.03	2.20	2.23	2.25	2.37

Finally, to study the effect of the I/O constraint on the speedup achievements of the selection algorithms, we considered the constraints of 2/2 and 3/3 for all the techniques. For this study, we considered our three largest benchmarks, namely, *IPsec*, *bitcounter*, and *MD5*. The ILP solver was not able to solve these benchmarks under the I/O constraint of 3/3 on this computer system. The values of the *PS* and *CCS* parameters for this study were assumed to be 25 and 200, respectively. Results are presented in Table VI, which shows that in the case of 3/3 I/O constraint, OPLE selects CIs with higher cycle savings compared to those of the Heuristic and Greedy methods. In the best case (i.e., *MD5* benchmark under an area constraint of 20%), the speedup of the OPLE method is 61% (60%) higher than that of the Greedy (Heuristic) method. Also, on average, for these benchmarks, OPLE selects CFU with 34% (35%) higher speed gain compared to that of the Greedy (Heuristic). In the case of the I/O constraint of 2/2, OPLE outperforms Heuristic and Greedy. On average, for these benchmarks, the OPLE technique selects CFU with 18% (16%) higher speedup compared to that of the Greedy (Heuristic) approach. This hints that the speedup improvement is lower for this I/O constraint compared to the previous constraint. By reducing the I/O size, the problem size becomes smaller, lowering the difference between the speedup of OPLE compared to those of Heuristic and Greedy. Finally, for this I/O constraint under all area constraints, OPLE managed to find the optimum solutions for all benchmarks (the same as those obtained by the ILP).

6. CONCLUSION

In this article, a heuristic custom instruction selection called OPLE was presented. The algorithm made use of both Greedy and optimal optimization techniques. The proposed algorithm searches for the near-optimal solution by using the divide and conquer approach. First, the search space is divided into parts. Then, the near-optimal candidate CIs from each part are found. During the generation of the candidate set of the parts, the recurrences of the CIs are also considered. The suggested CIs from different parts are combined to extract the final CI set. The final CI set is improved by iteratively running the proposed selection method for increasing the speedup of the extensible processor. To evaluate the efficacy of the proposed method, its speedups were compared to those obtained from the optimal and nonoptimal approaches for some benchmarks under different area budgets and I/O constraints. The results showed that the proposed method outperformed the nonoptimal approaches especially for large applications.

REFERENCES

N. Arora, K. Chandramohan, N. Pothineni, and A. Kumar. 2010. Instruction selection in ASIP synthesis using functional matching. In *Proceedings of International Conference on VLSI Design*. 146–151.

K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. 2008. CHIPS custom hardware instruction processor synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 3 (2008), 528–541.

K. Atasu, W. Luk, O. Mencer, C. Özturan, and G. Dündar. 2012. FISH: Fast instruction SyntHesis for custom processors. *IEEE Transactions on VLSI Systems* 20, 1 (2012), 52–65.

P. Biswas, N. D. Dutt, L. Pozzi, and P. Ienne. 2007. Introduction of architecturally visible storage in instruction set extensions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 3 (2007), 435–446.

P. Bonzini and L. Pozzi. 2007. Polynomial-time subgraph enumeration for automated instruction set extension. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'07)*. 1–6.

P. Bonzini and L. Pozzi. 2008. Recurrence-aware instruction set selection for extensible embedded processors. *IEEE Transactions on Very Large Scale Integrations (VLSI) Systems* 16, 10 (2008), 1259–1267.

N. Clark, A. Hormati, S. Mahlke, and S. Yehia. 2006. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 147–157.

N. T. Clark, H. Zhong, and S. A. Mahlke. 2005. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers* 54, 10 (2005), 1258–1270.

C. M. Fiduccia and R. M. Mattheyses. 1982. A linear-time for improving network partitions. In *Proceedings of the Design Automation Conference (DAC'82)*. 175–181.

FreePDK. 2010. A Free OpenAccess 45nm PDK and Cell Library for university. http://www.eda.ncsu.edu.

C. Galuzzi and K. Bertels. 2011. The instruction-set extension problem: A survey. *ACM Transactions on Reconfigurable Technology and Systems* 4, 18 (2011), 1–28.

R. E. Gonzalez. 2000. XTENSA: A configurable and extensible processor. *IEEE Micro* 20, 2 (2000), 60–70.

Gurobi. 2015. Gurobi Optimization. http://www.gurobi.com/.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*. 3–14.

M. Kamal, N. Kazemian-Amiri, A. Kamran, S. A. Hoseini, M. Dehyadegari, and H. Noori. 2010. Dual-purpose custom instruction identification algorithm based on particle swarm optimization. In *Proceedings of the 21st IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'10)*. 159–166.

M. Kamal, A. Afzali-Kusha, and M. Pedram. 2011. Timing variation-aware custom instruction extension technique. In *Proceedings of the Design, Automation and Test in Europe (DATE'11)*. 1517–1520.

K. Karuri, A. Chattopadhyay, M. Hohenauer, R. Leupers, G. Ascheid, and H. Meyr. 2007. Increasing data-bandwidth to instruction-set extensions through register clustering. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'07)*. 166–171.

K. Keutzer, S. Malik, and A. R. Newton. 2002. From ASIC to ASIP: The next design discontinuity. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*. 84–90.

S. K. Lam, T. Srikanthan, and C. T. Clarke. 2009. Selecting profitable custom instructions for area-time-efficient realization on reconfigurable architectures. *IEEE Transactions on Industrial Electronics* 56, 10 (2009), 3998–4005.

C. Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. 330–335.

T. Li, W. Jigang, S. Lam, T. Srikanthan, and X. Lu. 2009. Efficient heuristic algorithm for rapid custom-instruction selection. In *Proceedings of the IEEE/ACIS International Conference on Computer and Information Science*. 266–270.

S. Liao and S. Devadas. 1997. Solving covering problems using LPR-based lower bounds. In *Proceedings of the 34th Annual Conference on Design Automation (DAC'97)*. 117–120.

Y. S. Lu, L. Shen, L. B. Huang, Z. Y. Wang, and N. Xiao. 2009. Optimal subgraph covering for customisable VLIW processors. *IET Computer and Digital Techniques* 3 (2009), 14–23.

Y. Pan and T. Mitra. 2004. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the Design Automation Conference (DAC'04)*. 723–728.

A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli. 2003. Automatic instruction set extension and utilization for embedded processors. In *Proceedings of the Application-Specific Systems, Architectures, and Processors (ASAP'03)*. 108–118.

N. Pothineni, A. Kumar, and K. Paul. 2008. Exhaustive enumeration of legal custom instructions for extensible processors. In *Proceedings of International Conference on VLSI Design*. 261–266.

L. Pozzi, K. Atasu, and P. Ienne. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (2006), 1209–1229.

R. Ramaswamy and T. Wolf. 2003. PacketBench: A tool for workload characterization of network processing. In *Proceedings of the IEEE International Workshop on Workload Characterization*. 42–50.

J. Reddington and K. Atasu. 2012. Complexity of Computing Convex Subgraphs in Custom Instruction Synthesis. *IEEE Transactions on VLSI Systems* 20, 12 (2012), 2337–2341.

H. Scharwaechter, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr. 2011. A retargetable framework for compiler/architecture co-development. *Design Automation for Embedded Systems* 15 (2011), 1–32.

D. C. Schmidt and L. E. Druffel. 1976. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM* 23, 3 (1976), 433–445.

SNU. 2015. SNU Real Time Benchmarks. http://www.cprover.org/goto-cc/examples/snu.html.

A. Verma, P. Brisk, and P. Ienne. 2007. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'07)*. 125–134.

A. Verma, P. Brisk, and P. Ienne. 2010. Fast, nearly optimal ISE identification with I/O serialization through maximal clique enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 3 (2010), 341–354.

C. Xiao and E. Casseau. 2011. An efficient algorithm for custom instruction enumeration. In *Proceedings of the 21st Edition of the Great Lakes Symposium on VLSI*. 187–192.

Received April 2014; revised February 2015; accepted April 2015