Implementation-Aware Selection of the Custom Instruction Set for Extensible Processors

Amir Yazdanbakhsh[1*], Mehdi Kamal[1], Sied Mehdi Fakhraie[1], Ali Afzali Kusha[1], Saeed Safari[1], Massoud Pedram[2]

[1]School of Electrical and Computer Engineering, University of Tehran, Iran

[2]EE Department, University of Southern California, USA

a.yazdanbakhsh@gatech.edu, {mehdikamal, fakhraie, afzali, saeed}@ut.ac.ir, pedram@usc.edu

Abstract- This[*] paper presents an approach for incorporating the effect of various logic synthesis options and logic level implementations into the custom instruction (CI) selection for extensible processors. This effect translates into the availability of a piecewise continuous spectrum of delay versus area choices for each CI, which in turn influences the selection of the CI set that maximizes the speedup per area cost (SPA) metric. The effectiveness of the proposed approach is evaluated by applying it to several benchmarks and comparing the results with those of a conventional technique. We also apply the methodology to the existing serialization algorithms aimed at relaxing register file constraints in multi-cycle custom instruction design. The comparison shows considerable improvements in the speedup per area compared to the custom instruction selection algorithms under the same area-budget constraint.

Keywords: Extensible Processor; Design space exploration; Hardware/Software codesign; Application Specific Instruction set Processors; Microarchitecture.

I. Introduction

Increased rate of embedded applications calls for high performance, low power consumption, flexibility, and cost efficiency of systems that realize such applications. Extensible processors have emerged in the field of embedded computing as a promising approach to remedy many shortcomings of ASICs and general-purpose processors [1]. This approach exploits a simple general-purpose processor and extends its instruction set architecture with beneficial custom instructions (CIs) to provide flexibility and high performance [2]. In designing these processors, the runtime behavior of applications in the target domain are analyzed to determine the critical code segments of the applications. Based on this information, the base processor is augmented with a number of special instructions (custom instructions) for the computationally intensive parts of the code.

Various algorithms have been developed to identify and select the CIs in order to minimize the execution time of the underlying applications in the target domain [2]-[7]. In the CI identification phase, a pool of feasible CIs is determined subject to meeting pre-defined constraints (i.e., I/O constraint) whereas in the CI selection phase, a subset of identified CIs is chosen based on the specified objective function(s) and subject to constraint(s) on the layout area.

During the high-level synthesis process, different implementations of each primitive operation are explored in order to generate a area-delay Pareto optimal curve for that operation [8]. Previous works on CI selection for extensible processors has considered only a single point in the design space and ignored the role of subsequent logic synthesis and optimizations on the physical implementation of the design when identifying the most effective CIs.

In this paper, we propose a framework to improve the design of extended processors by considering different implementations of a primitive for a selected custom instruction. This is achieved by considering the Pareto optimal curve (delay versus area) of the CIs. Using this method, we are able to determine the best implementation where the area usage of the CI is minimum while the propagation delay of the CI does not violate the propagation delay constraint. This exploration which enables us to employ more CIs in a predefined area budget and also, more speed gain, is performed before the CI selection phase of the design flow. The results show that applying the technique gives rise to a considerable improvement in the speed enhancement of the extended processors compared to the case of the conventional design approach in which fixed delay and area is considered for each primitive operation. To the best of our knowledge, this investigation has not reported for ASIPs in the literature. Additionally, to have a reasonably low runtime, we had to use an efficiently fast heuristic technique, which is called WALK, to find (near) optimal implementation of each CI. This problem can be reduced to Knapsack problem, which is an NP-hard algorithm.

The remainder of this paper is organized as follows. In Section II, related works are briefly reviewed while motivation of the work along with the problem definition and formulation are presented in Section III.

[*]Presently at College of Computing, Georgia Institute of Technology, USA

Section IV describes the proposed approach and algorithm. Experimental setup and results are discussed in Section V. Finally, Section VI concludes the paper.

II. Related Works

There are several works in the literature focusing on CI identification and selection algorithm (see, e.g., [2], [10]-[18]). In [2], an enumeration algorithm aimed at generating all valid CIs considering just convexity and I/O constraints was presented. The concept of binary decision tree was utilized to search among all valid sub-graphs. Each internal node represented a potential sub-graph, which was analyzed based on the specified constraints to identify the validity of the enumerated CIs. The inclusion or exclusion of a node in a sub-graph was distinguished by the movement towards the right or the left branch, respectively. An approach similar to [2] and an exact algorithm to enumerate the entire feasible CIs in a reasonable time were provided in [10].

Different approaches to reduce the computation time in the CI identification phase have been proposed in [11]-[15]. While the architectural space is comprehensively explored in these works, the selection of the CIs is accomplished based on a constant area-delay table derived from the synthesis tool regardless of logic-level implementation of the primitive cells. Different algorithms for high-level CI identification for extensible processors have been proposed in [16]-[18]. In [19], the arithmetic operations of selected custom instructions are optimized. In this work, to improve the speedup, the normal hardware blocks were replaced by the delay-optimized ones.

In [20], a design flow for reconfigurable ASIPs (rASIPs) has been proposed. In the proposed design flow, where the processor was described by using LISA language, the custom instructions were extracted for mapping them to a coarse grained reconfigurable architecture. In this work, the CI extraction method of [22], which did not consider the area usage of the CIs during the selection phase, was used. A framework for performance and area trade-off evaluation in the CI extraction has been proposed in [24]. The information about how the area usage of each identified CI has been extracted was not presented in detail. Clearly, no area usage optimization of the identified CIs before the selection phase has been utilized. In [25], a reconfigurable transparent accelerator based on the look-up table was proposed. Designing this accelerator, called Programmable Carry Function Unit (PCFU), was the main focus of the paper. In [26], similar to [25], a transparent accelerator, named Configurable Compute Accelerator (CCA), has been proposed.

An ILP-based CI identification framework, which extracts CIs from the critical code segment, has been proposed in [27]. The extraction was performed using the available data bandwidth and transfer latencies between custom logic and a baseline processor. In [28], a method to expedite CI generation from the high-level application descriptions was discussed. An approach for increasing the number of I/O ports of the CFU to access the General Purpose Registers (GPRs) has been suggested in [29]. It was based on the existing idea of register clustering in VLIW processors without a significant increase in the size of the GPR files. In [30], an automated ISE synthesis, which consider both the user-specified and processor-specific constraints have been proposed. The authors did not provide the details of the CI area usage estimation and the way that the usage is considered in the selection phase. In [6], a framework for estimating the area utilization and latencies of custom instructions on lookup-table based commercial FPGAs was proposed. In [21], multiple implementations per each special (custom) instruction were added to the extensible processor. A run-time system was proposed to dynamically select the appropriate variation of the special instruction based on the available hardware resources. This work introduced a novel run-time adaptive extensible processor to increase the hardware usage efficiency.

While the aforementioned works have introduced novel algorithms and architectures to increase the flexibility and reconfigurability of custom instruction selection and implementation, they have not considered different synthesis constraints (in terms of area and delay) for each custom instruction during the CI selection phase in the ASIP design flow. We should mention that the idea of using different area-delay implementation of primitives has been used in other fields of digital circuit design such as high level synthesis [8] and reconfigurable architecture design [20]. In these fields, the objective functions and as well the granularity of the exploration are different than those used in the ASIP design flow considered in this work.

III. Motivation and Problem Formulation

In this section, first, we describe the motivation of the work by an example. Next, key concepts in the field of custom instruction are formally presented. Also, the problem of finding a subset of CIs that maximizes the total speedup per area while satisfying an area budget constraint is formulated.

A. Motivation

The CI selection algorithms select a subset of instructions from the generated CIs in the CI identification phase such that the *speedup per area* (SPA) metric is maximized while an area budget constraint is not violated [2][6]. Previous works consider a constant value for the delay and area of the primitive operations. Whereas, we consider different logic implementations (different delays and areas) of the primitive operations in the algorithm to maximize the SPA parameter. Note that the SPA merit function is utilized in the CI selection phase which is after the CI identification phase. The proposed technique is applied in the stage between identification and selection phases and its efficiency is independent from the merit function used in the selection phase. Hence, in this work, without loss of generality, we use the SPA metric for the CI selection under a predefined area budget.

Figure 1 shows two custom instructions generated from a data flow graph example under micro-architectural constraints. The CI selection algorithm should make a decision between these two CIs and select the one which is optimum in terms of the SPA metric. In the case of the first CI, we assume that the area and delay are fixed, and hence, SPA is fixed too. In the case of the second CI, the (32-bit) adder primitive is synthesized under different delay constraints assuming fixed areas and delays for the other primitives. Different delay constraints are synthesized with different area values due to different logic implementations. The area versus delay characteristic for this primitive is depicted in Figure 2. The characteristic is an area-delay Pareto optimal curve. As shown in this figure, the area of the synthesized 32-bit adder decreases as the delay constraint increases. The discontinuity in the characteristic originates from the use of two micro-architecture implementations of the adder (carry look-ahead adder on the left and ripple-carry adder on the right.)

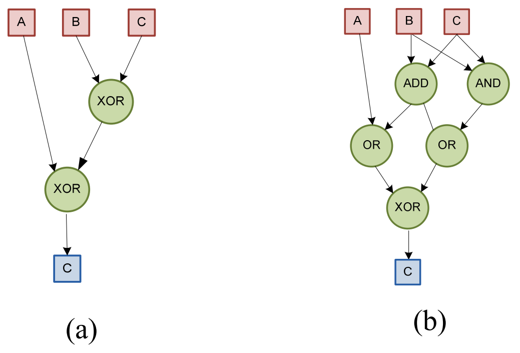


Figure 1. Two feasible custom instructions generated under micro-architectural constraints. a) CI(a) b) CI(b).

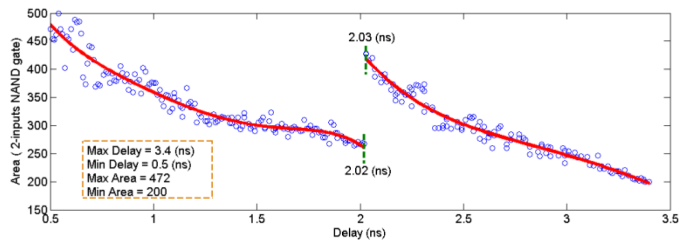


Figure 2. Area versus delay Pareto optimal curve for a 32-bit addition primitive.

The total area and critical path delay of the two CIs are calculated as follows:

Custom Instruction (a):

$$CP_a = D(XOR) + D(XOR) = 100\,ps$$

$$Area_a = 2\,A(XOR) = 128$$

Custom Instruction (b):

$$CP_b = D(ADD) + D(OR) + D(XOR) = D(ADD) + 90\,ps$$

$$Area_b = A(ADD) + 2\,A(OR) + A(AND) + A(XOR) = A(ADD) + 189$$

where CP stands for critical path delay of the CIs and $A(x)$ and $D(x)$ show the area and delay values of the operation x, respectively.

Table 1. Delay and area of some primitive operations. The area values are normalized to the area of a 2-inputs NAND gate.

Operation	Delay(ps)	Area (# of two-inputs NAND gate)
AND (32-bit)	40	41
OR (32-bit)	40	42
XOR (32-bit)	50	64

In Figure 3, the SPA versus delay curves of the adder primitive (obtained using Figure 2 and Table 1) for the two CIs are presented. These characteristics suggest that the decision on the CI selection depends on the delay constraint imposed by the system designer. The dashed line corresponds to the SPA of CI(a). It means that CI(b) are preferred to be selected in for the delays in the range of 1.21ns and 2.02ns and greater than 2.36ns while for the others the CI(a) should be selected.

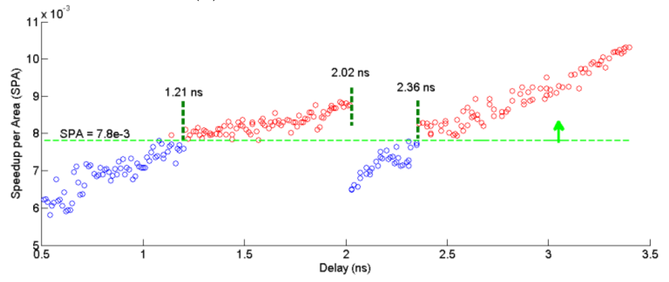


Figure 3. Speedup per Area (SPA) versus delay constraint for each of the CIs given in Figure 1.

This simple example demonstrates that integrating logic-level synthesis information into the architecture-level CI selection algorithms can potentially improve the SPA parameter. In this paper, we present a CI extraction method in which CIs are selected based on the logic-synthesis information of the primitives.

B. Problem Formulation

A data flow graph (DFG), G = (V∪V$_{in}$, E), is an acyclic directed graph where V and V$_{in}$ denote the sets of primitive operations in the basic processor and input variables of the basic blocks, respectively, and E, is the set of edges representing the data dependencies between the operations[5]. A custom instruction C is a sub-graph of G, induced by the subset of the nodes in V. The total number of predecessor nodes that are not in C and have at least one head endpoint to a node in C is denoted by IN$_C$. Similarly, the total number of nodes in C that have at least one head endpoint to the nodes that are not in C is denoted by OUT$_C$. In other words, IN$_C$ and OUT$_C$ represent the number of input values used by the primitive operations in C and the number of values produced by C to be used in other operations not in C, respectively. Also, the *convexity* of the custom instruction C means that the sequence of vertices in every path between two nodes in C comprises of nodes that also belong to C. The reason for considering convex CIs is to ensure the existence of a feasible scheduling solution [5].

Finally, counts of permissible read and write ports of the register file, which are imposed by the micro-architectural features, are denoted by R$_{in}$ and R$_{out}$. Since we do not allow accessing memory in the execution pipeline stage, we consider the memory operations (e.g. load and store) as forbidden nodes and exclude them from the custom instruction identification (see, e.g., [10][14]).

The problems of CI identification and selection may be stated as follows:

Problem 1: Given a DFG of an application, enumerate the set of feasible sub-graphs (CIs) considering the following constraints:

1) $IN_C \leq R_{in}$,
2) $OUT_C \leq R_{out}$,
3) There should not be any forbidden nodes in the enumerated sub-graphs, and
4) Each sub-graph should be convex.

Problem 2: Given the set of feasible CIs, the maximum delay constraint, Delay$_{max}$, and the area-delay Pareto optimal characteristic for the primitive operations, select the optimal CI set (in terms of the maximum performance and minimum area usage) that are pair wise disjoint and meet the specified area-budget constraint, Area$_{max}$. This problem can be formulated as follows:

$$Maximize \frac{\sum_{i \in set\ of\ selected\ CIs} P_i}{\sum_{i \in set\ of\ selected\ CIs} A_i} \equiv SPA$$

$$s.t. \sum_{i \in set\ of\ candidate\ CIs} A_i \leq Area_{max}; and$$

$$\forall i \in set\ of\ candidate\ CIs; CP_i \leq Delay_{max}$$

where *Delay*$_{max}$ is the maximum allowed latency for the computation of the custom instructions. In the cases where this delay is more than one clock cycle, one should customize the pipeline of the underlying processor to allow for the execution of multi-cycle CIs [34]-[36]. Note that due to the existence of the area-delay characteristic for each primitive operation, there is a function relating the area of each CI to the delays of primitive. The performance improvement, P_i, is calculated for each custom instruction. The maximum performance improvement is limited by the micro-architectural constraints that are formulated in [4].

IV. Proposed Approach

To bridge the gap between existing architecture-level custom instruction selection algorithms and the logic-level synthesis parameters such as delay and area, the framework, which is shown in Figure 4 was developed. Using this frame work, logic-level synthesis information is provided to high-level CI selection algorithms. The approach starts with the source code of the domain-specific benchmarks in C/C++, which are compiled to extract the DFG for future processing. Furthermore, the iteration number of each basic block and the code coverage of the benchmarks are extracted by applying inputs. Afterwards, all primitive operations (e.g. add, subtract, shift, etc.) are synthesized to obtain area-delay Pareto optimal characteristics. Next, the evaluated characteristics are stored into a logic-level synthesis library which is exploited during the architecture-level CI selection. The architecture-level CI selection comprises of the following two steps: (1) match enumeration (step (f)), and (2) template generation (step (g)). More information regarding these steps may be found in [13].

The template pool and area-delay Pareto optimal curve library obtained from logic-level synthesis are combined together in step (h) to form the integrated framework. In step (i), we use an optimization algorithm called WALK to search among the possible set of CIs in order to maximize the obtainable speed while not exceeding the specified area budget.

Thus, before applying the TemplateSelection function on the candidate custom instructions, the WALK algorithm is called. This algorithm would find the minimum area for each CI using the area-delay Pareto-optimal curve (space) of the primitive operations. The pseudo code of the proposed algorithm is shown in Figure 5. We should note that the area-delay curve is extracted by synthesis of the primitives under different delay constraints. Since the points obtained may not be necessarily continuous, instead of Pareto optimal curve, we may use Pareto optimal space (see, e.g., Figure 2).

The input of this algorithm is a CI, and its output is the area and delay of each nodes of the CI. The algorithm attempts to analyze different delays for the nodes of the CI to find the minimum area of the CI without violating the delay constraint. First, the algorithm assigns the minimum delay value to each node based on the area-delay curve/space (line 3). The main part of the algorithm consists of two loops. The inner loop (line 4) tries to minimize the area by increasing the delay of each node.

The node delay is increased by moving from one point to another point in the space for each call of the main loop. The jump step in the space is determined by the outer loop. Since there are different slopes for the area-delay curves of the primitives, one may not use a fixed jump step. For example, the adder area-delay curve in Figure 2 has positive and negative slopes in different regions. A fixed step in these cases may trap the algorithm in local minima. In each iteration, the while loop increases the delay of the node (line 6) which decreases the area more compared to those of the other nodes (lines 9-11). After finishing the inner loop, the new area of the CI is compared to its minimum area obtained before this jump step. If it is lower, then the new area becomes the minimum area (lines 20-23).

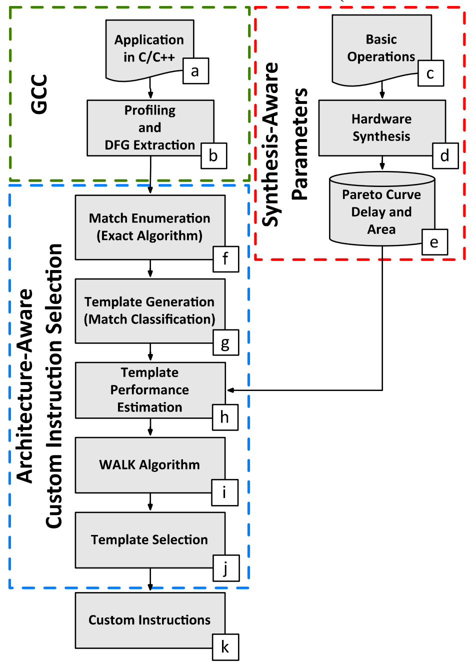

GCC
Application in C/C++
a
Profiling and DFG Extraction
b
Synthesis-Aware Parameters
Basic Operations
c
Hardware Synthesis
d
Pareto Curve Delay and Area
e
Architecture-Aware Custom Instruction Selection
Match Enumeration (Exact Algorithm)
f
Template Generation (Match Classification)
g
Template Performance Estimation
h
WALK Algorithm
i
Template Selection
j
Custom Instructions
k

Figure 4. Overview of the proposed architecture- and logic-level integration framework. The set of custom instruction is generated under various constraints regarding logic-level parameters.

```
1: WALK (Template T)
2:   FOR (JumpStep = 1; JumStep < MaxJumpStep; JumpStep++) //OuterLoop
3:     Assigning Minimum Delay to all Nodes of the T
4:     WHILE (True) //Inner Loop
5:       FOREACH (Node N of the T)
6:         Walk on the area-delay space of the N by jumping step of \
               JumpStep until reach the minimum area
7:         AreaReduction    = Reduced area of T by reducing the area of N
8:         PropagationDelay = Propagation Delay of T by reducing the \
                                   area of N
9:         IF (AreaReduction < Max_AreaReduction) && \
                 (PropagationDelay <= PropagationDelayConstraint)
10:          CandidateNode = N
11:        END IF
12:        IF (CandidateNode = NULL)
13:          BREAK; // Go to line 20
14:        ELSE
15:          Update the area-delay of CandidateNode
16:          Update the area-delay of T
17:        END IF
18:      END FOREACH
19:    END WHILE
20:    IF (T.Area < MinimumTArea)
21:      BestAreaDelay =  area-delay of Nodes
22:      MinimumTArea = T.Area;
23:    END IF
24: RETURN BestAreaDelay
```

Figure 5. The pseudo code of the WALK algorithm.

V. Results and Discussion

In this section, we compare conventional custom instruction selection approaches in which only one implementation of the primitive function is considered (fixed values for delay and area) with our proposed approach.

The area and delay of a CI depend on the area and delay of the primitives used in the CI. The area is equal to the summation of the area of all the primitives and the delay of the CI is equal to the summation of the delay of the primitives in the critical path. Hence, to calculate the area and delay of each CI, first, we need to find the area and delay of each primitive. The primitive area and delay are calculated using the synthesis tool. In the conventional approach, for each primitive, only one implementation is considered. Because in the ISA extension problem the goal is to increase speed enhancement, in the conventional approach, for each primitive, the implementation with the lower propagation delay is used. The area and delay of these implementations are used in the identification and selection phases. Hence, in this paper, for both the conventional and proposed approaches, first, the primitives were synthesized based on the minimum delay criterion. In the identification phase, only the CIs whose delays were smaller than the predefined maximum propagation delay constraint were identified. Then, the delay and the area of the CIs were calculated based on these area and delay values. Next, only for the proposed scheme, after the identification, the area-delay space was explored to find the best CI implementation. Note that the constraints and objectives of the ISA extension for both proposed and conventional method were similar.

These two approached were applied on some embedded applications as benchmark suits. For this purpose, we used seven domain-specific embedded applications from a wide range of domains. The selected benchmarks included IPSec and MD5 from PacketBench [31], LMS and ADPCM from the SNU-RT benchmark suits [32], and G271 Encode/Decode and BitCounter from MiBench[33]. All the applications were compiled by GCC to generate the 3-address intermediate representation, called GIMPLE. We also implemented a parser to extract the DFG of the basic blocks in all the benchmarks. As an example, Figure 6 shows one of the basic blocks of the G721decoder benchmark (a basic block in Quantize function).

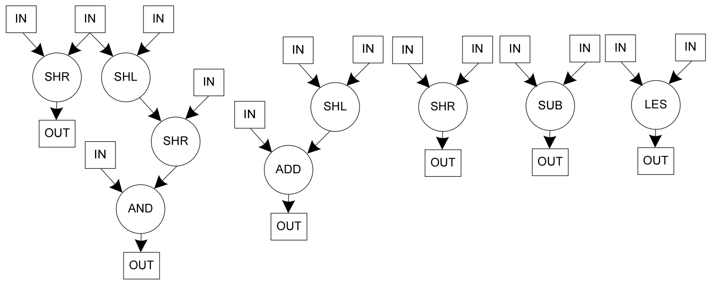


Figure 6. The DFG of a basic block in Quantize function of the G721decoder benchmark.

To obtain area-delay Pareto optimal characteristics, all primitive operations were synthesized using the Synopsys DC synthesis tool and a 90nm standard cell library. We imposed a wide range of delay constraints during the synthesis and selected those who met the constraint. All area values were normalized to the area of a two-input nand gate. Finally, the discrete set of 2-tuple points (delay, area) for each primitive was stored in the library to be utilized for the template selection algorithm of the framework depicted in Figure 4. In Table 2, some details about the 32-bit primitives which we use to generate the CIs are presented. The table contains the minimum and maximum values for the area and delay of each primitive as well as the number of the optimal Pareto curves that were considered during the area-delay exploration using the WALK algorithm.

In this work, for both conventional and proposed methods, the maximum propagation delay and also the I/O constraints were considered as 1 ns and 4/4, respectively. As an example, Figure 7 depicts the selected CIs by two methods for the G721decoder benchmark when the area constraint was 350. In this case, the performance improvements for the conventional and proposed approaches were 6.1% and 8%, respectively.

Table 2. Minimum and maximum values for the area and delay of each primitive and also, the number of the optimal Pareto curves.

Primitive Name	Minimum		Maximum		Points in Pareto-Optimal Curve
	Area	Delay(ns)	Area	Delay (ns)	
SUB	225	0.5	650	3.44	289
ADD	200	0.5	472	3.4	267
SHR / SHL	326	0.19	1358	0.47	28
EQT / NEQ	87	0.16	295	0.22	7
GRT / LES	115	0.21	315	0.69	44
AND	41	0.04	41	0.04	1
OR	42	0.05	42	0.05	1
XOR	64	0.05	64	0.05	1

In the following discussion, we use the term "*Saturation Point*" to denote the upper bound on area constraint value above which the attainable performance will not improve anymore.

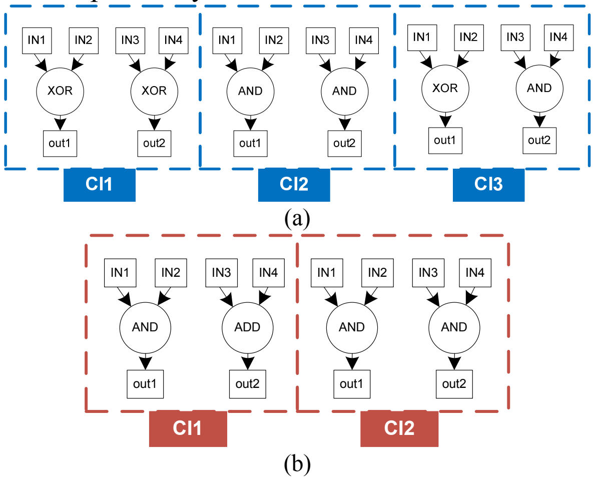


Figure 7. The CIs obtained by the (a) conventional and (b) proposed approaches in the case of the G721benchmark when the area constraint is 400.

SPA results of applying both approaches for the benchmarks are reported in Figure 8. The results show that the proposed algorithm outperforms the conventional approach under all area-budget constraints. When comparing the approaches for these benchmarks, two cases are encountered. In the first case, the two curves converge to each other by increasing the area constraint. This case happens for all benchmarks except for IPSec. In the second case, the performance improvement of the proposed approach is higher for all area-budget constraints. For the results belonging to the first case, the proposed algorithm always reaches the saturation point at lower area constraints (costs) than those for the conventional algorithm. The saturation points for each of the two algorithms are shown with arrows in Figure 8. In the second case (IPSec benchmark), except for the area constraint of 500 where the performance improvements of the two approaches are the same, the proposed algorithm leads to a higher improvement for a given area constraint.

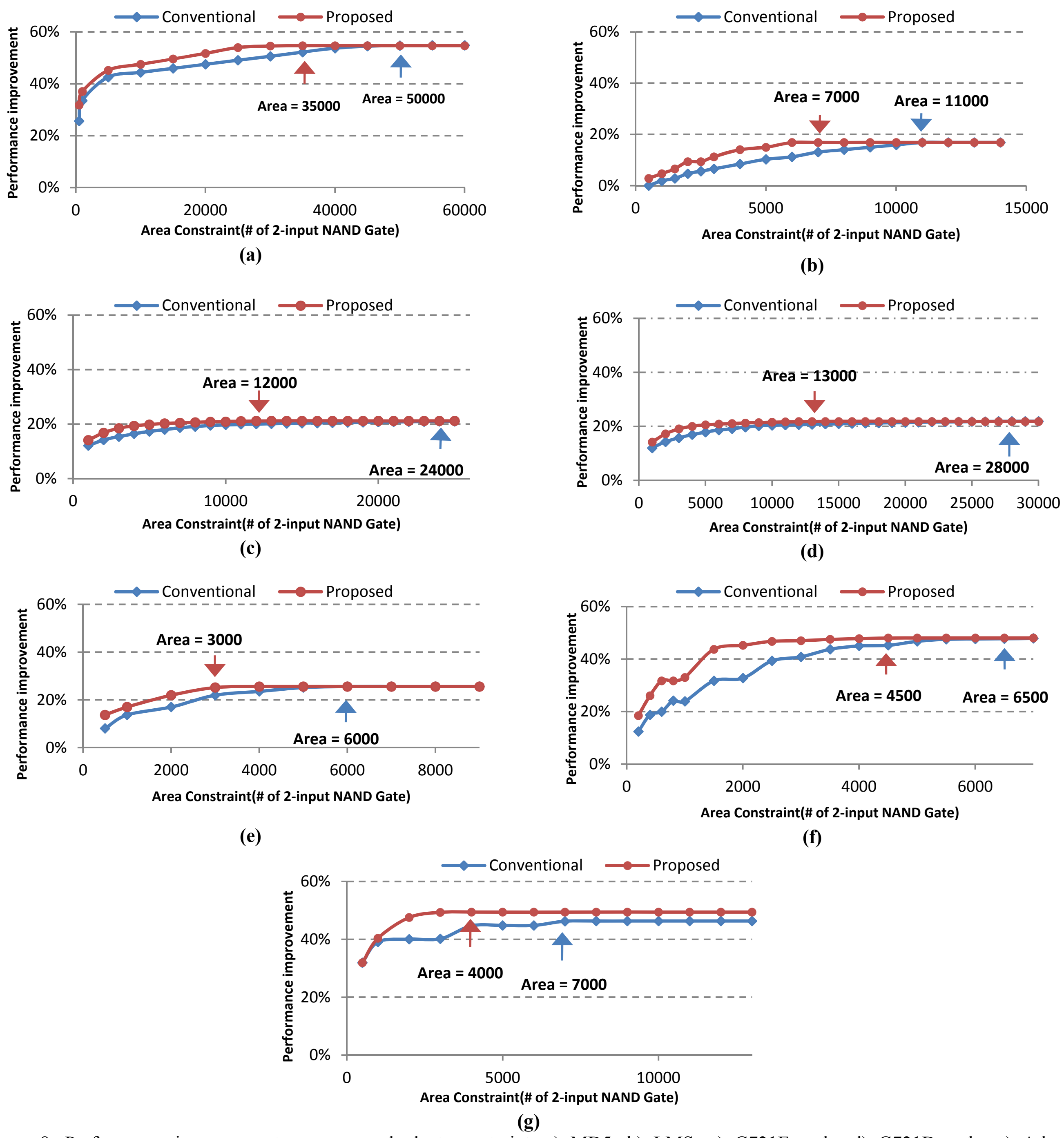
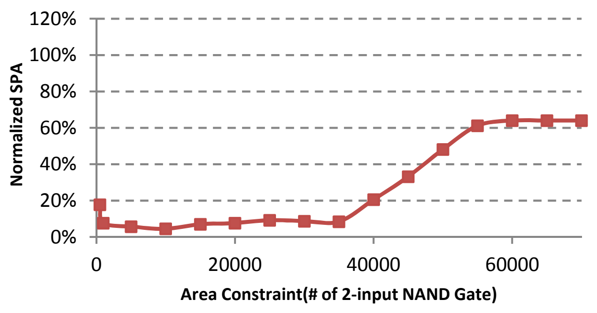
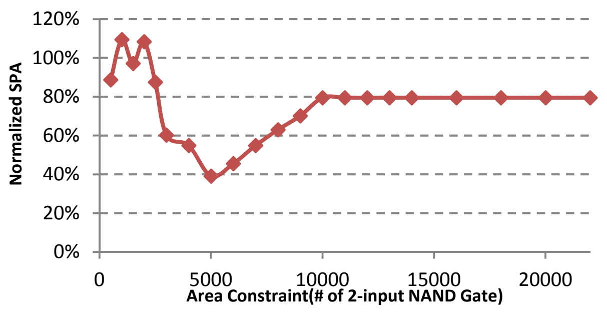


Figure 8. Performance improvement versus area-budget constraint. a) MD5, b) LMS, c) G721Encode, d) G721Decode, e) Adpcm, f) BitCounter, g) IPSec.

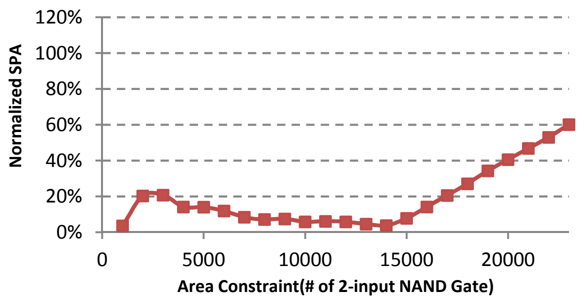
As mentioned before, selecting CIs is performed by maximizing the SPA metric. Figure 9 shows SPA values of the selected CIs obtained by the proposed algorithm normalized to those of the conventional method. As the results in this figure show, for all the benchmarks, the *normalized SPA* values are positive (except for few area-budget constraints which the values are zero) showing that the proposed method outperforms the conventional approach. However, after the saturation point, the performance gains of the two approaches become equal. The reason for the saturation of both approaches is that the conflicts due to overlaps between CIs prevent adding further CIs. Therefore, increasing area constraint does not enable us to add further CIs for improving the speedup. It should be noted that since our proposed approach is more efficient compared to the conventional approach, it reaches the saturation point at smaller areas.



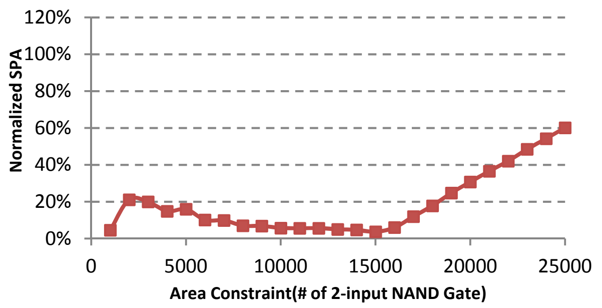
(a)



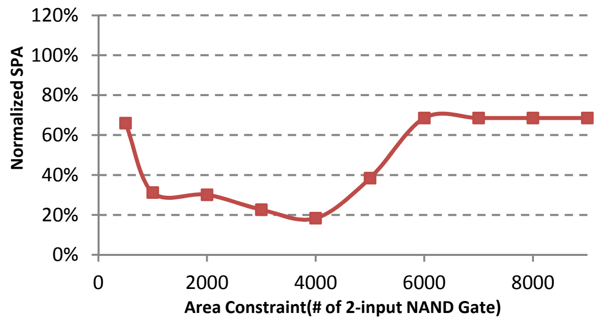
(b)



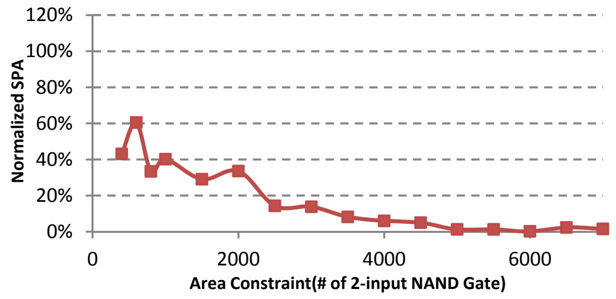
(c)



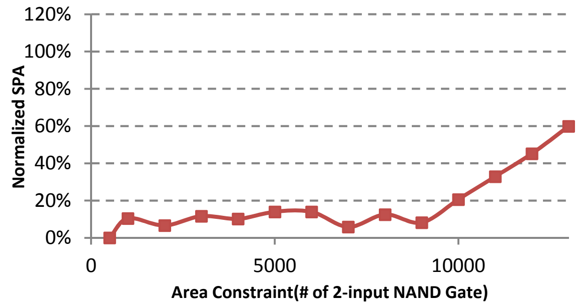
(d)



(e)



(f)



(g)

Figure 9. The speedup per area metric of the proposed algorithms normalized to that of the conventional algorithm. a) MD5, b) LMS, c) G721Encode, d) G721Decode, e) Adpcm, f) BitCounter, g) IPSec.

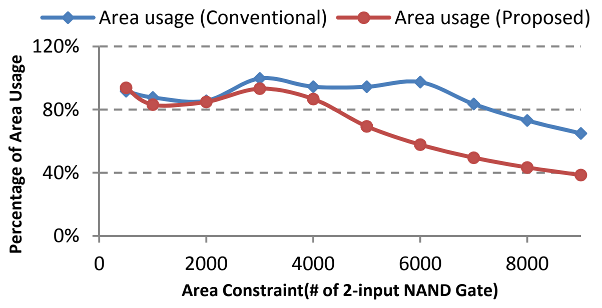
The increase in SPA may be achieved by either increasing the speedup factor or reducing the hardware cost. As shown in Figure 8, the speedup of the proposed algorithm is higher than that of the conventional approach. Additionally, the area usage should be less than or equal to the area-budget constraint. The average area usage of the two approaches under different area constraints are shown in **Error! Not a valid bookmark self-reference.**. In nearly all cases, the results show that the average area usage of the proposed is less than that of the conventional approach. In the case of Bitcounter, the proposed technique used more area on average while the performance improvement was higher too leading to higher SPA values. To make this concept clearer, the percentage of area budget usage in both approaches for Adpcm, LMS, and MD5 benchmarks are demonstrated in Figure 10 under different area constraints, respectively. These results indicate that for higher area-budget constraints, the proposed approach has a lower area usage and higher or equal performance improvements.

Table 3. Average area usage and area reduction percentage for the two approaches.

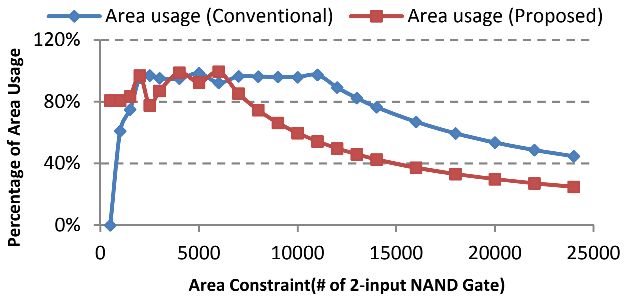
Benchmarks	Average Area Usage (# of 2-input NAND gate)		$\frac{Avg.\ AreaProposed}{Avg.\ AreaConventional}$
	Conventional	Proposed	
IPSec	6155	5404	0.8779
LMS	6917	4591	0.6637
G721Encode	12922	10539	0.8155
G721Decode	15933	11889	0.7461
Adpcm	3792	2660	0.7014
BitCounter	3332	3340	1.0024
MD5	30983	23473	0.7576

To study the efficacy of the proposed algorithm for multi-cycle operations, we again compared its results with those of the conventional algorithm. In multi-cycle operations, the custom instructions are executed in more than one cycle. These instructions become multi-cycled using methods that are categorized as serialization algorithms [5]. In this work, we consider instructions with a delay constraint of three clock cycles. We assumed the working frequency of the base processor was 1GHz in the 90nm CMOS technology.

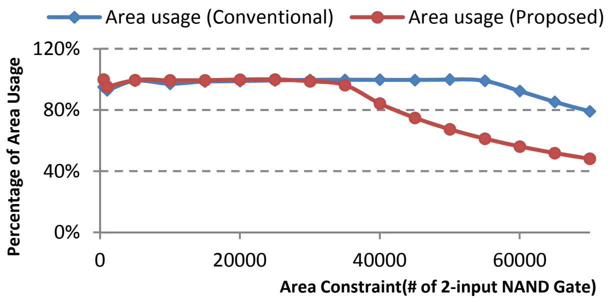
The saturation points for the multi-cycle CIs are presented in Table 4. The results reveal that our approach reaches the saturation point at lower area usage. In terms of performance, in the case of the approach presented in this work, the improvements are the same or higher when compared to those of the conventional technique. In Figure 11, the normalized SPA's and performance improvements are presented. The results show that under all area constraints the performance improvements obtained in our approach is either higher or identical to those of the conventional technique. In the case of normalized SPA values, except for a couple low area constraints in LMS and G721Encode, the achieved SPA metrics by our proposed method is higher than those of the conventional approach. Even for these negative normalized SPA values, (the area constraint 1000 in G721Encode and 1000 and 1500 in LMS benchmarks), the performance improvements are higher.



(a)



(b)

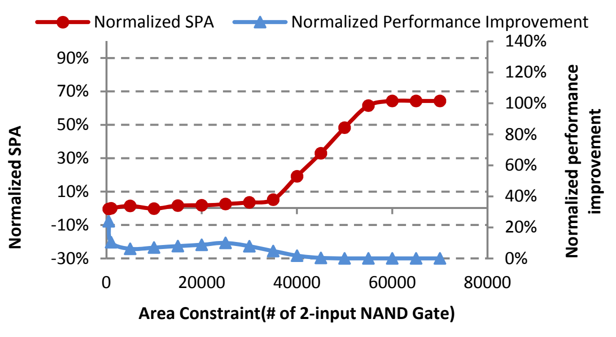


(c)

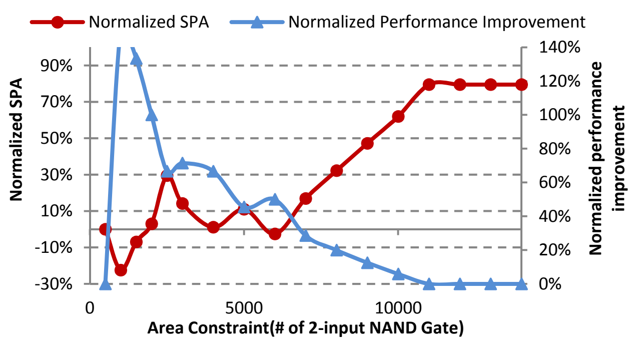
Figure 10. The percentage of area usage of the conventional and proposed approaches under different area constraints. a) Adpcm, b) LMS, c) MD5.

Table 4. Area constraint and Performance (Per.) of saturation points achieved by two approaches.

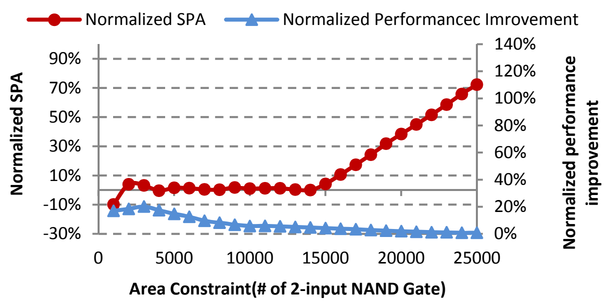
Benchmarks	Saturation Point			
	Conventional		Proposed	
	Area	Per.	Area	Per.
IPSec	8000	46.36%	4000	49.43%
LMS	11000	16.82%	6000	16.82%
G721Encode	24000	20.97%	12000	21.16%
G721Decode	28000	21.92%	13000	22.25%
Adpcm	6000	25.57%	4000	25.57%
BitCounter	9500	48.39%	7500	48.64%
MD5	55000	54.68%	35000	54.68%


Normalized SPA
Normalized Performance Improvement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
20000
40000
60000
80000

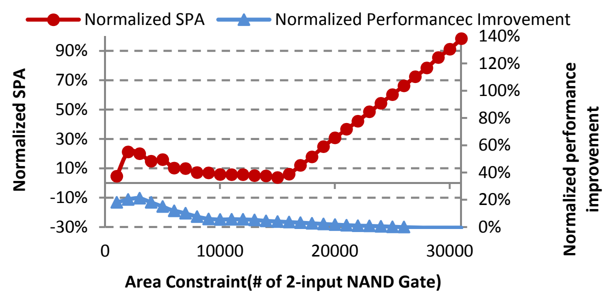
(a)


Normalized SPA
Normalized Performance Improvement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
5000
10000

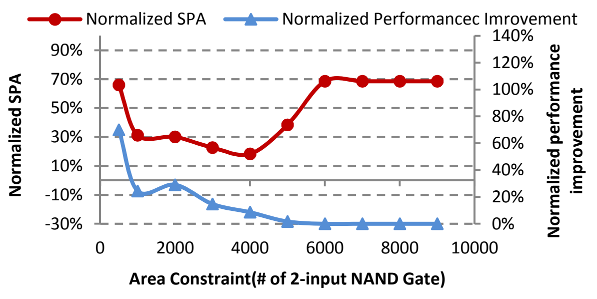
(b)


Normalized SPA
Normalized Performancec Imrovement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
5000
10000
15000
20000
25000

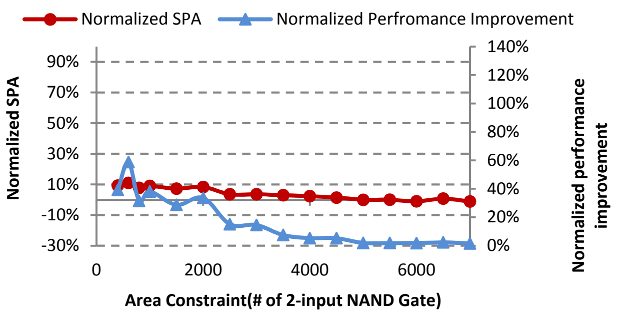
(c)


Normalized SPA
Normalized Performancec Imrovement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
10000
20000
30000

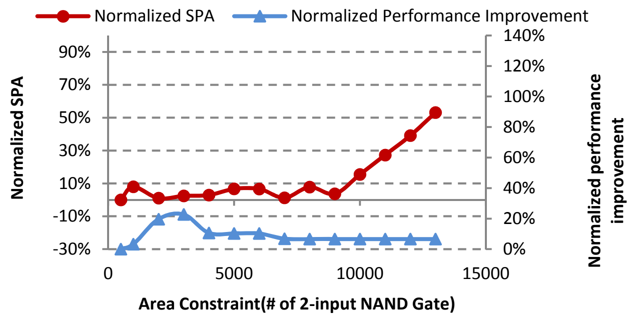
(d)


Normalized SPA
Normalized Performancec Imrovement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
2000
4000
6000
8000
10000

(e)


Normalized SPA
Normalized Perfromance Improvement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
2000
4000
6000

(f)


Normalized SPA
Normalized Performance Improvement
Normalized SPA
Normalized performance improvement
Area Constraint(# of 2-input NAND Gate)
90%
70%
50%
30%
10%
-10%
-30%
140%
120%
100%
80%
60%
40%
20%
0%
0
5000
10000
15000

(g)

Figure 11. Normalized performance improvement achieved by proposed algorithm in compare to conventional method for multi-cycle custom instructions imposing delay constraint equal to two times of clock cycle latency. a) MD5, b) LMS, c) G721Encode, d) G721Decode, e) Adpcm, f) BitCounter, g) IPSec.

VI. Conclusion

In this work, we proposed to integrate logic-level synthesis information into the existing high-level custom instruction selection algorithms for application specific instruction processors (ASIPs). The technique made use of the area-delay Pareto optimal characteristics of primitive operations obtained from the synthesis tool to generate the set of CIs that maximizes a predefined merit function. In this work, for this function, we utilized Speedup per Area (SPA) under the area-budget constraint as the metric function. To explore the area-delay Pareto optimal space, a heuristic algorithm called WALK was used. The algorithm searched among different area-delay possibilities for each primitive in all candidate CIs to reduce the CI area budget. We applied our proposed method to some benchmarks from different domains in embedded processors including seven domain-specific embedded applications from a wide range of domains. The selected benchmarks included IPSec and MD5 from PacketBench, LMS and ADPCM from the SNU-RT benchmark suits, and G271 Encode/Decode and BitCounter from MiBench. Our results, which were obtained for different area constraints, showed that up to 10% improvement in the achievable performance compared to the conventional custom instruction selection algorithms under the same area-budget constraint could be achieved. Furthermore, they revealed that, in most cases, our proposed algorithm reached the maximum attainable performance improvement using lower areas compared to those of the conventional approach. We also applied this methodology to the existing serialization algorithms. The comparison of the results indicated that up to 80% speed up per area improvement compared to the traditional multi-cycle algorithms under the same area constraints may be obtained.

References

[1] K. Keutzer, S. Malik, and A.R. Newton, "From ASIC to ASIP: the next design discontinuity," in Proceedings of International Conference on Computer Design: VLSI in Computers and Processors, 2002, pp. 84-90.

[2] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration," in *IEEE Trans. On Computers,* vol. 54, no. 10, pp. 1258-1270, Oct. 2005.

[3] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. 40th Des. Autom. Conf.*, Jun. 2003, pp. 256–261.

[4] A. Yazdanbakhsh, M. Kamal, M. E. Salehi, H. Noori, and S. M. Fakhraie, "Energy-Aware Registerfile Design Space Exploration for Extensible Processors," in *Proc. 10th International Conference on Embedded Computer Sysmtes: Architectures, Modling and Simulation*, July. 2010, pp. 273-281.

[5] C. Galluzi, and K. Bertels, "The Instruction-set Extension Problem: A Survey," in ACM Transaction on Reconfigurable Technology and Systems, vol 4, no. 2, pp. 18-1:18-28, May, 2011.

[6] S. K. Lam, T. Srikanthan, and C. T. Clarke "Selecting Profitable Custom Instructions for Area-Time-Efficient Realization on Reconfigurable Architectures," in *IEEE Trans. on Industrial Electronics,* vol. 56, issue 10, pp. 3998-4005, Oct. 2009.

[7] A. Yazdanbakhsh, M. E. Salehi, and S. M. Fakhraie "Customized Pipeline and Instruction Set Architecture for Embedded Processing Engines," in Journal of Supercomputing, vol. 68, No. 2, pp. 948-977, Mau. 2014.

[8] Ph. Coussy and A. Morawiec. *High-Level Synthesis from Algorithm to Digital Circtuit*. Springer Science, 2008.

[9] N. H. E. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective.* Pearson Education, 2005, pp. 678.

[10] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 7, pp. 1209–1229, Jul. 2006.

[11] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. Int. Conf. Compilers, Architectures, and Synth. Embed. Syst.*, Sep. 2004, pp. 69–78.

[12] X. Chen, D. L. Maskell, and Y. Sun, "Fast Identification of Custom Instructions for Extensible Processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 359– 368, Feb. 2007.

[13] A. Yazdanbakhsh, M. E. Salehi, S. Safari, and S. M. Fakhrair, "Locality Considerations in Exploring Custom Instruction Selection Algorithms," in *Proc. 2ndAsia Symposium on Quality Electronic DesignConf.*, August. 2010.

[14] P. Biswas, S. Banerjee, N. D. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: An Iterative Improvement-Based ISE Generation Technique for Fast Customization of Processors," *IEEE Trans. Very Large Scale Integration Syst.*, vol 14, no. 7, pp. 754-762, Jul. 2006.

[15] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," *in Proc. DATE*, Apr. 2007, pp. 1331–1336.

[16] T. Li, W. Jigang, S. Lam, T. Srikanthan, and X. Lu, "Efficient Heuristic Algorithm for Rapid Custom-Instruction Selection," in *Proc. 8th IEEE/ACIS International Conference on Computer and Information Science*, 2009, pp. 266-270.

[17] T. Li, W. Jigang, Y. Deng, T. Srikanthan, and X. Lu, "Fast Identification Algorithm for Application-Specific Instruction-Set Extensions," in *Proc. International Conference on Electronic Design*, 2008, pp. 1-5.

[18] K. Seto and M. Fujita, "Custom Instruction Generation with High-Level Synthesis," in *Proc. 6th IEEE Symposium on Application Specific Processors*, June. 2008, pp. 14-19.

[19] A. K. Verma, Y. Zhu, P. Brisk, and P. Ienne, "Arithmetic optimization for custom instruction set synthesis," in *Proc. 7th IEEE Symposium on Application Specific Processors*, July. 2009, pp. 54-57.

[20] L. Jozwiak, N. Nedjah, M. Figueroa, " Modern develpomnet methods and tools for embedded sytems: A survey", in the VLSI Journal of Integration, Vol(43), pp. 1-33, 2010.

[21] L. Bauer, M. Shafique, and J. Henkel, "Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation," in IEEE Transaction on Very Large Scale Integration (VLSI) Systems, vol. 16, 2008.

[22] [Ref-1] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, "A Design Flow for Architecture Exploration and Implementation of Partially Reconfigurable Processors", in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol(16), No.(10), pp. 1281-1294, 2008.

[23] [Ref-2] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained application source code profiling for ASIP design," in Proc. Des. Autom. Conf. (DAC), 2005, pp. 329–334.

[24] [Ref-3] U.D. Bordoloi, H.P. Huynh, S. Chakraborty, and T. Mitra, "Evaluating Design Trade-offs in Customizable Processors, in Proceedings of Design Automation Design (DAC), 2009, pp. 244-249.

[25] [Ref-4] S. Yehia, N. Clark, S. Mahlke, K. Flautner, "Exploring the Design Space of LUT based Transparent Accelerators", in Proceedings of international conference on Compilers, architectures and synthesis for embedded systems (CASES), 2005, pp.11-21.

[26] [Ref-5] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application Specific Processing on a General Purpose Core via Transparent Instruction Set Customization," in Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, 2004, pp. 30-40.

[27] [Ref-6] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk, "CHIPS: Custom Hardware Instruction Processor Synthesis," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), Vol. (27), No. (3), pp.528-541, 2008.

[28] [Ref-7] K. Atasu, W. Luk, O. Mencer, C. Ozturan, and G. Dundar, "FISH: Fast Instruction SyntHesis for Custom Processors," in IEEE Transaction on Very Large Scale Integration (VLSI) Systems, Vol. (20), No. (1), pp. 52-65, 2012.

[29] [Ref-8] K. Karuri, A. Chattopadhyay, M. Hohenauer, R. Leupers, G. Ascheid, and Heinrich Meyr, "Increasing Data-Bandwidth to Instruction-Set Extensions through Register Clustering," in Proc. of IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD), 2007, pp. 166-171.

[30] [Ref-9] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey, "A Design Flow for Configurable Embedded Processors based on Optimized Instruction Set Extension Synthesis," in Proc. Design, Automation and Test in Europe, 2006.

[31] R. Ramaswamy and T. Wolf, "PacketBench: A tool for workload characterization of network processing," *in Proc. of IEEE International Workshop on Workload Characterization*, October 2003, pp. 42-50.

[32] SNU-RT Real Time Benchmarks.[Online]. Available: http://archi.snu.ac. kr/realtime/benchmark/.

[33] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of Int. workshop on workload characterization*, 2001, pp. 3-14.

[34] H. Lin and Y. Fei, "Resource Sharing of Pipelined Custom Hardware Extension for Energy-Efficient Application-Specific Instruction Set Processor Design," in ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 17, no. 4, 2012.

[35] Altera, Nios II Custom Instruction User Guide, Apr. 2010 [Online]. Available: http://www.altera.com/literature/ug/ug nios2 custom in-struction.pdf

[36] C. Favi, T. Kluter, Ch. Mester, and E. Charbon, "Optionally-Clocked Instruction Set Extensions for High Efficiency Embedded Processors," in IEEE Transactions on Circuits and Systems, vol. 59, no. 3, 2012.