NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NU-EECS-12-05
December, 2012

Resource-Constrained High-Level Datapath Optimization in ASIP Design

Yuankai Chen and Hai Zhou

Abstract

In this work, we study the problem of optimizing the datapath under resource constraint in the high-level synthesis of Application-Specific Instruction Processor (ASIP). We propose a two-level dynamic programming (DP) based heuristic algorithm. At the inner level of the proposed algorithm, the instructions are sorted in topological order, and then a DP algorithm is applied to optimize the topological order of the datapath. At the outer level, the space of the topological order of each instruction is explored to iteratively improve the solution. Compared with an optimal brutal-force algorithm, the proposed algorithm achieves near-optimal solution, with only 3% more performance overhead on average but significant reduction in runtime. Compared with a greedy algorithm which replaces the DP inner level with a greedy heuristic approach, the proposed algorithm achieves 48% reduction in performance overhead.

Keywords: ASIP, high-level synthesis, datapath, optimization, resource constraint

Resource-Constrained High-Level Datapath Optimization in ASIP Design

Yuankai Chen and Hai Zhou
Electrical Engineering and Computer Science, Northwestern University, U.S.A.

***Abstract*—In this work, we study the problem of optimizing the datapath under resource constraint in the high-level synthesis of Application-Specific Instruction Processor (ASIP). We propose a two-level dynamic programming (DP) based heuristic algorithm. At the inner level of the proposed algorithm, the instructions are sorted in topological order, and then a DP algorithm is applied to optimize the topological order of the datapath. At the outer level, the space of the topological order of each instruction is explored to iteratively improve the solution. Compared with an optimal brutal-force algorithm, the proposed algorithm achieves near-optimal solution, with only 3% more performance overhead on average but significant reduction in runtime. Compared with a greedy algorithm which replaces the DP inner level with a greedy heuristic approach, the proposed algorithm achieves 48% reduction in performance overhead.**

I. Introduction

In recent years, the ever-increasing cost of designing and manufacturing System-on-Chip (SOC) drives the industry to look for alternative ways that are able to achieve higher volume, if the cost cannot be reduced [1]. Application-Specific Instruction-Set Processors (ASIPs) have emerged to serve this need. These ASIPs are processors that have customized instruction set and microarchitecture, so they can fit a wider range of applications than traditional Application-Specific Integrated Circuit (ASIC), while delivering better performance than general purpose processors.

Presently, high-level synthesis tools such as Architecture-Driven Languages (ADLs) are widely used to specify and synthesize ASIPs [2][3][4][5]. The architecture can be fully tailored to the aimed application domain. Such tools usually require the user to input both structural and behavioral specifications. Structural specification defines the microarchitectural constraints of the processor which usually includes the pipeline structure, register file interface, memory interface and available functional units for synthesis. In the behavioral specification, the user constructs the datapath of the processor explicitly or implicitly by specifying how instructions are implemented under those microarchitectural constraints.

Nowadays the performance is not the only synthesis objective. Efficent techniques for exploring the design space are desired to balance multiple system metrics such as resource usage, area, and power consumption simultaneously [6][7][8]. Given a budget of functional units, there are a great number of ways to construct the datapath. Unlike general purpose processors, instruction set for ASIP usually includes custom instructions that are designed for complex computations. An example in [9] shows that a custom instruction requires three multipliers and three adders. A careless design of the datapath will directly affect the efficiency of the instructions. Therefore the trade-off between resource usage and performance should be optimized in the datapath synthesis.

It is worth mentioning that in some ADLs such as LISA [4] and EXPRESSION [5], the synthesizer instantiates a new functional unit wherever it is used in instruction specification. It eases the user from the datapath design, but it results in significant resource overhead. To address this problem, some work proposed resource sharing techniques to combine the same functional units from different instruction operations [6][10][11]. However in this design methodology, the designers have little control over the resource usage. If the final resource usage exceeds the budget, they have to go back to the very first stage to re-design instruction set.

In this work, we study the problem of optimizing the datapath of a pipelined ASIP for a given custom instruction set and a set of functional units. We propose a two-level dynamic programming based heuristic algorithm that achieves near-optimal solution with short runtime.

II. Preliminary and Motivation Example

A custom instruction set for ASIP consists of base instructions (BIs) and custom instructions (CIs). The BIs are usually simple instructions adopted from state-of-art ISA. In the context of this work, we assume that the BIs are RISC-like, and can always be scheduled with one μop. The CIs are additional complex instructions that are generated from the targeted applications to perform special computation. The CIs are represented by Dataflow Graphs (DFGs), where each vertex represents a basic functional operation and each edge represents the data dependency between the connected operations.

With resource constraint, CIs may need to be scheduled into a series of μops. We assume that the synthesized datapath will be divided into a pipeline with an appropriate clock frequency in subsequent design steps. For simplicity, we do not consider the impact of data hazard in the pipeline. Ideally in every clock cycle, there is one μop completing its last pipeline stage. Therefore the total number of μops to execute an application reflects the runtime of the application.

The following example shows that different datapaths result in different performances, under the same resource constraint and the same instruction set. Consider the custom instruction 1 (CI1) in Fig. 1(a), the operations in this instruction are 2 floating point (FP) multiplications and 2 FP additions. For resource constraint with only 2 FP multipliers and 1 FP adder, Fig. 1(c) and (d) show two possible datapaths, DP1 and DP2. To map CI1 to DP1, optimally 2 μops are needed. In the first μop, fadd_1 in CI1 is mapped to fadd in DP1, and the result of the fadd is then stored in a register. In the second μop, fadd_2, fmul_1, and fmul_2 in CI1 are mapped to fadd, fmul_1, and fmul_2 in DP1, respectively, to complete the instruction. Note also that mapping CI1 to DP2 needs 3 μops, mapping CI2 (Fig. 1(b)) to DP1 needs 2 μops and mapping CI2 to DP2 needs only 1 μop.

Considering both instructions simultaneously, suppose in an application, CI1 is executed 30 times and CI2 is executed 50 times. If DP1 is adopted, the total number of μops for executing CI1 and CI2 is: $2 \times 30 + 2 \times 50 = 160$. If DP2 is adopted, the total number of μops is: $3 \times 30 + 1 \times 50 = 140$. Thus, DP2 is a better datapath than DP1.

III. Problem Formulation

The input to our problem consists of a set of functional unit $OS = \{O_i | i = 1 \dots m\}$, each of q_i units, and a set of custom instructions $IS = \{I_j | j = 1 \dots n\}$, each of frequency w_j in the

applications. Each instruction I_j is modeled as a DAG (V_j, E_j), where each vertex $v_{jt} \in V_j$ represents an operation and the type of the operation is defined by a mapping function $mf_j(\cdot) : V_j \rightarrow OS$, and each edge $e(v_{jt}, v_{jk}) \in E_j$ represents data dependency between the two operations, i.e., the operation of v_{jk} cannot start until v_{jt} finishes.

Datapath is a DAG $D = (V_D, E_D)$ where each vertex $v_{Dt} \in V_D$ represents an instance of functional unit with type mapping function $mf_D(\cdot) : V_D \rightarrow OS$, and each edge $e(v_{Dt}, v_{Dk}) \in E_D$ represents a physical interconnection in the datapath between the two functional units. The datapath should meet the resource constraint: the number of vertices that are mapped to every functional unit O_i should not exceed q_i.

To schedule an instruction I_j in the datapath, each operation in the instruction must be mapped to a functional unit in the datapath, which is expressed by a mapping function $g_j(\cdot) : V_j \rightarrow V_D$. For each mapped pair $g_j(v_{jt}) = v_{Dt'}$, the types of these two operations must be equal, i.e., $mf_j(v_{jt}) = mf_D(v_{Dt'})$.

Consider an edge $e(v_{jt}, v_{jk})$ in the instruction graph E_j, if there exists an edge from $g_j(v_{jt})$ to $g_j(v_{jk})$ in the datapath, the computation result of v_{jt} can reach v_{jk} in the same μop. Otherwise, at least two μops are needed. We define performance overhead $PO(j, D, g_j)$ for each instruction I_j with the datapath D and mapping function $g_j(\cdot)$ as the number of additional μops that are needed to complete the instruction. This can be computed as follows:

1) Let $c(v_{jt}, v_{jk})$ be the cost of edge $e(v_{jt}, v_{jk})$ in the instruction graph I_j. $c(v_{jt}, v_{jk}) = 0$ if there exists an edge from $g_j(v_{jt})$ to $g_j(v_{jk})$ in the datapath, otherwise $c(v_{jt}, v_{jk}) = 1$.
2) The length of a path in the instruction graph I_j, in terms of the edge costs, is equivalent to the number of edges that cannot be scheduled in the same μop, thus the length of the longest path is equivalent to $PO(j, D, g_j)$.

Taking the instruction frequencies into account, the weighted sum of the performance overheads of all instructions reflects the performance of the datapath $PO(D)$:

$$PO(D) = \sum_{j=1}^{n} w_j \cdot PO(j, D, g_j) \quad (1)$$

The problem is to synthesize the datapath D with the least $PO(D)$, for the given OS, q_i's, IS and w_j's.

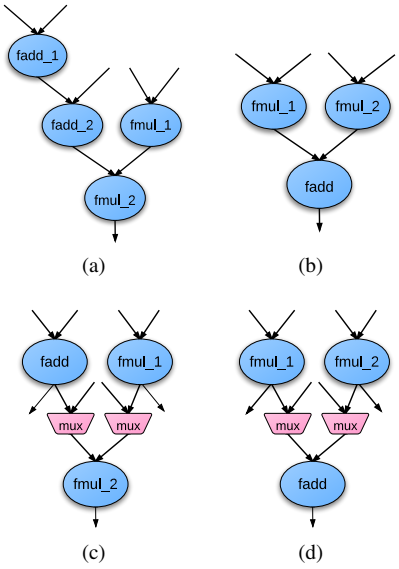

Fig. 1. (a) Custom instruction 1 (CI1). (b) Custom instruction 2 (CI2). (c) Datapath 1 (DP1). (d) Datapath 2 (DP2).

IV. Algorithm

We first study the sub-problem of instruction mapping where we assume the topological order of the datapath is fixed, and we present an algorithm that maps instructions optimally. Then we proceed to solve the datapath optimization problem. We show that a simplified problem where the instructions are chain graphs and all functional units are distinct can be solved optimally by a dynamic programming (DP) algorithm. The DP algorithm is then extended to a two-level heuristic algorithm to solve the general problem: at the inner level, the DP procedure is followed by a greedy algorithm that inserts additional functional units; at the outer level, the topological order space of each instruction is explored to improve the solution iteratively.

A. Instruction Mapping Problem

In this sub-problem, we assume that the topological order of the datapath is fixed. It means that the functional units are arranged in one-dimensional order, and no interconnection is allowed to be made from a functional unit to previous functional units in that topological order.

The complete datapath can be constructed from the topological order by adding necessary edges based upon the mapping functions of the instructions. For edge $e(v_{jt}, v_{jk})$ in instruction graph I_j, the corresponding functional units are $g_j(v_{jt})$ and $g_j(v_{jk})$. If $g_j(v_{jt})$ is placed before $g_j(v_{jk})$ in the topological order, an edge is allowed to be inserted from $g_j(v_{jt})$ to $g_j(v_{jk})$, otherwise those two operations need to be scheduled in two different μops.

Since the topological order of the datapath is fixed, the mappings of the instructions do not interfere with each other. To minimize the performance overhead of the instruction set is equivalent to minimize the performance overhead of each instruction individually.

In a *chain graph*, the nodes are arranged in a chain structure, and edges only exist between adjacent nodes. We first show an algorithm that optimally solves the instruction mapping problem for chain-graph instructions. Then we extend the algorithm to general instructions.

The algorithm for chain-graph instructions is presented in Algorithm 1. It maps one operation at a time in the chain order. The algorithm walks through P from the starting position pos, finds the first matched functional unit, and chooses it to map to the current operation. The next position of the chosen functional unit will be the starting point for the mapping of the next operation. If no matched functional unit is found until the end of P, it starts over from the beginning of P and increases performance overhead by one. The correctness of this algorithm is straightforward. Choosing the first matched functional unit leaves more resources for the subsequent operations. It increases the chance of mapping the subsequent operations in the same μop. The solution should be at least no worse than that if the first matched functional unit is not chosen.

The full *instruction mapping* procesure extends Algorithm 1 to general instruction. We examine all possible topological orders of the instruction, treat each topological order as a chain graph, and then apply Algorithm 1 to the chain graph. The minimum performance overhead among all topological orders is the optimal performance overhead for the instruction.

***Theorem** 1:* The above procedure returns the optimal performance overhead for the given instruction I and the topological order of datapath P.

Proof: First, if we apply Algorithm 1 to a topological order of I, the obtained mapping function is also a valid mapping function for I, because no operation can be scheduled before its predecessors. With the same mapping function, the performance overhead is also the same.

Suppose $\bar{g}(\cdot)$ is the optimal mapping function for I. We sort the operations in I by the following two rules:

- Operations are sorted by the μops that they belong to, earliest μop first.
- Operations in the same μop are sorted by the order of the functional units that they are mapped to.

We denote T as the resulted order. It is easy to prove that T is a topological order of instruction I: if otherwise, there exists an edge from an operation to an operation before it. However, the operations are sorted in the order that they are scheduled. It contradicts with the presumption that $\bar{g}(\cdot)$ is a valid mapping function.

Since $\bar{g}(\cdot)$ is the optimal mapping function, if we apply Algorithm 1 to T, we will get the same performance overhead. And it is impossible to get better performance overhead by applying Algorithm 1 to other topological orders of I. Therefore one can find the optimal mapping by applying Algorithm 1 to every topological orders of I. ■

Algorithm 1 Mapping algorithm for chain-graph instruction

Input: Chain-graph instruction I and the topological order of datapath P
Output: Minimum performance overhead

```
PO = 0
pos = 0
for each operation o in I in the chain order do
    repeat
        starting from position pos, walking down along P to find the first FU of the same type with o
        if no FU can be found until the end of P then
            start over from the beginning of P, pos = 0
            PO = PO + 1
        end if
    until FU is found
    map FU to o
    pos = position of the mapped FU + 1
end for
return PO
```

B. Optimizing Topological Order of Datapath

1) A Simplified Problem and DP Algorithm

We first study a simplified problem where all functional units are distinct, that is, $q_i = 1, i = 1 \dots m$, and the input instructions are all chain graphs.

We partition the functional unit set V_D into two subsets LV_D and RV_D, where the functional units in LV_D are placed before the functional units in RV_D in the topological order of datapath. Because no two functional units are of the same type, for every instruction operation there is only one functional unit that it can be mapped to. We say the operation is mapped to LV_D if the matched functional unit is in LV_D, otherwise the operation is mapped to RV_D.

Consider a pair of consecutive operations o_a and o_b in some instruction. If o_a is mapped to RV_D and o_b is mapped to LV_D, these two operations cannot be scheduled in the same μop, thus a performance overhead occurs. We call such a performance overhead is *caused* by the partition. We define the cost of a partition, $c(LV_D, RV_D)$, to be the total performance overhead of all instructions (weighted by instruction frequencies) that is caused by the partition.

With the partition of functional units, we can decompose each instruction into alternating segments where operations are either all mapped to LV_D or all mapped to RV_D. LI denotes the set of segments, from all instructions, where all operations are mapped to LV_D, and RI denotes the set of the rest of the segments. These segments can be viewed as smaller chain-graph instructions to be implemented by the functional units in LV_D and RV_D respectively.

We denote $f(V_D, IS)$ as the performance overhead of the optimal topological order of datapath for the given functional units V_D and instruction set IS. The following theorem gives the dynamic programming recursive equation for finding $f(V_D, IS)$.

***Theorem** 2:* Let PA be the set of all possible partitions for the given set of functional units V_D, the minimal performance overhead $f(V_D, IS)$ can be computed by:

$$f(V_D, IS) = \min_{PA}[f(LV_D, LI) + f(RV_D, RI) + c(LV_D, RV_D)] \tag{2}$$

Proof: After partitioning, edges in instructions can be classified into three categories:

- The edges that are from an operation mapped to LV_D to an operation mapped to RV_D. For this group of edges, the two operations can be scheduled in the same μop, therefore no performance overhead does not occur.
- The edges that are from an operation mapped to RV_D to an operation mapped to LV_D. The performance overhead occurs and is included in $c(LV_D, RV_D)$.
- The edges where both ends are mapped to LV_D, or both ends are mapped to RV_D. Whether the performance overhead occurs depends on the optimal topological order of functional units in LV_D or RV_D. It is computed in the sub-problem of finding $f(LV_D, LI)$ or $f(RV_D, RI)$.

Because LV_D and RV_D do not have intersection, $f(LV_D, LI)$ and $f(RV_D, RI)$ can be computed independently of each other. Therefore for a given partition, the minimal performance overhead is the sum of performance overheads from the aforementioned three edge categories. $f(V_D, IS)$ is the minimum sum among all possible partitions. ■

The number of possible partitions is exponential to the number of functional units. However, in the ASIP synthesis problem, the number of functional units is usually small. We can use one bit to mark each functional unit, therefore a single integer to mark each subset of functional units. With integral marking, the DP equation Eq. 2 can be efficiently computed as shown in Algorithm 2. $f(i)$ is the minimum performance overhead for the subset of functional units represented by integer i with corresponding instruction segments. The inner loop goes through each functional unit j in subset i, partitioning set i into two subsets: LV_D with the functional units in set i except j, and RV_D with j only. For each partitioning, the performance overhead is computed. $f(i)$ is the minimum performance overhead from all possible partitionings. The optimal partitioning is recorded in an array $prev(i)$. The partition cost of j in set i, $c(i, j)$, can be easily computed by examining the edges from j in all instructions. If the ending functional unit is in LV_D, $c(i, j)$ is increased by a performance overhead weighted by the instruction frequency.

Computing each $c(i, j)$ requires $O(n \cdot ML)$ runtime where ML is the maximal length of instructions. The complexity of the whole DP algorithm is $O(MAX \cdot m \cdot n \cdot ML)$, given by the complexity of computing $c(i, j)$ multiplied by the complexity of the two loops.

2) Extension to Two-Level Heuristic Algorithm

To extend Algorithm 2 to the original problem, one needs to consider: the quantity of each functional unit is not limited to 1 and the input instructions are general DAGs.

We extend the DP algorithm to a two-level heuristic algorithm. At the inner level, all instructions are sorted in topological order, and

Algorithm 2 DP Algorithm

Input: OS, chain-graph IS and w_j's

Output: The topological order of datapath with minimum performance overhead

1: $MAX = 2^m - 1$
2: $mark(i) = 2^i, i = 0..m-1$
3: initialize $f(i) = \infty, i = 1..MAX$ and $f(0) = 0$
4: **for** $i = 1$ to MAX **do**
5: **for** $j = 0$ to $m-1$ **do**
6: **if** $mark(j) \& i \neq 0$ **then**
7: **if** $c(i,j) + f(i - mark(j)) < f(i)$ **then**
8: $f(i) = c(i,j) + f(i - mark(j))$ and set $prev(i) = j$
9: **end if**
10: **end if**
11: **end for**
12: **end for**
13: **return** $f(MAX)$ and traceback in $prev$ to obtain the optimal topological order of the datapath.

Algorithm 3 Two-level Heuristic Algorithm

Input: OS, q_i's, IS and w_j's

Output: The optimal topological order of the datapath

1: sort instructions in topological order
2: **repeat**
3: call Algorithm 2
4: greedy insertion
5: topology order switch
6: **until** no improvement is achieved in recent N iterations
7: **return** The optimal topological order of the datapath

treated as chain graphs. We first assume that the quantities of all functional units are 1, and apply Algorithm 2 to find the minimum performance overhead. For functional units whose quantities are greater than one, we insert one additional instance at a time in a greedy manner: For each position in the topological order of datapath, we insert the instance and compute the new performance overhead by re-performing the instruction mapping. The instance is finally inserted to the position where the new performance overhead is minimal.

The outer level switches the topological order of each instruction. Using the current topological order of datapath, for each instruction, we can evaluate the performance overhead for each topological order. The topological order is then switched to the one with the least performance overhead. If multiple topological orders have the same least performance overhead, a random one is chosen from them. The randomness prevents the algorithm from getting stuck in a local minimum. The outer level explores the spaces of topological order of instructions. The updated topological orders are used in the next iteration. It guarantees that the total performance overhead is non-increasing through iterations.

We keep iterating through these two levels until no improvement is achieved in recent N iterations. The two-level algorithm is summarized in Algorithm 3.

V. Experimental Results

We selected a set of six benchmarks from a wide range of embedded applications featured with heavy computation to evaluate the proposed algorithm. The selected benchmarks are: FFT, LMS and SHA from SNU-RT [12], JPEG and GSM from mibench [13], and MESA from Express benchmarks [14]. For each benchmark, we first used LLVM [15] to obtain the DFG for every basic block and its execution times by profiling. We implemented algorithm in [16] to identify custom instructions from the DFGs of each benchmark. Note that the complexity of the input to our algorithm is not directly correlated to the complexity of the input benchmark, but correlated to the complexity of the identified custom instructions. We set 10 as the maximum number of primary inputs, and 4 as the maximum number of primary outputs. The execution times of each instruction is the occurrence of the instruction in the DFG times the execution times of the DFG. The frequency of an instruction is its execution times normalized against the whole instruction set.

TABLE I
Families of Operations

Families	Operations
Integer operations	add, sub, mul, div, shift left and shift right
FP operations	fadd, fsub, fmul and fdiv
Logic operations	and, or and xor
Comparisons	Int compare and FP compare
Type conversions	FP to Int and Int to FP

In our experiment, the resource constraint for each benchmark is determined as follows. We first classify operations into six families, as in Table I. An instruction is tagged by a family if it has an operation in that family. An instruction can be tagged by multiple families. For a family F, we denote $S(F)$ as the set of instructions that are tagged by F. And we denote $num(I_j, O_i)$ as the times of operation O_i appearing in instruction I_j. The number of instances of O_i is given by the weighted average appearance in $S(F)$ if $O_i \in F$, ceiled to the nearest integer:

$$q_i = ceil(\frac{\sum_{I_j \in S(F)} num(I_j, O_i) \times w_j}{\sum_{I_j \in S(F)} w_j}) \quad (3)$$

Because there is no existing work that formulates similar problem to ours, to demonstrate the efficiency of our proposed algorithm, we implemented an optimal algorithm and a greedy algorithm for comparison. The optimal algorithm exhaustively enumerates all possible orders of functional units. The complexity of this algorithm grows in factorial with respect to the total number of functional units. A pruning technique is applied to reduce the runtime: after each insertion, we evaluate the performance overhead of the current sub-sequence by running instruction mapping. If the performance overhead is greater than the currently achieved minimum, this branch is eliminated. The greedy algorithm replaces the inner level with a simple heuristic approach. It removes the DP algorithm, and greedily inserts *all* functional units one at a time.

The algorithms are implemented in JAVA, and run on an Linux machine with 2.8GHz Intel Pentium Dual-core and 2GB RAM. The statistics of the benchmarks is listed in Table II. We set the termination condition to be 10 iterations with no improvement, for both the proposed algorithm and the greedy algorithm.

The performance of the algorithms is shown in Table III. The second column of each benchmark lists the types of functional unit that have more than one instance for that benchmark, followed by the number of additional instances. For each algorithm, the performance overhead (PO) of the generated datapath and corresponding runtime are listed. It can be observed from the table that the proposed algorithm achieves near-optimal solution, with only 3% increase in performance overhead on average, compared with the optimal algorithm. For three out of six benchmarks, the proposed algorithm generates optimal datapath. The runtime of the proposed algorithm is much less than the optimal algorithm. Compared with the greedy algorithm, which is able to generate a solution within short runtime, the proposed algorithm achieves 48% less in performance overhead.

TABLE II
STATISTICS OF BENCHMARKS

Benchmarks	FFT	LMS	SHA	JPEG	GSM	MESA
Number of CIs	11	17	15	46	118	109
Max Number of operations	11	9	9	7	9	9
Number of types of FUs used	8	9	7	10	9	16

VI. CONCLUSION

In this paper, we proposed a two-level dynamic programming (DP) based heuristic algorithm that efficiently solves the problem of optimizing the performance of pipelined datapath under resource constraint in ASIP synthesis. Compared with optimal brutal-force algorithm, the proposed algorithm achieves near-optimal solution, with only 3% more performance overhead on average but significant reduction in runtime. Compared with a greedy algorithm which replaces the DP inner level with a greedy heuristic approach, the proposed algorithm achieves 48% less in performance overhead.

VII. ACKNOWLEDGEMENT

This work is supported by the NSF under CCF-0811270 and CCF-1115550.

REFERENCES

[1] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The next design discontinuity," *IEEE Proc. ICCD*, pp. 84–90, 2002.
[2] M. Imai, "ASIP Meister: A configurable processor core development system," *Proc. Intl. Conf. on Information and Communication*, 2005.
[3] S. Basu and R. Moona, "High level synthesis from Sim-nML processor models," *IEEE Proc. VLSI Design*, pp. 255–261, 2003.
[4] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.
[5] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven exploration of pipelined embedded processors," *IEEE Proc. VLSI Design*, pp. 921–926, 2004.
[6] M. Zuluaga and N. Topham, "Design-space exploration of resource-sharing solutions for custom instruction set extensions," *IEEE Trans. on CAD*, vol. 28, no. 12, pp. 1788–1801, 2009.
[7] Y. Lu and H. Zhou, "Efficient design space exploration for component-based system design," *IEEE Proc. ICCAD*, 2012.
[8] V. Zaccaria, G. Palermo, F. Castro, C. Silvano, and G. Mariani, "Multi-cube explorer: An open source framework for design space exploration of chip multi-processors," *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pp. 1 –7, feb. 2010.
[9] Q. Dinh, D. Chen, and M. D. Wong, "Efficient ASIP design for configurable processors with fine-grained resource sharing," *ACM/SIGDA Proc. Symposium on FPGA*, pp. 99–106, 2008.
[10] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," *IEEE Proc. DAC*, pp. 395–400, 2004.
[11] E. Witte, A. Chattopadhyay, O. Schliebusch, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr, "Applying resource sharing algorithms to ADL-driven automatic ASIP implementation," *IEEE Proc. ICCD*, pp. 193–199, 2005.
[12] SNU-RT, www.cprover.org/goto-cc/examples/snu.html.
[13] Mibench, www.eecs.umich.edu/mibench/.
[14] Express benchmarks, express.ece.ucsb.edu/benchmark/.
[15] LLVM, www.llvm.org.
[16] P. Biswas, S. Banerjee, and N. Dutt, "ISEGEN: Generation of high-quality instruction set extensions by iterative improvement," *IEEE Proc. DATE*, pp. 1246–1251, 2005.

TABLE III
EXPERIMENTAL RESULTS OF COMPARING TO GREEDY ALGORITHM AND OPTIMAL ALGORITHM

Benchmarks	Additional FUs. (Qty)	Greedy Algorithm		Optimal Algorithm		Proposed Algorithm			
		PO_g	runtime (sec.)	PO_{opt}	runtime (sec.)	PO_{prop}	runtime (sec.)	PO_{prop}/PO_{opt}	PO_{prop}/PO_g
FFT	fmul(1)	0.795	0.45	0.233	6.26	0.233	0.07	1.00	0.29
LMS	fmul(1)	0.835	0.11	0.298	6.57	0.298	0.09	1.00	0.36
SHA	add(2), xor(1)	1.000	0.40	0.621	12.70	0.621	0.03	1.00	0.62
JPEG	add(1)	0.451	0.40	0.338	2.82	0.384	0.37	1.14	0.85
GSM	add(1), mul(1)	0.835	0.89	0.503	1868	0.515	0.66	1.02	0.57
MESA	or(2), shr(1), shl(3), fmul(1)	0.92	1.43	0.352	53 hours	0.361	9.87	1.03	0.39
Average								1.03	0.52