Fast Generation of Multiple Custom Instructions under Area Constraints

Di Wu, Imyong Lee, Junwhan Ahn, and Kiyoung Choi

Abstract—**Extensible processors provide an efficient mechanism to boost the performance of the whole system without losing much flexibility. However, due to the intense demand of low cost and power consumption, customizing an embedded system has been more difficult than ever. In this paper, we present a framework for custom instruction generation considering both area constraints and resource sharing. We also present how we can speed up the process through pruning and library-based design space exploration.**

Index Terms—**Extensible processor, custom instruction, area constraint, resource sharing**

I. Introduction

In recent years, the demand on high performance, low power systems in the market of consumer electronics has been growing exponentially. In those systems, extensible or configurable processors can help overcome the challenges of cost, performance, and time-to-market pressure. Typically, an extensible processor is optimized to a specific application to obtain high performance and low power consumption that cannot be achieved by a general purpose processor (GPP), while maintaining a certain level of flexibility that is not possible with an application-specific integrated circuit (ASIC). There already exist several commercial products such as Altera Nios II [1], Xilinx MicroBlaze [2], Tensilica Xtensa [3] and ARC 700 [4].

Instead of designing the processor entirely from scratch, the designer of an extensible processor are given a base processor and allowed to add their own custom instructions (CIs) and/or other peripherals to meet the application and design constraints, which makes the design process much easier. However, due to the lack of a fully automated design tool chain that finds best CIs [5], configuring an extensible processor still depends much on the experience of the designers.

There have been many researches on automatic CI generation [5-13], and many different methodologies have been developed to solve partial problems. However, none of them studied generating CIs considering both area constraints and resource sharing. In this paper, we propose an efficient framework to explore the design space to generate an optimal set of CIs considering these two factors. This is an extension of our previous work [15], where only one CI is selected among the candidate CIs possibly overlapping with each other.

II. Previous Work

Typically, CI generation starts from execution time analysis. One simple method would be profiling the target application with an instruction set simulator (ISS) of the base processor. Time consuming basic blocks, known as kernels, will be identified from the application code. Then the code of each kernel will be converted to a directed acyclic graph (DAG) representation. This work can be easily done with the SUIF compiler [14]. Within each DAG, candidate CIs are identified and then most efficient CIs are selected.

There are two methodologies for CI generation. One

Manuscript received Nov. 30, 2010; revised Dec. 23, 2010.
This paper was invited from ISOCC 2010
School of EECS, Seoul National University, Seoul, 151-600, Korea
E-mail : kchoi@dal.snu.ac.kr

focuses on reusability of each CI [8], whereas the other focuses on performance gain of a single CI [7]. There is no guarantee that one methodology can beat the other one. And as state in [5], it is important to obtain a large CI in realistic cases. For these reasons, our work also takes the second approach.

An effective single CI generation algorithm based on the second approach is proposed in [7]. The basic idea of their algorithm is enumerating all possible solutions in the DAG with a search tree. After a topological sort, operations are arranged into corresponding levels. Fig. 1 shows a binary search tree of four levels, which is used for generating one CI from the given DAG. A notion of cut is introduced to represent a subset of nodes in a DAG. At each node, the edge (labeled 1) to the right child indicates "adding" the operation corresponding to that level to the cut, whereas the edge (labeled 0) to the left child indicates "ignoring" that operation. So, by visiting all the nodes in the tree, all possible solutions could be tested and an exact solution could be found. The cuts that satisfy all the constraints such as number of input ports, number of output ports, and convexity constraint are called candidate CIs. Notice that, the total number of cuts is O(2^n), where n is number of nodes in the DAG. A branch-and-bound technique is implemented to reduce the runtime of the exponential complexity algorithm. While traversing the search tree, whenever a constraint violation occurred, all the descendent nodes are pruned, because adding more nodes will not recover the violation. It is stated that the runtime is quite reasonable.

To extend the problem to multiple-CI selection, one can build up a tree with multiple branches (one branch for each CI). However, due to the exponential nature, such a method is unreasonably slow in practice. So in [5], a heuristic multiple-CI identification approach called *iterative selection* is proposed. This approach is based on a relatively fast single-CI identification algorithm. They use the single-CI identification algorithm iteratively on the given DAG and select the best CI for each iteration. Nodes of the CI selected in an iteration are set as forbidden nodes in the following iterations. The experiment shows that the performance difference between this heuristic and the optimal solution is reasonably small, and runtime is significantly reduced.

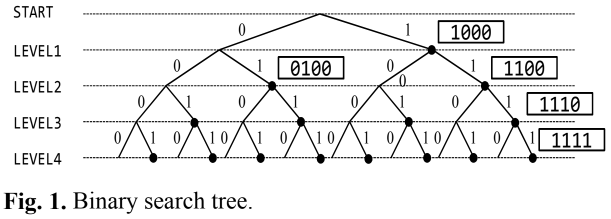

Fig. 1. Binary search tree.

Lee et al [9] extended the above work in three ways. First, CIs are implemented as sequential logic including memory load/store units. As a result, memory operations could be included into CIs, which had been forbidden in other approaches. Secondly, register operations are serialized. Thus I/O ports of CIs are not limited by the architecture. The direct impact of these two improvements is that CIs become larger. Thirdly, area is saved by resource sharing among CIs through the high-level synthesis (HLS) process. However, this algorithm still suffers from the complexity problem of [7]. And including memory operations and serializing register operations decreases the chance of pruning (a branch that includes a memory operation should not be pruned), which makes the algorithm even slower. What's more, if the designers need to set the area constraint, there is no better way to obtain optimal resource allocation (how many multipliers, how many adders, etc.), but to check all the combinations.

III. Motivations and Contributions

Die area is of great value in processor design, especially for embedded processor design where cost and (static/dynamic) power consumption are important. So every small piece of die area should be used carefully. Under an area constraint, there can be various combinations of resources that satisfy the constraint. Depending on the application, different combinations of resources can reveal very different performance numbers. For instance, let's consider two different combinations having similar total area: one containing three adders and one shifter, and the other containing one adder and three shifters. Apparently, for applications with more concurrent addition operations, the first combination will be preferable. In contrast, if the application has more concurrent shift operations, then the second combination would be better. Thus proper allocation of resources should be paid with great attention.

Given a kernel, we try to find a number of CIs and optimal resource allocation under an area constraint, considering resource sharing. Resource sharing is achieved through HLS. This problem can be formulated as follows:

Problem: Find cuts *C1*, *C2*, *C3*, ..., CN_{CI} for CIs and a resource allocation $(X_1, X_2, \ldots, X_n)$ in the set of feasible resource allocations (FRAs) given by

$$S_{FRA} = \{(X_1, X_2, \ldots, X_n) \mid A_1{*}X_1 + A_2{*}X_2 + \ldots + A_n{*}X_n < AREA\}$$

such that the performance gain

$$PG = T_{SW}(C1) - T_{HW}(C1) + T_{SW}(C2) - T_{HW}(C2) + \ldots + T_{SW}(CN_{CI}) - T_{HW}(CN_{CI})$$

is maximized, where A_i is area of a functional unit (FU) of type *i*, X_i is number of function units of type *i*, $T_{SW}(Cj)$ is the number of cycles required to execute *Cj* with base instructions, and $T_{HW}(Cj)$ is the number of cycles required to execute *Cj* with the generated CI for the cut.

The algorithm in [9] could be applied directly to explore the design space. However, there are two problems that cause this primitive method unfavorable. First, depending on the number of resource types and the total area constraint, the number of FRAs can be huge. Secondly, for a given resource allocation, the algorithm for finding the best CI is exponential, even though it can be made efficient by implementing it with the branch-and-bound technique. Since the algorithm should be executed repeatedly for every FRA, it can be too much time consuming, especially when the kernel is large in size.

In this paper, we consider two techniques to reduce the overall runtime of design space exploration. One is to prune the resource allocation space, and the other one is to decrease the runtime for finding a CI for each resource allocation.

IV. CI GENERATION FRAMEWORK

1. Pruning of Resource Allocation Space

Fig. 2 shows an example of resource allocation space with two types of resources: FU1 (e.g., multiplier) and FU2 (e.g., adder). Dashed line represents the area constraint. Each dot under the dash line represents an FRA. We assume only data-dominated circuits where the areas of registers, multiplexers, control units, and interconnects are ignorable. Although there are many FRAs in our example, not all of them need to be explored.

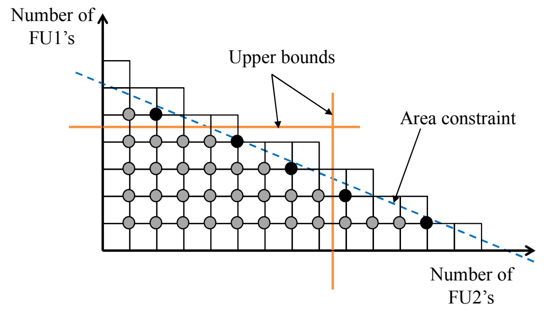

Fig. 2. An example of resource allocation space.

FRAs in the design space can be divided into two categories, namely tight FRAs (shown in black dots in Fig. 2) and loose FRAs (shown in gray dots in Fig. 2). Tight FRAs are those, to which adding any more resource will violate the area constraint. The rest of the dots are loose FRAs. Notice that, for any loose FRA, we can always find a tight FRA that has at least one more resource of a certain type. It is clear that allocating more resources will not decrease the performance, since if it does not help, then we can just ignore the extra resources during the resource binding phase. Thus loose FRAs can never give a better solution and therefore all of them are pruned from the resource allocation space.

Moreover, we can also prune some of tight FRAs when there is not enough concurrency in the kernel to exploit all the allocated resources. For each type of FU, we can set an upper bound of the number of instances used for implementing a CI as shown in Fig 2. We can obtain such upper bounds by performing HLS for every candidate CI with unlimited resources. As soon as possible scheduling (ASAP) is used for this purpose. For each type of FU, by recording the actual number of instances used in each candidate CI, we can obtain an upper bound for the CI. And by taking the maximum of the upper bounds over all candidate CIs, the global upper bound can be obtained.

2. Library Based Design Space Exploration

The approaches in [6] and [9] perform CI identification together with CI selection. However, it is observed that, separating these two processes can create another opportunity for decreasing the runtime for design space exploration.

Note that, CI identification simply enumerates all the possible cuts and check the constraints on number of input ports, number of output ports, and convexity. It has nothing to do with the performance gain that could be obtained by the CI implementation. Thus, for a given kernel, despite different FRAs, the result of CI identification does not change at all. However, CI selection is performed based on the performance gain of each CI and the performance gain is determined by the HLS process with given FRAs. Thus HLS and CI selection are independent of the CI identification. This feature provides an opportunity for improving the efficiency of design space exploration. More specifically, by reusing the identified CIs again and again during the CI selection, we can reduce the runtime significantly.

Fig. 3 shows the benefit of the proposed framework, where we assume two different FRAs and up to two CIs. In the traditional design space exploration approach (simply running the Lee's algorithm iteratively for each resource constraint), there are two iterations for each FRA. Within each iteration, CI identification is performed first to find all candidate CIs, HLS is performed over the candidates, and the CI with largest performance gain is selected. Notice that the runtime for the identification process of the second iteration is reduced compared to that of the first iteration, because the nodes included in the first CI are set as forbidden. The entire process is repeated for the second FRA, and the better solution is selected among the two best solutions. However, in our proposed method, we perform the CI identification process only once, and reuse the stored candidate CIs for HLS and selection process.

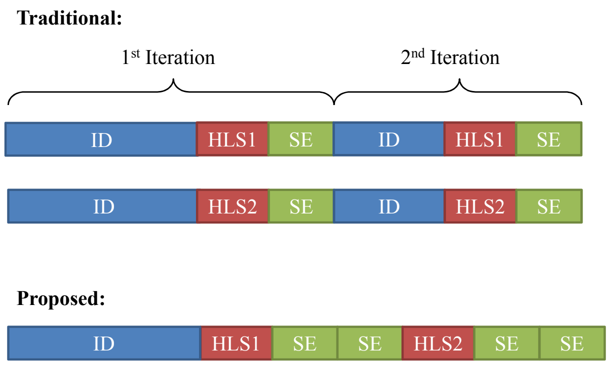

Fig. 3. Benefit of the proposed approach when considering multiple resource constraints. ID represents identification process, HLS represents high-level synthesis process, and SE represents selection process.

3. Multiple-CI Selection

In our framework, we adopt the idea of *iterative selection* from [5]. But we implement it in a different way. The basic principle of multiple-CI selection is that, any two cuts should not have overlapping nodes, in other words, two CIs should be disjoint. In [5], they set nodes from previously selected cuts as forbidden nodes. So in the next generation, forbidden nodes are not included in the search tree.

In our implementation, the path to each node from the root is encoded as a label of the node. Each '0' bit in the label indicates not including the corresponding operation, while '1' bit indicates including the operation. So if two candidate cuts overlap, both labels must have a '1' bit at the corresponding position. Based on this observation, we use the labels for the candidate cuts to select disjoint cuts. The pseudo code is shown in Fig. 4. In each iteration, the label of each candidate CI is compared with the mask. Among the candidate CIs that do not overlap with the mask, we find the CI with largest PG and update the solution and mask.

```
mask=0;
for i = 1 to $N_{CI}$
        for j = 1 to |CILibrary|
                CI = CILibrary[j]
                if CI.label & mask == 0
                        CI.PG = $T_{sw}(CI) - T_{hw}(CI)$;
                        if CI.PG > bestCI.PG
                                bestCI = CI;
        mask = mask | bestCI.label;
        $CI_i$ = bestCI;
```

Fig. 4. Pseudo code for implementing iterative selection using labels.

4. Proposed Framework

Fig. 5 shows the design flow in the proposed framework. CI identifier identifies candidate CIs following the approach in [9]. It receives DAG representation of the given kernel and checks the feasibility of considered cuts based on relaxed I/O constraint and the convexity constraint. All candidates are stored in the candidate library. HLS with unlimited

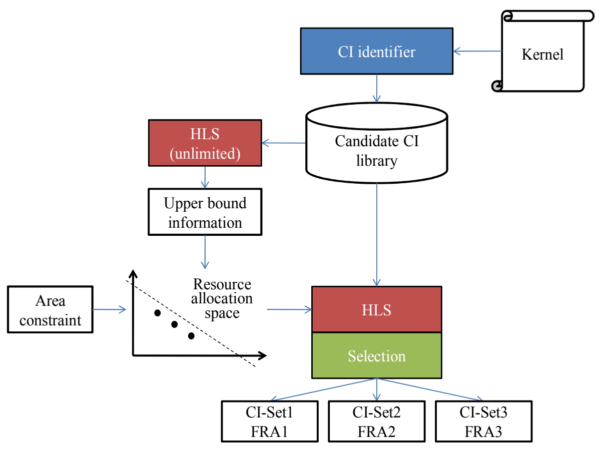

Fig. 5. Design flow in the proposed framework.

resources are done for each candidate CI. And upper bound information is obtained by recording the maximum number of FUs actually used. With the upper bounds, the resource allocation space is pruned. Then HLS process takes one FRA at a time to obtain scheduling and binding result, calculate the performance gain of all candidate CIs, and update the library. The best CI is selected and saved with the corresponding FRA. This process is repeated until we find all the N_{CI} CIs or no more CI is found. The entire process is repeated for each FRA in the resource allocation space. In the end, the FRA with the larget performance gain will be selected for the final design.

V. Experimental Results

We have selected basic blocks from the Mibench [16] and DSPstone [17] benchmarks to show the merit of our proposed multiple-CI generation framework. The kernels have been converted to DAG representations with the SUIF compiler [14]. We have used an extensible processor, whose instruction set architecture is compatible with ARM7. We assume that two register reads and one register write are supported in each cycle. Memory operations are also supported. Relaxed I/O constraint is set to 4/2 (at most four inputs and two outputs). We use the fastest implementation for each functional unit and obtain delay and area through synthesis using Synopsis Design Compiler.

Profiles of each kernel are shown in Table I. Size of each kernel is shown in the second row of the table. The number of visited nodes is the number of nodes in the binary search tree visited during the CI identification. As the number increases, the runtime of the identification process increases. The number is affected by the size of the kernel as well as the topology of the DAG. Usually, as the size of the kernel increases, the number of visited nodes grows exponentially. That is why our proposed framework is more effective for larger kernels. The number of candidate CIs indicates the speed of CI selection. The ratio of number of visited nodes to number of candidate CIs indicates the speedup achieved by our approach.

Fig. 6 shows the set of Pareto points obtained by the proposed approach for the CONVOLUTION example with different area constraints and up to four CIs. Horizontal axis represents the actual area cost for implementing the generated CIs, and vertical axis represents the number of cycles required to execute one iteration of CONVOLUTION. The point with zero area of CI represents no instruction extension, which takes 73 cycles. By using four CIs and using more area (more resources) for the CIs, we can improve the performance and reduce the execution time down to 22 cycles.

To see the efficiency of our approach, we have measured the runtime of the proposed method as well as the traditional method. In this experiment, the pruning of the resource allocation space has been performed for both methods. The area constraints have been set such that ten different FRAs remain and then only one best CI

Table 1. Profiles of example kernels

	REAL_UPDATES	DOT_PRODUCT	FIR	LMS	SHA	SUSAN	CONVOLUTION
Number of nodes	64	37	71	62	60	75	64
Number of visited nodes (A)	13,671,460	124,255	4,738,810	44,369,969	6,244,196	4,634,600	45,416,708
Number of candidate CIs (B)	5,514	20,523	10,998	22,043	8,210	49,124	24,520
Ratio (A/B)	2,479.4	6.1	430.1	2,012.9	760.6	94.3	1,852.2

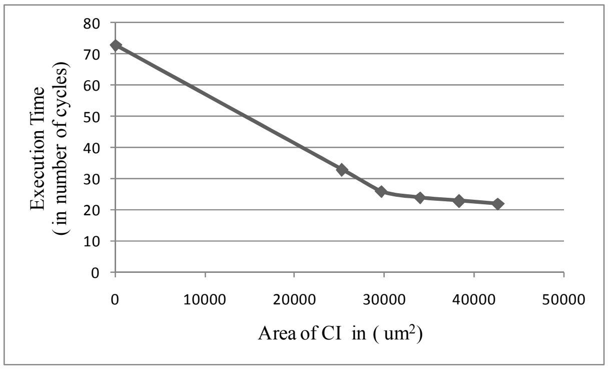

Fig. 6. Execution time of the CONVOLUTION example for CIs generated by the proposed approach with different area constraints.

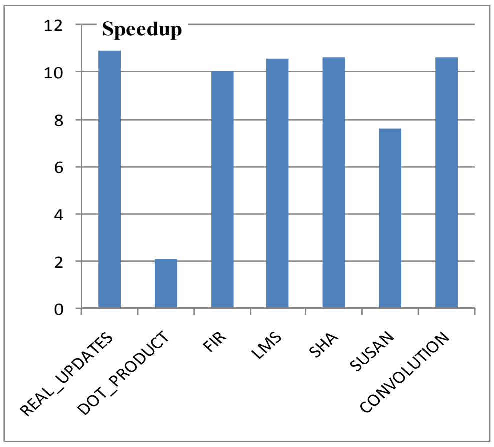

Fig. 7. Speedup achieved by the proposed method.

has been generated from each example kernel. Thus in the traditional method, Lee's algorithm is executed 11 times (once for the resource allocation space pruning, and 10 times for CI selection). For each example used in this experiment, we have generated three CIs.

Fig. 7 shows speedups (vertical axis) achieved by the proposed method compared to the previous approach [9]. For the DOT_PRODUCT example, due to the small size, our proposed method shows only 2.1 times of speedup. However, for other examples, the speedup is huge. This is because the runtime for CI identification is huge. Because of the relatively complex topology, SUSAN has the largest number of candidate CIs and relatively small ratio of A to B as shown in Table I. Thus the speedup is smaller than other examples, such as REAL_UPDATES, FIR, LMS, SHA and CONVOLUTION.

In our last experiment, to show the benefit of multiple-CI selection, we have set the area constraint to 30,000 μm^2 and generated one to four CIs for each example. Fig. 8 shows the experimental results. Notice that, for most of the examples, it is worth generating multiple CIs, since the performance increases as we generate more CIs. Especially in the SUSAN example, the second CI brings significant performance gain. However, we could not generate four CIs for some examples. We could generate only one CI for the DOT_PRODUCT example, and up to three CIs for the SUSAN and CONVOLUTION examples.

VI. Conclusions

In this paper, we have proposed an efficient framework for multiple CI generation, considering area constraint and resource sharing. To the best of our knowledge, this is the first attempt to find an optimal set of CIs under area constraint. To speed up the design

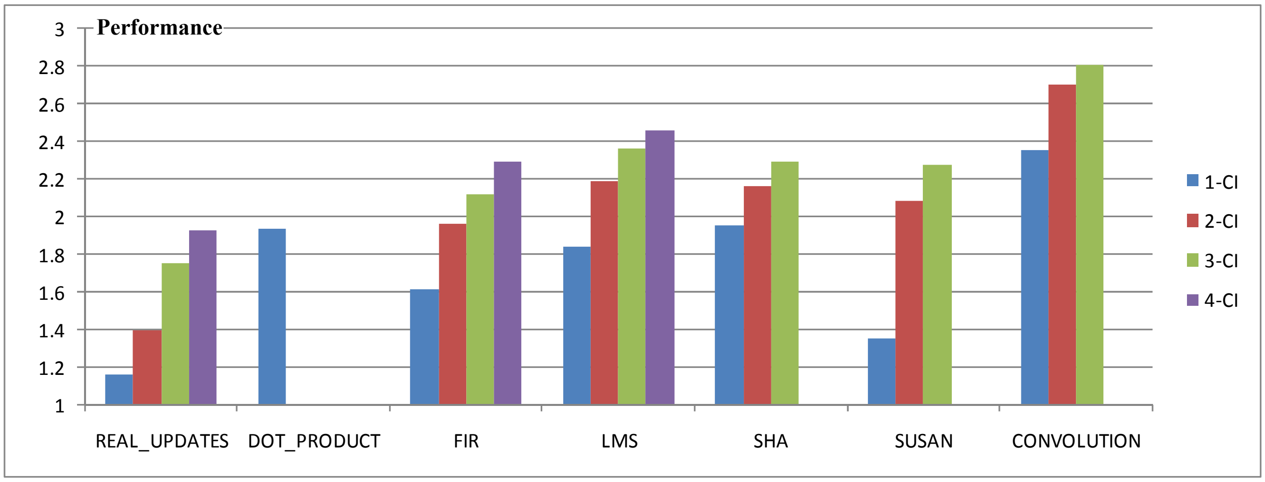

Fig. 8. Performance gain of multiple-CI selection.

space exploration, we have proposed techniques of pruning the resource allocation space and reusing the candidate CI information.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0019522) and by the Technology Innovative Program (Industrial Strategic technology development program, KI002168, Development of Configurable Device & SW Environment) funded by the Ministry of Knowledge Economy (MKE, Korea).

REFERENCES

[1] *Nios II Processor*. Available: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[2] *Xilinx MIcroBlaze*. Available: http://www.xilinx.com/tools/microblaze.htm

[3] *Tensilica Xtensa*. Available: http://www.tensilica.com

[4] *ARC 700*. Available: http://www.arc.com/ configurablecores/ arc700/

[5] L. Pozzi, K. Atasu, and P. Ienne, "Exact and Aproximate Algorithms for the Extension of Embedded Processor Instruction Sets," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* Vol.25, July, 2006.

[6] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A Scalable Application-Specific Processor Synthesis Methodology," in *Proc. ICCAD*, 2003.

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

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

[9] I. Lee, D. Lee, and K. Choi, "Memory Operation Inclusive Instruction-Set Extensions and Data Path Generation," in *Proc. ASAP*, 2007.

[10] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration Through Automated Instraction Set Customization," in *Proc. MICRO*, 2003.

[11] N. Cheung, S. Parameswaran, and J. Henkel, "INSIDE: INstruction Selection/Identification & Design Exploration for Extensible Processors," in *Porc. ICCAD*, 2003.

[12] K. Atasu, C. Öztüran, G. Dündar, O. Mencer, and W. Luk, "CHIPS: Custom Hardware Instruction Processor Synthesis," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* Vol.27, 2008.

[13] K. Atasu, R.G. Dimond, O. Mencer, W. Luk, C. Ozturan, and G. Dundar, "Optimizing Instruction-set Extensible Processors under Data Bandwidth Constraints," in *Porc. EDAA*, 2008.

[14] *SUIF Compiler*. Available: http://suif.stanfor.edu

[15] D. Wu, I. Lee, and K. Choi, "Fast custom instruction generation under area constraint," in *Proc. ISOCC*, 2010.

[16] *MiBench*. Available: http://www.eecs.umich.edu/mibench/

[17] *DSPstone*. Available: http://www.iss.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html

Di Wu received the B.S. degree in school of Microelectronics from Xidian University, Xi'an, China, in 2009. He is currently pursuing M.S degree in department of Electronical Engineering and Computer Science from Seoul National University, Seoul, Korea. His research Interests include processor design and design automation.

Imyong Lee received the B.S. degree in electrical engineering from the Seoul National University, Seoul, Korea, in 2003, where he is currently pursuing the Ph.D degree in electrical engineering. His research interests include computer architecture, design automation, and hardware/software codesign.

Junwhan Ahn is an undergraduate student in electrical engineering from Seoul National University, Republic of Korea, in 2011. His current research interests include computer architecture and design automation.

Kiyoung Choi received the B.S. degree in electronics engineering from Seoul National University, Republic of Korea, in 1978, the M.S. degree in electrical and electronics engineering from KAIST, Republic of Korea, in 1980, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1989. From 1989 to 1991, he was with Cadence Design Systems, Inc. In 1991, he joined the faculty of the Department of Electrical Engineering and Computer Science, Seoul National University. His primary interests include various aspects of computer-aided electronic systems design including embedded systems design, high-level synthesis, and lower-power systems design. He is also interested in computer architecture and especially in configurable and reconfigurable computer architecture design.