Custom Instruction Generation Using Temporal Partitioning Techniques for a Reconfigurable Functional Unit

Farhad Mehdipour†, Hamid Noori††, Morteza Saheb Zamani†, Kazuaki Murakami††, Koji Inoue††, Mehdi Sedighi†

†Computer and IT Engineering Department, Amirkabir University of Technology, Tehran, Iran
{mehdipur,szamani,msedighi}@aut.ac.ir

††Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan
noori@c.csce.kyushu-u.ac.jp
{murakami,inoue}@i.kyushu-u.ac.jp

Abstract. Extracting appropriate custom instructions is an important phase for implementing an application on an extensible processor with a reconfigurable functional unit (RFU). Custom instructions (CIs) are usually extracted from critical portions of applications. It may not be possible to meet all of the RFU constraints when CIs are generated. This paper addresses the generation of mappable CIs on an RFU. In this paper, our proposed RFU architecture for an adaptive dynamic extensible processor is described. Then, an integrated framework for temporal partitioning and mapping is presented to partition and map the CIs on RFU. In this framework, two mapping aware temporal partitioning algorithms are used to generate CIs. Temporal partitioning iterates and modifies partitions incrementally to generate CIs. Using this framework brings about more speedup for the extensible processor.

1. Introduction

An extensible processor with a reconfigurable functional unit (RFU) can be an alternative to General Purpose Processors (GPPs), Application-Specific Integrated Circuits (ASICs) and Application-Specific Instruction set Processors (ASIPs) to achieve enhanced performance in embedded systems. ASICs are not flexible and have an expensive and time consuming design process. On the other hand, GPPs are very flexible but may not offer the necessary performance. ASIPs are more flexible than ASICs and have more potential to meet the high-performance demands of embedded applications, compared to GPPs but the synthesis of ASIPs traditionally involved the generation of a complete instruction set architecture for the targeted application. This full-custom solution is too expensive and has long design turnaround times.

Another method for providing enhanced performance is application-specific instruction set extension. By creating application-specific extensions to an instruction set, the critical portions of an application's dataflow graph (DFG) can be accelerated by using custom functional units. The nodes of these DFGs are the instructions of critical potion of applications and the edges of DFGs represent the dependency between instructions. In our method, custom instruction is a sequence of instructions that are extracted from hot basic blocks (HBBs). HBBs are basic blocks which are executed more than a predefined number of times. A basic block is a sequence of instructions that is terminated by a control instruction. Instruction set extension improves performance and reduces energy consumption of processors but not as effective as ASICs. Instruction set extension also maintains a degree of system programmability, which enables them to be utilized with more flexibility. Using an extensible processor with a reconfigurable functional unit proposes favorable tradeoff between efficiency and flexibility, while keeping design turnaround time much shorter. The reconfigurable part of an extensible processor executes critical portions of an application to gain higher performance. It can be coarse grain or fine grain. The latter is more flexible but it is slower comparing with the coarse grain one. Extracting CIs from applications is an important stage in accelerating application execution. Some generated CIs cannot be mapped on reconfigurable hardware because some RFU constraints, like physical constraints, cannot be considered at the CI generation phase. We call this kind of CIs *rejected CIs*.

Identifying optimal set of custom instruction to improve the computational efficiency of applications has received significant attention recently. Research in reconfigurable computing is often more in line with our goal. Some papers in reconfigurable computing investigate the identification of application sections that are mapped to a reconfigurable fabric. In [7], the authors combine template matching and generation based on the occurrence of patterns which usually led to small templates. Methods presented in [5] and [8] impose further constraints by allowing multiple input-single output patterns. Arnold et al. [1] avoid the exponentially increase of these patterns by using an iterative technique that detects 2-operator patterns, replace their occurrences in the DFG and repeats the process. Atasu et al. [2] search a full binary tree and decide at each step whether or not to include a particular instruction in a pattern, but they do not take into account the underlying hardware architecture. Clark et al. [4] search possibly good patterns by starting with small patterns and expanding them considering the input, output and convexity constraints [16].

In this paper, we propose a novel framework for generating CIs. Our main goal is proposing a framework for generating CIs for AMBER, an adaptive dynamic extensible processor presented in [11]. However, this framework can be used for CI generation as a general methodology. AMBER uses a coarse grain reconfigurable functional unit with fixed resources. Initial CIs are generated by a CI generation tool and some of them might be rejected because of violating RFU constraints. Rejection of CIs decreases the speedup. We do not use any pruning algorithm for making smaller CIs from rejected CIs because obviously, by using bigger CIs, more speedup can be obtained. We use a mapping-aware *temporal partitioning* algorithm to generate CIs.

Temporal partitioning can be stated as partitioning a data flow graph into a number of partitions such that each partition can fit into the target hardware and also, dependencies among the graph nodes are not violated [3, 6]. Different algorithms have been presented for temporal partitioning. Karthikeya et al. [6] proposed algorithms for temporal partitioning and scheduling of large designs on area constrained reconfigurable hardware. *SPARCS* [12] is an integrated partitioning and synthesis framework, which has a temporal partitioning tool to temporally divide and schedule the DFGs on a reconfigurable system. Tanougust et al. [15] attempted to find the minimum area while meeting timing constraints during temporal partitioning. In [14], Spillane and Owen focused on finding a sequence of conditions for an optimized scheduling of configurations to achieve the desired trade-offs among reconfiguration time, operation speed and area. In [9], a design flow was proposed for the compilation of data flow graphs for a reconfigurable system. In this paper, we propose a modified version of this framework for generating appropriate CIs for the RFU of AMBER. In this framework, temporal partitioning is done iteratively and gets feedbacks from mapping to modify partitions and map them onto the RFU. Also, it takes advantages of the basic design flow of [9] to generate CIs and improve target extensible processor speedup.

In Section 2, the architecture of RFU proposed for AMBER is described. Section 3 discusses the design flow proposed for generating CIs and details of temporal partitioning algorithms and their incremental versions. In Section 4, experimental results are presented and finally, Section 5 concludes the paper.

2. AMBER RFU Architecture

In [11] an adaptive extensible processor (AMBER) was presented which has the capability of tuning its extended instructions to the running application. AMBER uses a RISC processor as the base processor. In the first stage of this work, a coarse grain reconfigurable functional unit (RFU) was designed for AMBER using a quantitative approach [4]. The presented RFU architecture is an array of functional units (FUs) (Fig. 1). FUs support all fixed point instructions of the base processor except multiplication, division and load. Twenty-two applications from Mibench [17] were used to provide quantitative analysis. In addition, a mapping tool was developed to map CIs on the RFU. The details of RFU design is out of scope of this paper, and therefore we describe the specification of the final architecture. According to the obtained results, eight inputs, six outputs and 16 FUs brought about a reasonable CI rejection rate. Rejection rate represents the percentage of CIs that cannot be mapped on the RFU according to its defined constraints. In the proposed architecture, there are left to right connections in the 4th row and right to left connections in the 3rd row. The outputs of FUs in each row are fully connected to the inputs of FUs in the subsequent row. Moreover, there are extra vertical connections, as in Fig. 1, between non-subsequent rows to keep the CI rejection rate low.

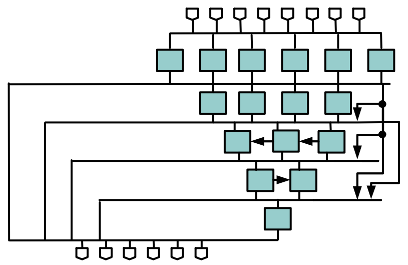


Fig. 1. Architecture of the RFU designed for AMBER

3. Integrated Temporal Partitioning and Mapping

As mentioned in Section 1, in this paper, our main focus is on a method for CI generation. In the following sections, we explain the details of the approaches used for generating CIs.

3.1. Overview

Initial CIs were extracted from hot basic blocks of applications according to the algorithm presented in [11]. Two different approaches for generating appropriate CIs are presented. Appropriate CI set means the set of CIs which satisfy the RFU primary constraints and may have the capability of being mapped successfully on the RFU. RFU *primary constraints* are the architectural constraints including the number of inputs, outputs and nodes. In the first approach (*CIGen*) (Fig. 2(a)), appropriate CIs are generated for each application considering the RFU primary constraints by using the CI generation tool. For rejected CIs, *CIGen* follows a conservative method to generate appropriate CIs. One important drawback of this CI extraction procedure is that it cannot consider all of the constraints such as routing resources constraints. Therefore, some of these CIs may not be mapped to the RFU and should be executed on the base processor.

Integrated Framework is the second CI generation approach that performs an integrated temporal partitioning and mapping process to generate mappable CIs. The proposed design flow for *Integrated Framework* is shown in Fig. 2(b). This design flow takes rejected CIs and attempts to partition them to appropriate CIs with the capability of being mapped on the RFU. In our methodology, a DFG corresponds to a CI and partitions obtained from the integrated temporal partitioning process are the same appropriate CIs which are mappable on the RFU. In the first stage, RFU primary constraints are considered to generate initial CIs. Then for each CI generated in the first step, the mapping process is done and the generated CIs are accepted and finalized if they can be mapped on the RFU. Otherwise, an incremental temporal partitioning algorithm modifies the CI (partition) by moving some of the nodes to the subsequent CI (partition). Mapping algorithm attempts to reduce total connection length between the nodes and satisfy the RFU architectural constraints simultaneously. In the next step, the mapping process is repeated. This process is done iteratively until all partitions are mapped successfully on RFU. This framework gains the following advantages:

1. Reducing the number of rejected CIs that can affect the overall performance by partitioning the rejected CIs to appropriate CIs which can be mapped on the RFU.
2. Using a mapping-aware temporal partitioning process to prohibit the rejection of CIs by modifying CIs according to the feedbacks obtained from the mapping process.

In the *Integrated Framework,* two algorithms were developed for temporal partitioning which are described in the following section.

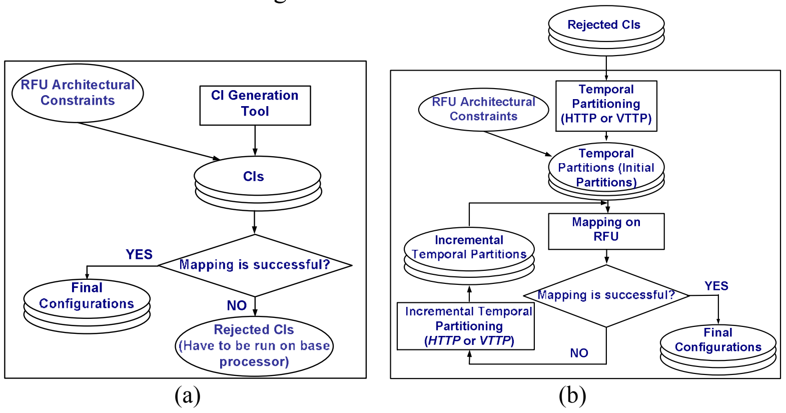

RFU Architectural Constraints
CI Generation Tool
CIs
YES
Mapping is successful?
Final Configurations
NO
Rejected CIs (Have to be run on base processor)
(a)
Rejected CIs
Temporal Partitioning (HTTP or VTTP)
RFU Architectural Constraints
Temporal Partitions (Initial Partitions)
Mapping on RFU
Incremental Temporal Partitions
Mapping is successful?
YES
Incremental Temporal Partitioning (HTTP or VTTP)
NO
Final Configurations
(b)

Fig. 2. Design flows for *CIGen* (a) and the *Integrated Framework* (b)

3.2. Horizontal and Vertical Traversing Temporal Partitioning (*HTTP* and *VTTP*)

HTTP [9] is used as the first temporal partitioning algorithm in the *Integrated Framework*. This algorithm traverses DFG nodes horizontally according to the *ASAP* (As Soon As Possible) level of the nodes and adds them to the current partition while architectural constraints are satisfied. The *ASAP* level of nodes represents their order to execute according to their dependencies [2, 9]. For example, a parent node should be executed before its descents because of data dependencies between them. *HTTP* algorithm partitions the input DFG by horizontally traversing of DFG nodes. This algorithm usually brings about more parallelism for instruction execution that may result in increasing required intermediate data size. On the other hand, intermediate data size can affect data transferring rate and configuration memory size. We present another temporal partitioning algorithm to vertically traversing of DFG nodes. Although using this algorithm creates partitions with longer critical paths it reduces the intermediate data size.

3.3. Partitions Modification

In *Integrated Framework*, each partition which does not satisfy RFU constraints, is modified by selecting and moving proper nodes to the subsequent partition and then a new iteration starts. For *HTTP* and *VTTP* algorithms, two different strategies are used as incremental algorithms for partition modification.

Incremental HTTP. This complementary algorithm selects a node and moves it to the next partition. A new partition is created and the number of partitions is increased by 1 if there is no more partition. The best choice for moving nodes are the nodes with highest *ASAP* level. All nodes in a partition are sorted according to their *ASAP* level and the node with the highest *ASAP* level is selected to move to the subsequent partition. In Fig. 3(a), the nodes 15, 13, 11, 9, 14, 12, 10, 8, 3, 7 are selected in order and moved to the next partition.

Incremental VTPP. This algorithm chooses another strategy for selecting and moving the nodes. The most important characteristic of *VTTP* is extracting long paths by in-depth traversing of DFG nodes. Therefore, selecting nodes from a partition should be done according to this property; otherwise the results of the modification process will converge to those of the *HTTP* algorithm. Our experiments justify this statement. In the first attempt, a node with the highest *ASAP* level is selected and moved to the next partition. In other attempts for modifying the same partition, the nodes are selected from the path where the previous moved node had been located. A node with the highest *ASAP* level from another path is selected if there is still any node belonging to the current partition on the processing path. In Fig. 3(b), the nodes 15, 14, 6, 13, 12, 5, 11, 10, 4, 7 are selected in-order and moved to the next partition during the incremental *VTTP*.

3.4. Mapping Procedure

In our *Integrated Framework*, the mapping process is the same as the well-known placement problem [13]. Mapping process can be defined as the placement of the DFG nodes on a fixed architecture RFU, to determine the appropriate positions for DFG nodes on the RFU. Minimizing the connection length, area and the longest wire are usually the main goals in this process [13]. Assigning CI instructions or DFG nodes to FUs is done based on the priority of the nodes. We calculated *slack* of nodes [10] to determine their priority for partitioning. Slack of each node represents its criticality. For example, slack equal to *0* means that it is on the critical path of DFG and should be scheduled with the highest priority. *ASAP* level of nodes determines the order of partitioning for the nodes with equal slack value.

The nodes with lower value of *ASAP* level should be scheduled according to their execution order in the DFG. Therefore, in the first step, *ASAP, ALAP* (As Late As Possible) and *slack* values of each node in the DFG are determined [9, 10]. Assigning a position for each selected node starts by determining an appropriate row for that node. Row number is set to the last row if the selected node is on a critical path with the length more than or equal to the RFU depth. Otherwise, row number is selected according to *slack* and *ALAP* of the selected node and the number of unoccupied cells available in the RFU rows. For the nodes which do not belong to any critical path longer than the RFU depth, their starting row is set to *ALAP- slack -1*. This means that we reserve FUs of the lower rows for the nodes belonging to the critical path. Therefore, spiral shaped mapping of nodes are possible for long critical paths. After determining the row number, an appropriate column is determined for the selected node. Column number is determined according to the minimum connection length criterion. All unoccupied cells of the RFU in the determined row are checked to find an FU which gives the minimum connection length. For each row, a maximum capacity is considered to prohibit gathering many nodes in a row. Capacity of rows is determined with respect to the longest critical path and the number of critical paths in the DFG. Referring to the RFU architecture in Fig. 1 and its routing resources, though the RFU depth is equal to 5, our mapping algorithm can map CIs whose critical path length are at most equal to 8. Fig. 3 show examples of mapping of CIs on the RFU. Corresponding CI of the first partition in Fig. 3(b) has been mapped on RFU in a spiral shaped path because of its long length.

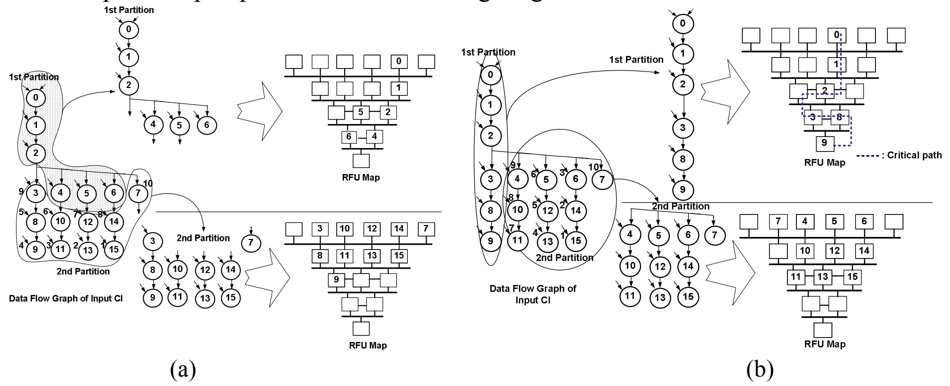


Fig. 3. Examples of *HTTP* (a) and *VTTP* (b) and their related incremental versions

4. Experimental Results

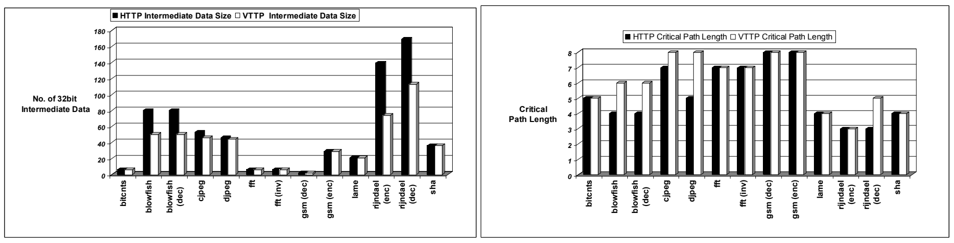
SimpleScalar tool set (PISA configuration)[18] and 22 applications of Mibench [17] were used for doing experiments. Initial CIs were generated according to the method proposed in [11]. CI rejection rate with respect to RFU architectural constraints was about 10%. Table 1 shows the minimum and maximum length of initial CIs. Also it shows the minimum length of rejected CIs which are applied to *Integrated Framework*. Application names include rejected CIs are shown in bold face. Last column of Table 1 depicts in 9 of the 22 applications, there was not any rejected CI, which means that all CIs in these applications were mapped on the RFU successfully. Also, for 13 of the 22 applications that include rejected CIs, CI rejection percentage is at least 1.9% for *sha* and at most 43.2% for *blowfish* and *blowfish(dec).*

The base line processor was a 4-way in-order RISC processor with a 32KB L1 data cache (1 clock cycle latency); a 32KB L1 instruction cache (1 clock cycle latency) and a 1MB unified L2 cache (6 clock cycles latency). On the other hand, it was assumed that the RFU has a variable latency based on the length of the longest critical path [11]. It was presumed that the first row of the RFU takes one clock cycle and the other rows take 0.5 clock cycles for execution. For generating appropriate CIs, as mentioned in Section 3, we used two different approaches. First, we used *CIGen* to generate CIs with respect to RFU constraints. For these CIs, the mapping process was done and some of them were rejected again at the mapping stage because of the RFU routing resource constraints. Experiments show that 10 of 13 applications already have some rejected CIs using the *CIGen*.

Table 1. CIs length for Mibench applications

Application Name	**Min. CI length**	**Max. CI length**	**Min. Rejected CI length**	**CI Rejection %**
adpcm(enc)	5	7	-	0
adpcm(dec)	5	7	-	0
bitcounts	**4**	**20**	**20**	**14.3**
blowfish	**5**	**16**	**15**	**66.7**
blowfish (dec)	**5**	**16**	**15**	**66.7**
basicmath	3	11	-	0
cjpeg	**5**	**59**	**11**	**20.7**
crc	5	5	-	0
dijkstra	4	9	-	0
djpeg	**4**	**48**	**8**	**23.8**
fft	**3**	**16**	**16**	**8.3**
fft (inv)	**3**	**16**	**16**	**8.3**
gsm (dec)	**5**	**14**	**14**	**6.3**
gsm (enc)	**4**	**26**	**13**	**9.5**
lame	**3**	**13**	**7**	**8.3**
patricia	3	6	-	0
qsort	5	7	-	0
rijndael (enc)	**5**	**16**	**10**	**46.3**
rijndael (dec)	**5**	**18**	**10**	**57.1**
sha	**5**	**18**	**7**	**18.5**
stringsearch	5	9	-	0
susan	6	10	-	0

In the second approach, we used the *Integrated Framework* to partition the rejected CIs and generate appropriate CIs, which are successfully mapped on the RFU. We compared two *HTTP* and *VTTP* algorithms with respect to critical path length of generated CIs, intermediate data size and speedup. Fig. 4(a) compares two algorithms with respect to intermediate data size. For 6 of 13 applications, intermediate data size is smaller using *VTTP*. For 7 remaining applications intermediate data size is the same. Another comparison was done with respect to critical path length. Fig. 4(b) shows that *VTTP* generated CIs with critical length equal to or more than *HTTP* because it traverses DFG nodes in depth, whereas *HTTP* traverses them horizontally. Finally, we compared both algorithms regarding the speedup obtained from the extensible processor. Fig. 5 depicts the comparison of speedup achieved using *HTTP* and *VTTP*. Using both of these algorithms, all CIs were mapped successfully on the RFU but *HTTP* resulted in better speedup, since it benefits from parallelism more in the instruction execution. In other words, critical path length is less using *HTTP*, and therefore, RFU execution latency was smaller. In addition, according to Fig. 5, both *HTTP* and *VTTP* offer better speedup compared to *CIGen*.



(a) (b)

Fig. 4. Intermediate data size (a) maximum critical path length for CIs (b)

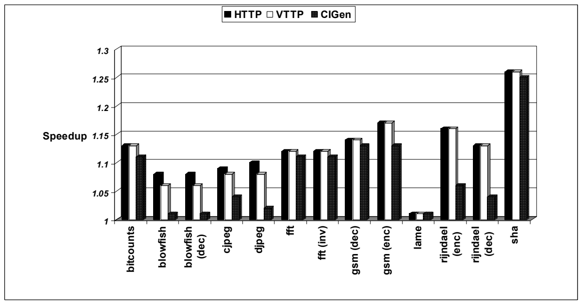


Fig. 5. Speedup comparison between *HTTP, VTTP* and *CIGen*

5. Conclusion

In this paper, an integrated framework was presented to address generating appropriate custom instructions and mapping them on the RFU of an adaptive extensible processor.
First, an RFU was presented for AMBER, a dynamic adaptive extensible processor. Generating appropriate CIs by applying the RFU constraints to the CI generation tool may still cause some generated CIs to be rejected. This approach does not have the capability of considering constraints such as routing resource constraints before mapping. *Integrated Framework* is the second approach we used to generate CIs. This framework can be used as a general approach for generating CIs. *Integrated Framework* uses mapping-aware temporal partitioning algorithms for generating appropriate CIs. In this framework, each rejected CI is partitioned to multiple partitions and is iteratively modified to meet the RFU constraints. CI modification is done using incremental versions of *HTTP* and *VTTP* algorithms. Our proposed mapping algorithm uses spiral shaped paths to cover CIs including critical path lengths more than the RFU depth. The experimental results showed that for the attempted benchmarks, this framework successfully mapped all CIs on the RFU. Also, the *Integrated Framework* using both *HTTP* and *VTTP* brought about more speedup enhancement compared to *CIGen*. In addition, *HTTP* gained higher performance in comparison with *VTTP* because of more instruction parallelism.

Acknowledgement

The authors would like to thank *System LSI* Laboratory of Kyushu University for providing the necessary facilities and equipments. This work has been supported by Iran Telecommunication Research Center *(ITRC)*.

References

1. Arnold, M., Corporaal, H., Designing domain-specific processors. In Proceedings of the Design, Automation and Test in Europe Conf, 2001, pp. 61-66.
2. Atasu, K., Pozzi, L., Lenne, P., Automatic application-specific instruction-set extensions under microarchitectural constraints, 40th Design Automation Conference, 2003.
3. Bobda, C., Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement, Ph.D thesis, Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, 2003.
4. Clark, N., Kudlur, M., Park, H., Mahlke, S., Flautner, K., 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.
5. Halfhill, T.R., MIPS embraces configurable technology, Microprocessor Report, 3 March 2003.
6. Karthikeya, M., Gajjala, P., Dinesh, B., Temporal partitioning and scheduling data flow graphs for reconfigurable computer, IEEE Transactions on Computers, vol. 48, no. 6, 1999, pp.579–590.
7. Kastner, R. Kaplan, A., Ogrenci Memik, S., Bozorgzadeh, E., Instruction generation for hybrid reconfigurable systems, ACM TODAES, vol. 7, no. 4, 2002, pp. 605-627.
8. Lee, C., Potkonjak, M., Mangione-Smith, W.H., MediaBench: A tool for evaluating and synthesizing multimedia and communications systems, In Proceedings of the 30-th Annual Intl. Symp. On Microarchitecture, 1997, pp 330-335.
9. Mehdipour, F., Saheb Zamani, M., Sedighi, M., An integrated temporal partitioning and physical design framework for static compilation of reconfigurable computing system, International Journal of Microprocessors and Microsystems, Elsevier, vol. 30, no. 1, Feb 2006, pp. 52-62.
10. Micheli, G.D., Synthesis and optimization of digital circuits, McGraw-Hill, 1994.
11. Noori, H., Murakami, K., Inoue, K., General Overview of an Adaptive Dynamic Extensible Processor Architecture, Workshop on Introspective Architecture (WISA'2006) , 2006.
12. Ouaiss, I., Govindarajan, S., Srinivasan, V., Kaul M., Vemuri R., An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures, In Proceedings of the Reconfigurable Architecture Workshop, 1998, pp. 31-36.
13. Sherwani N, Algorithms for VLSI physical design automation, Kluwer-Academic Publishers, 1999.
14. Spillane, J., Owen, H., Temporal partitioning for partially reconfigurable field programmable gate arrays, IPPS/SPDP Workshops, 1998, pp. 37-42.
15. Tanougast, C., Berviller, Y., Brunet, P., Weber, S., Rabah, H., Temporal partitioning methodology optimizing FPGA resources for dynamically reconfigurable embedded real-time system, International Journal of Microprocessors and Microsystems, vol. 27, 2003, pp. 115-130.
16. Yu, P., Mitra, T., Characterizing embedded applications for instruction-set extensible processors, In Proceedings of Design and Automation Conference, 2004, pp. 723- 728.
17. http://www.eecs.umich.edu/mibench.
18. http://www.simplescalar.com.