An Integrated Temporal Partitioning and Mapping Framework for Improving Performance of a Reconfigurable Instruction Set Processor[*]

Farhad Mehdipour[a **], Hamid Noori[b], Morteza Saheb Zamani[c], Hiroaki Honda[d], Koji Inoue[a], Kazuaki Murakami[a]

[a] *Faculty of Information Science and Electrical Engineering, Department of Informatics, Kyushu University, Fukuoka, Japan*
[b] *School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran*
[c] *Department of Computer Engineering and IT, Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran*
[d] *Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka, Japan*

Received 5 December 2008, revised 9 September 2009, accepted 21 September 2009

Abstract

Reconfigurable instruction set processors allow customization for an application domain by extending the core instruction set architecture. Extracting appropriate custom instructions is an important phase for implementing an application on a reconfigurable instruction set processor. A custom instruction (CI) is usually extracted from critical portions of applications and implemented on a reconfigurable functional unit. In this paper, our proposed RFU architecture for a reconfigurable instruction set processor is introduced. As the main contribution of this work, an integrated framework of temporal partitioning and mapping is introduced that partitions and maps CIs on the RFU. Temporal partitioning iterates and modifies partitions incrementally to generate CIs. The proposed framework improves the timing performance particularly for the applications comprising a considerable amount of CIs that could not be implemented on the RFU due to architectural limitations. Furthermore, exploiting similarity detection and merging as two complementary techniques for the integrated framework brings about reduction in the configuration memory size.

Keywords: Reconfigurable instruction set processor, Custom instruction, Reconfigurable functional unit, Temporal partitioning.

1. Introduction

Synthesis of application-specific instruction-set processors (ASIPs) has been an important design methodology for system-on-chip processors in the last decade. ASIPs have more potentials to meet the high-performance demands of embedded applications, compared to general purpose processors (GPPs) but the synthesis of ASIPs traditionally involved the generation of a complete instruction set architecture for the targeted application. On the other hand, GPPs are very flexible but may not offer the necessary performance. Reconfigurable instruction set processor is one of the recent trends in customization. An important feature of this design method is extending an existing processor core with units specialized for a given domain, rather than designing a custom processor completely. The main motivation toward customization of existing processors versus the design of complete ASIPs is to avoid the complexity and cost of complete processor and tool set development.

Reconfigurable instruction set processors allow addition of a functional unit or co-processor and application-specific custom instructions (CIs) to the core instruction set architecture to meet the critical computational demands of a target application [3]. Instruction set customization is an effective way to improve processor performance and also to meet the power demands of embedded applications. It also maintains a degree of system programmability, which enables them to be utilized with more flexibility. Using a reconfigurable instruction set processor with a reconfigurable functional unit proposes favorable tradeoff between efficiency and flexibility while keeping design turnaround time much shorter.

The reconfigurable part of a reconfigurable instruction set processor executes critical portions of an application to gain higher performance. It can be fine-grained or coarse-grained [7]. In fine-grained systems processing elements are typically logic gates, flip-flops and look-up tables. They operate at the bit level, implementing a Boolean function of a finite-state machine. Also, they are more flexible but they are slower compared with the coarse-grained one. On the

[*] A portion of this article appeared in Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference (ACSAC'06). This is an expanded version that includes over 30% new materials and results.
[**] Corresponding author. E-mail: farhad@c.csce.kyushu-u.ac.jp

other hand, in coarse-grained hardware, the processing elements may contain complete functional units, like ALUs that operate upon multiple-bit words. Coarse-grained hardware demand for less configuration memory and mapping of instructions on them is easier [7].

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 potions of applications and the edges of DFGs represent data dependencies between instructions corresponding to 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 such as branch [13].

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*. Two different strategies are used for rejected CIs. In the first case, rejected CIs are run on the base line processor, and so, this offers no speedup. As the second strategy, we suggest using approaches to recover and execute rejected CIs on the RFU rather than the base processor. To achieve this goal, two approaches are proposed. In the first approach, a CI generation tool is used to regenerate the CIs from HBBs according to the RFU constraints.

As another approach, we propose a framework for generating CIs. This framework generates CIs in such a way that they can be executed on the RFU. Besides, it partitions rejected CIs to multiple mappable CIs. The same well-known temporal partitioning [4][6][10] concept is utilized to this end. Temporal partitioning can be stated as partitioning a data flow graph (DFG) 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 [4][6].

Previous work in the domain as well as our contributions will be reviewed in Section 2. In Section 3, an overview of our target reconfigurable instruction set processor called AMBER, which was introduced in [13], its components and the proposed RFU architecture for AMBER are highlighted. Section 4 discusses the design flow proposed for generating CIs and the details of the temporal partitioning algorithms and their incremental versions. In Section 5, custom instruction similarity detection and merging algorithm for reducing configuration memory size are discussed. In Section 6, experimental results are presented and, finally, Section 7 concludes the paper.

2. Related Literature

Research in reconfigurable instruction set processors has mainly been revolving around automatic instruction generation/selection. Identifying an optimal set of custom instructions to improve the computational efficiency of applications has received significant attention more recently. PRISC [15] and Chimaera [21] provide compilation tools that attempt to automatically generate and map the custom instructions on the reconfigurable logic. Their generated custom instructions tend to be relatively small, due to the difficulty of the matching problem and the size of the available programmable fabric.

Research in reconfigurable computing is often more in line with our goal. Some research papers on reconfigurable computing have addressed the identification of application sections that are mapped to a reconfigurable fabric. Most CI extraction methods attempt to identify patterns within a basic block. In [7], the authors combine template matching and generation based on the occurrence of patterns which usually led to small templates. Template matching is done based on graph isomorphism. Arnold et al. [1] avoid the exponential increase in these patterns by using a technique that iteratively detects two-operator patterns and replaces their occurrences in the DFG. 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. The potential large search space is pruned based on input/output constraints. They attempt to find maximal sub-graphs of an application data flow graph but they do not take into account the underlying structure of the execution hardware. Clark et al. [5] search possibly good patterns by starting with small patterns and expanding them considering the input, output and convexity constraints.

The general goal of this study is presenting methods for CI generation, specifically for recovering the rejected CIs. Moreover, the main intuition behind this work which helps to justify our claims is generating CIs for AMBER, a reconfigurable instruction set processor introduced in [13]. AMBER uses a coarse-grained reconfigurable functional unit with fixed resources. Initial CIs are generated by a simple CI generation tool presented in [13] some of the generated CIs 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. In fact, in our proposed approach an integrated temporal partitioning and mapping framework is used for CI generation. CIs generated by this framework are the maximal sub-graphs extracted from the data flow graph of the corresponding rejected CI.

Temporal partitioning is a basic function of our proposed framework. Several algorithms have been presented for temporal partitioning. SPARCS [14] is an integrated partitioning and synthesis framework, which has a temporal partitioning tool to temporally divide and schedule DFGs on a reconfigurable system. Bobda [4] proposed a temporal partitioning approach based on positioning of modules in a three-dimensional vector space. Karthikeya et al. [6] proposed algorithms for temporal partitioning and scheduling of large designs on area constrained reconfigurable hardware. Tanougust et al. [19] attempted to find the minimum area while meeting timing constraints during temporal partitioning. Spillane and Owen [18] 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.

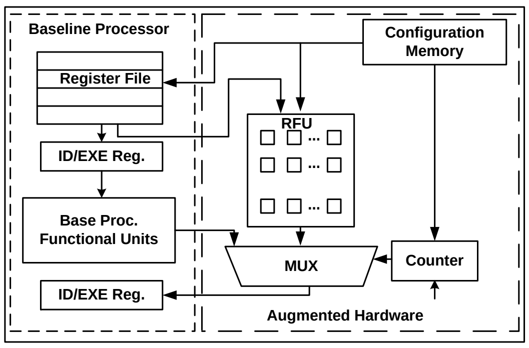
In this paper, we propose a framework for CI generation relying on the integrated framework introduced in [10]. This framework can be used as a general methodology for CI generation in which temporal partitioning is done iteratively and gets feedbacks from the mapping process to modify partitions and map them onto the RFU to improve target reconfigurable instruction set processor's speedup. Another contribution of this work is to introducing similarity detection and merging as two complementary techniques to reduce the number of generated CIs and thus the configuration memory size.

3. General Overview of AMBER

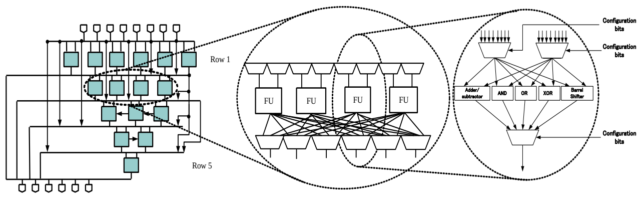
AMBER was introduced in [13] and falls in the category of the reconfigurable instruction set processors. It has been developed by integrating a base processor with three other main components. The base processor is a general RISC processor and the other three components are: profiler, sequencer and a coarse-grained reconfigurable functional unit (RFU) (Figure 1.a). AMBER has two operational modes: a training mode and a normal mode. In the training mode, the applications are run on the base processor and profiled by the profiler. Then, the start addresses of HBBs are detected. They are read from the object code using the start addresses of HBBs and the CIs are extracted from the HBBs. In this mode, configuration data for RFU are generated and a sequencer table is initiated. When these processes are done, the processor switches to the normal mode. In the normal mode, using the RFU, its configuration data (which is stored in the configuration memory), sequencer and its table, the custom instructions are executed on the RFU. AMBER enters the training mode once, and learns about custom instructions and then it switches to the normal mode.

The *base processor* is an in-order RISC processor that supports MIPS instruction set. The *profiler* performs the profiling for running applications through monitoring the program counter (PC). The profiler looks for jumps and taken branches by monitoring PC to detect critical regions. The profiler has a table with a counter for each entry that keeps the execution frequency for basic blocks. Using the profiler table and a threshold value, the start addresses of HBBs are detected [13].

The *sequencer* mainly determines the microcode execution sequence by selecting between the RFU and the processor functional unit. It has a table in which the start addresses of custom instructions (which are going to be executed on the RFU) are specified. The table of the sequencer is initialized according to the locations of the custom instructions in the object code at the training mode when they are generated. In the normal mode, it checks whether the corresponding configuration is available in the multi-context memory or not. In the case of availability, the sequencer selects the configuration bits to read registers which are required by the CI, from the register file. Then it switches from processor functional unit to the RFU, waits for some specified clock cycles and lets the RFU finish the execution of the CI. If the configuration is not available in the multi-context memory, the original code will be executed on the processor functional unit as usual. In this case, only the expected speedup for that instruction will be missed and there will be no penalty. Figure 1 illustrates the integration of different components in AMBER.



(a)



(b)

Fig. 1. AMBER's architecture (a) Integration of main components in AMBER; (b) RFU detailed architecture.

The *RFU* is an array of functional units (FUs) that is located inside of the base processor in parallel with standard functional units (Figure 1.b). According to the size of data in a processor, an array of functional units (FUs) seems an efficient and reasonable hardware for accelerating dataflow subgraphs as CIs [5]. Using coarse-grained reconfigurable accelerators demands less configuration memory. In addition, they are faster comparing with fine-grained programmable hardware and mapping instructions on them is easier. Each FU of our proposed coarse-grained RFU supports all fixed-point instructions of the base processor except multiplication, division and load.

In the first stage of this work, a quantitative approach was followed to determine the architectural specifications of RFU using the twenty-two applications from Mibench [11].

A mapping tool was developed to map CIs on the RFU. We merely describe the specification of the final architecture. According to the obtained results, eight inputs, six outputs and 16 FUs were chosen for the RFU because they brought about a reasonable CI rejection rate (about 10%). Rejection rate represents the percentage of CIs that cannot be mapped on the RFU according to its defined constraints. In addition, a proper topology for RFU connections was achieved based on the quantitative analysis. In the proposed architecture, there are left to right connections in the fourth row and right to left connections in the third row. As mentioned above, 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 Figure 1.b, between non-subsequent rows to keep the CI rejection rate low. Obviously, detecting unnecessary connections through quantitative analysis of the DFGs and removing them result in less area and power consumption. More details on AMBER and its components can be found in [13].

4. CI Handling Approaches

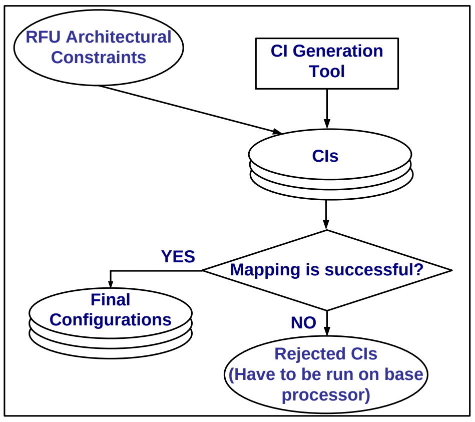
As mentioned in previous sections, our main focus is on a method for handling CIs. In the following sections, the details of the CI generation approaches are explained.

4.1. Overview

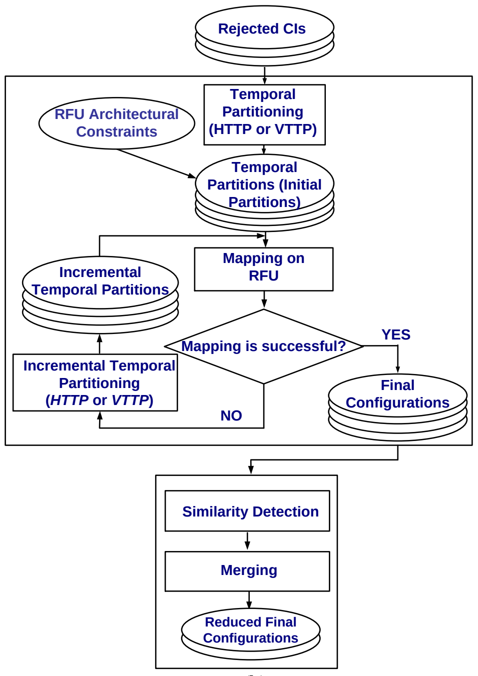
Basic CI generation approach has been proposed in [13]. This CI generator algorithm gets the DFG of each HBB as an input and then looks for the biggest sequence of instruction that can be executed on the RFU. Then after checking the flow and anti dependencies, *mappable instructions* are moved and added to the head and tail of the detected biggest instruction sequence. *Mappable instructions* are those instructions that can be implemented by RFU. It should also be checked that the area where the instructions are going to be moved, are not target of branch instructions. If these conditions are met then for those parts of the object code that instructions are moved, object code is rewritten (binary rewriting). In this algorithm CIs are generated in such a way that they can become as big as possible and RFU architectural constraints are not considered during CI generation process, therefore some of the generated CIs do not satisfy the RFU architectural constraints and cannot be mapped on the RFU. These CIs are executed on the base processor during run time; therefore they offer no more speedup.

To reduce the CI rejection rate and generating appropriate CIs, two different approaches are presented [9]. Appropriate CI set means the set of CIs which satisfy the RFU primary constraints and might 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*) (Figure 2.a), appropriate CIs are generated for each application considering the RFU *primary constraints* by using a modified version of simple CI generation algorithm [13] described above. *CIGen* uses the basic CI generation algorithm and considers RFU *primary constraints* for generating appropriate CIs. In this method, the decision on accepting or rejecting CIs is made after their mapping on the RFU since the CI generation process is unaware of the mapping process results. 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 ultimately mapped to the RFU. These CIs are rejected and should be executed on the base processor.


RFU Architectural Constraints
CI Generation Tool
CIs
Mapping is successful?
YES
NO
Final Configurations
Rejected CIs (Have to be run on base processor)

(a)


Rejected CIs
RFU Architectural Constraints
Temporal Partitioning (HTTP or VTTP)
Temporal Partitions (Initial Partitions)
Incremental Temporal Partitions
Mapping on RFU
Mapping is successful?
YES
NO
Incremental Temporal Partitioning (HTTP or VTTP)
Final Configurations
Similarity Detection
Merging
Reduced Final Configurations

(b)

Fig. 2. Design flow for (a) *CIGen*; (b) *IntegFrame*.

IntegFrame is the second CI generation approach that is based on the framework in [10]. It performs an integrated temporal partitioning and mapping process to generate mappable CIs. The proposed design flow for *IntegFrame* is shown in Figure 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. Moreover, the 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 partitions (CIs). Then for each partition (CI) generated in the first step, the mapping process is done. These CIs are accepted and finalized if they can be mapped on the RFU. Otherwise, an incremental temporal partitioning algorithm modifies the partition (CI) by moving some of the nodes to the subsequent partition (CI). The mapping process is repeated to map the modified partition (CI) on the RFU. It attempts to reduce total connection length between the nodes and satisfy the RFU architectural constraints simultaneously. This process is repeated until all partitions are mapped successfully on the RFU. This framework can have the following advantages:

Reducing the number of rejected CIs: This can affect the overall performance by partitioning the rejected CIs to CIs which can be mapped on the RFU.

Using a mapping-aware temporal partitioning process: this process attempts to prevent the rejection of CIs by modifying CIs according to the feedbacks obtained from the mapping process. In fact, only primary constraints of the RFU can be considered in the *CIGen* but it is unaware of such mapping information as routing resource constraints. In the *IntegFrame*, CIs are partitioned in such a way that they can be mapped on the RFU.

Two temporal partitioning algorithms and their incremental versions were developed specially for this framework. The mapping-aware incremental versions can be used during iterative CI generation process. The time complexity of both temporal partitioning algorithms is $O(n^2)$, where n is the number of instructions in the CI.

4.2. Horizontal Traversing Temporal Partitioning (HTTP)

The *IntegFrame-HTTP* (briefly referred as *HTTP*) is the first temporal partitioning algorithm. 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 [12]. In other words, a parent node should be executed before its descents because of data dependencies between them. For example, in Figure 3, by using the *HTTP* algorithm, a DFG corresponding to a rejected CI has been partitioned into two smaller CIs which are mappable on the RFU. The number located at the left top side of each node stands for the selecting and moving order of that node.

4.3. Vertical Traversing Temporal Partitioning (VTTP)

HTTP algorithm partitions a given DFG by horizontally traversing of the DFG nodes and usually brings about more parallelism for instruction execution. However, this may require large intermediate data to be transmitted to the subsequent partition. The size of intermediate data can affect data transfer rate and the size of configuration memory. We presented *IntegFrame-VTTP* (briefly referred as *VTTP*) as an alternative to *HTTP* to vertically traversing the DFG. Although using this algorithm creates partitions with longer critical paths, it reduces the size of intermediate data. Figure 4 shows an example for *VTTP* algorithm.

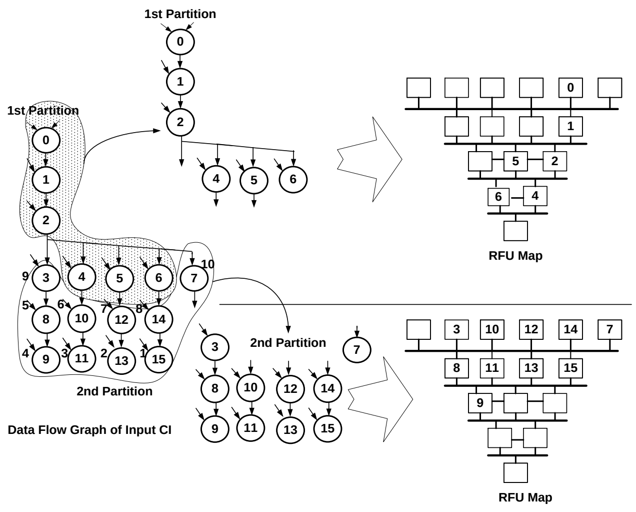


Fig. 3. Example of *HTTP* and its related incremental algorithm.

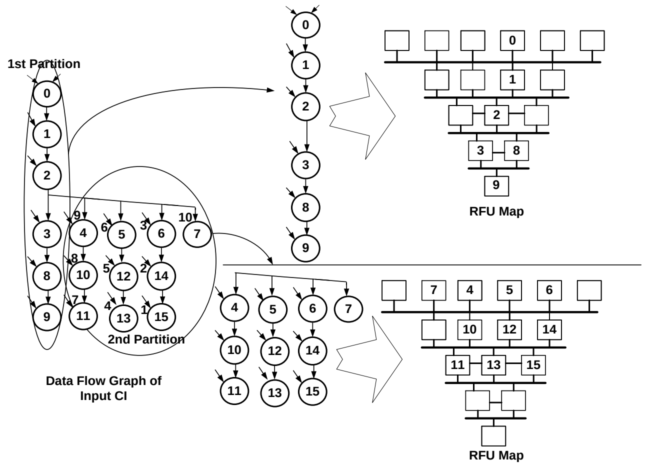


Fig. 4. Example of *VTTP* and associated incremental algorithm.

4.4. Partitions Modification

In the *IntegFrame*, an incremental temporal partitioning process is accomplished iteratively until all partitions are mapped on the RFU successfully. 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, we propose two different partition modification strategies.

The main difference between these two algorithms is in the way of selecting the nodes to be moved to the next partition. The time complexity of this process is $O(n^2)$.

Incremental HTTP: Firstly, this algorithm chooses 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 and moved to the subsequent partition. In Figure 3, the order in which are selected and moved to the next partition is 15, 13, 11, 9, 14, 12, 10, 8, 3 and 7. The nodes are moved until all the generated partitions satisfy the RFU architectural constraints.

Incremental VTPP: Incremental *VTPP* algorithm chooses another strategy for selecting and moving nodes from current non-mappable partition to the next partition. The goal of *VTTP* was to reduce the size of intermediate data by in-depth traversing of DFG nodes. Therefore, selecting nodes from a partition should be done according to this goal; otherwise the results of the modification process may converge to those of the *HTTP* algorithm. Our experiments demonstrate this statement. In the first attempt, a node with the highest ASAP level is selected and moved to the next partition. If the constraints are still not met, the nodes are selected from the path where the previous moved node had been located in their ASAP level order. A node with the highest *ASAP* level from another path is selected if there is not any more node belonging to the current partition on the processing path. In Figure 4, the order of node selection is shown. The nodes 15, 14, 6, 13, 12, 5, 11, 10, 4 and 7 may be selected in-order and moved to the next partition during the incremental *VTTP*.

4.5. Mapping Algorithm

The mapping process in the *IntegFrame* is the same as the well-known placement problem [16]. Placement problem can be defined as determining appropriate positions for hardware modules on the target device. Minimizing the connection length, area and the longest wire are usually the main goals in this process [16]. Here, the mapping process can be defined as the placement of DFG nodes on a fixed architecture RFU, to determine the appropriate positions for DFG nodes on the RFU**.** Assigning CI instructions or DFG nodes to FUs is done based on their priority. We calculated *slack* of nodes [12] to determine their priority for mapping. 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. On the other hand, for the nodes with the same criticality, their *ASAP* level determines their mapping order.

In the first step of mapping process, *ASAP, ALAP* (As Late As Possible) and *slack* values of each node in the DFG are calculated [12]. 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 the *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. For this purpose, it is tried to prevent the occupation of FUs in the lower RFU rows by the nodes that do not belong to critical paths. Therefore, spiral shaped mapping of nodes is possible for long critical paths thanks to the horizontal connections in the third and fourth rows. After setting the row number, an appropriate column is determined for the selected node. Column number is calculated according to the minimum connection length criterion. All unoccupied units of the RFU in the selected row are checked to find an FU which gives the minimum connection length between the selected node and its dependent nodes already positioned on the RFU.

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. Row number is decreased and a new attempt starts if there is not any FU in that row to assign the selected node.

Referring to the RFU architecture in Figure 1.a and its routing resources, though the RFU depth is equal to 5, our mapping algorithm can map CIs whose critical path length is at most equal to 8. In Figure 3 and Figure 4, examples of mapping of CIs on the RFU have been shown. Corresponding DFG of the first partition in Figure 4 has a critical path longer than the RFU depth, and so it takes advantage of a spiral shaped mapping. This kind of mapping results in effective usage of routing resources (horizontal connections of the third and forth rows) and FUs. At the same time, the mapping algorithm attempts to reduce total connection length between DFG nodes.

5. Reducing Configuration Memory Size

The large number of custom instructions generated for applications can potentially be an important issue in the implementation of RFU and configuration memory being used for storing configuration data. Employing the *IntegFrame* is effective in increasing the number of CIs due to splitting a CI into multiple CIs.

To support an application containing large number of CIs, a large size of configuration memory is needed. Increasing configuration memory size can impose more hardware cost as well as more power consumption for the reconfigurable instruction set processor. On the other hand, according to observations obtained from experiments, we came to following conclusions:

- Usually, there are a considerable number of similar CIs in applications.
- There is some wasted area in the RFU with respect to existing small CIs in which the number of nodes are

noticeably less than the number of FUs available in the RFU.

Therefore, two different techniques for reducing the size of configuration memory based on similarity detection and merging of CIs are proposed. In following sections, these two complementary methods are described.

5.1. Exploring Similarity of Custom Instructions

The number of custom instructions for applications can be reduced by detecting the similarity between CIs. Similarity of two CIs can be detected based on four kinds of similarity: a) similarity of nodes (FUs) and connections in corresponding DFGs, b) similarity of inputs, c) similarity of outputs and d) similarity of immediate inputs. In most cases, CIs have similar configuration bits related to FUs and connections but those bits for inputs or outputs are different. In addition, detecting the similarity of nodes (FUs) and connections needs a special similarity detection procedure. Therefore, we isolate nodes and connections similarity from other kinds of similarity. In this way, two types of FUs and connections similarity are defined:

- *Complete Similarity:* The functionality of nodes (FUs) and connections of both CIs are completely the same. In this case, considering corresponding DFG of each CI, there is a one to one similarity between nodes and also connecting edges of the two DFGs.
- *Subset Similarity:* One of the CIs is completely similar to a subset of another CI in terms of the nodes and connections.

We use a naïve graph isomorphism algorithm to explore the nodes and connections similarity of two DFGs. The graph isomorphism problem is to determine whether there is a one to one correspondence between the nodes of the graphs while preserving the edges. In other words, two graphs are isomorphic if there is a one-to-one correspondence between their nodes and also, there is an edge between two nodes of one graph if and only if there is an edge between the two corresponding nodes in the other graph [20]. To detect the subset similarity of CIs, we were also more interested in similar sub-graphs that are not necessarily isomorphic. Sub-graph isomorphism is deciding if there is a sub-graph of one graph which is isomorphic to another graph [20]. Sub-graph isomorphism is a general form of maximal sub-graph detection problem.

A simple approach to identify maximal similar sub-graphs is considering all cycle free paths starting at two vertices from two graphs and doing a pair wise comparison afterwards [8]. This approach constructs maximal similar sub-graphs by induction from the starting vertices and by matching length limited similar paths. What makes this approach feasible is that it considers all possible matches at once.

According to four types of similarity and also for reducing the size of configuration memory as well, it is divided to four parts to support partial programmability. Paying attention to the RFU architecture, each CI configuration needs 512 bits; 155 bits are used for selecting functions (FUs) and routing resources (selectors of MUXes), 92 bits for selecting inputs, 61 bits for selecting outputs and finally, 204 bits for keeping immediate inputs. Configuration bits of four parts of each CI are generated and indexed by an index number. The configuration of a CI is determined according to the four index numbers stored in an index table. This table has four columns with the number of entries equal to the maximum number of generated CIs (more information can be found in [13]).

In the first stage of our similarity detection algorithm, only the similarity of nodes and connections are considered. As a matter of fact, by considering all types of similarities altogether, only a small amount of similar CIs can be resulted. For detecting the similarity of two CIs according to their FUs and routing resources, a graph isomorphism algorithm from [8] was used. The first CI is replaced by the second one, if the two CIs have *Complete* or *Subset Similarity*. In this stage, for each application, the size of the first part of configuration memory can be reduced due to reduction in the number of CIs.

Regarding the amount of bits needed to maintain CI inputs and outputs configurations and also immediate input values, reducing the size of these parts can lead to a considerable reduction in the overall size of configuration memory. Therefore, in the second stage, another similarity detection algorithm is used for exploring similarity of two CIs based on the similarity of inputs, outputs and immediate operands, separately. Similarity of two CIs according to their inputs means that they have a similar set of inputs. In other words, two CIs are examined to have a one to one correspondence between their inputs. CIs similarity based on outputs and also immediate operands are detected in a similar manner. Figure 2.b shows that similarity detection process can be done after generating mappable CIs to reduce the final number of CIs and hence, the size of configuration memory.

5.2. Merging

According to our observations, the length of generated CIs varies between 5 and 59. CIs with the length of 4 or less are rejected in the initial CI generation stage. Therefore, some CIs which are successfully mapped on the RFU have a small size compared with the RFU size that has 16 FUs. This may result in a noticeable unused space in the RFU. Experiments show that the percentage of unused space of the RFU for twenty-two applications of Mibench [Mibench] is almost 63% without considering the similarity of CIs.

According to this observation, we attempted to reduce unused space of the RFU by merging small CIs to larger ones. We call this process merging. In fact, merging is conceptually opposite to temporal partitioning. Our *IntegFrame* uses a temporal partitioning based approach to split CIs to smaller ones. On the contrary, merging tries to merge small CIs to larger ones that can be mapped on the RFU successfully. Merging of CIs not only decreases the unused spaces of RFU but also reduces configuration memory size because of reduction in total number of CIs.

For merging, a simple approach is presented which finds mergeable CI pairs for each application. This process is done in two steps. In the first step, for every CI pair, a combined CI is formed and is checked to satisfy RFU primary constraints. In the second stage, these CIs would be replaced by the combined one if it could be mapped successfully on the RFU. Figure 5 shows two CIs, each of which has been successfully mapped on the RFU. These CIs are merged and form a larger CI which in turn is mappable on the RFU. Figure 5 also shows the mapping of the resulted CI on the RFU.

The merging procedure can be exploited in two cases; a) independently without using the similarity detection process or b) after the similarity detection phase (Figure 2.b). Experiments show more reduction in the size of configuration memory by using both the similarity detection and the merging processes. The similarity detection process is performed before the merging process because merging of CIs may destroy the similarity of primary CIs, while considerable number of similar CIs can be detected before their merging.

6. Experimental Results

Simplescalar tool set (PISA configuration) [17] and 22 applications of Mibench [11] were used in the experiments. The base line processor of AMBER was MIPS324K with a five stage pipeline, 32KB L1 data cache (1 clock cycle latency); 32KB L1 instruction cache (one clock cycle latency) and 1MB unified L2 cache (6 clock cycle latency). The RFU was implemented using Synopsys tools with Hitachi 0.18μm library. The RFU area size was 1.15mm^{2}. It was assumed that the RFU has a variable latency based on the length of the longest critical path. Regarding the base processor frequency (166MHz) and RFU delay, CIs with critical path length less than or equal to 5 take one clock cycle and CIs including critical path length more than 5 take 2 clock cycles for execution on the RFU [13]. For example, for the corresponding CI of the first partition in Figure 4, due to the critical path's length that is equal to six, it takes two clock cycles to execute on the RFU.

Initial CIs were generated using the method proposed in [13]. CI rejection rate with respect to RFU architectural constraints was around 10%. 0 shows the minimum and maximum length of initial CIs. It also shows the minimum length of rejected CIs which were applied to the *IntegFrame*. Application names with rejected CIs have been shown in bold face. Figure 6 depicts that in 9 of the 22 applications, there was not any rejected CI. This means that all CIs in these applications were mapped on the RFU successfully. However, 13 of the 22 applications included rejected CIs and some of them like *blowfish* and *blowfish(dec)* included a remarkable rejection rate.

A couple of algorithms were introduced in Section 4 for recovering rejected CIs. First, *CIGen* was used to regenerate CIs with respect to RFU constraints. For these CIs, the mapping process was performed and some of them were rejected again at the mapping stage. In fact, this method generates the CIs considering only RFU primary constraints but it cannot consider the routing resource constraints before mapping. Figure 7 indicates that 10 out of 13 applications already included CIs which were not mappable on the RFU. These rejected CIs should be executed on the base processor and offer no more speedup.

In the second approach, the *IntegFrame* is utilized to partition the rejected CIs and generate appropriate ones. It is shown that the *IntegFrame* can result in higher speedup in comparison with the *CIGen* due to successful mapping of all CIs on the RFU. This is one of the most important achievements of the proposed framework.

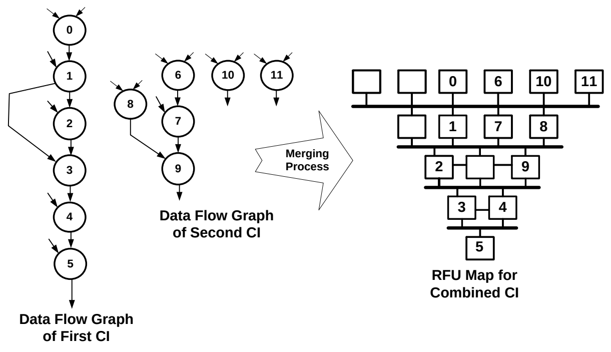


Fig. 5. An instance of merging of two mergeable CIs.

Table 1
CIs length for Mibench applications

Application Name	Min. CI length	Max. CI length	Min. Rejected CI length
cm(enc)	5	7	-
adpcm(dec)	5	7	-
bitcounts	**4**	**20**	**20**
blowfish	**5**	**16**	**15**
blowfish (dec)	**5**	**16**	**15**
basicmath	3	11	-
cjpeg	**5**	**59**	**11**
crc	5	5	-
dijkstra	4	9	-
djpeg	**4**	**48**	**8**
fft	**3**	**16**	**16**
fft (inv)	**3**	**16**	**16**
gsm (dec)	**5**	**14**	**14**
gsm (enc)	**4**	**26**	**13**
lame	**3**	**13**	**7**
patricia	3	6	-
qsort	5	7	-
rijndael (enc)	**5**	**16**	**10**
rijndael (dec)	**5**	**18**	**10**
sha	**5**	**18**	**7**
stringsearch	5	9	-
susan	6	10	-

Two algorithms including *IntegFrame-HTTP* and *IntegFrame-VTTP* were compared with respect to initial and final number of partitions (CIs), critical path length of generated CIs and intermediate data size. Figure 8 shows that for *cjpeg*, *reijndael(dec)* and *reijndael(enc), HTTP* generated larger number of initial partitions. Comparing final number of CIs generated shows that in most cases, the two algorithms generated equal number of CIs except for *cjpeg* that *VTTP* generates more CIs.

Figure 9 compares the two algorithms with respect to intermediate data size. For 6 out of 13 applications, intermediate data size was smaller using *VTTP*. For the seven remaining applications, intermediate data size was the same. Another comparison was done with respect to critical path length. Figure 10 shows that *VTTP* generated CIs with critical path length equal to or more than *HTTP* because it traverses DFG nodes in depth, whereas *HTTP* traverses them horizontally.

Figure 11 depicts the comparison of speedup achieved using *HTTP*, *VTTP* and *CIGen* (speedup is obtained compared to the applications run-time on the base processor). Using both *HTTP* and *VTTP* algorithms, all CIs were mapped successfully on the RFU and the fraction of applications which were accelerated was the same for the two algorithms. However, in most cases, *HTTP* resulted in better speedup since it benefits more from parallelism in the instruction execution. In other words, critical path length was less using *HTTP*, and therefore, RFU execution latency was smaller. Moreover, according to Figure 11, both *HTTP* and *VTTP* offer better speedup compared to *CIGen*. *IntegFrame* can be run in the training mode and it has a small overhead time ($O(n^2)$) due to using incremental algorithms.

To examine the effect of the two proposed algorithms on reducing the size of configuration memory, including similarity detection and merging, additional experiments were conducted. 0 shows the number of similar CIs considering similarity of FUs and connections, inputs, outputs and immediate inputs. For example, for *blowfish(dec)*, there are 15 similar CIs considering the FUs and connections similarity, seven CIs with similar set of inputs, 14 CIs with similar outputs and 14 CIs with similar set of immediate input values. Therefore, total configuration bits needed to this application are $25 \times 155+(40-7)\times 92+(40-14) \times 61+ (40-14) \times 204=1.7KB$ in comparison with $40\times 512= 2.5KB$, which are obtained without using the similarity detection algorithm. Obviously, the size of configuration memory is determined according to the largest size of memory required to support an application.

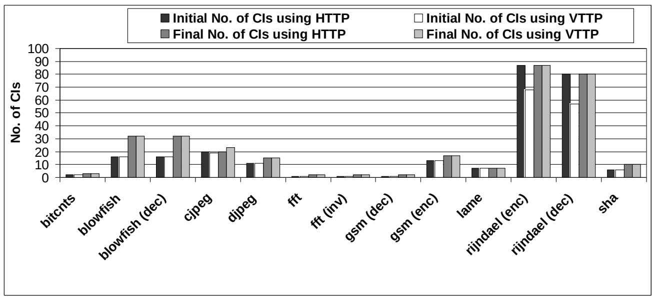


Fig. 8. Initial and final number of partitions using *HTTP* and *VTTP*.

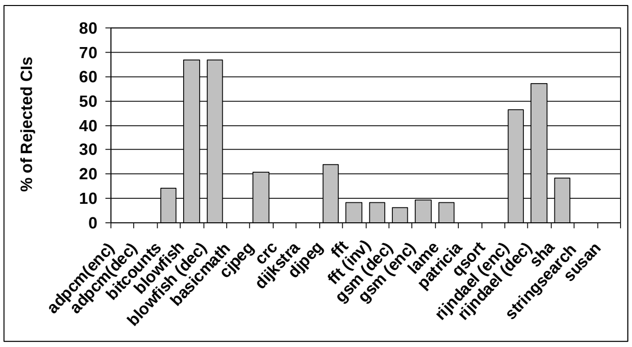


Fig. 6. CI rejection rate for Mibench applications.

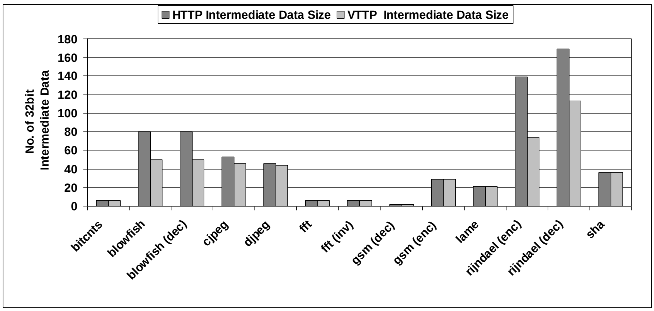


Fig. 9. Comparison of the intermediate data size.

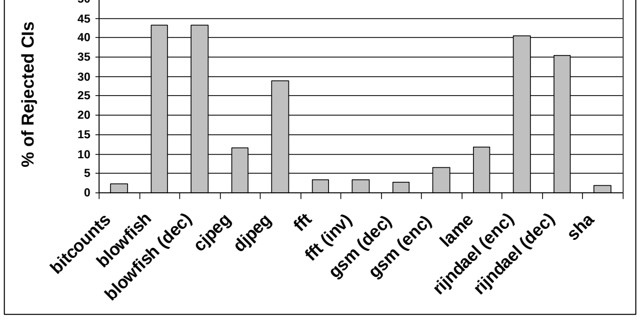


Fig. 7. Rejection rate for CIs generated by CIGen.

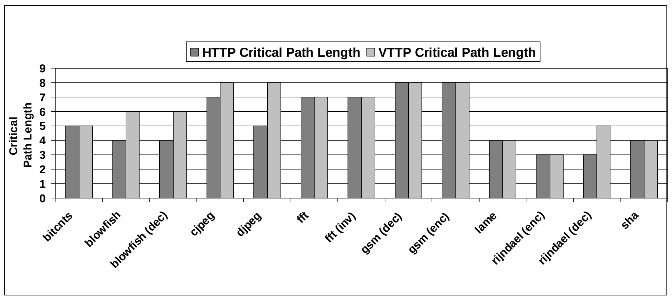


Fig. 10. CIs maximum critical path length for *HTTP* and *VTTP*.

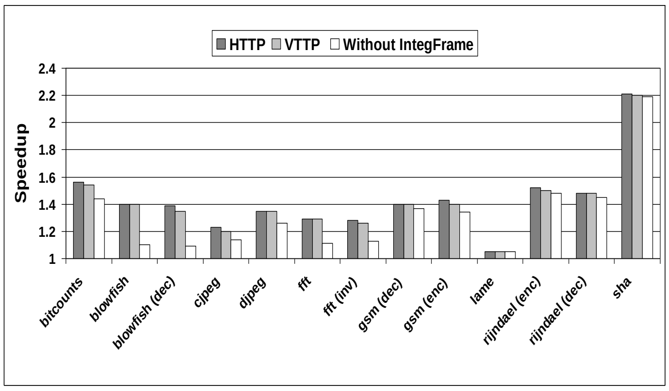


Fig. 11. Speedup comparison between *HTTP*, *VTTP* and *CIGen*.

Table 2
Results obtained using similarity detection and merging techniques

Applications	No. of CIs	No. of CIs after Similarity Detection	No. of Inp/Out/Immediate Similarities	No. of CIs after Similarity Detection & Merging
adpcm(enc)	4	3	0/0/0	2
adpcm(dec)	3	3	0/0/0	2
bitcounts	9	7	0/0/0	5
blowfish	40	25	6/14/14	22
blowfish (dec)	40	25	7/14/14	22
basicmath	10	6	0/0/0	3
cjpeg	41	22	0/5/3	16
crc	1	1	0/0/0	1
dijkstra	9	7	0/1/1	4
djpeg	31	18	0/5/3	12
fft	13	6	0/2/1	4
fft (inv)	13	6	0/2/1	4
gsm (dec)	17	8	0/1/0	5
gsm (enc)	84	24	21/18/2	17
lame	39	20	3/5/3	12
patricia	10	7	0/0/1	4
qsort	17	10	1/1/3	5
rijndael (enc)	117	14	0/9/32	8
rijndael (dec)	110	15	0/19/26	10
sha	32	9	1/4/4	6
stringsearch	5	4	0/1/1	2
susan	9	4	0/0/0	3

The maximum size of the configuration memory is up to 4.5KB which is obtained for *rijndael(enc)* using similarity detection algorithm, while the initial memory size is 7.4KB without using the reduction techniques. In other words, similarity detection process reduces the configuration memory size by 38.5%. In addition, using merging process after applying the similarity detection algorithm reduces configuration memory size to 4.4KB, which indicates the overall 40% reduction. The final stage of experiments was performed to observe the particular effect of the merging process on the configuration memory size. Experiments show at least 7% and at most 50% reduction in the number of CIs. Therefore, using only merging process can substantially reduce the configuration memory size.

7. Conclusion

This paper addressed handling of CI generation concerns for a reconfigurable instruction set processor. Some of the CIs which are extracted from hot basic blocks of applications were rejected due to RFU primary constraints. Two approaches were presented to support the rejected CIs. The first approach (*CIGen*) generates CIs by applying the RFU constraints to the CI generation tool. It may still cause rejection of some generated CIs. The *IntegFrame* is the second approach introduced to perform CI generation task. This framework can be used as a general approach for generating CIs as well. It uses mapping-aware temporal partitioning algorithms for generating CIs. *HTTP* is a temporal partitioning algorithm that takes advantage of parallel instruction execution on an RFU. *VTTP* is an alternative which reduces intermediate data size. In the *IntegFrame,* CI modification is done using incremental versions of *HTTP* and *VTTP* algorithms. This framework successfully mapped all CIs on the RFU. Therefore, the *IntegFrame* brought about more speedup compared to *CIGen* in both cases of using *HTTP* and *VTTP*.

This paper also addressed the configuration memory size reduction, which can decrease area, cost and power consumption of hardware augmenting to the base processor. Two techniques comprising similarity detection and merging were presented in this paper to reduce the size of configuration memory. The first method attempted to detect similarity of CIs based on four different types of similarity. The latter one used an algorithm to merge CI pairs and replace them with a combined CI. These two approaches resulted in fewer CIs as well as smaller configuration memory.

Acknowledgment

This research was supported in part by Core Research for Evolutional Science and Technology (CREST) of Japan Science and Technology Corporation (JST).

References

[1] M. Arnold and H. Corporaal, Designing domain-specific processors, In Proc. of the Design, Automation and Test in Europe Conf, 61-66, 2002.

[2] K. Atasu, L. Pozzi and P. Lenne, Automatic application-specific instruction-set extensions under microarchitectural constraints, In Proc. of the Design, Automation and Test in Europe (DATE), 256-261, 2003.

[3] F. Barat, R. Lauwereins and G. Deconinck, Reconfigurable instruction set processors from a hardware/software perspective, IEEE Trans. on Software Engineering, vol. 28, no. 9, 847-861, 2002.

[4] C.Bobda, Synthesis of Dataflow Graphs for Reconfigurable Systems Using Temporal Partitioning and Temporal Placement, Ph.D thesis, University of Paderborn, 2003.

[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 Proc. of IEEE/ACM Int. Symp. on Microarchitecture, 30-40, 2004.

[6] M. Karthikeya, P. Gajjala and B. Dinesh, Temporal partitioning and scheduling data flow graphs for reconfigurable computer, IEEE Trans. on Computers, vol. 48, no. 6, 579-590, 1999.

[7] R. Kastner, A. Kaplan, S. Ogrenci Memik and E. Bozorgzadeh, Instruction generation for hybrid reconfigurable systems, ACM TODAES, vol. 7, no. 4, 605-627, 2002.

[8] J. Krinke, Identifying Similar Code with Program Dependence Graphs, In Proc. 8th Working Conf. on Reverse Engineering, 301-309, 2001.

[9] F. Mehdipour, H. Noori, M. Saheb Zamani, K. Murakami, M. Sedighi and K. Inoue, An integrated temporal partitioning and mapping framework for handling custom instructions on a reconfigurable functional unit, The 11th Asia-Pacific Computer Systems Architecture Conf. (ACSAC'06), Lecture Notes in Computer Science, vol. 4186/2006, 219-230, 2006.

[10] F. Mehdipour, M. Saheb Zamani and M. Sedighi, An integrated temporal partitioning and physical design framework for static compilation of reconfigurable computing systems, Microprocessors and Microsystems, vol. 30, no. 1, 52-62, 2006.

[11] Mibench. http://www.eecs.umich.edu/mibench.

[12] G.D. Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.

[13] H. Noori, F. Mehdipour, K. Murakami, K. Inoue and M. Saheb Zamani, An architecture framework for an adaptive extensible processor, The Journal of Supercomputing, Springer Netherlands, vol. 45, no. 3, 313-340, 2008.

[14] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul and R. Vemuri, An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures, In Proc. of the Reconfigurable Architecture Workshop, 31-36, 1998.

[15] R. Razdan and M.D. Smith, A high-performance microarchitecture with hardware-programmable functional units, In Proc. of the 27th Annual Int. Symp. on Microarchitecture, 172-180, 1994.

[16] N. Sherwani, Algorithms for VLSI Physical Design Automation, Kluwer-Academic Publishers, 1991.

[17] Simplescalar. http://www.simplescalar.com.

[18] J. Spillane and H. Owen, Temporal partitioning for partially reconfigurable field programmable gate arrays, IPPS/SPDP Workshops, 37-42, 1998.

[19] C. Tanougast, Y. Berviller, P. Brunet, S. Weber and H. Rabah, Temporal partitioning methodology optimizing FPGA resources for dynamically reconfigurable embedded real-time system, Microprocessors and Microsystems, vol. 27, 115-130, 2003.

[20] W.Weisstein,Graphisomorphism,http://mathworld.wolfram.com/GraphIsomorphism.html.

[21] Z.A.Ye et al, Chimaera: A high-performance architecture with tightly-coupled reconfigurable functional unit, In Proc. of the 27th Annual Int. Symp. on Computer Architecture, 225-235, 2000.