Improving Performance and Energy Saving in a Reconfigurable Processor via Accelerating Control Data Flow Graphs

Mehdipour, Farhad
Computing and Communication Center, Kyushu University

Noori, Hamid
Department of Informatics, Kyushu University

Zamani, Morteza Saheb
Department of IT and Computer Engineering, Amirkabir University of Technology

Inoue, Koji
Department of Informatics, Kyushu University

他

出版情報：IEICE Transactions on Information and Systems. E90-D (12), pp.1956-1966, 2007-12-01. IEICE
バージョン：
権利関係：

PAPER *Special Section on Reconfigurable Systems*

Improving Performance and Energy Saving in a Reconfigurable Processor via Accelerating Control Data Flow Graphs

Farhad Mehdipour[†a)], Hamid Noori[††], Morteza Saheb Zamani[†††], *Nonmembers,* Koji Inoue[††] and Kazuaki Murakami[††], *Members*

Summary Extracting frequently executed (hot) portions of the application and executing their corresponding data flow graph (DFG) on the hardware accelerator brings about more speedup and energy saving for embedded systems comprising a base processor integrated with a tightly coupled accelerator. Extending DFGs to support control instructions and using Control DFGs (CDFGs) instead of DFGs results in more coverage of application code portion are being accelerated hence, more speedup and energy saving. In this paper, motivations for extending DFGs to CDFGs and handling control instructions are introduced. In addition, basic requirements for an accelerator with conditional execution support are proposed. Then, two algorithms are presented for temporal partitioning of CDFGs considering the target accelerator architectural constraints. To demonstrate effectiveness of the proposed ideas, they are applied to the accelerator of a reconfigurable processor called AMBER. Experimental results approve the remarkable effectiveness of covering control instructions and using CDFGs versus DFGs in the aspects of performance and energy reduction.

Key words: *Reconfigurable accelerator, Conditional execution, Control data flow graph, Temporal partitioning, reconfigurable processor.*

1. Introduction

Using an accelerator for executing critical or hot (most frequently executed) portions of applications is an effective technique to enhance the performance and energy saving of processors in embedded systems. In this technique, data flow graphs (DFGs) extracted from critical portions of an application are executed on an accelerator and remaining portions on the base processor, correspondingly. By executing hot portions on an accelerator performance improvement is obtained through exploiting potential parallelism and reducing the latency of critical paths and the number of intermediate results read/written to the register file. The accelerator can be implemented as a reconfigurable hardware with fine or coarse granularity or as a custom hardware (such as Application Specific Instruction-set Processors or Extensible Processors) [8]. Finer granularity brings about more flexibility compared to coarse one; however it suffers from long latency and high reconfiguration overhead time compared to coarse grained accelerators. Moreover, it needs large amount of memory for configuration bits. The custom hardware accelerators are faster and consume less energy compared to reconfigurable accelerators; however they need a long and costly design and manufacturing process.

The integration of accelerator and the processor can be tightly [3][4][11][21][27] or loosely coupled [8][15]. For loosely-coupled systems, there is an overhead for transferring data between base processor and accelerator. In a tightly coupled accelerator, data is read and written directly to and from the processor's register file, making the accelerator an additional functional unit in the processor pipeline. This makes the control logic simple, as almost no overhead is required in transferring data to the programmable hardware unit; however, it increases the read/write ports of the register file. This paper focuses especially on tightly coupled reconfigurable accelerator.

DFG extraction can be done at high level or binary level of the source code. In our analysis, we concentrate on the latter one which means that the DFG nodes are the primitive instructions of the base processor and each functional unit in accelerator implements the instruction level operations. Using binary level brings about more transparency, more accurate software estimation and similar speedup to source-level for numerous applications while keeping the binary compatibility [24]. The Control DFG (CDFG) is a DFG containing control instructions (e.g. branch instruction). For a branch instruction, two succeeding paths can be considered. If the branch is taken (branch result is true), a sequence starting from branch target address called taken path is executed. Otherwise, not-taken path including a sequence of instructions starting from the next address to the branch is executed. Therefore, depending to the result of a branch instruction one of the instruction sequences from taken or not taken paths are executed. Handling branches (conditional execution) is a challenge in CDFG acceleration. It means executing CDFG instead of DFG on accelerator which includes conditional execution facilities due to existing control instructions in CDFG. We consider two types of CDFGs:

a) CDFGs including at most one branch instruction as its last instruction. In this case, the accelerator does not need to support conditional execution.

[†] The author is with the Computing and Communication Center, Kyushu University, Fukuoka, Japan.
[††] The authors are with the Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka, Japan.
[†††] The author is with the Department of IT and Computer Engineering, Amirkabir University of Technology, Iran.
a) Email: farhad@c.csce.kyushu-u.ac.jp

b) CDFGs containing more than one branch instructions. This case necessitates featuring conditional execution in accelerator for executing CDFGs.

In CDFG generation process one can only follow the frequently executed (hot) directions of branches. For each branch, one of its taken or not-taken paths or both of them might be hot. We suggest adding hot directions of branches into the CDFG without being limited to selecting just one or all of the directions. This can hide branch misprediction penalty.

One main intuition behind this work is twofold regarding two following questions:

a) Does executing CDFGs instead of DFGs on accelerator (which means acceleration based on CDFGs vs. DFGs) obtain higher performance?

b) How can the conditional execution be supported on an accelerator, if the answer to the first question is Yes?

To answer the first question, effect of extending DFGs and covering control instructions on the speedup is studied and reasonable arguments for extending DFGs and using CDFGs instead of DFGs are presented. Moreover, as an answer to the second question, basic requirements for an accelerator with conditional execution support are studied.

Another contribution of this paper is our investigation on CDFG generation considering limitations of resources in the accelerator (e.g. number of inputs, outputs, logics, connections and etc). CDFGs extracted from various applications have different sizes and in most cases the whole CDFG can not be mapped on the accelerator. Therefore, algorithms for partitioning CDFGs under the accelerator resource constraints are studied. We present a couple of CDFG temporal partitioning algorithms to partition large CDFGs to smaller and mappable ones. Mappable CDFGs satisfy the accelerator architectural constraints; hence, can be mapped and executed on the accelerator. Moreover, our proposed ideas are applied to a reconfigurable processor referred as AMBER and its accelerator is extended to support conditional execution. Then, effects of using CDFGs on speedup and energy consumption are evaluated. Finally, the extended version of AMBER is compared with other approaches to show the effectiveness of our proposed ideas.

The paper outline is as follows. Section 2 introduces a reconfigurable processor referred as AMBER. In Section 3, motivations for extending DFGs on control instructions and using CDFGs are illustrated. General requirements for an accelerator architecture featuring conditional execution are proposed in Section 4. Section 5, describes the algorithms proposed for CDFG temporal partitioning which are used for generating appropriate CDFGs for executing on the accelerator. Also, comparison of the algorithms and a performance evaluation approach are presented in Section 5. Section 6, after a review on related work, explains the results of applying the proposed ideas to a reconfigurable processor called AMBER and he experimental results obtained. Finally, the paper is concluded in Section 7.

2. General Overview of AMBER

AMBER is a reconfigurable processor [17] targeted for embedded systems. It has been developed by integrating a base processor with two other main components[17]. The base processor is a general RISC processor and the other two components are: sequencer and a coarse grain reconfigurable functional unit (RFU). Fig. 1.a illustrates the integration of different components in AMBER.

The *base processor* is a 4-issue in-order RISC processor supporting MIPS instruction set. The *sequencer* mainly determines the microcode execution sequence by selecting between the RFU and the processor functional unit. The *RFU* is based on array of 16 functional units (FUs) with 8 input and 6 output ports. It is used in parallel with other processor's ALUs (Fig. 1.b). RFU reads (write) from (to) register file. Each FU can support all fixed-point instructions of the base processor except *multiply, divide* and *load*. In the RFU, the output of each FU in a row can be used by all FUs in the subsequent row.

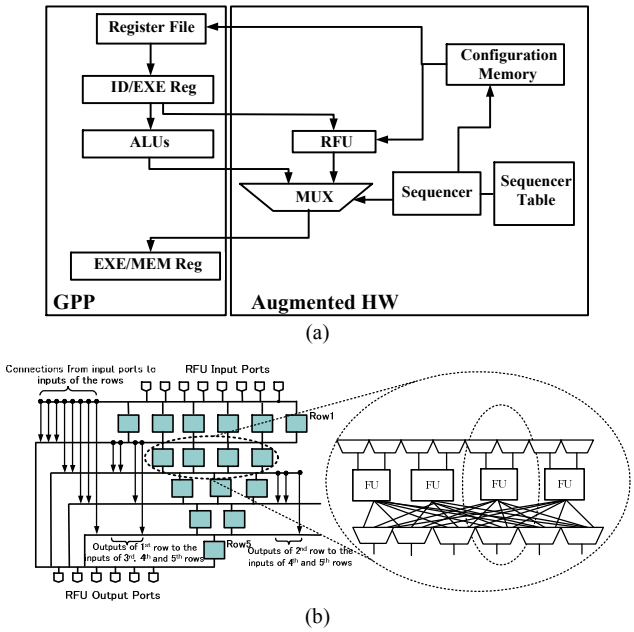


Fig. 1. Main components in AMBER (a) RFU architecture (b)

AMBER has two operational modes: the *training* and the *normal mode*. The training mode is done offline. In this phase, target applications are run on an instruction set simulator (ISS) and profiled. Then, the start addresses of hot basic blocks (HBBs) are detected. HBB is a basic

block that is executed more times than a given threshold and a basic block is a sequence of instructions that terminates in a control instruction. After generating configuration bit-streams for extracted DFGs and initiating sequencer tables in training mode it switches to the normal mode. DFGs are executed on the RFU in the normal mode utilizing the RFU, its configuration data (stored in the configuration memory) and sequencer. More details on AMBER can be found in [17].

3. Motivations

We follow a quantitative analysis to explore the motivations for extending DFGs over control instructions and generating CDFGs to be executed on an accelerator. We use some applications of Mibench [16] for the analysis. As mentioned formerly, DFGs are extracted from the frequently executed portions of application and a control instruction (e.g. branch instruction) may terminate DFG generation process. Therefore, control instructions located in a short distance result in generation of small size DFGs (SSDFG). In fact, SSDFGs are not suitable for improving performance in application execution and have to be run on the base processor. Authors showed in [13] that the small length DFGs (including less than or equal to five instructions) offer no more speedup.

In Fig. 2, a piece of a main loop of *adpcm(enc)* is shown. *adpcm(enc)* is an application program containing a loop which consumes 98% of total execution time. The critical portion of application contains 12 branch instructions. According the location of branch instructions, four DFGs can be extracted from the piece of loop that has been shown in Fig 1. In this figure, three out of four DFGs are SSDFGs. These SSDFGs do not gain more speedup and have to be run on the base processor.

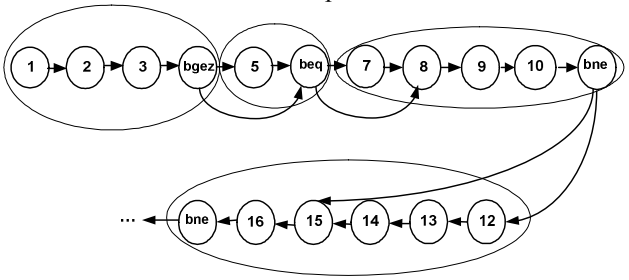


Fig. 2. Control data flow graph of hot portion of *adpcmc(enc)*

This kind of analysis was accomplished for 17 applications of Mibench [16]. Fig. 3 shows the overall percentage of frequently executed (hot) portion of each application. In addition, this figure shows the fraction of applications that could not be accelerated because of SSDFGs. For example, for *bitcount* application, almost 92% of application is hot. On the other hand, 32% out of 92% of hot portions do not worth to be accelerated due to the SSDFGs; therefore, they are dismissed from execution on the accelerator. However, analyses show for some applications like *fft*, *fft(inv)* and *sha* which includes few branch instructions, supporting conditional execution no considerable speedup is achievable, because the small portion of generated DFGs are removed due to SSDFGs.

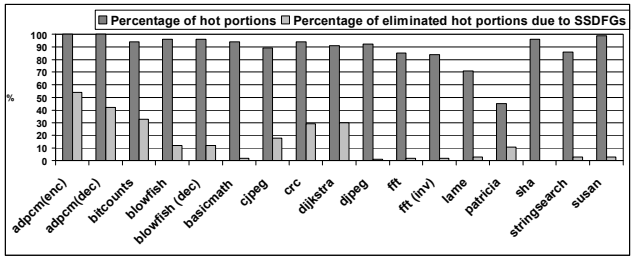


Fig. 3. Fraction of hot portions and eliminated hot portions in some applications of Mibench [16]

Extending DFGs to contain more than one branch instruction and generating the CDFGs vs. DFGs is one solution to amortize the number of generated SSDFGs. As mentioned before, according to the result of a branch instruction, one of the instructions sequence located in taken or not-taken paths of the associated branch might be executed. In some case, only one of these paths is frequently executed (hot) and in some other cases both paths are hot. In latter case, covering both directions can aid the generation of larger CDFGs, hence more parallelism, as well as eliminating branch misprediction penalties. On the other hand, for supporting CDFGs architecture of the accelerator should be able to support conditional execution. In addition, appropriate algorithms are required to generate CDFGs considering the specifications of the accelerator.

4. Basic Requirements for Architecture Featuring Conditional Execution Support

As it was mentioned in Section 3, conditional execution support in accelerator is needed for CDFG execution. For this purpose, the capability of branch instruction execution should be added to the accelerator. The target accelerator is assumed to be a coarse grained reconfigurable hardware which is a matrix of functional units (FUs) with specified connections. CDFG nodes are the base processor instructions, since we assumed that our concentration is on binary level of the applications. Therefore, each FU like the processor's ALUs can execute instruction level operations.

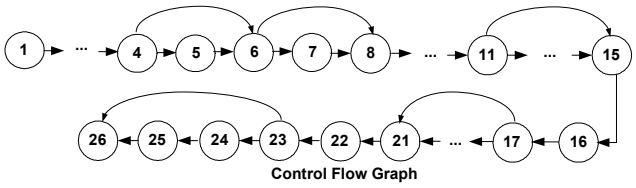
In a DFG, the nodes (instructions) receive their input from a single source whereas, in the CDFG, nodes can have multiple sources with respect to the different paths

generated by branches. The correct source is selected at run time according to the results of branches.

Fig. 4 shows a piece of *adpcm(enc)*'s critical portion, a part of its corresponding DFG (Fig. 4.a) and the CFG comprising only control flow of instructions (Fig. 4.b). In Fig. 4.a, each node of DFG corresponds to one instruction in the code. Inside circles, instruction number and instruction itself has been depicted. Each instruction has at most two sources and one destination. According to the results of branch instructions various values for an instruction source could be obtained. For example, 13th instruction (13:subu) receives its first source (register R3) from 3rd (3:subu) and 7th (7:subu) instructions. Its output may be routed to instructions (16:slt) and (19:subu) or (22:slt) depending on the result of branch instruction (17:bne). As another example, instruction (22:slt) may receive its first source (R3) through the instructions (3:subu), (7:subu) or (19:subu) depending on the result of branch instructions (4:bgez), (6:beq), (11:bne) and (17:bne). Also, it receives the second source (R9) from (21:sra). Consequently, the nodes that generate output data of a CDFG are altered according to the results of branches as well. Therefore, the accelerator should have some facilities to generate valid output data.

inst. #	*address*	*inst.*	*operands (dest, src1, src2)*
1	400418	addu	R13 R0 R0
2	400420	addiu	R4 R4 2
3	**400428**	**subu**	**R3 R2 R11**
4	400430	bgez	400440 R3
5	400438	addiu	R13 R0 8
6	400440	beq	400450 R13
7	**400448**	**subu**	**R3 R0 R3**
8	400450	addu	R10 R0 R0
9	400458	sra	R8 R9 0x3
10	400460	slt	R2 R3 R9
11	400468	bne	400488 R2
12	400470	addiu	R10 R0 4
13	**400478**	**subu**	**R3 R3 R9**
14	400480	addu	R8 R8 R9
15	**400488**	**sra**	**R9 R9 0x1**
16	**400490**	**slt**	**R2 R3 R9**
17	**400498**	**bne**	**4004b8 R2**
18	4004a0	ori	R10 R10 2
19	**4004a8**	**subu**	**R3 R3 R9**
20	4004b0	addu	R8 R8 R9
21	**4004b8**	**sra**	**R9 R9 0x1**
22	**4004c0**	**slt**	**R2 R3 R9**
23	**4004c8**	**bne**	**4004e0 R2**
24	**4004d0**	**ori**	**R10 R10 1**
25	**4004d8**	**addu**	**R8 R8 R9**

(a)



(b)

Fig. 4. A piece of *adpcm(enc)* code and a part of its corresponding DFG and (a) its control flow graph (b)

Predicated execution is one technique [19] which effectively removes control dependency of programs running on ILP (Instruction level parallelism) processors. Proposed architecture in [9] uses predicated instructions. With predicated execution, control dependency is essentially turned into data dependency using predicates. A predicated variable is a boolean variable which represents the control information of a control instruction. The following instructions become no-ops if the predicated variable is evaluated to be false. To support predicated execution on a microprocessor two sets of modifications are needed in the instruction set architecture of the processor. First, the original instructions need to be replaced with their predicated versions. A predicated instructions performs the same operations as its original (non-predicated version), but it does not change the processor state if the associated predicated variable is evaluated to be false at the time of its execution. Second, predicate defining instructions need to be added to set the predicate variables to appropriate boolean variables for the following instructions to behave correctly.

The architecture with predicated execution features should have radical changes, since every instruction can be predicated and a separated predicated register file is needed. Partial architectural support has also been studied [12] to solve this issue. In [12], Mahlke et al. proposed architecture with two new instructions added to the original instruction set to support predicated execution. Instead of making all instructions predicated, only two instructions are defined to perform depending on the predicated while the others remain the same. In this architecture every predicated instruction can be rewritten (without altering the behavior at the end of instructions execution) using non-predicated instructions and the two newly introduced ones. In other words, the behavior of a predicated instruction is evaluated and its destination is transferred to a temporary register and then the result is conditionally copied to the destination.

In this section, we propose basic requirements of an architecture which can support conditional execution. In the general architecture with conditional execution features, following characteristics are found:

a) An FU in the accelerator can receive its inputs directly from accelerator primary inputs or from output of the other FUs.

b) According to the condition of branch instructions, output of each node can be directed to the other nodes from different paths. For example, in Fig. 4.b, output of instruction (13:subu) can be routed to nodes (16:slt), (19:subu) and (22:slt). It means instruction (19:subu) receives the value of R3 (output of instruction 13) if branch instruction (17:bne) is not-taken, otherwise R3 is obtained by instruction (22:slt). Therefore, there may be several outputs for a CDFG and some of them may be valid as accelerator's final outputs.

According to aforementioned properties, the accelerator architecture must have these following inevitable requirements:

a) Capability of selective receiving of inputs from both accelerator primary inputs and output of other instructions (FUs) for each node.

b) Possibility of selecting the valid outputs from several outputs generated by accelerator according to conditions made by branch instructions.

c) Accelerator should be equipped by control path besides to data path which provides the correct selection of inputs and outputs for each FU and entire accelerator.

We will give more details on the architecture designated for a reconfigurable processor in Section 6.

5. Algorithms for CDFG Temporal Partitioning

Extending DFGs to cover hot directions of branch instructions indeed, results in large CDFGs which may not satisfy the accelerator resource constraints. In other words, CDFG extracted from various applications have different sizes and some times the whole CDFG can not be mapped on the accelerator due to the resource limitations of the accelerator (e.g. number of inputs, outputs, logics and specifically routing resource constraints). Using temporal partitioning algorithms which consider the accelerator constraints is a solution to this issue. Temporal partitioning can be stated as partitioning a DFG/CDFG 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 [1][7][14]. A temporal partitioning algorithm can consider the accelerator architectural specifications to generate executable DFGs on the accelerator. Even if the logic resource limitations are considered, some of them like the routing resource constraints are not applicable in DFG generation phase. Satisfying or violating routing resource constraints can be specified after trying to map a DFG on the accelerator.

Integrated Framework presented in [13] (based on design flow proposed in [14]) performs an integrated temporal partitioning and mapping process to generate mappable DFGs. It takes *rejected* DFGs and attempts to partition them to appropriate ones with the capability of being mapped on the accelerator. The DFGs which are called *rejected* (vs. *mappable*) are ones that can not be mapped on the accelerator due to resource constraints [13]. Moreover, the partitions obtained from the integrated temporal partitioning process are the same appropriate DFGs which are *mappable* on the accelerator.

In the first stage of *Integrated Framework*, temporal partitioning algorithm generates initial partitions applying accelerator primary constraints (e.g. number of FUs, inputs, outputs). Then in the second stage, for each partition generated in the first step, the mapping of DFG nodes on the accelerator's FUs is done. The mapping tool attempts to reduce total connection length between the nodes and satisfy the accelerator architectural constraints simultaneously. These DFGs are accepted and finalized if they can be successfully mapped and routed on the accelerator.

The routing process is unsuccessful if there are r routing resources between ith and jth row while demanding more than r resources between these rows. In other words, the routing process fails if one or more connections in DFG could not be routed due to limitation of the routing resources. In case of unsuccessful routing, *Integrated Framework* utilizes an iterative process to change the partitions incrementally and repeats upon performing a successful mapping and meeting routing constraints. During the incremental temporal partitioning algorithm each partition is modified by moving some of the nodes to the subsequent partition (for more details refer to [13]).

We modified the *Integrated Framework* introduced in [13] by replacing DFG temporal portioning algorithm with a temporal partitioning algorithm applicable to CDFGs. The modified *Integrated Framework* partitions large CDFGs and generates mappable and executable CDFGs on the accelerator. Indeed, temporal partitioning algorithm has a key role in *Integrated Framework.* As the authors knowledge there are small number of algorithms for CDFG partitioning, though a lot of works have been done around the DFG temporal partitioning [1][7][14]. In [1] a temporal portioning algorithm has been presented that partitions a CDFG considering target hardware with non-homogenous architecture. In this approach, each node of CDFG may have several implementation models on hardware and software. Setting control signal values determines a specific path of the data and converts a CDFG to sub-graphs that do not include control instructions. This algorithm tries to consider all states of the control instructions in application to convert corresponding CDFG to a set of DFGs. Then it tries to minimize the number of states to reduce the number of generated DFGs. For each DFG a temporal partitioning algorithm is used for partitioning. One of the important disadvantages of this algorithm is that the large number of DFGs may be obtained during CDFG to DFG conversion. In addition, an exact knowledge to different states in application is required to reduce the number of DFGs.

In this section, a couple of algorithms are introduced for CDFG temporal partitioning. The main goal is generating the minimum number of partitions to reduce the reconfiguration overhead time as well as configuration memory size. The proposed algorithms may be used as general CDFG temporal partitioning algorithms and also

can be superimposed on the modified *Integrated Framework*. First, some definitions are presented:

- ***Terminator instruction*:** An instruction which changes execution direction of the program including procedure or function call instructions and also backward branch and return (to prevent making cycles in CDFG).

- ***Critical instruction*:** An instruction is *critical* or hot if its execution frequency is more than the predefined frequency *threshold*. In fact, execution frequency of instructions is achievable through profiling phase. One method for profiling is running the application on an instruction set simulator (ISS) and gathering required information like the approaches used in [17][18].

- ***Frequency threshold*:** defines a boundary to identify whether an instruction is *critical* or not.

5.1. TP Based on Not-Taken Paths (NTPT)

This algorithm as our first CDFG temporal partitioning algorithm adds instructions from not-taken path of a control instruction to a partition until violating the target hardware architectural constraints (e.g. number of logic resources, inputs and outputs) or reaching to a *terminator* control instruction. In fact, a *terminator* instruction is an exit point for a CDFG. Therefore, in our methodology a CDFG can include one or more exit-points according the different paths achieved based on control instructions conditions. The concept of non-atomic multi-exit CDFGs has been introduced in [18]. Generating a new partition is started with branch instructions which at least one of their taken or not-taken instructions has not been located in the current partition. The pseudo code for NTPT algorithm is as follows:

Not Taken Path Traversing Temporal Partitioning Algorithm:

1. Create a new empty partition and add initial instruction to the current partition
2. If the current partition does not satisfy the accelerator primary architectural constraints, remove the last instruction added to the partition and close current partition and go to step 3, otherwise if the last instructions is not a terminator then consider its next instruction from the not-taken path, add it to the current partition and repeat step 2.
3. Create *StartNodeList* as an empty list.
4. If the last instruction of the current partition is a branch instruction, add it to *StartNodeList.*
5. For each branch instruction in current partition, if it's taken or not-taken instructions are not in current partition, add it to *StartNodeList.*
6. Repeat steps 1 to 5 for each of instructions in *StartNodeList* as the initial instruction of new partition.

Fig. 5 exemplifies how this algorithm works for a piece of a CDFG. If the first partition generation stops in instruction 14 due to resource limitation of the accelerator, then, second partition is started from instruction 11. Because, for branch instructions located in nodes 4 and 6, both taken and not-taken paths has been inserted in the first partition, but for instruction 11, only its not-taken path are located in the first partition. Therefore, it is inserted in *StartNodeList* and used as an initial instruction of the next partition. The time complexity of this algorithm is $O(n^2)$ where, n is the number of nodes in CDFG.

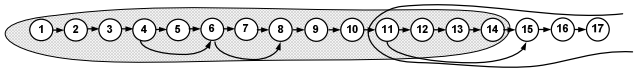


Fig. 5. Applying NTPT algorithm on a sample CDFG

5.2. Execution Frequency-Based Algorithms

In NTPT algorithm, instructions were selected only from not-taken paths of branches regardless of their criticalities, whereas, execution frequency of taken and not-taken instructions may be different. Here, another temporal partitioning algorithm is proposed with the aim of taking into account the execution frequency of taken and not-taken instructions. Execution frequency of instructions is an effective factor for selecting the instructions being added to the current partition. In other words, selecting the next instruction to the branch is done according to the frequency of branch succeeding instructions. For a branch instruction according to execution frequency of its succeeding taken or not-taken instructions one of them or both can be *critical*.

In our frequency-based temporal partitioning algorithm, instructions are added one after another until observing a terminator or a branch instruction. For each instruction, list of all instructions located on its taken and not-taken paths stopping at a *terminator* are created. All instructions of two lists are added to the current partition if enough space is available. Otherwise, the list with higher execution frequency is selected and the other list is used to create a new partition. If two lists are terminating in a unique instruction, it is both added to the current partition, so, it necessitates reconfiguration during execution of the instructions located in the current partition. The time complexity of this algorithm is $O(n^3)$ where, n is the number of nodes in CDFG. Fig. 6 clarifies this partitioning technique. According to partitions produced and shown in the figure, both taken and not-taken parts of branch instructions 4 and 6 are in the first partition. On the other hand, for instruction 11, only taken instructions are located in the same partition. Therefore, in its execution a reconfiguration for loading the next partition is needed if the branch is not-taken.

Frequency Based Temporal Partitioning Algorithm:

1. Create an empty partition and add initial instruction

to the current partition while the architectural constraints are satisfying, or the instruction is not a *terminator*.
2. if the last instruction in current partition is branch, create two lists for its succeeding taken and not-taken paths (these lists should be stopped at a *terminator*).
3. if both lists are *critical* and can be added to the current partition and are terminating in a unique branch instruction, add both to the current partition, otherwise add one of them that is more *critical* (has more execution frequency).
4. For the list which its instructions have not been added to the current partition, create a new partition and insert its instructions in the new partition.
5. Repeat steps 1 to 4 starting the instructions are next to the last nodes of the recently generated partitions as the initial instructions.

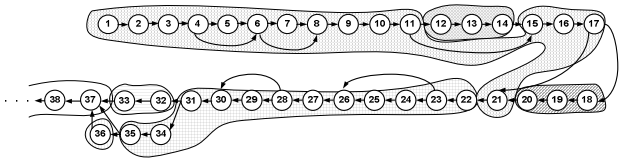


Fig. 6. An example of frequency-based CDFG temporal partitioning

5.3. Evaluating Proposed Algorithms

The above algorithms were compared according to a) the number of generated partitions and b) efficiency factor. The former is a factor that determines the number of reconfigurations during run-time. The latter has been defined as a factor to show the efficacy of executing CDFGs on the accelerator. First, we introduce some definitions and then present equations for calculating the efficiency factor.

- **Branch Taken Part (BTP):** Set of instructions in a given CDFG that are started from the target address of a branch and is terminated by a *terminator*.
- **Branch Not-Taken Part (BNTP):** Set of instructions in a given CDFG that are started from the succeeding instruction of a branch and is terminated by a *terminator*.

We introduce Eq. 1 to calculate the efficiency factor:

$$EfficiencyFactor = \frac{CDFGCycleCountOnCPU.\tau_{CPU}}{\tau_{CPU}\sum_{i=1}^{N} f_{p_i}.CC(P_i) + \gamma_{CRFU}.\sum_{i=1}^{N}\sum_{j}^{n_i}(b_{ji}(t).\alpha_{ji}t + b_{ji}(nt).\alpha_{ji}(nt))} \quad (1)$$

In Eq. 1:
N is the number of partitions generated by the temporal partitioning algorithm
τ_{CPU} : the base processor clock cycle time
γ_{CRFU} : a timing factor that represents the time is needed for loading a configuration (partition) on the accelerator
n_i : the number of branches in partition P_i
$\alpha_{ji}(t)$: taken frequency of jth branch in partition P_i
$\alpha_{ji}(nt)$: not-taken frequency of jth branch in partition P_i
$b_{ji}(t)$: is equal to 0 if BTP of the jth branch instruction in P_i has been located at the same partition and it is equal to 1 if some or all instructions of BTP are not located in P_i.
$b_{ji}(nt)$: is equal to 0 if BNTP of the jth branch instruction in P_i has been located at the same partition and it is equal to 1 if some or all instructions of BNTP are not located in P_i.
$CC(P_i)$: the number of clock cycles required for execution of P_i on the accelerator.
f_{p_i} : execution frequency of P_i. Total execution frequency of P_i is the summation of the execution frequency of all other partitions which their execution may cause to starting P_i. For each partition which its execution is independent from other partitions, execution frequency of its first instruction is considered.
CDFGCycleCountOnCPU: number of clock cycles spent for CDFG execution on the CPU in aggregate.

Eq. 2 can be obtained by dividing numerator and denumerator of Eq. 1 to τ_{CPU}.

$$EfficiencyFactor = \frac{CDFGCycleCountOnCPU}{\sum_{i=1}^{N} f_{p_i}.CC(P_i) + (\frac{\gamma_{CRFU}}{\tau_{CPU}}).\sum_{i=1}^{N}\sum_{j}^{n_i}(b_{ji}(t).\alpha_{ji}t + b_{ji}(nt).\alpha_{ji}(nt))} \quad (2)$$

$\frac{\gamma_{CRFU}}{\tau_{CPU}}$ denotes the reconfiguration time of the accelerator in terms of processor clock cycle time. For example, for the ratio equal to 2, accelerator reconfiguration time is equal to duration of the two base processor clock periods. Eq. 2 represents the number of clock cycles spent for DFG execution on the base processor to the number of clock cycles on the accelerator. In other words, it represents the execution time ratio; therefore, larger amount of this factor means lower delay and correspondingly higher speedup. The first term of the denumerator represents the number of clock cycles for CDFG execution on the accelerator and the second term represents the time required to reconfigure the accelerator in terms of the number of clock cycles. Moreover, $\sum_{i=1}^{N}\sum_{j}^{n_i}(b_{ji}(t).\alpha_{ji}t + b_{ji}(nt).\alpha_{ji}(nt)))$ denotes the total number of reconfigurations.

Six applications of Mibench [16] were selected for evaluation of the two proposed algorithms. These applications have considerable number of branch instructions and high potential to get enhanced performance using the conditional execution supporting features (Fig. 3). In addition, in these applications the large

numbers of SSDFGs are generated due to the many short distance branch instructions. Comparison of two NTPT and execution frequency-based temporal partitioning algorithms was accomplished with respect to the average number of partitions (CDFGs) generated and the efficiency factor defined by Eq. 2. According to Fig. 7, using NTPT algorithm, fewer partitions are obtained for all of the applications. We remove all small size CDFGs (CDFGs with the length less than or equal to 5 instructions) from the CDFGs generated by temporal partitioning algorithms.

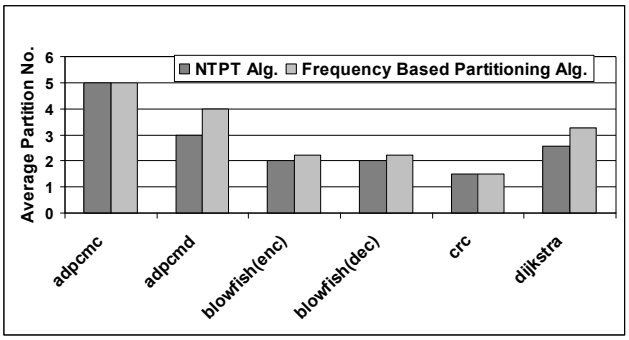


Fig. 7. Comparison of the number of partitions

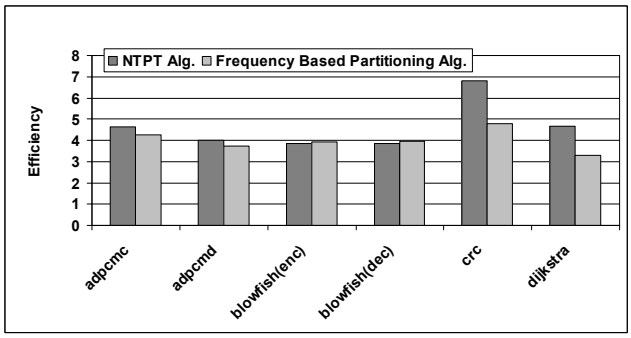


Fig. 8. Comparison of the efficiency factor

Fig. 8 depicts the NTPT algorithm has more or equivalent efficiency compared to frequency-based algorithm. Though, the NTPT algorithm is a simpler approach for temporal partitioning, but it may bring about more efficiency comparing with the more complicated frequency-based algorithm. Some compilers employing for VLIW processors move hot instructions to the not-taken part of branch instructions to avoid the pipeline flushing [11][23]. For the applications have been modified by this kind of compliers, NTPT algorithm is suggested. However, we do not claim that the NTPT algorithm does better for all CDFGs. The following example elucidates this fact:

Analyzing different CDFGs show that for some of them, NTPT is not a good choice. For example, Fig. 9 shows two CDFGs extracted from *blowfish(dec)* application with different properties. Their hot directions have been shown in bold arcs. Using both of algorithms generates similar results for CDFG of Fig. 9.a, since hot direction of branch instructions are in the same direction of not-taken paths. On the other hand, for CDFG of Fig. 9.b, hot direction of branch instruction 2 corresponds to its taken path. Therefore, different outputs are obtained for the two temporal partitioning algorithms. Certainly, more efficiency is achieved for the CDFG of Fig. 9.b through frequency based algorithm.

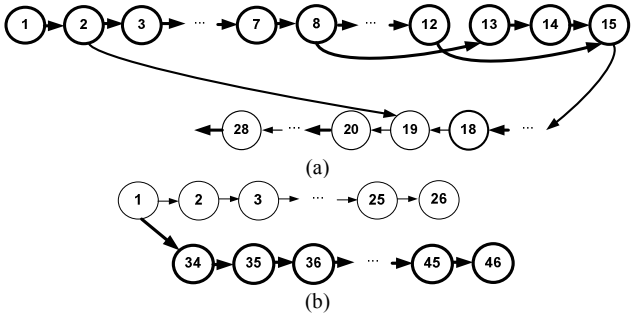


Fig. 9. Sample CDFGs extracted from *blowfish(dec)*

6. Case study: Extending an Accelerator of an Reconfigurable Processor to Support Conditional Execution

As mentioned in Section 2, AMBER comprises a tight integration of a reconfigurable functional unit (RFU) to a RISC processor[17]. Performance enhancement is achievable by executing hot portions on RFU and remaining portions on the base processor. AMBER's RFU can not support conditional execution, here; we propose an extended version of RFU with conditional execution support. The basic requirements introduced in Section 4 are applied to the AMBER's RFU.

6.1. Related Work

Several studies have examined the design of reconfigurable processors and systems. In the case of reconfigurable processors, PRISC [21], OneChip [3], MOLEN [27], and XiRisc [11] are instances of tightly coupled integration of a GPP with fine-grained programmable hardware and ADRES [15] is a sample of a tightly coupled coarse-grained accelerator. AMBER falls in the coarse-grained category. AMBER unlike other reconfigurable processors, neither needs for a new programming model and new compiler nor rewriting and recompiling the source codes. Consequently, the approach is applicable to cases where the source code is not available. Loosely coupled systems like MorphoSys [10] and Garp [6] suffer from the overhead of transferring data between the base processor and the coprocessor. Chimaera [5] adds a shadow register to solve this issue. In AMBER, the input/output resources are shared between the RFU and the processor functional units. In the work proposed by Clark et al. [4] as the most similar work to AMBER, hot portions of the application

are detected (limited to one basic block) using rePLay framework [20] and then executed on a hardware accelerator. rePLay unlike AMBER selects only one direction in branches. When both directions are hot and the branch does not bias in one direction, the hot trace is terminated. In contrast to their design, the control signals or configuration data for AMBER's RFU are generated offline to be more energy efficient. This eliminates the need for more hardware. They also need to extend the *branch target address cache* (BTAC) to store additional information for replacing the DFG by an invocation of a subgraph function. In our extension, generated CDFGs are non-atomic and can include branches and multiple exit points [18]. CDFGs also can accommodate both directions of a branch if both are hot. This feature can save the penalty cycles due to the misprediction of branches. Finally, unlike their proposed accelerators, AMBER's extended RFU can support conditional execution.

6.2. Extending RFU to Support Conditional Execution

In Section 4, basic requirements were presented for the accelerator architecture featuring conditional execution. Here, we extend AMBER's RFU according to these requirements and construct a conditional RFU (CRFU). First, we propose conditional data selection muxes for controlling selectors of muxes used for FU inputs and outputs of the RFU. Fig. 10 (top portion) shows a RFU (with 5 FUs) without conditional execution facilities. On the other hand, the hardware has been modified as shown in bottom part of Fig. 10 to support conditional data execution. In the proposed architecture, the selector signals of muxes used for choosing data for FU inputs (the Data-Selection-Mux), along with the CRFU output and exit point (not shown in the figure) are controlled by other muxes (the Selector-Mux). The inputs of Selector-Mux (one-bit width) originate from the FUs (which execute branches) of the upper rows and the configuration memory in order to control the selector signals conditionally, as well as unconditionally. The selectors of Selector-Mux are controlled by configuration bits. It should be noted the outputs of FUs are only applied to the Selector-Muxes in the lower-level rows, not in the same or upper rows. A similar structure is used for selecting the valid output data of the CRFU.

For example, suppose a CDFG containing nodes (instructions) (3:subu), (6:beq), (7:subu) and (13:subu) (Fig. 4) is to be mapped on the CRFU. The first source of instruction 13 (R3) uses the output of instruction 3 when instruction 6 is taken otherwise uses the output of instruction 7. Instructions 3, 7, 6, and 13 are mapped to FU1, FU2, FU3, and FU5, respectively, using the mapping algorithm presented in [13]. In this architecture, the selection bits for input muxes of FU4 and FU5 are controlled by configuration bits. Assuming that outputs of FU1, FU2, FU3, and the immediate value have been assigned to inputs 1, 2, 3, and 0 of the *Data Selection Mux* in the second input of FU5. The selector signals of *Selector-Mux* i.e. *Sel1* and *Sel0* are configured to be driven by *Not Branch result from FU3* and *Branch result from FU3*, respectively, using configuration bits. When FU3 (instruction 6) is taken, *Sel1* is 0 and *Sel0* is 1, therefore the output of FU1 (instruction 3) is selected. When FU3 is not-taken *Sel1* is 1 and *Sel0* is 0, therefore the output of FU2 (instruction 7) is selected.

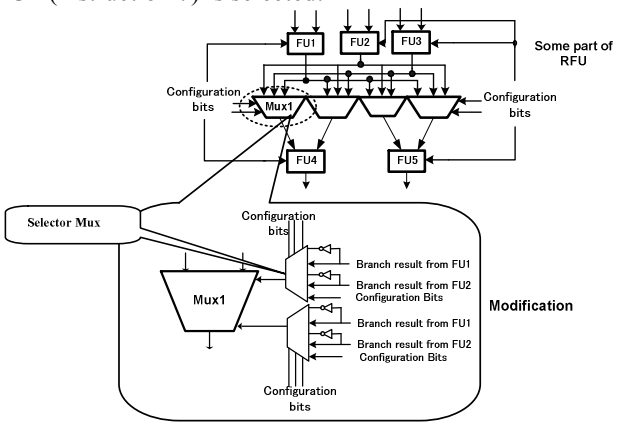


Fig. 10. Equipping the RFU to support conditional execution

6.3. Performance and Energy Evaluation

The CRFU was developed and synthesized using Synopsys tools [25] and Hitachi 0.18μm. Its area is 2.1 mm^2 (gate count of CRFU is 68,407 and 231,236 gates for the base processor). Each CDFG totally needs 615 bits (~80bytes) for its configuration on the CRFU. Profiling data was provided by executing applications on the Simplescalar [22] as ISS. *Integrated Framework* introduced in [13] was modified to use NTPT temporal partitioning algorithm to generate mappable CDFGs for executing on the modified AMBER's RFU (CRFU). Total elapsed time by the modified *Integrated Framework* for processing 487 CDFGs from 22 applications on a host machine (Intel Core 2 Duo 6600, 2400MHz, 2GB RAM) was less than 7sec.

The CRFU has variable delay for CDFG execution. This idea has been proposed in [17]. The delay of CRFU for CDFGs with various depths (critical path lengths) from **1** to 5 (maximum supportable depth) are 2.2*ns,* 4.2*ns,* 6.1*ns,* 7.9*ns and* 9.8*ns,* respectively. The required number of clock cycles for executing each CDFG is determined according to the depth of CDFG and base processor clock frequency. Therefore, according to the clock frequency of the AMBER's base processor which is 300MHz (according to Table 1) the number of clock cycled required for executing CDFGs with depths 1 to 5 are 1, 2, 2, 3, 3

clock cycles, respectively. Configuration of the AMBER's base processor is as Table 1.

Table. 1. Base processor configuration

Issue	4-way
Clock frequency	300MHz
L1- I cache	32K, 2 way, 1 cycle latency
L1- D cache	32K, 4 way, 1 cycle latency
Unified L2	1M, 6 cycle latency
Execution units	4 integer, 4 floating point
RUU size & Fetch queue size	64
Branch predictor	Bimodal
Branch prediction table size	2048
Extra branch misprediction latency	3

We evaluated the effectiveness of CDFGs versus DFGs in the aspects of speedup and total energy reduction. The average number of instructions included in DFGs is 5.43 instructions and for CDFGs is 8.32 instructions. Therefore, extending DFG and covering control instructions results in larger data flow graphs for acceleration, hence promising more speedup. Fig. 11 shows the speedups obtained based on CDFG and DFG compared to the base processor for some applications. According to Fig. 11, using CDFG achieves remarkable speedup compared to DFGs as expected. The reason for the high speedup obtained by *adpcm* is that it has a main loop with 56 instructions, including 12 branches. For 7 of these branches, both taken and not-taken instructions are hot, so that 27% of branches are mispredicted. Therefore, a big part of executed clock cycles belongs to penalty of the mispredicted branches (18%). For those branches with both directions being hot, the CDFGs include both directions, and hence, the CRFU architecture eliminates cycles of mispredicted branches. Also, since CDFGs are longer than DFGs, more ILP can be exploited.

Other comparison was done based on the effect of employing CDFG versus DFG in total energy reduction. In our measurement, the configuration memory is assumed to keep up to 100 CDFG configurations. Therefore, the size of the configuration memory is 80x100 bytes SRAM with a 640-bit width data bus and in one clock cycle the configuration can be loaded to the CRFU.

Verilog-XL from Cadence, Power Compiler from Synopsys and 0.18µm technology cell library from Hitachi were exploited to measure the power of CRFU. The power consumption of the CRFU for 100,000 different test vectors is 246.335mW. The configuration memory was modeled using CACTI [26] in 0.18µm. The area is 0.77mm^2 and the energy for each access is 0.198nJ. Also, Wattch [2] which is based on Simplescalar [22] was used for energy estimation of the base processor. The Wattch was targeted for 0.18µm as well. Fig. 12 shows the total energy reduction for the AMBER using CDFG compared to the DFG for the clock frequency of 300MHz. This figure concludes that using CDFG brings about noticeable reduction in total energy compared to DFG.

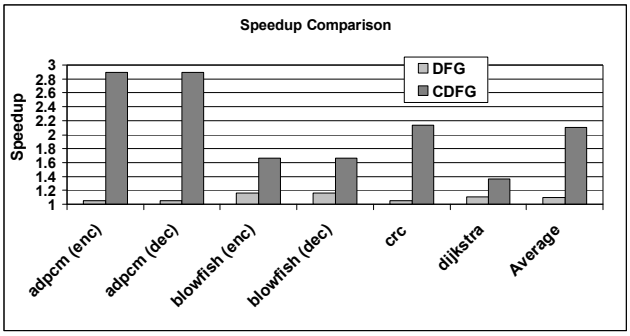


Fig. 11. Speedup comparison of DFG vs. CDFG

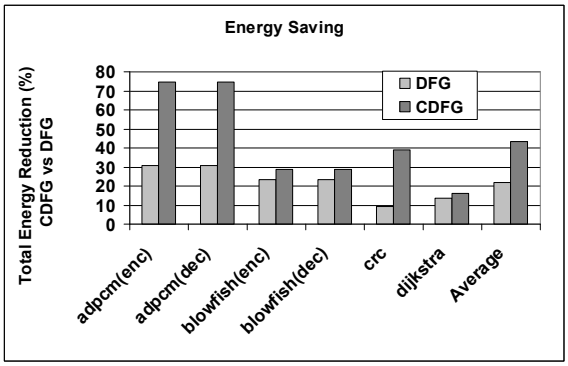


Fig. 12. Comparison of energy reduction using CDFG vs. DFG

Finally, we compared our proposed architecture with two most similar architectures which employ the tightly coupled reconfigurable accelerators augmented to a RISC processor inside a reconfigurable processor. Table 2 summarizes this comparison. Although, our architecture uses 0.18µm technology and variable delay accelerator and on the other side they used 0.13µm technology with one clock cycle delay accelerator, higher speedup in average has been attained by ours. Moreover, in our architecture, total power dissipation is reduced by 43.5% in average whereas, no measurement is available for others.

7. Conclusion

In an effective method to enhance performance of an embedded system, data flow graphs extracted from frequently executed portions of an application are mapped and executed on an accelerator. With respect to the result of a branch in instruction sequences, one of its taken or not-taken paths should be executed. In this paper, we highlighted the main motivations for handling branch instruction in DFGs and extending DFGs to CDFGs. In addition, basic requirements for developing an accelerator with conditional execution support were pointed out.

Table 2: Comparison of two similar approaches with ours

Approach	Base Processor	Accelerator Coupling to Processor	Technology Size (µm)	Accelerator Granularity	Accelerator Delay (# of clock cycles)	Average Speedup	Energy Reduction (%)
Yehia et al. [28]	ARM 1-issue, 250MHz	Tight	0.13	Fine (LUT-based)	1	**1.47**	N/A
Clark et al. [4]	ARM 4-issue, 250MHz	Tight	0.13	Coarse (FU-based)	1	**1.28**	N/A
Ours	MIPS 4-issue, 300MHz	Tight	0.18	Coarse (FU-based)	Variable	**2.1**	**43.5**

Also, two algorithms for CDFG temporal partitioning and generating mappable CDFGs were proposed. Mappable CDFGs satisfy the accelerator hardware constraints and can be executed on accelerator. NTPT is a temporal partitioning algorithm which tries to traverse not-taken path of the branch instructions and partitions the input CDFG. On the contrary, frequency-based temporal partitioning algorithm considers the taken and not-taken frequencies to partition input CDFG. In this approach, both taken and not-taken paths associated to a branch can be added to a partition simultaneously. Comparison of these algorithms shows that though NTPT is a simple partitioning algorithm but it generates small number of CDFGs which bring about a comparable and even higher speedup. To show the effectiveness of supporting conditional execution in hardware, we applied our proposals to the accelerator of an extensible processor called AMBER. RFU was a matrix of functional units which was extended (CRFU) to support the conditional execution. We used an integrated framework based on NTPT algorithm to generate mappable CDFGs on CRFU. These CDFGs are executed on CRFU to accelerate the application execution. Experimental results show the noticeable effectiveness of covering branch instructions and using CDFGs versus DFGs. Also, total energy degrades by 43%. In addition, the designated architecture obtains higher speedup in comparison with two similar reconfigurable processors.

Acknowledgment

This research was supported in part by Core Research for Evolutional Science and Technology (CREST) of Japan Science and Technology Corporation (JST) and Grant-in-Aid for Encouragement of Young Scientists (A), 17680005.

References

[1] Auguin, M. Bianco, L. Capella, L. Gresset, E. Partitioning conditional data flow graphs for embedded system design, Proc. of ASAP 2000 (2000) 339-348

[2] Brooks D. Wattch: a framework for architectural-level power analysis and optimizations, In Proc. ISCA (2000)

[3] Carrillo, J. E. Chow, P. The effect of reconfigurable units in superscalar processors, Proc. of the ACM/SIGDA FPGA (2001) 141-150.

[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 Proc. of the 37th Annual International Symp. on Microarchitecture (2004) 30-40.

[5] Hauck, S. Fry, T. Hosler, M. Kao, J. The Chimaera reconfigurable functional unit, IEEE Symp. on FPGAs for Custom Computing Machines (1997) 206-217.

[6] Hauser, J.R. Wawrzynek, J. GARP: A MIPS processor with a reconfigurable coprocessor, IEEE Symp. On FPGAs for Custom Computing Machines (1997) 12-21.

[7] Karthikeya, M. Gajjala, P. Bhatia, D. Temporal partitioning and scheduling data flow graphs for reconfigurable computers, IEEE Transactions on Computers, 48 (6) (1999) 579-590.

[8] Kastner, R. Kaplan, A. Sarrafzadeh, M. Synthesis techniques and optimizations for reconfigurable systems, Kluwer-Academic Publishers (2004).

[9] Lee, J.E. Kim, Y. Jung, J. Choi, K. Reconfigurable ALU array architecture with conditional execution, International SoC Design Conference (2004) 222-226.

[10] Lee, M.H. Singh, H. Lu, G. Bagherzadeh, N. Kurdahi, F.J Design and implementation of the MorphoSys reconfigurable computing processor, Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology (2000).

[11] Lodi, A. Toma, M. Campi, F. Cappelli, A. Canegallo, R. Guerrieri, R. A VLIW processor with reconfigurable instruction set for embedded applications, IEEE Journal of Solid-State Circuits, Vol. 38, No. 11 (2003) 1876–1886.

[12] Mahlke, S. A. Hank, R. E. McCormick. J.E. August. D. A comparison of full and partial predicated execution support for ILP processors. In Proc. ISCA (1995) 138-150.

[13] Mehdipour, F. Noori, H. Saheb Zamani, M. Murakami, K. Sedighi, K. Inoue, K. Custom instruction generation using temporal partitioning techniques for a reconfigurable functional unit, Proc. of EUC'06 (2006).

[14] Mehdipour, F. Saheb Zamani, M. Sedighi, M. An integrated temporal partitioning and physical design

framework for static compilation of reconfigurable computing systems, Int. J. of Microprocessors and Microsystems, Elsevier, Vol. 30, No. 1 (2006) 52-62.
[15] Mei, B. Vernalde, S. Verkest, D. Lauwereins, R. Design methodology for a tightly coupled VLIW/Reconfigurable matrix architecture, DATE (2004) 1224-1129.
[16] Mibench, www.eecs.umich.edu/mibench.
[17] Noori, H. Mehdipour, F. Murakami, K. Inoue, K. Saheb Zamani, M. A reconfigurable functional unit for an adaptive dynamic extensible processor, Proc. of FPL (2006) 781-784.
[18] Noori, H. Mehdipour, F. Murakami, K. Inoue, K. Goudarzi, M. Generating and Executing Multi-Exit Custom Instructions for an Adaptive Extensible Processor, Design-Automation and Test in Europe (DATE'07) (2007).
[19] Park, J.C. Schlansker, M.S. On predicated execution, Technical Report HPL-91-58. Hewlett Packard Laboratories (1991).
[20] Patel, S. Lumetta, S. rePLay: A hardware framework for dynamic optimization, IEEE Transanction Computer 50 (6) (2001) 590-608.
[21] Razdan, R. Smith, M.D. A high-performance microarchitecture with hardware-programmable functional units, MICRO-27 (1994).
[22] Simplescalar, www.simplescalar.com
[23] Smith J.E, Sohi, G.S. The microarchitecture of superscalar P. In Proc. IEEE, Vol. 83, (1995) 1609- 1624.
[24] Stitt, G. Lysecky, R. Vahid, F. Dynamic hardware/software partitioning: a first approach, In Proc. Design Automation Conference (2003) 25-255.
[25] Synopsys Inc. http://www.synopsys.com/roducts/logic/design_compiler.html.
[26] Tarjan, D. Thoziyoor, Sh. Jouppi, N.P. Cacti 4.0, HP Laboratories, Technical Report (2006).
[27] Vassiliadis, S. Gaydadjiev, G. Kuzmanov, G. The MOLEN polymorphic processor, IEEE Transactions on Computers, Vol. 53, No. 11 (2004) 1363-1375.
[28] Yehia, S. Clark, N. Mahlke, S. Flautner, K. Exploring the design space of LUT-based transparent accelerators. In Proc. of CASES'05 (2005).

Farhad Mehdipour received the B.Sc. degree from Sharif University of Technology in 1996, and the M.Sc. and Ph.D. degrees in Computer Systems Architectures from the Amirkabir University of Technology in 1999 and 2006, respectively. He joined to Research Institute for Information Technology, Kyushu University in December 2006. His research interests include reconfigurable computing systems, physical design and reconfigurable embedded processors.

Hamid Noori received the B.Sc. degree from Sharif University of Technology in 1996, and the M.Sc. degree in Computer Systems Architectures from the Amirkabir University of Technology in 1999. He is currently PhD candidate in Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan, since 2004. He is student member of IEEE and his research interests include reconfigurable processors and devices and high performance, low power embedded processors.

Morteza Saheb Zamani received the B.Sc. degree in Computer Engineering from Isfahan University of Technology in 1989, and the M.Eng.Sc. and Ph.D. degrees in Computer Engineering from the University of New South Wales, Australia in 1992 and 1996, respectively. He joined Amirkabir University of Technology in 1996 and he is now an assistant professor at IT and Computer Engineering department and the head of "Computer Systems Architectures" group.

Koji Inoue was born in Fukuoka, Japan in 1971. He received the B.E. and M.E. degrees in computer science from Kyushu Institute of Technology, Japan in 1994 and 1996, respectively. He received the Ph.D. degree in Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan in 2000. In 1999, he joined Halo LSI Design & Technology, Inc., NY, as a circuit designer. He is currently an associate professor of the Department of Informatics, Kyushu University. His research interests include power-aware computing, high-performance computing, dependable processor architecture, and secure computer systems. He is a member of the ACM, the IEEE, the IEEE Computer Society, the IEICE, and the IPSJ (Information Processing Society of Japan).

Kazuaki Murakami was born in Kumamoto, Japan in 1960. He received the B.E., M.E., and Ph.D. degrees in computer science and engineering from Kyoto University in 1982, 1984, and 1994, respectively. From 1984 to 1987, he worked for the Fujitsu Limited, where he was a Computer Architect of the mainframe computers. In 1987, he joined the Department of Information Systems of Kyushu University, Japan. He is currently a Professor of the Department of Informatics, and also the Director of the Computing and Communications Center. He is a member of the ACM, the IEEE, the IEEE Computer Society, the IPSJ, and the JSIAM.