九州大学学術情報リポジトリ
Kyushu University Institutional Repository

High Performance, Low Power Reconfigurable Processor for Embedded Systems

Mehdipour, Farhad
Computing and Communication Center, Kyushu University

Noori, Hamid
Department of Informatics, Kyushu University

Inoue, Koji
Department of Informatics, Kyushu University

Murakami, Kazuaki
Department of Informatics, Kyushu University

http://hdl.handle.net/2324/8315

出版情報：International SoC Design Conference, pp.51-55, 2007-10
バージョン：
権利関係：

KYUSHU UNIVERSITY

High Performance, Low Power Reconfigurable Processor for Embedded Systems

Farhad Mehdipour
Computing and Communication Center,
Kyushu University, Fukuoka, Japan
farhad@c.csce.kyushu-u.ac.jp

Hamid Noori , Koji Inoue , Kazuaki Murakami
Department of Informatics, Kyushu University, Fukuoka, Japan
noori@c.csce.kyushu-u.ac.jp
{inoue, murakami}@i.kyushu-u.ac.jp

Abstract - *Using an extensible processor in which data flow graphs (DFGs) are generated from frequently executed portions (hot portions) of applications and are executed after chip-fabrication provides flexibility as well as addressing the time-to-market and significant non-recurring engineering costs issues. In this paper, the effect of extending DFGs to control data flow graphs (CDFGs) through covering control instructions on the speedup is studied. Moreover, basic requirements for an accelerator with conditional execution support are presented. A temporal partitioning algorithm is introduced to partition the large CDFGs to smaller mappable ones under the accelerator resource constraints. To demonstrate effectiveness of the proposed ideas, they are applied to the accelerator of an extensible processor called AMBER which utilizes a matrix of functional units to accelerate the execution of the DFGs. Experimental results approve the considerable effectiveness of covering control instructions and using CDFGs versus DFGs in the aspects of performance and energy reduction.*

Keywords: 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) [6]. The integration of accelerator and the processor can be tightly [3][4][7][13] or loosely coupled [6][10]. This paper focuses especially on tightly coupled reconfigurable accelerator. In the lattar one, 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 eliminating data transferring overead.

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. 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.

Some applications of Mibench [11] are used for the analysis to explore motivations for extending DFGs over control instructions and generating CDFGs to be executed on an accelerator. 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 [8] that the small length DFGs (including less than or equal to five instructions) offer no more speedup. In Figure 1, 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.

This kind of analysis was accomplished for 17 applications of Mibench [11]. Figure 2 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 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. 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. According to the result of a branch instruction, one of the instructions sequence located in taken or not-taken paths of the associated branch or both of them might be executed. Covering both directions can aid the generation of larger CDFGs, hence more parallelism, as well as eliminating branch misprediction penalties.

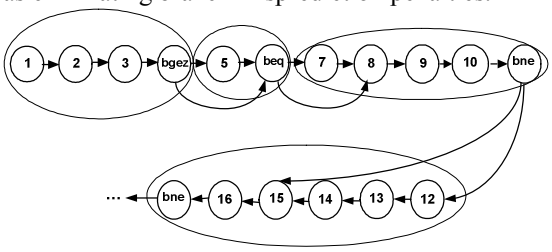


Figure 1. Control DFG of hot portion of *adpcmc(enc)*

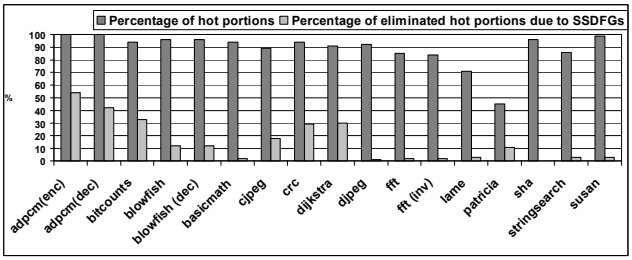
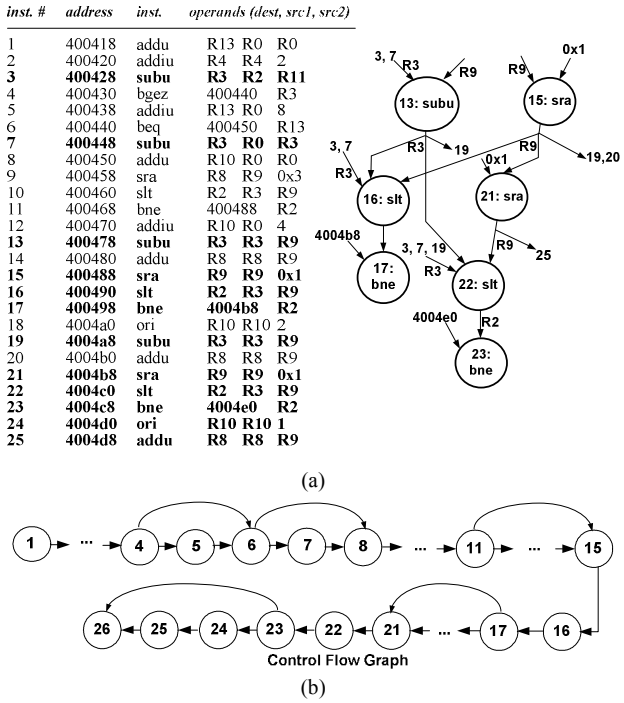


Figure 2. Fraction of hot portions and eliminated hot portions in some applications of Mibench [11]

2 Basic Requirements for Conditional Execution Support in Hardware

For conditional execution support, the accelerator integrating to the base processor should be equipped by the capability of branch instruction execution. 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 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. Figure 3 shows a piece of *adpcm(enc)*'s critical portion, a part of its corresponding DFG (Figure 3.a) and the CFG comprising only control flow of instructions (Figure 3.b). In Figure 3.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)

Figure 3. A piece of *adpcm(enc)* code, a part of its corresponding DFG and (a) its control flow graph (b)

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 Figure 3.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.

3 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][5]. A temporal partitioning algorithm can consider the accelerator architectural specifications to generate executable DFGs on the accelerator. As the authors' knowledge there are few algorithms for CDFG partitioning, though a lot of works have been done around the DFG temporal partitioning [1][5]. In [1] a temporal portioning algorithm has been presented that partitions a CDFG considering target hardware with non-homogenous architecture. This algorithm considers all states of the control instructions in application to convert corresponding CDFG to a number of DFGs. Then it minimizes 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.

Here, an algorithm is 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 algorithm which is refferd as not-taken path traversing temporal partitioning algorithm (NTPT) 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. *Terminator* instruction is an exit point for a CDFG and changes execution direction including procedure or function call instructions and also backward branch and return (to prevent making cycles in CDFG). 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. Figure 4 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 used as an initial instruction of the next partition.

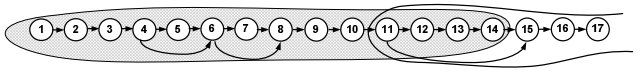
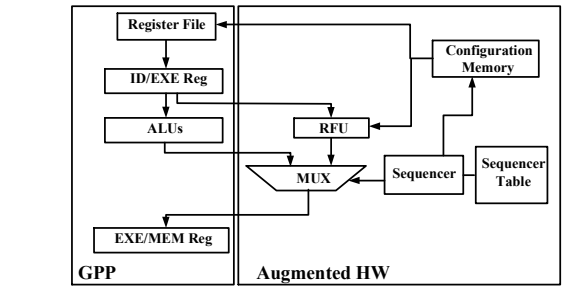


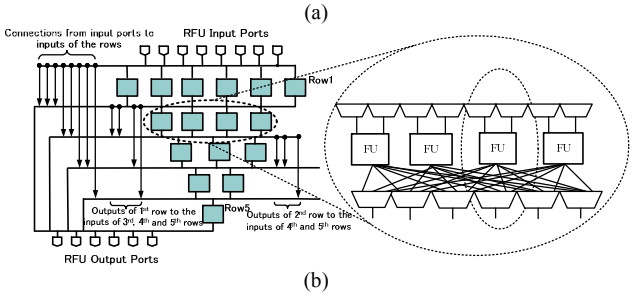
Figure 4. Applying NTPT algorithm on a sample CDFG

4 Extending an Accelerator of a Reconfigurable Processor

AMBER is a reconfigurable processor [12] targeted for embedded systems. It has been developed by integrating a base processor with two other main components[12]. The base processor is a general RISC processor and the other two components are: sequencer and a coarse grain reconfigurable functional unit (RFU). Figure 5.a illustrates the integration of different components in AMBER.



(a)



(b)

Figure 5. Main components in AMBER (a) RFU architecture (b)

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* (Figure 5.b) 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. RFU reads (write) from (to) register file. In the RFU, the output of each FU in a row can be used by all FUs in the subsequent row. Performance enhancement is achievable by executing hot portions on RFU and remaining portions on the base processor. More details on AMBER can be found in [12]. AMBER's RFU can not support conditional execution, therefore; we propose an extended version of RFU through applying the basic requirements mentioned in Section 2 to support conditional execution.

First, we propose conditional data selection muxes for controlling selectors of muxes used for FU inputs and outputs of the RFU. Figure 6 (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 Figure 6 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 more details refer to [9].

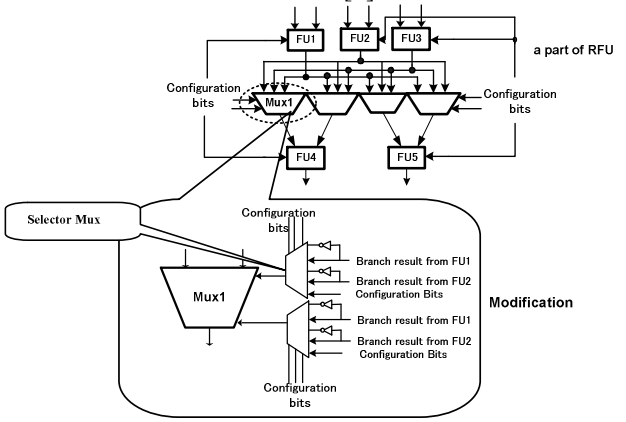


Figure 6. Modifying RFU architecture

For example, suppose a CDFG containing nodes (instructions) (3:subu), (6:beq), (7:subu) and (13:subu) (Figure 3) 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 [8]. 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.

5 Experimental Results

The CRFU was developed and synthesized using Synopsys tools [15] and Hitachi 0.18µm. Its area is 2.1 mm^2. NTPT temporal partitioning algorithm was used to generate mappable CDFGs for executing on the CRFU. The CRFU has variable delay for CDFG execution. This idea has been proposed in [12]. 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. 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. Figure 7 shows the speedups obtained based on CDFG and DFG compared to the base processor for a number of applications. According to Figure 7, using CDFG achieves remarkable speedup compared to DFGs as expected.

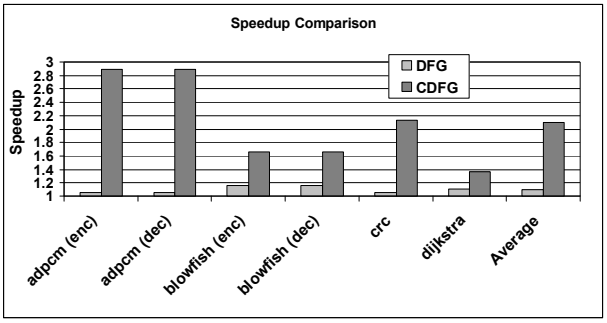


Figure 7. Speedup comparison of DFG vs. CDFG

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 [16] 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 [14] was used for energy estimation of the base processor. The Wattch was targeted for 0.18µm as well. Figure 8 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.

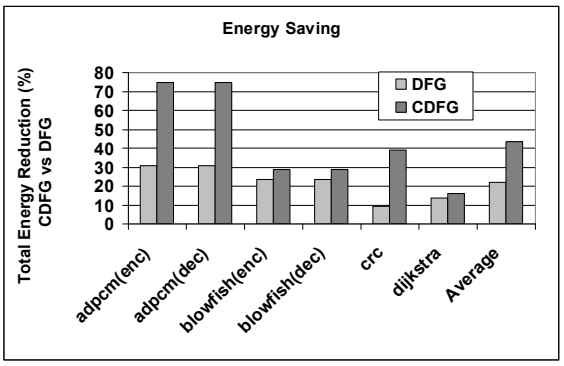


Figure 8. Comparison of energy reduction

6 Conclusion

In this paper, the main motivations for handling branch instruction in DFGs and extending DFGs to CDFGs were highlighted. In addition, basic requirements for developing an accelerator with conditional execution support were pointed out. NTPT is a temporal partitioning algorithm which was introduced for generating mappable CDFG. Mappable CDFGs satisfy the accelerator hardware constraints and can be executed on accelerator. 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. Experimental results show the noticeable effectiveness of covering branch instructions and using CDFGs versus DFGs in acheiving higher speedup. Also, total energy degrades by 43% by using CDFGs.

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] M. Auguin, L. Bianco, L. Capella and E. Gresset, ”Partitioning conditional data flow graphs for embedded system design”, Proc. of ASAP’00, pp. 339-348, 2000.

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

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

[4] 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 the 37th Annual International Symp. on Microarchitecture, pp. 30-40, 2004.

[5] M. Karthikeya, P. Gajjala and D. Bhatia, “Temporal partitioning and scheduling data flow graphs for reconfigurable computers”, IEEE Transactions on Computers, Vol. 48, No. 6, pp. 579-590, 1999.

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

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

[8] F. Mehdipour, H. Noori, M. Saheb Zamani, K. Inoue, and K. Murakami, “Custom instruction generation using temporal partitioning techniques for a reconfigurable functional unit”, Proc. of EUC’06, pp. 249-260, 2006.

[9] F. Mehdipour, H. Noori, M. Saheb Zamani, K. Inoue and K. Murakami, “Improving Performance and Energy Saving in a Reconfigurable Processor via Accelerating Control Data Flow Graphs”, IEICE Transaction on Information and Systems, to be appeared, 2007.

[10] B. Mei, S. Vernalde, D. Verkest and R. Lauwereins, ”Design methodology for a tightly coupled VLIW/Reconfigurable matrix architecture”, DATE’04, pp. 1224-1129, 2004.

[11] Mibench, www.eecs.umich.edu/mibench.

[12] H. Noori, F. Mehdipour, K. Murakami, K. Inoue, and M. Saheb Zamani, “A reconfigurable functional unit for an adaptive dynamic extensible processor”, Proc. of FPL’06, pp. 781-784, 2006.

[13] R. Razdan and M.D. Smith, “A high-performance microarchitecture with hardware-programmable functional units”, MICRO-27, 1994.

[14] Simplescalar, www.simplescalar.com

[15] Synopsys Inc. http://www.synopsys.com/roducts/logic/design_compiler.html.

[16] D. Tarjan, S. Thoziyoor, N.P. Jouppi, “Cacti 4.0, HP Laboratories”, Technical Report, 2006.