Handling Control Data Flow Graphs for a Tightly Coupled Reconfigurable Accelerator

Noori, Hamid
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

Mehdipour, Farhad
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

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

Inoue, Koji
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

他

出版情報：Proceedings of 3rd International Conference on Embedded Software and Systems, pp.345-356, 2007-05-15. International Conference on Embedded Software and Systems
バージョン：

Handling Control Data Flow Graphs for a Tightly Coupled Reconfigurable Accelerator

Hamid Noori[1], Farhad Mehdipour[1], Morteza Saheb Zamani[2], Koji Inoue[1], and Kazuaki Murakami[1]

[1] Department of Informatics,
Graduate School of Information Science and Electrical Engineering,
Kyushu University, Fukuoka, Japan
{noori,farhad}@c.csce.kysuhu-u.ac.jp,
{inoue,murakami}@i.kyushu-u.ac.jp
[2] Department of IT and Computer Engineering, Amirkabir University of Technology,
Tehran, Iran
szamani@aut.ac.ir

Abstract. In an embedded system including a base processor integrated with a tightly coupled accelerator, extracting frequently executed portions of the code (hot portion) and executing their corresponding data flow graph (DFG) on the accelerator brings about more speedup. In this paper, we intend to present our motivations for handling control instructions in DFGs and extending them to Control DFGs (CDFGs). In addition, basic requirements for an accelerator with conditional execution support are proposed. Moreover, some algorithms are presented for temporal partitioning of CDFGs considering the target accelerator architectural specifications. To show the effectiveness of the proposed ideas, we applied them to the accelerator of an extensible processor called AMBER. Experimental results represent the effectiveness of covering control instructions and using CDFGs versus DFGs.

1 Introduction

Using an accelerator for accelerating the execution of frequently executed portions of applications is an effective technique to enhance the performance of a processor in embedded systems. In this technique, data flow graphs (DFGs) extracted from critical portions of an application are executed on an accelerator. Similar technique has been presented in [3, 4, 9, 13, 15, 22, 2, 5, 17, 21]. 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) [7]. The integration of accelerator and the processor can be tightly or loosely coupled [7, 13]. For loosely-coupled systems, there is an overhead for transferring data between the base processor and the accelerator. When an accelerator is tightly coupled [9, 2, 5, 17, 21], 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

Y.-H. Lee et al. (Eds.): ICESS 2007, LNCS 4523, pp. 345–356, 2007.

programmable hardware unit, however, it increases the read/write ports of the register file. Our main focus in this paper is on a tightly coupled reconfigurable accelerator.

DFG extraction can be done at high level or binary level of the source code. In our analysis, we focus on the latter one which means that the DFG nodes are the primitive instructions of the base processor. The DFG containing control instructions (e.g. branch instruction) are called Control Dataflow Graphs (CDFGs). Handling branches (conditional execution) is a challenge in CDFG acceleration, because, due to the result of a branch instruction, the sequence of execution changes. We consider two types of CDFGs:

1. CDFGs which contain at most one branch instruction as its last instruction. In this case the accelerator does not need to support conditional execution.
2. CDFGs containing more than one branch instructions. Accelerators used for executing CDFGs should have conditional execution support.

As mentioned before, accelerators are used for executing hot portions of applications. Therefore, while generating CDFG we only follow hot directions of branches. For a control instruction, the only taken, or not-taken might be hot which means that they have a considerable execution frequency (more than a specified threshold). In some other cases, both directions can be hot. We propose adding only hot directions of branches into the CDFG without being limited to selecting just one or all of the directions. This brings about more instruction level parallelism (ILP) and can hide branch misprediction penalty.

In this work, we intend to answer these two following questions.

a) Does acceleration based on CDFG vs. DFG obtain higher performance?
b) How can the conditional execution be supported on an accelerator?

To answer the first question, we investigate the effect of extending DFGs and covering control instructions on the speedup and present some important motivations for extending DFGs over basic blocks (using CDFGs instead of DFGs). Moreover, as an answer to the second question, we introduce basic requirements for an accelerator with conditional execution support.

Due to the limitations of hardware resources of the accelerator (e.g. the number of inputs, outputs, logics, connections and etc) and different size of extracted CDFGs from various applications, in most cases the whole CDFG can not be mapped on the accelerator. As another contribution in this paper, we present 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.

2 Motivations

In this section, basic arguments to extend DFGs over control instructions and supporting CDFGs are investigated. We follow a quantitative analysis approach and use some applications of Mibench [14] for these analyses. As mentioned above, DFGs are extracted from the frequently executed portions of an application and a control instruction (e.g. branch instruction) may cause the DFG generation to be stopped.

Therefore, short distance control instructions may 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 [11] because they do not offer any more speedup.

In Fig. 1, a piece of a main loop of *adpcm(enc)* has been shown. *adpcm(enc)* is an application program which includes a loop which consumes 98% of total execution time. The critical portion of application contains 3 loads and 12 branch instructions. According the location of branch instructions, 4 DFGs can be extracted from the piece of loop that has been shown in Fig 1. Three DFGs from four depicted DFGs in Fig. 1 are SSDFGs (have the length less than or equal to 5 (we only execute DFGs which have more than 5 nodes on the accelerator). These SSDFGs do not bring about more speedup and have to be run on the base processor.

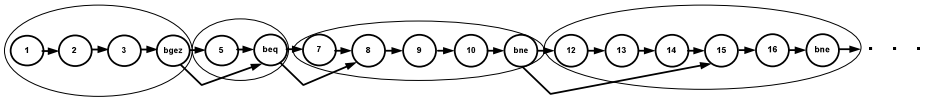

Fig. 1. Control data flow graph of hot portion of *adpcmc(enc)*

This kind of analysis was accomplished for 17 applications of Mibench [14]. Results of analysis motivate us to use CDFGs instead of DFGs for acceleration. Fig. 2 shows the overall percentage of frequently executed (hot) potion of each application. In addition, this figure shows the fraction of applications that can not be accelerated due to 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 results in no considerable speedup, because a small portion of generated DFGs are removed due to SSDFGs.

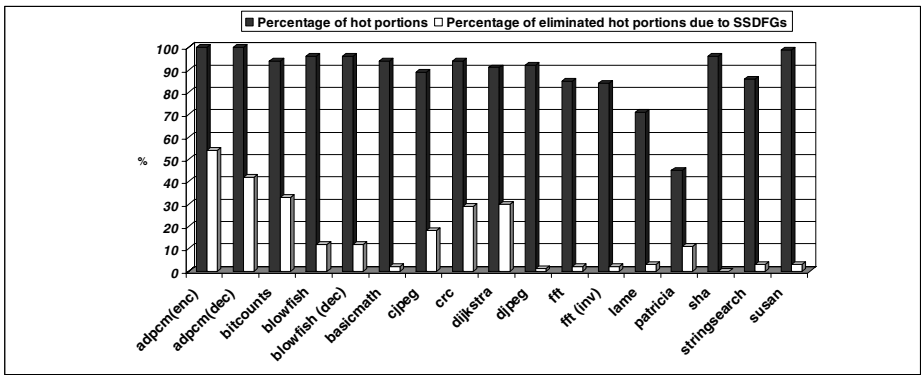

Fig. 2. Fraction of hot portions and eliminated hot portions in applications

Extending DFGs to contain more than one branch instruction and generate the CDFGs vs. DFGs is one solution to prevent many SSDFGs generation. For a control instruction, in some cases only taken or not-taken might be hot and for some others both directions are hot. In latter case, covering both directions may help to the generation of larger CDFGs, hence more parallelism and elimination of branch misprediction penalties. In addition, architecture of the accelerator should be modified to execute the CDFGs. Indeed, appropriate algorithms are required to generate CDFGs considering the specifications of the accelerator.

3 Basic Requirements for Architecture with Conditional Execution Support

To support conditional execution in an accelerator, the capability of branch instruction execution should be added to the accelerator. It is assumed that the proposed accelerator is a coarse grain reconfigurable hardware which is a matrix of functional units (FUs) with specific connections between the FUs. Moreover, each FU like the processor's ALUs can execute an instruction level operation.

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. 3 shows the CFG (contains only control flow of instructions) and DFG for a section of an *adpcm(enc)* loop. Node 8 may receive one of its inputs from nodes 5 or 7. The result of the branch that located in node 6 determines which one should be selected. 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 support conditional execution and generate valid output data. Predicated execution is one technique [16].

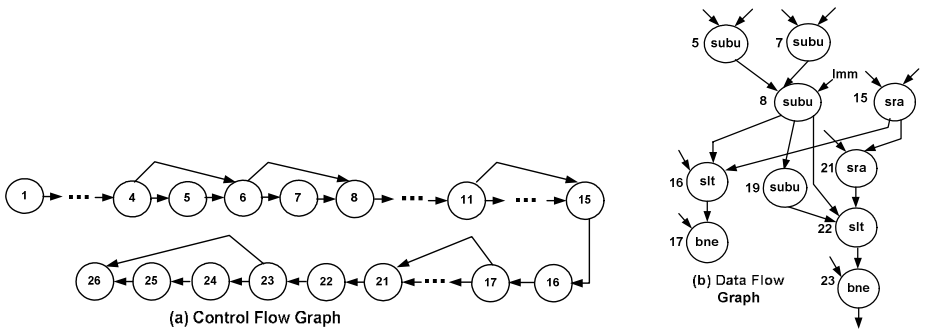

Fig. 3. Control flow (a) and data flow graphs (b) for a part of *adpcm(enc)* loop

Predicated execution is an effective technique to remove control dependency of programs running on ILP (Instruction level parallelism) processors. Proposed architecture in [8] uses predicated instructions. With predicated execution, control

dependency is essentially turned into data dependency using predicates. A predicate is a Boolean variable used to represent the control information of a control instruction and to nullify the following instructions associated with it. The following instructions become no-ops if the predicated variable is evaluated to be false. The architecture featuring predicated execution should have radical changes, since every instruction can be predicated and a separated predicated register file is needed. Also, partial architectural support has also been studied [10]. In [10], Mahlke et al. proposed architecture with two new instructions added to the original instruction set to support predicated execution.

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

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. 3 output of node 8 can be routed to nodes 16, 19 or 22. Node 19 will receive the output of node 7 if branch instruction in node 7 is not-taken, otherwise it will be obtained by node 22. Therefore, there may be several outputs for a CDFG and some of them may be valid as its output accelerator final outputs.

According to above mentioned properties, the accelerator architecture must have these following 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. In this case, no need to modify the FUs.
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 specifically proposed for an extensible processor in Section 5.

4 Algorithms for CDFG Temporal Partitioning

CDFG extracted from various applications are in different sizes and for some of the CDFGs the whole of it can not be mapped on the accelerator due to the limitations of hardware resources of the accelerator (e.g. number of inputs, outputs, logics and specifically routing resource constraints). Even if the logic resource limitations are considered, some constraints like the routing resource constraints are not applicable in

CDFG generation phase. Satisfying or violating routing resource constraints can be specified after trying to map a CDFG on the accelerator. Therefore, we investigate some algorithms for partitioning CDFGs under the different hardware resources constraints of the accelerator and introduce a mapping-aware framework which considers the routing resource constraints in CDFG generation process. 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 nodes are not violated [6].

Integrated Framework presented in [11] (based on design flow proposed in [12]) performs an integrated temporal partitioning and mapping process to generate mappable DFGs. This framework 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*) DFGs are ones that are not mappable on the accelerator due to hardware constraints [11]. Moreover, the partitions obtained from the integrated temporal partitioning process are the same appropriate DFGs which are *mappable* on the accelerator.

Extending the CDFGs to cover hot directions of branch instructions will result in larger CDFGs. Using temporal partitioning algorithms considering the accelerator constraints is a solution to this issue. 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, 6, 12].

In [1] a temporal portioning algorithm has been presented that partitions a CDFG considering target hardware with non-homogenous architecture. 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 considers all states of the control instructions in application to convert corresponding CDFG to a set of DFGs and then it tries to reduce the number of generated DFGs. Using this algorithm the large number of DFGs may be obtained during CDFG to DFG conversion. In addition, the knowledge to different states in application is required to reduce the number of DFGs. In this section, we propose some CDFG temporal partitioning algorithms. The proposed algorithms can also be used as general CDFG temporal partitioning algorithms.

4.1 TP Based on Not-Taken Paths (NTPT)

This algorithm adds instructions from not-taken path of a control instruction to a partition until violating the target hardware architectural constraints (e.g. the number of logic resources, inputs and outputs) or reaching to a terminator control instruction. A terminator instruction is an instruction which changes execution direction of the program, e.g. procedure or function call instructions and also backward branches (to prevent cycles in CDFG). 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. 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.

4.2 Execution Frequency-Based Algorithms

In NTPT algorithm, instructions were selected only from not-taken paths of branches, whereas execution frequency of taken and not-taken instructions may be different. Our second temporal partitioning algorithm considers the execution frequency (obtained through profiling) of taken and not-taken instructions as an effective factor for selecting the instruction and adding them to the current partition. A frequency threshold is defined to determine that whether instruction is critical or not. Critical instruction means an instruction with a frequency more than the defined threshold. For a branch instruction one of its taken or not-taken instructions or both of them 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. In this case, the other list is used to create a new partition. If two lists are terminating in a unique instruction, it is attempted to add them to the current partition, so, there is no need to reconfiguration during execution of the current partition instructions.

4.3 Evaluating Proposed Algorithms

The proposed 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 required during run-time. The latter has been defined as a factor to show the efficiency of executing CDFGs on the accelerator. Efficiency factor is ratio of the number of clock cycles spent for DFG execution on the base processor to the number of clock cycles on the accelerator. Because of the space limitation we omitted the details of efficiency factor calculation. Larger amount of this factor means lower delay and correspondingly higher speedup. Six applications of Mibench [14] 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. 2). 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. According to Fig. 4, using NTPT algorithm, small number of partitions is obtained for all of the benchmark applications. We removed all small length CDFGs (SSDFGs) from the CDFG set generated by the temporal partitioning algorithms.

On the other hand, results obtained show that the NTPT algorithm has more or equivalent efficiency in comparing with frequency-based algorithm (Fig. 5). Though, the NTPT algorithm is a simpler approach for temporal partitioning, but it may bring about more efficiency comparing with the frequency-based algorithm which is more complicated. Some compilers which are used for VLIW processors move hot instructions to the not-taken part of branch instructions to avoid the pipeline flushing [9, 19]. For the applications have been modified by this kind of compliers, using NTPT algorithm is suggested. However, we do not claim that the NTPT algorithm does better for all critical portions of applications.

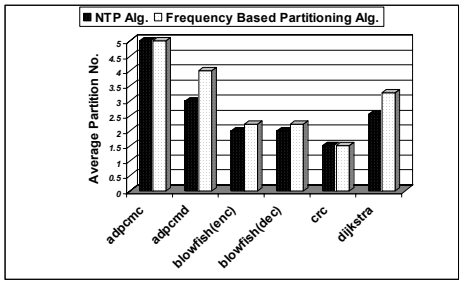

Fig. 4. Comparison of the number of partitions

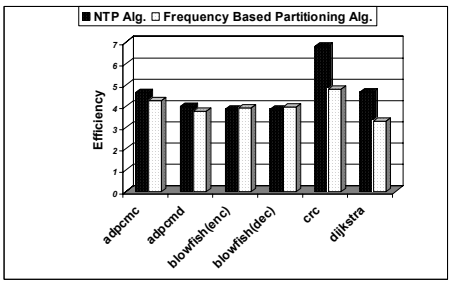

Fig. 5. Comparison of the efficiency factor

5 Case Study: Extending an Extensible Processor to Support Conditional Execution

AMBER is an extensible processor introduced in [15] targeted for embedded systems with the aim of accelerating application execution. Other tightly coupled accelerators have been proposed in [3, 4, 13, 19, 21]. The reconfigurable functional unit (RFU) in AMBER acts as an accelerator and can not support conditional execution. The basic requirements represented in Section 3 are applied for extending the AMBER's RFU to support conditional execution.

5.1 General Overview of AMBER

AMBER has been developed by integrating a base processor with three other main components [15]. The base processor is a general RISC processor and the other three components are: profiler, sequencer and a coarse grain reconfigurable functional unit (RFU). Fig. 6(a) illustrates the integration of different components in AMBER.

The *base processor* is an in-order RISC processor that supports MIPS instruction set. The *profiler* does the profiling for running applications through looking for hot portions which are usually in loops and functions. 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 functional units (Fig. 6(b)). 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 [15].

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. AMBER enters the training mode once and after detecting start addresses of the hot portions, generating configuration bit-streams for extracted DFGs and initiating sequencer tables it switches to the normal mode. In the normal mode, using the RFU, its configuration data (from configuration memory) and sequencer DFGs are executed on the RFU. More details on AMBER and its components are out of scope of this paper and can be found in [15].

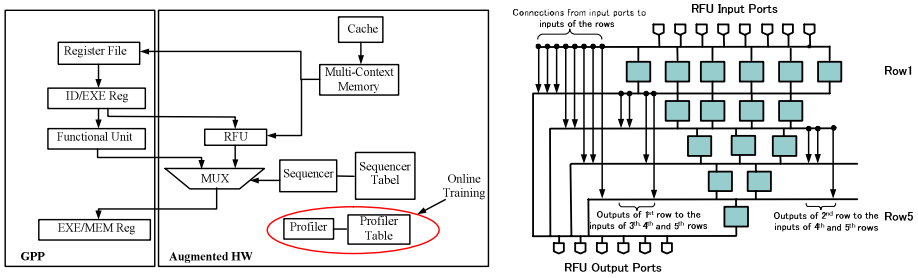

Fig. 6. Integration of main components in AMBER (a) RFU general architecture (b)

5.2 Extending AMBER RFU to Support Conditional Execution

In this section, we apply the basic requirements introduced in Section 3 to RFU used in AMBER. First, we propose conditional data selection muxes for controlling selectors of muxes used for FU inputs and outputs of the RFU. Fig. 7 shows an example of a RFU (with 5 FUs) without supporting conditional execution. On the other hand, the hardware has been modified as shown in bottom part of Fig. 7 to support conditional data selection.

In the proposed architecture, the selector signals of muxes used for choosing data for FU inputs (the Data-Selection-Mux), along with the RFU output and exit point (not shown in the figure) are each controlled by another mux (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 RFU.

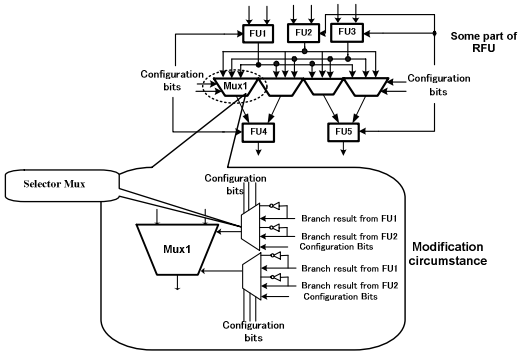

Fig. 7. Equipping the RFU to support conditional execution

For example, suppose a CDFG containing nodes 5, 6, 7, and 8 (Fig. 3) is to be mapped on the modified RFU. The second input of node 8 uses the output of node 5 when node 6 is taken otherwise uses the output of node 7. Nodes 5, 7, 6, and 8 are mapped to FU1, FU2, FU3, and FU5, respectively. 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* for 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 (node 6) is taken, *Sel1* is 0 and *Sel0* is 1, therefore the output of FU1 (node 5) is selected. When FU3 is not-taken *Sel1* is 1 and *Sel0* is 0, therefore the output of FU2 (node 7) is selected.

5.3 Performance Evaluation

The extended RFU was developed and synthesized using Synopsys tools [20] and Hitachi 0.18µm. The area of the extended RFU is 2.1 mm^2. Each CDFG needs 615 bits in total for its configuration on the RFU. 375 out of 615 bits is used for control signals. Profiling data was provided by executing applications on the Simplescalar as ISS [18]. Integrated Framework based on NTPT temporal partitioning algorithm is used to generate mappable CDFGs. The required number of clock cycles for executing each CDFG is determined according to depth of CDFG and base processor clock frequency.

We compared the effectiveness of CDFGs versus DFGs. The average number of instructions included in DFGs is 6.39 instructions and for CDFGs is 7.85 instructions. Fig. 8 shows the speedups obtained based on CDFG and DFG compared to the base processor for some applications. 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 extended RFU architecture eliminates cycles of mispredicted branches. Also, since CDFGs are longer than DFGs, more ILP can be extracted.

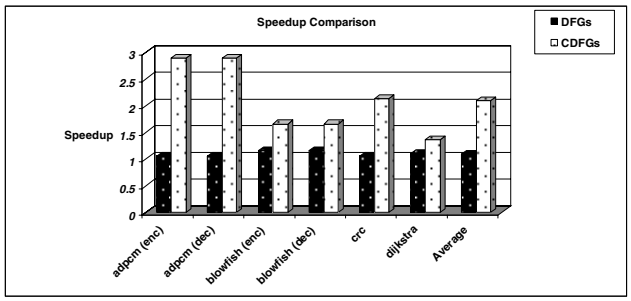

Fig. 8. Speedup comparison of acceleration approaches based on DFGs and CDFGs

6 Conclusion

In this paper, we presented motivation for handling branch instruction in DFGs and extending them to CDFGs. In addition, basic requirements for developing an accelerator featuring conditional execution were presented and some algorithms for CDFG temporal partitioning and generating executable CDFGs on the accelerator were proposed. NTPT is a temporal partitioning algorithm which tries to traverse not-taken path of the branch instructions and partitions the input CDFG. On the other hand, frequency-based temporal partitioning algorithm considers the taken and not-taken frequencies to partition input CDFG. Using this approach it is possible to add both taken and not-taken paths of a branch instruction to a partition. 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 to support the conditional execution. We used an integrated framework based on NTPT algorithm to generated mappable CDFGs on the RFU. These CDFGs are executed on the RFU to accelerate the application execution. Experimental results show the effectiveness of covering branch instructions and using CDFGs versus DFGs.

Acknowledgement

This research was supported in part by the Grant-in-Aid for Creative Basic Research, 14GS0218, Encouragement of Young Scientists (A), 17680005, and the 21st Century COE Program.

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

[3] Clark, N, Blome, J, Chu, M, Mahlke, S, Biles, S, Flautner, K. An architecture framework for transparent instruction set customization in embedded processors, Proc. ISCA (2005) 272-283

[4] Clark, N, Zhong, H, Mahlke, S. Processor acceleration through automated instruction set customization, MICRO-36 (2003)

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

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

[8] Lee, J.E, Kim, Y, Jung, J, Choi, K. Reconfigurable ALU array architecture with conditional execution, International SoC Design Conference (2004) 222-226
[9] 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
[10] Mahlke, S. A, Hank, R. E, McCormick, J.E, August, D. I, Hwu, W. W. A comparison of full and partial predicated execution support for ILP processors. In Proc. ISCA (1995) 138-150
[11] 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, Int. Conference on Embedded and Ubiquitous Computing (2006)
[12] 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
[13] Mei, B, Vernalde, S, Verkest, D, Lauwereins, R. Design methodology for a tightly coupled VLIW/Reconfigurable matrix architecture, DATE (2004) 1224-1129
[14] Mibench, www.eecs.umich.edu/mibench
[15] Noori, H, Mehdipour, F, Murakami, K, Inoue, K, Saheb Zamani, M. A reconfigurable functional unit for an adaptive dynamic extensible processor, Proc. of IEEE International Conference on Field Programmable Logic and Applications (2006) 781-784
[16] Park, J.C, Schlansker, M.S. On predicated execution. Technical Report HPL-91-58. Hewlett Packard Laboratories (1991)
[17] Razdan, R, Smith, M.D. A high-performance microarchitecture with hardware-programmable functional units, MICRO-27 (1994)
[18] Simplescalar, www.simplescalar.com
[19] Smith J.E, Sohi, G.S. The microarchitecture of superscalar P. In Proc. IEEE, Vol. 83, (1995) 1609- 1624
[20] Synopsys Inc. http://www.synopsys.com/products/logic/design_compiler.html
[21] Vassiliadis, S, Gaydadjiev, G, Kuzmanov, G. The MOLEN polymorphic processor, IEEE Transactions on Computers, Vol. 53, No. 11 (2004) 1363-1375
[22] P. Yu and T. Mitra, Characterizing embedded applications for instruction-set extensible processors, In Proc. Design Automation Conference (2004) 723-728