Generating and Executing Multi-Exit Custom Instructions for an Adaptive Extensible Processor

Noori, Hamid
Kyushu University

Mehdipour, Farhad
Amirkabir University of Technology

Murakami, Kazuaki
Kyushu University

Inoue, Koji
Kyushu University

他

出版情報：Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07, 2007-04-17. European Design and Automation Association
バージョン：

Generating and Executing Multi-Exit Custom Instructions for an Adaptive Extensible Processor

Hamid Noori† Farhad Mehdipour†† Kazuaki Murakami† Koji Inoue† Maziar Goudarzi†
†Kyushu University, Fukuoka, Japan
††Amirkabir University of Technology, Tehran, Iran
noori@c.csce.kysuhu-u.ac.jp mehdipur@ce.aut.ac.ir murakami,inoue@i.kyushu-u.ac.jp
goudarzi@slrc.kyushu-u.ac.jp

Abstract

To improve the performance of embedded processors, an effective technique is collapsing critical computation subgraphs as application-specific instruction set extensions and executing them on custom functional units. The problems of this approach are immense cost and long time of designing. To address these issues, we propose an adaptive extensible processor in which custom instructions (CIs) are generated and added after chip-fabrication. To support this feature, custom functional units are replaced by a reconfigurable matrix of functional units with the capability of conditional execution. Unlike previous proposed CIs, ours can include multiple exits. Experimental results show that multi-exit CIs enhance the performance by 46% in average compared to CIs limited to one basic block. A maximum speedup of 2.89 compared to a 4-issue in-order RISC processor, and a speedup of 1.66 in average, was achieved on MiBench benchmark suite.

1 Introduction

An effective way to enhance the performance of a processor for embedded SoCs, is adding CIs to the instruction set of the base processor. A CI encapsulates the computation of a frequently executed subgraph of the program's dataflow graph (DFG). While CIs can be effective, the time and cost of designing and verifying a base processor with added CIs causes many issues associated with designing a new processor from scratch, such as significant non-recurring engineering costs.

To mitigate the immense design cost and time of each new extensible processor, we propose an **AD**aptive **EX**tensible process**OR** (ADEXOR) in which, CIs are generated and added after chip-fabrication automatically.

Almost all methods for identifying and generating optimal set of CIs such as [1, 4, 7, 17, 20, 21] focus on CIs with a single entry and a single exit however, ours are single entry but multiple exits. Consequently, we can cover hot directions of several branches into the CI without being limited to selecting just one or all of the directions. This brings more instruction level parallelism (ILP) and can hide branch misprediction penalty. Moreover, we use a *reconfigurable functional unit* (referred in this paper as CRFU) instead of custom functional units, which enables us to implement more CIs, but results in more constraints that should be considered while generating them. In our approach, unlike other reconfigurable processors (e.g. [3, 8, 10, 11, 14, 18]) there is no need to new opcodes, new programming model or new compiler which obviate rewriting or recompiling the source codes.

Both [6] and [21] show that, by extending CIs over multiple basic blocks higher speedup can be obtained, but our approach is different. Our multi-exit CIs (MECIs) can be extended in multiple hot directions and unlike [6] can include the branch instruction itself. We have also added capability of conditional execution to the reconfigurable functional unit. In order not to increase the number of read/write ports of the register file after adding CRFU, we propose an architecture to share input/output resources between CRFU and processor functional units (Fig. 1).

2 Related work

PRISC [14], Chimaera[8], OneChip[3], MOLEN [18] and XiRisc [10] are some instances of tightly coupled integration of a general purpose processor with fine-grain programmable hardware and ADRES [11] is a sample of tightly coupled coarse-grain accelerator. Fine-grain accelerators allow for very flexible computations, but they have a long latency and reconfiguration time compared to coarse grain counterparts. Furthermore, they need a large amount of memory for storing configuration bits. ADEXOR falls in the coarse-grain category. However, we follow a quantitative approach to determine the specifications of our CRFU.

978-3-9810801-2-4/DATE07 © 2007 EDAA

Adaptive dynamic optimization systems such as Turboscalar [2], rePLay [13], PARROT [15], and Warp Processors [19] select frequently executed regions of the code through dynamic profiling, optimize the selected regions and cache/rewrite the optimized version for future occurrences. The execution of the optimized version is carried out by extra tasks sharing the main processor and/or by extra hardware. Warp Processor uses fine-grain hardware for accelerating whole loops, while Turboscalar and PARROT use wide VLIW for accelerating hot paths and traces. Hot paths and traces used in [2], [13] and [15] are longer than MECIs but contain only one direction of branches and one exit point, however our MECIs can contain both directions of a branch.

Clark et al. [5] detect hot portions of the application (limited to one basic block) using rePLay framework and execute them on a hardware accelerator. rePLay is not a good choice for our approach because it selects only one direction of branches and when both directions are hot and the branch does not bias in one direction, it terminates the hot trace. They extend their work in [6] and relax the CIs to cross over basic blocks. In this work, although a CI can include more than one basic block but it is atomic and does not include the branch instructions. To support the execution of their CIs, they modify the compiler to insert *call* instructions to the CIs before or after the branch. Code motion should be applied to the object code for supporting the misprediction of the branches. However, our multi-exit CIs are non-atomic and can include both directions of a branch. This feature can save the penalty cycles generated due to misprediction of branches. We do not need to any compiler modifications or code motions.

3 Overview of ADEXOR architecture

ADEXOR, targeted for embedded systems, is composed of four main components: *i*) a base processor which is a 4-issue in-order RISC processor, *ii*) a coarse grain reconfigurable functional unit (CRFU) whose functions and connections are controlled by configuration bits, *iii*) a configuration memory for keeping the configuration bits of the CRFU for each MECI and *iv*) counters for controlling the read/write signals of the register file and selecting between processor functional units and the CRFU (Fig. 1).

CRFU is in parallel with other functional units. It is based on a matrix of functional units (FUs) with multiple inputs and outputs. CRFU reads (write) from (to) register file. Each FU of CRFU can support all fixed-point instructions of the base processor except *multiply, divide* and *load*. It can support MECIs including at most one *store*. CRFU uses configuration memory to update the program counter (PC) and find the valid exit point, after executing each MECI. CRFU specifications are determined using a quantitative approach at *design phase*.

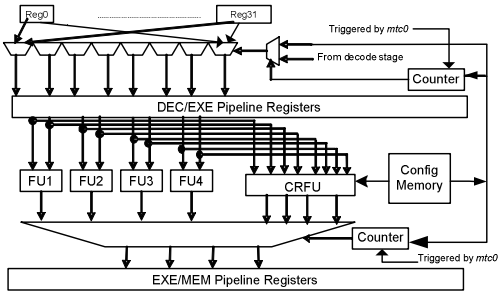

Fig. 1. Integrating the base processor with the CRFU

The counters are activated as soon as a MECI is detected. At this time the required clock cycles for executing the corresponding MECI is loaded from configuration memory into the counters. During the specified cycles the counters select the configuration bits for choosing the input and output registers of the MECI for the CRFU and also select CRFU outputs as well.

For using ADEXOR, there are two phases: *configuration phase* and *normal phase* (Fig. 2). In the configuration phase, done offline, target applications are run on an instruction set simulator (ISS) and profiled. Then, the start addresses of hot basic blocks (HBBs) are detected [22]. An HBB is a basic block with an execution frequency more than a given threshold. MECIs are generated by linking these HBBs. Mapping the MECIs to generate configuration bits for the CRFU is done in this phase. In the normal phase, the CRFU, its configuration memory and counters are employed for executing MECIs.

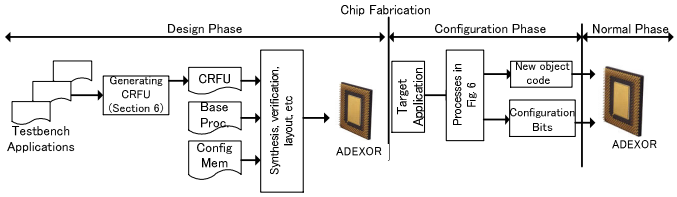

Fig. 2. Different phases for designing and using ADEXOR

4 Supporting conditional execution

In the DFG of CIs, the nodes (primitive instructions) receive their input data only from one source whereas in the control dataflow graph (CDFG) of a MECI, nodes can have multiple sources. The valid source is selected at run time according to the outcome of branches. Fig. 3 shows the control flow graph (CFG) and DFG of a part of *adpcm* loop. Node 8 gets its input from nodes 5 and 7. The result of branch in node 6 determines which one should be selected. The nodes that generate output data and exit of a MECI alter according to the outcome of branches as well. The CRFU should have some facilities to support conditional execution and generate valid output data and exit point. Predicated execution is one such technique [9].

We propose conditional data selection multiplexers

(muxes) which are added to the inputs of FUs, outputs of the CRFU and exit point selectors. Fig. 4 (a) shows a 2x2 CRFU without supporting conditional execution. In this architecture the selection bits for input muxes of FU3 and FU4 are controlled by configuration bits. To support conditional data selection, we have replaced configuration bits with *Selector-Muxs* (Fig. 4(b)).

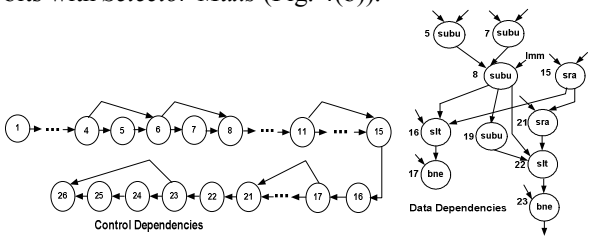

Fig. 3. CFG and DFG for a part of adpcm loop

In the proposed architecture, the selector signals of muxes used for selecting data for FU inputs (the *Data-Selection-Mux*), along with the CRFU output and exit point, are each controlled by another mux (the *Selector-Mux*). The inputs of Selector-Mux (one-bit width) come from FUs (executing branches) of the upper rows and configuration bits to be able to control the selector signals conditionally, as well as unconditionally. The selectors of Selector-Mux are controlled by configuration bits. It should be noted that the outputs of FUs are only applied to the Selector-Mux of the FUs in the lower-level rows, not the same or upper rows. Similar structure is used for selecting valid output data of the CRFU and valid exit.

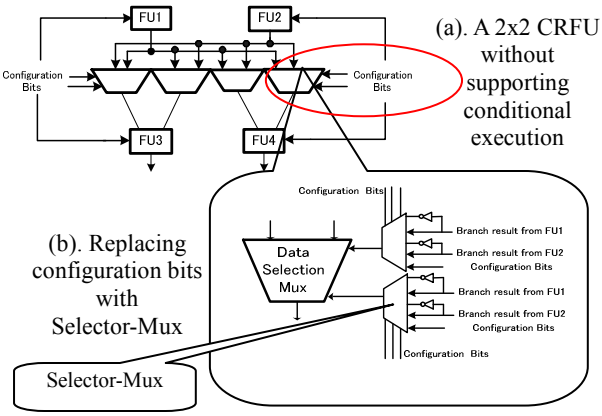

Fig. 4. Adding more hardware to the CRFU (a) to support conditional execution (b)

5 Generating multi-exit custom instructions

Fig. 6 shows the chain of main functions and tools that are used for generating MECIs. First the applications are run on the ISS and profiled. Using the profiling data, the HBBs are detected and linked to make hot instruction sequence (HIS). MECIs should not cross loop boundaries. Therefore, first we detect hot loops and sort them from the innermost loop to the outermost in the ascending order of their start addresses. To generate a HIS, the start address of the first HBB of the loop is passed and checked whether it has been covered by previous MECIs or not. If it has not been covered, the HBB is read from the object code and added to the current HIS. Reading an HBB terminates, when a control instruction is seen. Then the algorithm in Fig. 5 is applied to its last instruction:

```
1 if (it is indirect jump, return or call) then terminate HIS;
2 elsif (it is direct jump) then call recursively the function with the target address of jump;
3 elsif (it is a branch) then
3-1 if (it is hot backward) then terminate HIS;
3-2 elsif (its not-taken direction is hot) then call recursively the function with the target address of not-taken direction else terminate HIS;
3-3 if (its hot taken direction is hot) then call recursively the function with the target address of taken direction else terminate HIS;
4 else terminate HIS;
```

Fig. 5. Checking the control instruction of an HBB

This process is repeated for each new added HBB until HIS reaches to the end (terminal) points in all directions. When generating HIS is done for the loops, the process is continued for the remaining HBBs.

While generating CDFG for a HIS, we generate all possible sources for each input plus the effective branches for selecting those sources. The CDFG is passed to the MECI generator. In current implementation, each MECI can include only fixed-point instructions except *multiply*, *divide* and *load*. It can support at most one *store* instruction and up to five branches.

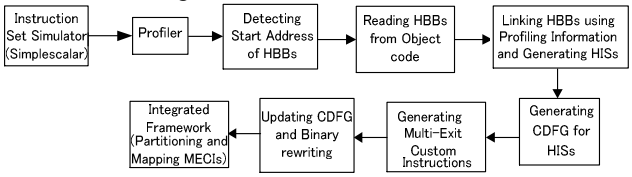

Fig. 6. Tool chain for generating MECIs

MECI generator looks for the largest sequence of instruction (subgraph) that can be executed on the CRFU, in the CDFG. Then, after checking the flow dependence and anti-dependence, *executable instructions* in each HBB are moved and added to the entry point (head) and exit point(s) (tails) of the detected largest instruction sequence (subgraph). *Executable instructions* are those instructions that can be executed by the CRFU and *non-executable* (i.e. *floating point, load, divide, multiply,* second *store*) are those that are not supported by the CRFU. It should also be checked that the region where the instructions are going to be moved in the object code, are not target of branch instructions.

For those parts that instructions are moved, the object code is rewritten (Fig. 7). Moving instructions should be

limited inside a basic block. In current version, a MECI can have up to four exit points. The exit points of a MECI are: *i) branch* with only one hot direction, *ii) indirect jump* and *return*, *iii) call*, *iv)* hot backward *branch* and *v)* an instruction where its next instruction is *non-executable*. Sometimes more than one MECI can be extracted from a HIS. MECIs with less than 5 instructions are rejected. Corresponding to the instructions movement in the object code, CDFG should be updated as well. Then it is passed to the integrated framework to partition large MECIs and map generated MECIs on the CRFU. The integrated framework and mapping tools are the extension of our previous work [23] for CDFGs instead DFGs.

To support the execution of MECIs on the CRFU in the normal phase, the entry instruction of the subgraph of each MECIs is over written by *mtc0* (move to coprocessor) instruction in the object code (Fig. 7). In the normal phase when the *mtc0* is decoded, its operand is used for indexing and loading configuration bits from configuration memory of the CRFU for the corresponding MECI.

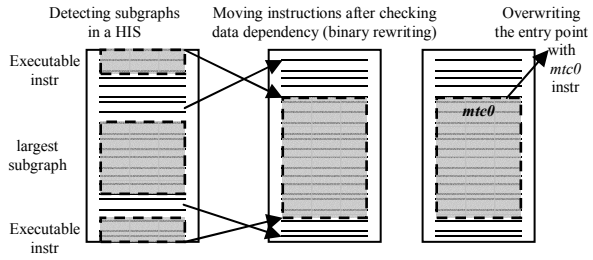

Fig. 7. Generating a MECI

6 Proposed architecture for the CRFU

The tool chain in Fig. 6 was used for our quantitative approach to determine the CRFU architecture.

Our simulation environment is based on Simplescalar (PISA configuration) [16]. 22 applications of Mibench [12] were selected as inputs for our quantitative approach. In this paper, we use *mapping rate* frequently by which we mean the percentage of generated MECIs for 22 applications that can be mapped on the CRFU. We have considered the execution frequency of MECIs while calculating the mapping rate. All 22 applications of Mibench executed to completion. Since execution time varies for each application, a weight was assumed for each of them so that the product of the execution time and the weight is equal for all.

The graphs in Fig. 8 and 9 show the mapping rate of 22 applications for different numbers of input, output and FUs. These diagrams show that setting the number of inputs, outputs, and FUs to respectively 8, 6, and 16 results in a mapping rate of 88.21%.

Two other parameters that should be determined for the CRFU are its depth and width. Width (number of columns) and depth (number of rows) show the maximum instructions that can be executed in parallel and the length of the critical path in MECIs (depth of MECIs), respectively. Six and five were selected for the width and depth of the CRFU, respectively. Measuring the mapping rate for different numbers of FUs in each row shows that 6, 4, 3, 2 and 1 are proper numbers for the first to fifth rows.

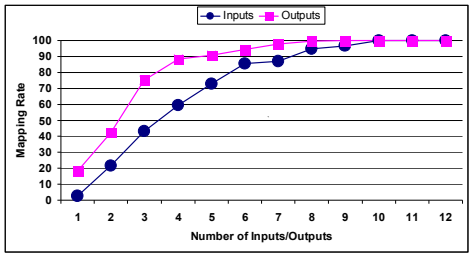

Fig. 8. The effect of different number of inputs, outputs on the mapping rate for 22 applications

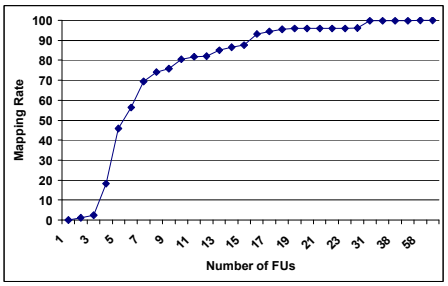

Fig. 9. The effect of different number of FUs on the mapping rate for 22 applications

The same above approach was repeatedly employed to find the number of inputs connected to each row as well as the number of connections among different rows. The final architecture of CRFU is shown in Fig. 10. Considering all of these constraints, the mapping rate decreases to 81.43%.

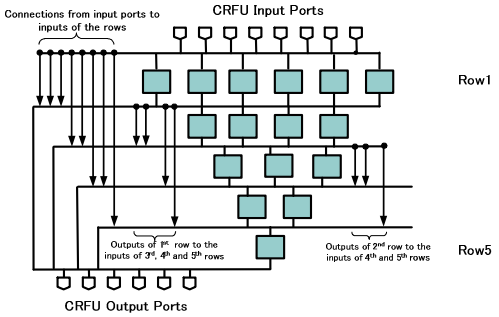

Fig. 10. Proposed architecture for the CRFU

The 8 input ports have been replicated and distributed among different rows to facilitate data accesses (7, 3, 2, 2, and 1 for Row1 to Row5, respectively). In the CRFU, the output of each FU in a row can be used by all FUs in the subsequent row (connections with length one). Besides, there are four connections with length two, two

connections with length three and one connection with length four. The Selector-Muxes (Section 4) are available for Row3, Row4 and Row5 (not shown in the Fig. 10).

Experiments show that all FUs do not need to support all functions. Table 1 shows the distribution of functions among rows according to the experimental results.

Table. 1. Number of required function in each row

	Row1	Row2	Row3	Row4	Row5
and	2	3	1	1	0
or	1	2	1	1	1
xor	2	1	1	1	0
nor	1	0	0	0	0
add/sub	5	4	3	2	1
shift	4	3	2	2	1
compare	2	2	2	2	1

The VHDL code of the proposed architecture was developed and synthesized using Synopsys tools and Hitachi 0.18 µm. The area of the CRFU is 2.1 mm^2. Since each FU output can be accessed directly via the output ports of the CRFU and also the *depth* (length of critical path in the DFG) of each MECI is known after mapping, we can have a CRFU with variable latency which depends on the depth of each MECI. The delay of the CRFU for MECIs with various depths from 1 to 5 are **2.2 ns, 4.2 ns, 6.1 ns, 7.9 ns** and **9.8 ns**, respectively. The required clock cycles for executing each MECI is determined according to these numbers, depth of DFG and base processor clock frequency. The CRFU needs 375 bits for control signals and 240 bits for immediate values and exit points. So each MECI needs 615 bits in total for its configuration.

In order not to increase the number of read/write ports of the register file, the inputs and outputs of processor functional units (FU1 to FU4) are shared with the CRFU (Fig. 1). In a conventional processor, the signals for reading/writing registers are generated by decode stage. Here, two signals control the ports: *i*) signals from the decode stage and *ii*) configuration bits. When the *mtc0* is detected, its operand is used for indexing configuration memory. Then specified number of cycles for executing the corresponding MECI is loaded to counters as well as CRFU configuration. The counters activate the selectors of muxes and select configuration bits for enabling input and output registers for the CRFU and select CRFU output simultaneously, during specified cycles. Fig. 1 shows that CRFU has four outputs but we mentioned that it needs 6 outputs. To support CRFU with six outputs without adding more write ports to the register file, we added two registers to the CRFU. When a MECI has more than four outputs, extra writes are stored in these registers, four of them are done in one cycle and the remaining ones in the next cycle.

7 Experiment results

The configuration of the base processor is in Table 2.

Table. 2. Base processor configuration

Issue	4-way
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

To see the effectiveness of MECIs compared to CIs limited to one HBB, we limited the MECI generator to one HBB and regenerated the CIs. Then we redesigned the CRFU using the same quantitative approach. The number of inputs, outputs and FUs are the same as before, but it has simpler connections and FUs and does not support conditional execution. The area of CRFU reduces to 1.15 mm^2 and its delay for a CI with a critical length of five is 7.66 ns. Each CI configuration needs 512 bits. The average number of instructions included in CIs (one HBB) is 6.39 instructions and for MECIs is 7.85 instructions. Fig. 11 shows the speedups obtained by MECIs and CIs compared to the base processor for some applications.

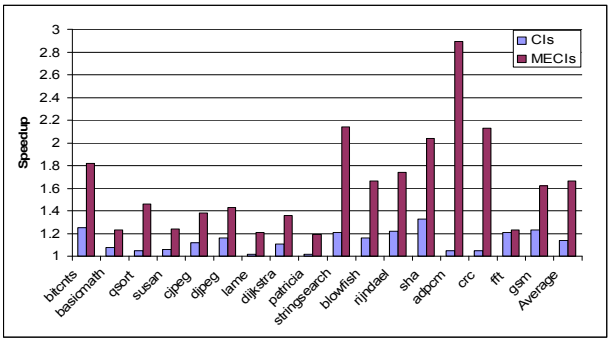

Fig. 11. Speedup for CIs (one HBB) and MECIs compared to base processor (clock freq 300 MHz)

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 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 MECIs include both directions, and hence, the CRFU architecture eliminates cycles of mispredicted branches. Also, since HBBs are linked and longer MECIs are generated, more ILP can be extracted. For applications like *basicmath, susan, lame, patricia,* and *fft* that most of the dynamic instructions are floating point, multiply, divide and load (69%, 45%, 79%, 44% and 57%, respectively), the speedup enhancement is less than average which is already expected.

To see the effect of base processor clock frequency on

the speedup obtained using MECIs, five different frequencies were tried (Fig. 12). The speedup diminishes in higher frequencies since clock period becomes smaller but the CRFU delay remains unchanged.

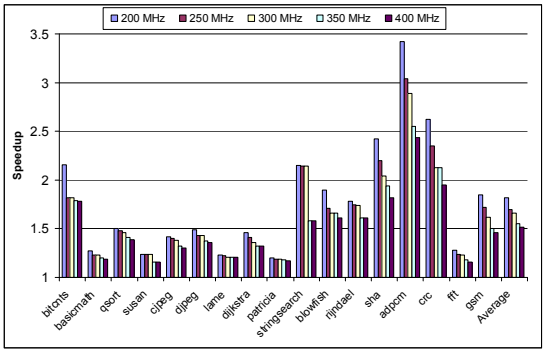

Fig. 12. Effect of the clock frequency on speedup

To measure the required time for the configuration phase, the tool chain in Fig. 6 was run on a Pentium 4 (CPU 3.6 GHz) with 1GB main memory. For the most complex benchmark, it takes less than 9 minutes (Table 3). For the applications with larger number of HBBs and branches, profiling is longer.

Table. 3. Execution time of offline configuration

Application	Exec. time (Seconds)	Application	Exec. time (Seconds)
adpcm	225	gsm	461
bitcounts	331	lame	526
blowfish	94	patricia	84
basicmath	34	qsort	233
cjpeg	75	rijndael	68
crc	132	sha	29
dijkstra	101	stringsearch	3
djpeg	9	susan	122
fft	36	Average	150.8

To see the effect of connections of the CRFU with length more than one, all of them were deleted and only connections with length one were kept. In this architecture for passing data from one FU to another FU in a non-subsequent row, or for passing data of input ports to FUs placed in rows other than the first row, *move* instructions are inserted in the intermediate FUs (similar to [5, 6]). We applied MECIs generated for the proposed CRFU to this architecture. 24.2% of the MECIs could not be mapped due to the limitation of number of FUs.

8 Conclusions

We have presented an approach for generating and executing custom instructions including multiple basic blocks. These custom instructions can include branch instructions and have multiple exit points. We architected a reconfigurable functional unit with conditional execution capability to support the execution of these custom instructions. Our experimental results show that by extending custom instructions over multiple HBBs the average speedup increases by 46% compared to the custom instructions which are limited to only one HBB. This is achieved in return for 83% more hardware and 20% more configuration bits. Utilizing connections with different length are helpful for supporting larger custom instructions with the available number of FUs.

9 Acknowledgments

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] K. Atasu et al, Automatic application-specific instruction-set extension under microarchitectural constraints, *DAC* 2003.
[2] Black, and J. Shen, Turboscalar: A High Frequency High IPC Microarchitecture, *ISCA 2000.*
[3] J. E. Carrillo E. and P. Chow, The effect of reconfigurable units in superscalar processors, *Proc. of the 2001 ACM/SIGDA FPGA*, 2001.
[4] N. Clark et al., Processor acceleration through automated instruction set customization, *MICRO-36*, 2003.
[5] N. Clark et al, Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization, *MICRO-37,* 2004.
[6] N. Clark et al., An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors, *ISCA 2005.*
[7] D. Goodwin et al., Automatic generation of application specific processors, *CASES 2003.*
[8] S. Hauck et al., The Chimaera reconfigurable functional unit, *IEEE Symp. FPGAs for Custom Computing Machines,* 1997.
[9] J. Lee et al., Reconfigurable ALU Array Architecture with Conditional Execution, *International SoC Design Conference, 2004.*
[10] A. Lodi et al., A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications, *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1876–1886, 2003.
[11] B. Mei et al., Design Methodolody for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study, *DATE 2004.*
[12] Mibench, www.eecs.umich.edu/mibench
[13] S. Patel et al., rePLay: A Hardware Framework for Dynamic Optimization, *IEEE Trans.on Comp.*, vol. 50. no. 6, 2001.
[14] R. Razdan et al., A high-performance microarchitecture with hardware-programmable functional units, *MICRO-27*, 1994.
[15] R. Rosner et. al, Power Awarness through Selective Dynamically Optimized Traces, in *Proc. ISCA-31*, 2004.
[16] Simplescalar, www.simplescalar.com
[17] F. Sun et al., Synthesis of custom processors based on extensible platforms, *ICCAD 2002.*
[18] S. Vassiliadis et al., The MOLEN Polymorphic Processor, *IEEE Transactions on Computers*, vol. 53, no. 11, Nov. 2004.
[19] Warp Processors, http://www.cs.ucr.edu/~vahid/warp/
[20] P. Yu et al., Scalable Custom Instructions Identification for Instruction-Set Extensible Processors, *CASES 2004.*
[21] P. Yu and T. Mitra, Characterizing Embedded Applications for Instruction-Set Extensible Processors, *DAC 2004.*
[22] H. Noori et al., A General Overview of an Adaptive Dynamic Extensible Processor, *Workshop on Introspective Architectures*, 2006.
[23] F. Mehdipour et al., Custom Instruction Generation Using Temporal Partitioning Techniques for a Reconfigurable Functional Unit, *Int. Conference on Embedded and Ubiquitous Computing*, 2006.