2006－ＡＲＣ－168（13）
2006／6／9

A Reconfigurable Functional Unit for Adaptable Custom Instructions

Hamid Noori[†] Farhad Mehdipour[§] Kazuaki Murakami[†] Koji Inoue[†]

and Morteza SahebZamani[§]

[†]Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-1 Kasuga-koen, Kasuga, Fukuoka, 816-8580, Japan

[§]Computer Engineering and Information Technology Department, Amirkabir University of Technology, #424 Hafez Ave., Tehran, Iran

E-mail: noori@c.csce.kyushu-u.ac.jp, {murakami, inoue}@i.kyushu-u.ac.jp, {mehdipur,szamani}@aut.ac.ir

Abstract This paper presents a reconfigurable functional unit (RFU) for an adaptive dynamic extensible processor. The processor can tune its extended instructions to the target applications, after chip-fabrication, which brings about more flexibility. The custom instructions (CIs) are generated deploying the hot basic blocks during the training mode. In the normal mode, CIs are executed on the RFU. A quantitative approach was used for designing the RFU. The RFU is a matrix of functional units with 8 inputs and 6 outputs. Performance is enhanced up to 1.5 using the proposed RFU for 22 applications of Mibench. The size of configuration memory has been reduced by 40% through making the RFU partially reconfigurable, finding subsets of CIs and merging small CIs into one configuration. This processor needs no extra opcodes for CIs, new compiler, source code modification and recompilation.

Keyword Extensible Processor, Hot Spots, Online Profiling, Custom Instruction, Reconfigurable Accelerator

1. Introduction

ASICs, general purpose processors (GPPs), ASIPs, and extensible processors are various approaches for designing embedded systems. Although ASICs have higher performance and lower power consumption, they are not flexible and have an expensive and time consuming design process. For GPPs, although availability of tools, programmability, and ability to rapidly deploy them in embedded systems are good reasons for their common use, usually they do not offer the necessary performance. ASIPs are more flexible than ASICs and have more potential to meet the challenging high-performance demands of embedded applications, compared to GPPs. However, the synthesis of ASIPs traditionally involved the generation of a complete ISA for the targeted application which is too expensive and has long design turnaround time.

Another method for providing enhanced performance is application-specific instruction set extension. In this method, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to custom functional units. Instruction set extension improves performance and also maintains a degree of system programmability, which enables them to be utilized with more flexibility. The main problem with this method is that there are significant non-recurring engineering costs associated with their implementation.

In our approach, an Adaptive dynaMic extensiBlE processoR (AMBER) is presented in which the CIs are adapted to the target applications and generated after chip-fabrication, fully transparently and automatically. This approach reduces the design time and cost drastically. Our CIs are generated by exploiting the HBBs. An HBB is a basic block that is executed more than a given threshold. A basic block is a sequence of instructions that terminates in a control instruction. We propose an RFU architecture to support a wide range of generated CIs. Our 8-input, 6-output RFU is a coarse grain accelerator based on a matrix of functional units (FUs). It is tightly coupled with the base processor. In this method, there is no need to add extra opcodes for CIs, develop a new compiler, change the source code and recompile it.

In Section 2, we highlight some related work. The general overview of AMBER is described in Section 3. Section 4 discusses our quantitative approach for designing RFU and the proposed architectures. Section 5 covers the integration of RFU and the base processor. Section 6 overviews the configuration memory. Performance evaluation results can be found in Section 7 and finally the paper is closed by conclusions and future work.

2. Related Work

PRISC[1], Chimaera[2], OneChip[3], XiRisc[4] and MOLEN [5] are some instances of tightly coupled integration of a GPP with fine-grain programmable hardware and ADRES [6] is a sample system with coarse-grain hardware. Fine-grain accelerators allow for very flexible computations, but there are several drawbacks to using them. They have long latency and reconfiguration time. Furthermore, they need a large amount of memory for storing configuration bits. To overcome the computational inefficiency and reconfiguration latency, most of them deal with very large

subgraphs. This work differs in that we focus on acceleration at finer granularity.

For loosely coupled systems like Garp[7] and MorphoSys [8], there is an overhead for transferring data between the base processor and the coprocessor. For tightly coupled designs, data transferring takes less time but adding RFUs usually demands for more register file read/write ports. Chimaera adds a shadow register to solve this issue. In our case, we share the input/output resources between them.

All of these designs require a new programming model, a new compiler, new opcodes for new instructions, source code modification or recompilation. In our approach, we do not encounter these issues. The user just runs the applications on the base processor, and then, generation of custom instructions and handling their execution are done transparently and automatically.

Adaptive dynamic optimization systems such as Turboscalar [9], rePlay [10], PARROT [11], and Warp Processors [12] 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 on by extra tasks sharing the main processor and/or by extra hardware. To overcome the overhead of dynamic optimization, we have defined two modes for our processor.

The similar design to ours has been proposed by Clark [13]. However, we use different methods for profiling and generating, mapping and handling execution of CIs. Our RFU is not integrated like other functional units. It shares the available read/write ports. By applying some modifications in the routing resources and locations of inputs, our RFU can handle more CIs. In addition we tried to go for more details such as structure and size for the configuration memory of our RFU.

3. General Overview of AMBER Architecture

By *adaptive* we mean that the processor can tune its extended instructions to the target applications. Moreover, we claim it is *dynamic*, because instruction set extension is done based on the profiling of dynamic code. AMBER has been designed and developed by integrating three main components to the *base processor*, namely *profiler*, *RFU* and *sequencer*. The base processor is a 4-issue in-order RISC processor that supports MIPS instruction set.

Profiling of running applications is done by the *profiler* through monitoring the program counter (PC). In every clock cycle, the profiler compares the current value and the previous value of the PC. If the difference of these two values is not equal to the instruction length, a taken branch or jump has occurred. The profiler has a table with a counter for each entry (start address of basic blocks) to keep the execution frequency of basic blocks. In the case of a taken branch or a jump, the profiler table is checked. If the target address (current PC) is in the table, the corresponding counter is incremented; otherwise it is added as a new entry and its counter is initialized to one. Using the profiler table, start addresses of HBBs can be obtained.

RFU is a matrix of functional units (FUs) plus a configuration memory. According to the size of data in the processors, a matrix of FUs seems efficient and reasonable enough for accelerating dataflow subgraphs as CIs. Each CI updates the PC after its execution finishes, considering original sequence execution, so that the processor can continue from the correct address.

The *sequencer* mainly determines the microcode execution sequence by selecting between the RFU and the processor functional unit. It has a table in which the start addresses of CIs in the object code are specified. The table of the sequencer is initialized according to the locations of the CIs in the object code in training mode. The sequencer monitors the PC and compares it to its table entries. When it detects that a CI is going to be executed, it switches from processor functional unit to the RFU, waits for specified clock cycles and lets the RFU finish the execution of the CI and then again switches to the processor functional unit.

AMBER has two operational modes: training mode and normal mode (Fig. 1). In the training mode, applications are run on the base processor and profiled. Then, the start addresses of HBBs are detected. Using these addresses, HBBs are read from the object code. CI generation has been limited to one HBB. During CI generation, some changes may be applied to the object code and therefore, some parts of the object code may be rewritten. Generating configuration data for RFU and initializing sequencer table is done in this mode. When these processes are done, the processor switches to the normal mode.

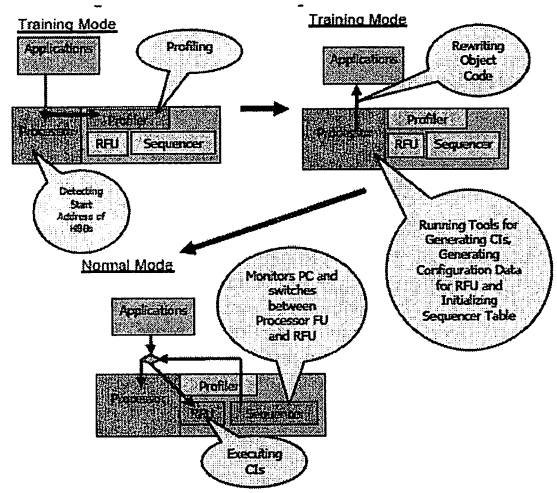

Fig. 1. AMBER operational modes

Training mode can be done online or offline. In the case of online training, profiler needs some hardware. In this case, all processes (detecting HBBs, generating CIs, etc) are performed on the base processor. For offline training, a simulator (e.g. Simplescalar [16]) is needed. All processes are executed on the host machine.

In the normal mode, using the RFU, its configuration data, sequencer and its table, the CIs are executed on the RFU. For more details on AMBER, refer to [1].

4. RFU Architecture: A Quantitative Approach

In this section, we explain our tool flow. Also, CI generation method and mapping algorithm are described and finally, the proposed architectures for RFU are presented.

4.1. Tool Chain for Quantitative Approach

We followed a quantitative approach by applying the tool chain of Fig. 2 for designing RFU, using 22 applications of Mibench [15]. Simplescalar was utilized as our simulator. The simulator was modified to generate a trace of taken branches and jumps as an input for the profiler for detecting start addresses of HBBs [1]. For each HBB start address, its corresponding basic block is read from the object code. Reading the HBBs terminates when encountering a control instructions. Then DFG is generated for each HBB and passed to the CI generator tool. The mapping tool receives the optimized CIs and maps them on RFU. The results of mapping tool guide us to RFU architecture.

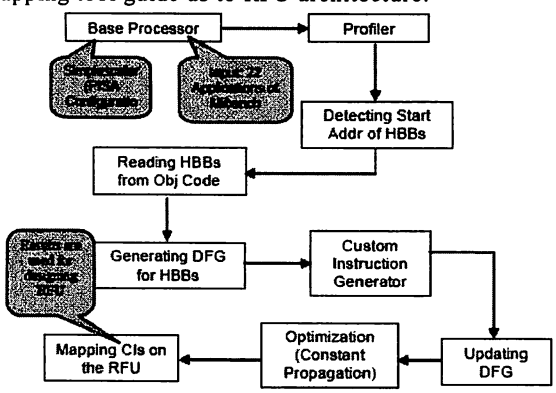

Fig. 2. Tool Chain

Our CI generator, mapping tool and RFU were developed in two phases. In the first phase, we assumed some primary constraints for both CIs and RFU. CIs were generated and mapped on RFU considering these constraints. We concluded a proper architecture for RFU, by analyzing the feedbacks resulted from mapping. After finalizing the RFU, an integrated temporal partitioning and mapping framework was developed for generating CIs. The details of the framework are out of the scope of this paper.

Primary constraints for generating CIs are: a) supporting only fixed-point instructions excluding *multiply*, *divide* and *load* and b) including at most one *store* and at most one *control* instructions. Multiply and divide were excluded due to their low execution frequency and large area required for hardware implementation and *loads* were ignored because of the cache misses and long memory access time, which makes the execution latency unpredictable.

As the primary constrains for the RFU, a matrix of FUs which can support only fixed-point instructions of the base processor was assumed without any limitations on the number of inputs, outputs (I/O) and FUs. The output of each FU was supposed to be used by the neighbors in the same row and by all other FUs in the lower level rows.

Our CI generation algorithm does not need to be complicated for two reasons. First, the length of HBBs is usually between 10 to 40 instructions, and so they are not too big to need a complicated algorithm. Secondly, in the online training mode, the CI generator is going to be executed on the base processor.

Our CI generator receives the DFG of each HBB as an input and then looks for the longest sequence of instructions that can be executed on the RFU. After checking the flow dependence and anti-dependence, more instructions are added to the head and tail of the detected instruction sequence by moving *executable instructions* in the object code. *Executable instructions* are those instructions that can be executed by the RFU. It is also checked whether or not the areas where the instructions are going to be moved, are the target of branch instructions. For those parts of the object code where instructions are moved, the object code is rewritten, if these conditions are met. In this phase we try to make CI as large as possible. Sometimes HBBs are so large that more than one CI may be extracted. This process is repeated until all nodes are covered or no CI longer than five can be generated. Our CI generator ignores CIs with less than five instructions.

Mapping is the appropriate positioning of DFG nodes on FUs. Assigning instructions of CI or DFG nodes to FUs is done based on the priority of nodes. The nodes assigned lower value of ASAP (As Soon As Possible) [2] have to be executed earlier. ASAP represents the execution order of nodes according to their dependencies. After calculating ASAP of each node, a preliminary mapping of nodes is done, starting with lower level nodes to higher level nodes. Using this simple algorithm does not guarantee the minimum connection length between nodes.

To overcome this issue, after the initial mapping, nodes are moved to other FUs to achieve shorter connections. We restrict the moving scope of nodes to its connections bounding box. Slack [2] and position of parents and descents of each node determine the boundary of moving to other FUs. Slack of each node represents its criticality. The nodes with zero slack value are on a critical path and can not be moved to other FUs. A node with slack value equal to 1 can only move to one row above or below. This node has to be located under or at the same row of its parents and above or at the same row of its descents. By moving nodes to unoccupied locations at the scope of its bounding box, appropriate locations in terms of minimum connections criterion can be found.

4.2. Proposed Architectures for RFU

In this paper, *mapping rate* is defined as the percentage of generated CIs that can be mapped on the RFU for 22 applications of Mibench. We have considered the execution frequency of CIs for measuring the mapping rate as well. All 22 applications were executed till completion. Because execution time varies for each application, for a fair comparison, a weight was assumed for each so that the production of execution time and

weight is equal for all.

To determine the proper numbers for RFU inputs and outputs, we mapped our generated CIs on the RFU without considering any constraints. The curves in Fig. 3 show the mapping rate for different numbers of inputs and outputs.

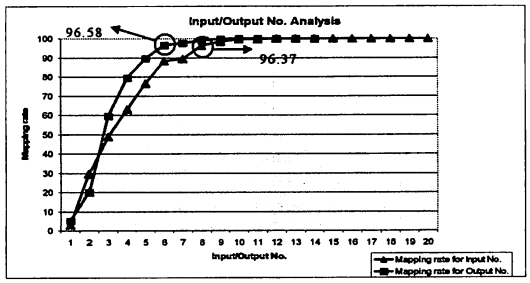

Fig. 3. Mapping rate for different numbers of I/O

According to the results, eight and six are good candidates for the number of inputs and outputs, respectively. The graph becomes almost flat after these numbers. To find the appropriate number for FUs, we similarly measured the mapping rate for various numbers of FUs. The measurement was done for two cases. Once it was done for CIs that meet our I/O constraints obtained from last experiments, and in the second case, we did not assume any limitation. The two graphs in Fig. 4 show that 16 is a good candidate. The curve marked by triangles, which shows the mapping rate without considering any constraints, illustrates that most of the remaining CIs are so large that even 35 FUs are not enough for executing them. The curve dotted by square symbols, depicts that by 16 FUs 99.84% of CIs that meet the I/O constraints can be handled by RFU.

We continued similar procedure to specify the width and depth of RFU. Experimental results specify that 6 and 5 are appropriate for width and depth, respectively. By adding the width and depth constraints to previous constraints, the mapping rate will reduce from 94.74% to 93.51%.

However, we have 16 FUs that should be laid in a 6x5 matrix. Measuring the mapping rate for different numbers of FUs in each row shows that 6, 4, 3, 2 and 1 for first to fifth rows, respectively, are proper candidates. By adding these new constraints, the mapping rate reaches to 92.28%. However, in this architecture, we have assumed that the inputs of the RFU can be accessed by any FU directly and there are direct connections from the output of each row to the input of other lower rows. Moreover, each FU can have inputs (outputs) from (to) the left and right FUs.

To make the architecture more realistic, we assumed that all inputs are applied only to the first row. We also limited the number of connections. The outputs of each row can be used only by all FUs in the subsequent row. All connections to and from neighboring FUs were deleted. In this architecture, for transferring input data to rows below the first row, or transferring the output of one row to the input of FUs in a non-subsequent row, *move* instructions should be inserted on the intermediate FUs. With these limitations, the mapping rate decreases to 77.53%.

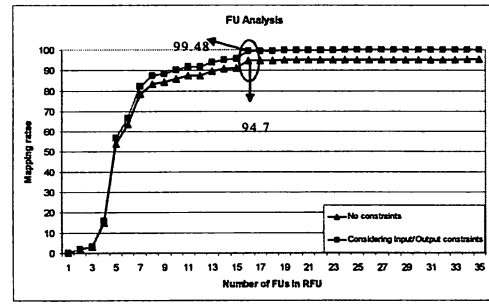

Fig. 4. Mapping rate for different number of FUs

To improve the mapping rate, we examined many different configurations and structures. According to the mapping rate results, we reached to the following architecture depicted in Fig. 5.

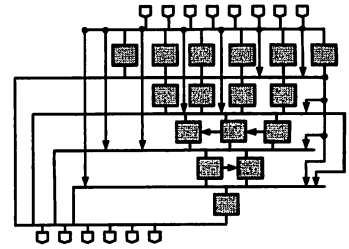

Fig. 5. Optimized RFU architecture

In this architecture, to facilitate data accessing for FUs and reduce the inserted move instructions (which occupy FUs), four other longer connections were added to the base connections. Base connections connect the output of each row to the inputs of subsequent rows. These four longer connections connects row 1 to rows 3, 4 and 5 and row 2 to row 4. We also distributed the input ports among rows. 7, 2, 2, 2, 1 are the number of inputs for the first to fifth rows, respectively that can facilitate access to inputs directly for all rows. The number of inputs for the RFU is 8 and these 14 inputs are generated by replicating the main 8 inputs. In the third and fourth rows, three uni-directional connections to the neighboring FUs, were added to support CIs with critical path longer than 5.

Experiments show that each FU of RFU does not need to support all the operations. We defined three types of operations: logical operations (type 1), add/sub/compare (type 2) and shift operations (type 3). Distribution and the number of operation of each type for each row are given in Table 1. Considering all the constraints for the second proposed architecture, the mapping rate increases to 90.48% which is almost 13% better than that for the first architecture. Each configuration needs 308 bits for control signals and 204 bits for immediate values.

Table 1. Number of required functions in each row

Row No.	Type 1	Type 2	Type 3
1	2	6	4
2	3	3	2
3	1	3	2
4	1	2	1
5	1	1	0

5. Integrating RFU with the Base Processor

Fig. 6 depicts how the RFU is connected to the base processor. The I/O ports of the processor functional units have been shared by the RFU. Using this technique, there is no need to add more read/write ports to the register file.

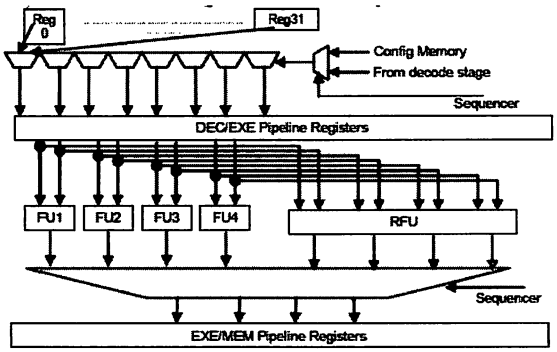

Fig. 6. Integrating RFU with the base processor

In a conventional processor, the signals for reading registers are generated by the decode stage. In this design, two signals control reading the register file, one comes from decode stage and the other from configuration bits. Fig. 6 shows four outputs for RFU whereas we had mentioned that RFU had 6 outputs. To support RFU with six outputs without adding write port to the register file, we added two registers to the RFU. When the custom instruction has more than four outputs, extra write values are registered. Four of them are written in one cycle and the remaining ones in the next cycle. Therefore, for CIs with more than four outputs, the execution takes one more cycle.

6. Configuration Memory

Two techniques were used to reduce the size of configuration memory: similarity detection and merging of CIs. Two CIs can be similar according to their: nodes (FUs) plus connections, inputs, outputs and immediates. In most cases, the configuration bits related to FUs and connections for CIs are the same but the inputs, outputs or immediates are different. Moreover, detection of FUs and connections similarity is done in a different manner. Two FUs and connections similarity types were defined: *complete* and *subset similarities*. Two CIs are completely similar if their functionality of nodes (FUs) and connections are the same and they have subset similarity if one of the CIs is completely similar to a subset of another CI considering the nodes and connections similarities.

To support similarity detection, 512 bits of CI configuration data was divided into four parts; 155 bits for the selecting operations of FUs plus selectors of muxes (connections) (P_1), 92 bits for selecting inputs (P_2), 61 bits for outputs (P_3) and 204 bits for immediate values (P_4). Two CIs have similar inputs, outputs or immediates if their P_2, P_3 and P_4 are exactly the same, respectively. By generating one configuration bits for similar P_1, P_2, P_3 and P_4, the size of configuration memory is decreased.

Experiments show that the percentage of unused space of RFU for 22 applications is almost 63% without considering the similarity of CIs. According to this observation, we tried to reduce unused space of RFU by merging small CIs of each application.

Fig. 7 includes the first 7 attempted applications with the largest number of CIs. For each application, the leftmost bar in Fig. 7 shows the number of initial generated CIs and the second to fifth specify the number of required P_1, P_2, P_3 and P_4, respectively, after detecting similarities and merging CIs. By this method, the size of configuration memory was decreased from 7.4 KB to 4.4 KB which means 40% improvement.

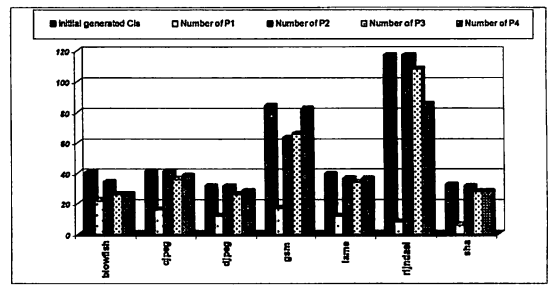

Fig. 7. Number of P_1, P_2, P_3 and P_4 compared to initial CIs

7. Performance Evaluation

Simplescalar was used as our simulator framework. As for the base processor, we assumed a 4-issue in-order RISC processor supporting MIPS instruction set with 32KB L1 data cache (1 cycle hit), 32KB L1 instruction cache (1 cycle hit), 1MB unified L2 cache (6 cycle hit), 64 RUU size and 64 fetch queue size. We assumed a variable delay for our RFU which depends on the length of the critical path after mapping a CI on the RFU. We developed the VHDL code of RFU and synthesized the code with Synopsys tools using Hitachi 0.18μ technology. The area of RFU is 1.1534 mm^2. Table 2 shows the delay of RFU for CI with different length.

Figure 8 shows the obtained speedup for four base processors with different clock frequencies (200, 300, 400 and 500 MHZ). For programs like *lame* and *patricia* in which 53% and 22% of dynamic instructions are floating point and 26% and 22% are load, respectively, there was not much opportunity for generating effective CIs. Other applications like *qsort* and *adpcm* have small HBBs so that either they were rejected or small performance improvement could be obtained using them. In *susan* and *gsm*, multiplication covered 13% and 7% of dynamic instructions. For applications with higher speedup such as *sha*, *stringsearch*, *rijndael* and *gsm*, CIs covered high percentage of dynamic instructions namely are 41%, 51%,

45% and 32%, respectively.

Table 2. RFU Delay for CI with different length

CI Length	RFU Delay (ns)
1	1.38
2	2.28
3	3.12
4	4.89
5	6.47
6	7.57
7	8.65
8	9.66

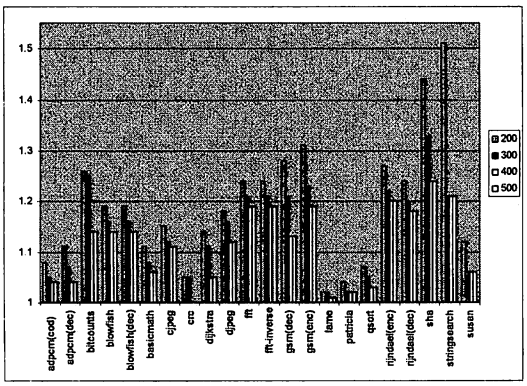

Fig. 8. Speedup for four different clock frequency

8. Conclusions and Future Work

Using a quantitative approach, we proposed an RFU for an adaptive dynamic extensible processor. The RFU has 8 inputs and six outputs with 16 FUs. By adding few longer connections between rows and facilitating the input access we could improve the mapping rate by 13%. Furthermore, the configuration memory was reduced by making the RFU partially reconfigurable, generating one configuration for similar/subset CIs and merging small CIs. The performance improvement was up to 25% and the average speedup was 1.10. As expected, when the dynamic instructions are covered more by CIs, the speedup is higher. To increase the speedup for more applications, CIs and the RFU can be extended to support multiplications and floating point instructions. Another solution, which is our next step, is relaxing CIs over HBBs.

Acknowledgment

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

Reference

[1] R. Razdan, and M. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. MICRO-27*, Nov. 1994, pp. 172–180.

[2] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera reconfigurable functional unit," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, Apr. 1997, pp. 87–96.

[3] J. E. Carrillo, and P. Chow, "The effect of reconfigurable units in superscalar processors," in *Proc.of the 2001 ACM/SIGDA FPGA*, 2002, pp. 141–150,.

[4] 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.

[5] S. Vassiliadis, et al., "The MOLEN Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, Nov. 2004.

[6] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereinsg, "Design Methodolody for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study," in *Proc. Design, Automation and Test in Europe*, 2004, pp. 1224-1229.

[7] J. R. Hauser, and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable processor," in *IEEE Symp. on FPGAs for Custom Computing Machines*, Apr. 1997, pp. 12-21.

[8] M. H. Lee, et al., "Design and implementation of the MorphoSys Reconfigurable Computing Processor," *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*, pp. 147-164, Mar. 2000.

[9] B. Black, and J. P. Shen, "Turboscalar: A High Frequency High IPC Microarchitecture", in *Proc. ISCA-27*, 2000.

[10] S. Patel, and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization", *IEEE Transaction on Computers*, vol. 50. no. 6, pp. 590-608, 2001.

[11] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson, "Power Awarness through Selective Dynamically Optimized Traces", in *Proc. ISCA-31*, 2004, pp. 162-175.

[12] http://www.cs.ucr.edu/~vahid/warp/

[13] N. Clark, et al., "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization", in *Proc. MICRO-37*, 2004, pp. 30-40.

[14] H. Noori, et al., "A General Overview of an Adaptive Dynamic Extensible Procesor", in *Proc. Workshop oon Introspective Architecture*, 2006.

[15] www.eecs.umich.edu/mibench

[16] www.Simplescalar.com

[17] G. De Micheli, "Synthesis and optimization of digital circuits", McGraw-Hill, 1994.