An Efficient Dynamic Switching Mechanism (DSM) for Hybrid Processor Architecture

Akanda Md. Musfiquzzaman, Ben A. Abderazek, Sotaro Kawata, and Masahiro Sowa

Graduate School of Information Systems
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, 182-8585 Tokyo, Japan
email: akanda@sowa.is.uec.ac.jp

Abstract. Increasing the processor resources usability and boosting processor compatibility and capability to support multi-executions models in a single core are highly needed nowadays to benefit from the recent developments in electronics technology. This work introduces the concept of a dynamic switching mechanism (DSM), which supports multi-instruction set execution models in a single and simple processor core. This is achieved dynamically by execution $mode-switching$ scheme and a $sources-results$ locations computing unit for a novel queue execution model and a well-known stack based execution model. The queue execution model is based on queue computation that uses queue-registers, a circular queue data structure, for operands and results manipulations and assigns queue words according to a single assignment rule. We present the DSM mechanism and we describe its hardware complexity and preliminary evaluation results. We also describe the DSM target architecture.

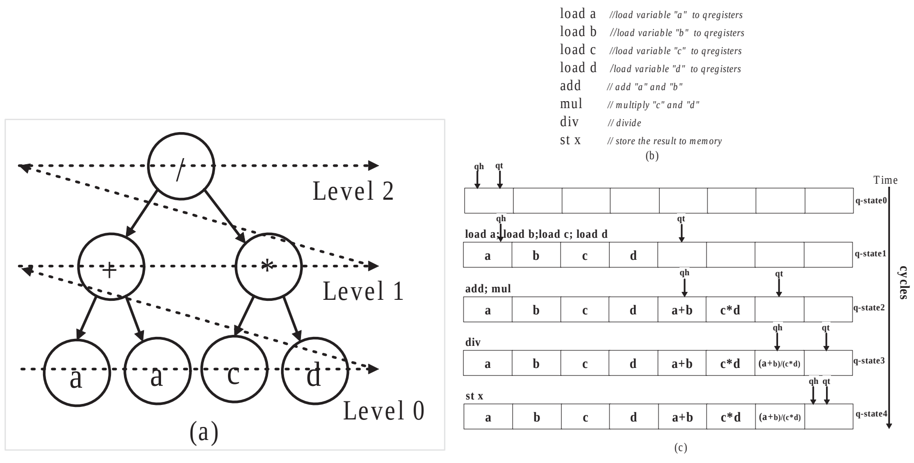
Index Words: Hybrid processor, compatibility, design, dynamic switching mechanism, FaRM computing algorithm.

1 Introduction

Generally, the motivation for the design of a new architecture arose from the technological development, which changed gradually the architecture parameters traditionally used in the computer industry. With this growing changes, the computer architect is faced with answering the question what functionality has to be put on a single chip, giving them an extra performance edge. Nowadays, as we enter into an era of constant demand for faster and compatible processors as well as different Internet and network appliances using different processor architectures, it becomes extremely complicated and costly to develop a separate processor for every execution model that satisfy this demand. Internet applications, which are generally *stackbased* need high execution speed or high performance as defined by the literature. However, recently the term "high performance" is questioned again by many processor designers and computer users. Some consider that high

performance means high execution speed or low execution time of some given applications. Other define "high performance" differently. They consider that processors which support several execution models are the favourite candidates for high performance "award", since switching from processor to processor lead to difficulty and waste of time. This is true especially when users have different applications written for different execution models (Stack and RISC model for example). In this case, users are obliged to run these two applications separately on different machines. In conventional machines, this problem was *somehow* solved by a direct software translation techniques. However, these techniques still suffer from slow translation speed. Sun Microsystems proposed another alternative and designed its Stack-based Java processor, so that Java code can execute directly [4, 5]. According to its designers, the JavaChip-I, for example, is a highly efficient Java execution unit design. It delivers up to 20 times the Java performance for x86 and other general-purpose processor architectures, as well as up to five times the performance obtained by $just-in-time(JIT)$ compilers. It is evident that in term of reduced execution time (ET), the solution is better than the indirect way (translation) or the JIT scheme, but in term of compatibility, the processor still suffers from not being able to execute other codes. Therefore, reduced ET, compatibility, and cost are questioned again. Supported another execution model will eventually lead to more complex hardware. We realized, that supporting different instruction sets can yield superior operational attributes to those architectures that support a single instruction set. Our objectives are clear. First, we knew that to reduce die size and improve performance, DSM (Queue and Stack switching algorithm) would be implemented in the FaRM pipeline as a finite state machine rather than a traditional microcoded engine. Second, the solution would have to dynamically calculate Queue and Stack locations to architectural registers called shared storage unit (SSU). Thus avoiding the need for a translation stage. Finally, the resultant architecture would have to perform 16-bit fetches in order to fetch up to four instructions at once. To this end, we proposed a Hybrid processor architecture that addresses this and other problems as a pure-play architectural paradigm, which will integrate Stack and Queue execution models right into the FARM core[1, 2].
In this work we introduce the concept and the architecture of a dynamic switching mechanism (DSM), which supports multi-execution models in a single and simple processor core. This is achieved dynamically by execution $mode-switching$ scheme and a $sources - results$ locations computing unit for a novel queue execution model and a well-known stack based execution models.
This work has two major contributions:

- The first contribution is the proposal of a dynamic switching mechanism for multi-execution models support. To evaluate its efficiency, we designed it in register transfer level using Verilog HDL language. Then, we evaluated its functional correctness and its hardware complexity when compared with the Parallel Queue Qrocessor (PQP). Examples for FQM and FSM execution models are also given.



```
load a  //load variable "a" to qregisters
load b  //load variable "b" to qregisters
load c  //load variable "c" to qregisters
load d  /load variable "d" to qregisters
add     // add "a" and "b"
mul     // multiply "c" and "d"
div     // divide
st x    // store the result to memory
```

Fig. 1. Sample example of parallel execution in FARM Queue execution Model (FQM). (a) Data flow graph of the expression $X = (a+b)/(c*d)$. (b) QEM instruction sequence using Level order traversal, and (c) Queue contents at each execution state.

– Our second contribution, which is related to the first one, is a detailed architectural description of the target hybrid processor system and its complexity analysis

2 Hybrid Mode Execution Overview

The FARM architecture design is categorized by co-ordinates along a three-axis system. The three dimensions of the design space are: (1) number of instructions set supported by the architecture, (2) the storage scheme, and (3) the number of operands permitted for each mode[2]. The FARM system is a highly 32-bit processor that support a subset of the Queue instruction set (QEM) and Stack instruction set (SEM)[1].
The QEM mode uses a first-in-first-out Queue data structure as the underlying control mechanism for the manipulation of operands and results. In addition, the QEM is analogous to the stack execution model (SEM) in that it has operations in its instructions set, which implicitly reference an operand Queue, just as a stack machine has operations, which implicitly reference an operand Queue. Each instruction removes the required number of operands from the front of the Queue operand, performs some computations, and stores the result of computation into the Queue of operands at the specified offsets from the head of the Queue. The Queue of operand occupies continuous storage locations. A special register, called the Queue Head (QH), contains the address of the first operand in the operand Queue. Operands are retrieved from the front of the Queue by

reading the location indicated by the QH pointer. Immediately after retrieving an operand, the QH is incremented so that it points at the next operand in the Queue. Results are returned to the rear of the operand Queue indicated by the Queue Tail (QT).
When switched for Stack-based mode, the switching circuitry and the functional computing unit (FCU) perform the job of execution model switching. Hence, the FCU calculates the sources and destination for corresponding instructions.
In FSM, implicitly referenced operands are retrieved from the head of the operand stack and results are returned back onto the head of the stack. For example consider a *sub* instruction. In SEM model, the *sub* instruction pops two operands from the top of the stack (TOS), computes the difference and pushes the results back onto the top of the stack. In QEM mode, the *sub* instruction removes two operands from the front of the Queue (FOQ), computes their difference, and puts the results at the rear of the Queue (ROQ) indicated by the RQP. In the former case, the result of the operation is available at the TOS. In the later case, the result is behind any other operand in the Queue. This will have an enormous potential to effectively exploit pipelined ALU with a simple hardware, which, due to their hardware structure, normal SEM obviously cannot guaranty.
For better understanding of the novel QEM, we show in Fig. 1 a sample example that calculates a simple mathematical expression $x = (a + b)/(c * d)$. In the above example, the QEM instruction sequences were generated using the level order scan algorithm (LOST) published in our earlier work[2]. That is, in order to get instruction sequence that can be correctly executed in FQM, the data flow graph for the given expressing should be traversed from the deep level to the top level and from left to right. In Fig. 1(c) the queue contents (later reffered as shared storage unit) at each cycle are shown. As illustrated in Fig. 1(a) and Fig. 1(b),the first four instructions are independent. Hence , they are processed in parallel (q-sate1). The second level in the data flow graph also has two independent instructions (+ and *). These two instructions are also executed in parallel (q-state2). At the third level, only *div* instruction is generated and is separately executed (q-state3). The result is finally stored into the PROG/DATA memory (not shown in the figure).

3 DSM Mechanism Overview

The DSM mechanism is implemented in a so-named hybrid processor architecture (FARM)[1]. Before we describe the DSM mechanism and its functional description, we first give a brief architectural overview about the FARM core that adopts the DSM mechanism. The target processor is based on Consume-produce Order Queue Computational Model(FQC) and Stack computational model(FSC)[1]. The processor has six pipeline stages and is based on 16-bit instruction set architecture. It supports two execution modes: Queue mode (FQM) and Stack mode (FSM). The basic block diagram of FaRM is given in Fig. 2. The FARM core consists of the following units: (1) Instruction Fetch unit (FU), (2) Decode Unit (DU), (3) FaRM Computation unit (FCU), (4) Issue Unit (IU),

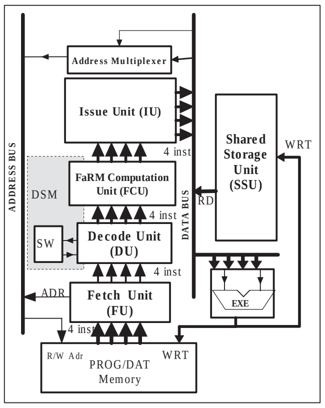


Fig. 2. Hybrid FARM system architecture.

(5) Execution Unit (EXE) and (6) Shared Storage Unit (SSU). The basic block diagram of FaRMqs is given in Fig. 2.
Instruction Fetch: The fetch unit fetches 4*16-bits instructions/cycle from the program memory and inserts them into the fetch buffer.
Instruction Decode: Decodes the instruction opcode and operands. The DU has four decode circuits (DC) and one Mode Selector Register (MS) that is set to zero or one according the type of instruction being decoded.
FaRM Computing Unit: The FaRM Computation unit gathers information mainly from the decode unit and uses them to compute the instruction sources and destination locations for FQM and FSM models.
Issue stage: The Issue stage issues ready instructions to the execution unit. Memory and registers dependency are checked by this unit/stage. This unit also checks the sources availability for each instruction.
Execution Unit: The EXE executes issued instructions and sends the results to the SSU or the data memory. The EXE consists of: (1) Arithmetic logical unit (ALU), (2) Shift unit, (3) Set register unit, (4) Load/Store unit, (5) Move and Compare unit and (7) Branch unit.
Write back Unit: The write back unit writes the result back to the PROG/DATA memory or the shared storage unit (SSU). The SSU is a 32*256 registers and is shared by both FQM and FSM execution models. In FQM, the SSU behaves as a conventional register file-like. However, in FSM, the system organizes the SSU access as a last-in-first-out.

4 DSM Architectural Details

As we earlier mentioned, the DSM system consists of a switching circuitry (SW) and a dynamic computation unit (FCU). It is implemented in a hybrid processor

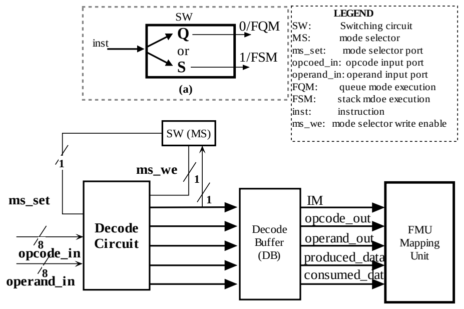
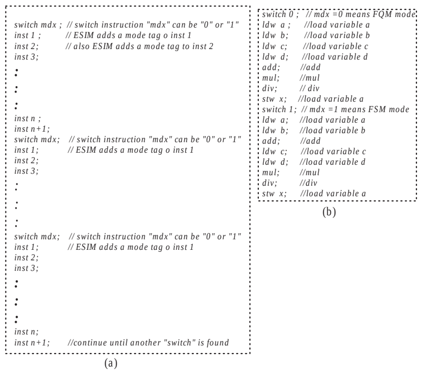


Fig. 3. DSM Basic Block Diagram



```
switch mdx ;  // switch instruction "mdx" can be "0" or "1"
inst 1 ;      // ESIM adds a mode tag o inst 1
inst 2;       // also ESIM adds a mode tag to inst 2
inst 3;
:
:
:
inst n ;
inst n+1;
switch mdx;   // switch instruction "mdx" can be "0" or "1"
inst 1;       // ESIM adds a mode tag o inst 1
inst 2;
inst 3;
:
:
:
switch mdx;   // switch instruction "mdx" can be "0" or "1"
inst 1;       // ESIM adds a mode tag o inst 1
inst 2;
inst 3;
:
:
:
inst n;
inst n+1;     //continue until another "switch" is found
```
(a)

```
switch 0 ;  // mdx =0 means FQM mode
ldw  a ;    //load variable a
ldw  b;     //load variable b
ldw  c;     //load variable c
ldw  d;     //load variable d
add;        //add
mul;        //mul
div;        // div
stw  x;     //load variable a
switch 1;   // mdx =1 means FSM mode
ldw  a;     //load variable a
ldw  b;     //load variable b
add;        //add
ldw  c;     //load variable c
ldw  d;     //load variable d
mul;        //mul
div;        //div
stw  x;     //load variable a
```
(b)

Fig. 4. FARM programming Models:(a) Programming Model, (b) Example of FQM and FSM assembly program

architecture. The FCU unit calculates $on-the-fly$ the sources and destinations for instructions in both FQM and FSM executions modes. A block diagram of the DSM mechanism is illustrated in Fig. 3. The DSM detects the instruction mode by decoding the operand of the *switch* instruction. After the mode detection, it inserts a $mode - bit$ for all instructions between the current and the next *switch* instruction. Hence, the same operation can be used for both modes. This will increase the resources usability and the overall system performance (discussed later).

Dynamic switching example: In Fig. 4, we show the FARM programming model and an assembly program example. The first instruction at a given program should be always a "switch" instruction. This instruction will guide the decode unit to dynamically *reprogram* itself according to the *switch* instruction's operand value (*zero* for FQM mode and *one* for FSM mode). Instructions that follow *switch* instruction continue, then, execution in the FSM or FQM until another *switch* instruction is found (refer to Fig. 4). The DSM (major

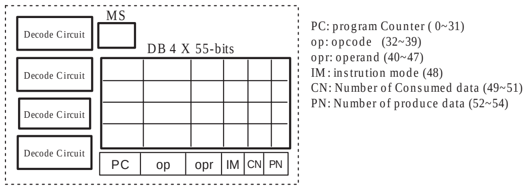
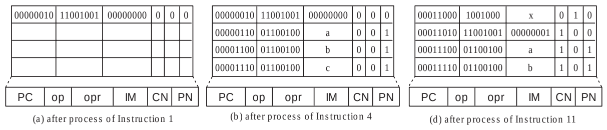


Fig. 5. DSM hardware components



PC	op	opr	IM	CN	PN
00000010	11001001	00000000	0	0	0

(a) after process of Instruction 1

PC	op	opr	IM	CN	PN
00000010	11001001	00000000	0	0	0
00000110	01100100	a	0	0	1
00001100	01100100	b	0	0	1
00001110	01100100	c	0	0	1

(b) after process of Instruction 4

PC	op	opr	IM	CN	PN
00011000	1001000	x	0	1	0
00011010	11001001	00000001	1	0	0
00011100	01100100	a	1	0	1
00011110	01100100	b	1	0	1

(d) after process of Instruction 11

Fig. 6. Various DSM states over hybrid execution modes.

hardware components) mechanism are illustrated in Fig. 5. This figure shows the initial state of the decode buffer (DB)and mode switch (MS) register before switch instruction is decoded. Fig. 6 shows the contents of the DB at different states for different instructions.

4.1 Dynamic Computation Algorithm

Dynamic Computation Algorithm (DCA) has two mapping algorithms: (1) FARM Queue Computing and (2) FARM Stack Computing.

FQM sources and destination computing: In order to have a correct execution, each instruction needs to know the values of the QH (Queue Head) and the QT (Queue Tail). The above values are easy to know in serial Queue execution model, since the QH is always used to fetch instruction from the operand queue

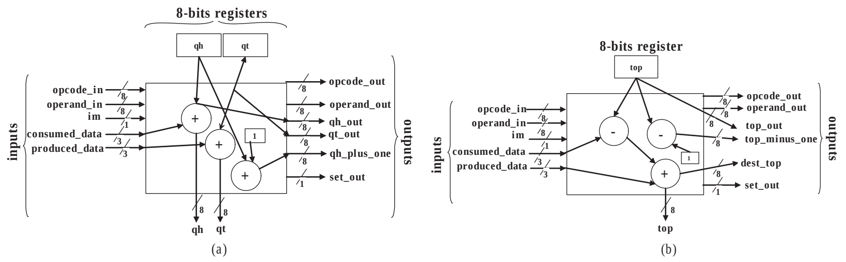


Fig. 7. FQM and FSM computing circuits: (a) FQM computing circuit; (b) FSM computing circuit

(OPQ) and the QT is always used to store the result of the computation in to the tail of the OPQ. However, in the parallel execution scheme the above pointers are not explicitly determined. This is due to the fact that previous instructions are simultaneously executed and may not be completed in order. Fig. 7(a) shows the block diagram of queue computing circuits (FQC). The FQC is based on the Produce-consume order Queue Computation Mechanism.
The QH calculation for an instruction I(n+1) is given by:

$$QH(n+1) = QH(n) + (consumed\_data) \qquad (1)$$

Where, Consumed Data is the number of fetched operands of an instruction.

The QT calculation of an instruction I(n+1) is calculated by:

$$QT(n+1) = QTP(n) + (produced\_data) \qquad (2)$$

Where, Produced Data is the number of the results generated by an instruction
FSM sources and destination computing: FARM Stack Mechanism (FSM) is based on the pure stack Mechanism. For mapping the source and destination FSM used only one register named Top. Fig. 7(b) shows the block diagram of FaRM Stack (FSM) mechanism. The Top pointer calculation for an instruction (n±1)is given bellow:

$$TOP(n \pm 1) = TOP(n) - consumed\_data + produced\_data \qquad (3)$$

Here, we use ± because top pointer does not only increase but also decrease. For *load* instruction Top pointer will increase and for *store* instruction Top will decrease. After assigning instruction, the FSM sends instructions to the FARM Computation Buffer (FCB).
Example for FQM Computing: To calculate the sources and destination location for FQM execution, we use equation (1) and (2) to assigning the QH and QT pointers value. To illustrate the idea, we use the simple expression $X = (a + b)/(c * d)$ as an example. The assembly code for the above expression in shown in Fig. 4(b). In FARM programming model, the first instruction should be always a *switch* type instruction. In this example, the operand of the *switsh* is 0 which indicates that the being loaded program should be executed in FQM. As a result, the mode-bit for all following instruction will be set to FQM mode.
Example for FSM Computing: We use formula 3 in Stack Computing to calculate the TOP pointer value for each instruction. The assembly code of a simple expression $X = (a + b)/(c * d)$ is also shown Fig. 4. In this case, the operand of *switch* instruction is *one*, which indicates a stack program instructions sequence. The DCA uses the stack mapping mechanism. For this case, the DCA sets a "*setsignal*" for each instruction ("1" indicates the stack instructions) and requests the issue stage for sequential issue.

Table 1. Hybrid FARM processor complexity (in flip-flops) evaluation results over speed and area optimizations. The DSM mechanism is implemented on Decode and FCA units. SOPT means speed optimizations; AOPT means area optimizations

Description	Unit Name	SOPT	AOPT
Fetch Module	FU	201	166
Decode Module	DU	1039	975
Computation Module	FCA	848	800
Issue module	IU	589	326
Execution module	EXE	1140	1140
Shared storage module	SSU	6545	5730

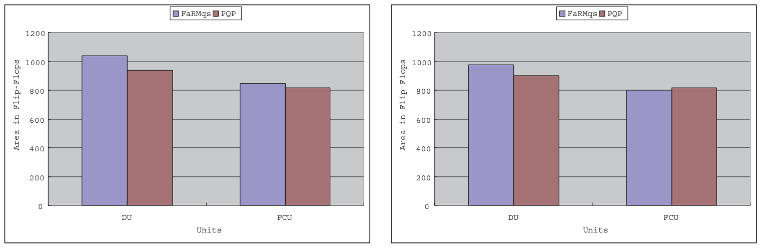


Fig. 8. Hardware Complexity evaluation and comparison: (a) The left side figure shows the area the comparison over Speed optimizations, (b) The right side figure shows the area comparison over Area optimizations

5 DSM Implementation Evaluation

We designed the DSM mechanism in register transfer level using Verilog HDL. In order to evaluate the effectiveness and the correctness of the mechanism, we performed two types of evaluations: (1) Functional evaluation level, (2) Hardware evaluation level.

Functional verification: We described the DSM in RTL level and used Verilog-XL and Simvision simulators to evaluate the functional simulation result. We captured the input and output signals changes for several cases.

Hardware level evaluation : The DSM and the whole system core were designed in RTL level and correctly integrated and synthesized in a Altra STRATIX-EP1S25F1020 device. The design results for area and speed optimizations are shown in Table 1. Notice that the DSM is integrated in the DU and FCU units. In order to know the real hardware complexity of the DSM, we compared the units that integrated it with other units in a similar queue processor architecture that do not include switching capability. From the above comparison, we found that the DSM mechanism uses about 60% of DU and 40% of the FCU. Fig. 8 shows the comparison results of the area with a conventional Queue processor core over speed and area optimizations respectively. From the above simulation, we noticed that the additional area for DSM mechanism is acceptable for both speed and area optimizations. We found that the DU and the FCU use about 9.72% and 3.77% extra area when compared with conventional queue processor's decode and queue computation modules.

6 Concluding Remarks

In this paper, we have proposed a dynamic switching mechanism (DSM) for a hybrid processor architecture that supports Queue and Stack computation modes in a shared resources single processor core. This is achieved dynamically by execution $mode-switching$ scheme and a $sources-results$ locations computing unit. We have presented the novel aspects of the DSM mechanism as well as the the architecture description of a hybrid processor system that adopts the DSM mechanism.
We designed the hardware of the DSM and verify its functionality and overall complexity. From the design and evaluation results, we proved that the DSM mechanism can be used without enormous additional hardware when compared with conventional queue processor core (about 9.72% for DU and 3.77% for FMU extra hardware when compared with Queue processor core). We also conclude that the overall performance of the hybrid FARM architecture, which is still in its infancy, is acceptable. Hence, the DSM system increases the resources usability and the overall performance of the FARM processor.
Our future work is to evaluate the whole system with real benchmark programs and estimate the real complexity of the whole system.

References

1. M. M. Akanda , B. A. Abderazek , T. Yoshinaga, M. Sowa, "FaRMqs: Hybrid Processor Architecture in Verilog HDL" IPSJ, 67th Conf., March 2 3, 2005. for Parallel Execution in Parallel Queue Processor", IPSJ, 66th Conf., March 9 11, 2004, pp.1-69 1-70.
2. B. A. Abderazek, Kirilka Nikolova, and M. Sowa: "FARM-Queue Mode: On a Practical Queue Execution Model", Proceedings of the Int. Conf. on Circuits and Systems, Computers and Communications, Tokushima, Japan, pp.939-944, July 2001
3. M. Sowa, B. A. Abderazek, S. Shigeta, K. Nikolova, and T. Yoshinaga, " Proposal and Design of a Parallel Queue Processor Architecture (PQP) ", 14th IASTED Int. Conf. on Parallel and Distributed Computing and System, Cambridge, USA, pp. 554-560, Nov. 2002.
4. N. VijayKrishnan, "Issues in the Design of JAVA Processor Architecture". PhD dissertation, University of South Florida, Tampa, FL-33620. December 1998.
5. R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural issues in java runtime systems," Tech. Rep. TR-990719, 1999.
6. Sowa Laboratory, www.sowa.is.uec.ac.jp