3P-7

FARM-Queue Execution Model: Toward an Alternative Computing Paradigm

Abderazek BEN ABDALLAH, Kirilka NIKOLOBA, and Masahiro SOWA
Graduate School of Information Systems
The Electro-communication University
1-5-1, Choufugaoka, Chufu, 1828585 Tokyo, Japan
E-mail: {baa, nikol}@sowa.is.uec.ac.jp, sowa@is.uec.ac.jp

Abstract

For several decades, processor designers have used the well-known first-in-first-out (FIFO) Queue data structure to mainly resolve two processor-memory interfaces- long latency and low bandwidth. In this letter, a processing element architecture that directly incorporates hardware queue, as it's primarily data handling structure is proposed. Such processing element is called Queue execution model, which uses FIFO data structure as the underlying mechanism for data manipulations. The above execution model is implemented in a so-named Functional Assignment Register Microprocessor (FARM), that embraces two integrated programming environments. First, we present a mathematical formal specification for generating queue execution model's instruction sequence for evaluating an arbitrary arithmetic or Boolean expression. Then, we show that queue based execution model has the potential to take advantage of a pipelined ALU, when the computation queue is not empty, and support recursion and program reentrancy without intermingling data parameter list. Finally, we present the novel aspects of the above execution model as well as the principle underlying the architecture and the constraints that must be met. To characterize the behavior of the proposed architecture, we simulate the machine in software. Our preliminary evaluation results are performed through a full-level-architectural simulator and a number of test benchmark programs.

1. Introduction

Queue is well known to computer architecture designers as a first-in-first-out (FIFO) data structure. It has a number of interesting properties that make it useful in supporting processor design functions [3]. So far, as an attempt to improve overall processor performance, designers have used these data structure as matching devices to interface two subsystems with different duty cycles. That is, they have used queues in numerous architectures to support various processor functions and to hold instructions (both decoded and non decoded), branch target buffer instructions, and data during reads and writes. In this research paper, we extend the use of this data structure by designing a processing element architecture that directly incorporates hardware queue, as it's primarily data handling structure. Such processing element is called Queue execution model, which uses FIFO data structure as the underlying mechanism for data manipulations. Queue execution model is analogous to the stack execution model (S-Model) that it has operations in its instructions set, which implicitly reference an operand queue, just as a stack execution model has operations, which implicitly reference an operand stack. The basis of the Queue execution model (Q-Model) is the use of queue data structure to handle operands [1,2]. Each instruction removes the required number of operands from the head of an operand queue (OPQ), performs some computations, and stores the results of computation into the rear of the OPQ, which occupies continuous storage locations. A special counter, called the queue-front-counter (QFC), 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 QFC. Immediately after retrieving an operand, the QFC 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-rear-counter (QRC). In S-Model, 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. On a stack execution model, the above instruction pops two operands from the top of the stack, computes the difference and pushes the results back onto the top of the stack. However, on a Q-Model, the `sub' instruction removes two operands from the front of the queue, compute their difference, and put the result at the rear of the queue indicated by the RQP. In the former case, the result of the operation is available at the head of the stack. In the later case, the result is behind any other operand in the queue. We will show that Q-Model has the potential to take advantage of a pipelined ALU when the two queue counters (QRC and QHC) do not point to the same Queue location. On the other hand, the pure stack execution model cannot exploit a pipelined ALU, since the results of one operation must be returned to the top (head) of the stack before they can become the operands of the next operation. That is, each instruction depends on the effect of the previous instruction on the stack. This property has profound implications in the area of program compactness, hardware simplicity and execution speed. We will show that the instructions' sequences of the proposed execution model are easily generated from expression binary trees.

2. Preliminary Evaluation Result

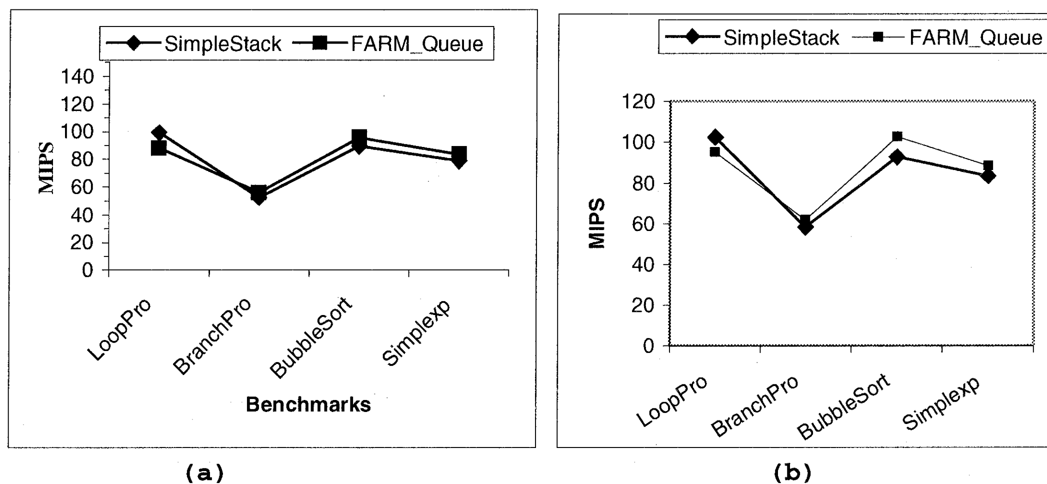

Fig. 1 Number of instruction executed per second for overlapped and no overlapped fetch/execute ALU and a range of benchmark programs (a) Overlapped, (b) No overlapped.

Figure 1 (a) shows the average number of instructions executed per second (MIPS) for Bubble-sort, BranchPro, LoopPro, and SimpleExp benchmark programs for both cases. Bubble-sort is assorting technique program in which pairs of adjacent values in the list to be sorted are compared and interchanged if they are out of order; thus, list entries "bubble upward" in the list until they bump into one with a lower sort value. In all of these benchmark programs, the queue based execution model always meets or exceeds the performance of the stack based execution model except for the Bubble-Sort benchmark. For example, for SimpleExp, the number of instructions executed per second (MIPS) is about 5 % more that that of the Stack based execution model, which means also a less execution time.

In Figure 1 (b) , the same measurement is done for overlapped fetch/execute ALU. This case performs better than the first one (no overlapped fetch/execute ALU). This is as expected, since the queue based execution model takes better advantage of overlapped fetch and execution when the computation queue (operand queue) is not empty. This is due to the fact that, in stack based execution model, the results of one operation must be returned to the top of the stack before they can become the operands of the next operation. That is, each instruction depends on the effect of the previous instruction on the stack.

3. Conclusion

Our preliminary result show that the benefit of Queue based execution model over stack-based execution model is seen with longer ALU pipeline. The speedup-range is from 1.01 to 1.1 for the mentioned benchmark programs. On the other hand, under overlapped fetch/execute pipelined ALU, the benefit of the queue-based execution model is optimum (about 1.1) for two-stages pipelined ALU. Our proposed architecture is estimated to be implemented without considerable hardware addition and complexity. Therefore, the total power consumption and die area, which are under Investigation, are estimated to be satisfactory. Since the above execution's model is a serial architecture (programs are executed sequentially), our future work is to improve the above architecture by studying the interaction between the hardware architecture and the compiler. Furthermore, we expect to study its behavior in other environments.

4. Reference

1. Shusuke O., Hitoshi S., Atsui M., and Masahiro S.. Design of a Super scalar Processor Based on Queue Machine Computing Model. In proceeding of IEEE PACRIM, August 22-24, 1999.
2. Abderazek B.A, Mudar S., and Masahiro S.. Dynamic Fast Issue (DFI) Mechanism For Dynamic Scheduled Processors. In Transaction of IEICE on VLSI and CAD Design, December 12, 2000.
1. Michael K.M. ``Processor Implementation Using Queue", IEEE Micro, Aug. 1995, pp. 58-66.