PROPOSAL AND DESIGN OF A PARALLEL QUEUE PROCESSOR ARCHITECTURE (PQP)

Masahiro Sowa, Ben A. Abderazek, Soichi Shigeta, Kirilka Nikolova and Tsutomu Yoshinaga
Graduate School of Information Systems
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, 182-8585 Tokyo, Japan
E-mail: {sowa, ben, shigeta, nikol, yosinaga}@is.uec.ac.jp

ABSTRACT
In this paper, we propose a parallel Queue processor architecture (PQP) that uses Queue data structure for operands and results manipulations. The above architecture project, which started a couple of years ago here at Sowa laboratory, features simple pipeline, compact Queue based instruction set architecture, and is targeted for Internet applications and new class of terminals requiring small memory footprints and short programs run-times.
The proposed architecture has been designed and then evaluated in software.
First, 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. Second, to characterize the behavior of the proposed architecture, we present the preliminary evaluation results over a range of benchmark programs.

KEY WORDS
Parallel Queue Processor, Design, Queue Computing, Instruction Level Parallelism

1. Introduction

As demand for increased performance in microprocessor continues, new architectures are required to meet this demand. Implementations of existing architecture are quickly reaching their limits as increases in current superscalar out of order (OOO) issue are bounded by circuit complexity [3, 13], and performance increases due to technology improvement are approaching their limit.
From another hand, the most widely known and used applications in the Internet and mobile computing nowadays, are generally based on java programming language; that is based on stack computing principle. While Java applications are typically run through a compiler with the resulting bytecode interpreted in software. Recently the Sun's PicoJava and other stack-based processors execute *Stack* based applications (Generally Java Bytecode) in hardware without software interpreters or code translators. This provides fast execution of Java programs with minimum hardware and operating system support. However, execution of stack-based applications on stack-based processors is invariably constrained by the limitations of the stack architecture for accessing operands [8, 6]. Thus, the conventional stack based processors cannot take advantage of a pipelined arithmetic logic unit (ALU), since the result of one operation must be returned to the top of the stack before it becomes the operand of the next operation. To cope with the above problem, several hardware mechanisms were proposed [8, 6, 16]. However, most these techniques lead to hardware complications [16, 11].
Here in this research work, we propose a parallel Queue processor (PQP) architecture that effectively deals with the complexity of superscalar processors and the lack of parallelism support in stack based architectures. It will be shown that the PQP has the potential to take advantage of a pipelined ALU when the operand queue (OPQ) is not empty. The PQP architecture uses a first-in-first-out (FIFO) data structure as the underlying mechanism for results and operands manipulations.
Our architecture design idea was inspired from the original research works of Sowa [9, 14, 17] and Bruno [8]. In these researches, Queue data structure was proposed for operand and results manipulations. However, when the original Queue computing model was proposed more than two decades ago, neither Stack based applications nor Queue computing model seemed to be an important paradigms in the foreseeable future. Things have changed since then: Internet, Embedded, and home network (HON) applications are becoming very attractive nowadays.
To this end, we propose a parallel queue processor (PQP) architecture targeted for Internet applications and new class of terminals requiring small memory footprint and short programs run-times.
The proposed PQP architecture is in its earlier design stage here at the UEC Sowa Laboratory [7, 4]. It is a full new computing paradigm made of a PQP compiler, a PQP simulator as well as several benchmarks programs especially developed to evaluate the performance behavior of the architecture.
The rest of this paper is organized as follow: In section two, we review the related work. In section three, we describe the Queue computing fundamental. A sample Queue instruction sequences is also given in this section.

Section four gives the system architecture overview. The evaluation result of the PQP processor is given in section five. Finally, the last section gives our concluding remarks and future work.

2. Related Work

Historically, the Queue computing idea is traced back to several decades ago. Sowa [1, 9, 14, 17] investigated the design constraints of a super scalar processor architecture based on Queue computation model. Another research work was proposed by Bruno [8, 12]. He investigated a serial indexed Queue machine architecture that uses the FIFO as the underlying mechanism for operands and results manipulations. At the execution stage of the above architecture, each instruction removes the required number of operands from the front of the operand Queue, performs some computation and stores the result back into the operand Queue at a specified offsets from the front of the Queue. A major problem with the above indexed Queue architecture is that it requires the relocation of a potentially large number of operands. In addition, the operand queue within the above architecture is implemented in the main memory.
The work described herein investigates the design and the evaluation of a novel parallel Queue processor architecture (PQP), which executes data and control flow programs and is targeted for Internet applications and new class of terminals.

3. Queue Fundamental

The Queue computing model (QEM) is analogous to the usual stack computing model (SEM). The QEM has operations in its instructions set which implicitly reference to an operand Queue (OPQ), just as a SEM has operations, which implicitly reference an operand stack. Each instruction removes the required number of operands from the head of the OPQ, performs some computations, and stores the results of the computations at the tail of the OPQ, which occupies continuous storage locations (described later). That is, the execution order of instructions coincides with order of the data in the OPQ. A special register, called the Queue head pointer (QHP), contains the address of the first operand in the OPQ. The operands are retrieved from the front of the OPQ by reading the location indicated by the QHP. Immediately after retrieving an operand, the QHP is incremented so that it points to the next operand in the OPQ. Results are returned to the rear of the OPQ indicated by the Queue tail pointer (QTP). However, in the conventional SEM, implicitly referenced operands are retrieved from the head of the operand stack (OPS) and the results are returned back to the head of the OPS.

3.1 Sample Queue Instructions Sequence

In Figure 3.1, a sample example to demonstrate the basic Queue operations is shown. In Figure 3.1 (a), the arithmetic operations correspond to the internal nodes of the tree and the fetch operations correspond to the leaf node of the binary tree. Informally, the LOST traversal is done by visiting the nodes of the binary tree from the deepest to the shallowest levels and from the left to the right within each level as indicated in Figure 3.1 (a) (level 0 to level 2). The (*) is a multiply operator, the (+) is an addition operator, and the (/) is a division operator.
The Queue content after each Queue instruction processing is shown in Figure 3.1 (b). The "*mul*" *m*nemonic is a multiply operation, "*add*" is an addition operation, "*div*" is a division operation, and "d" is a 2-bit *index address* (00 for index register *a1*, 01 for index register a2, 10 for index register *a3* and 11 for index register *a4)*.

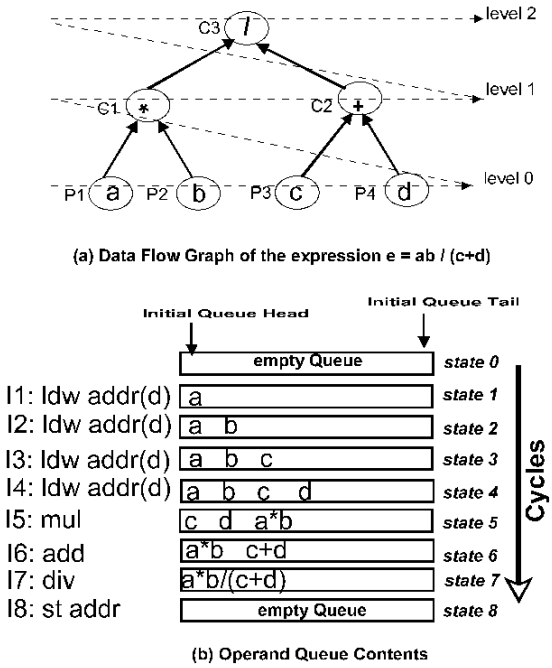

Figure 3.1 Basic Serial Queue computing

3.2 Parallel Execution Fundamental

In Figure 3.2, a parallel execution model of the previous example is shown. In the above example, parallel processing of instructions is obtained by allowing the four load instructions to be simultaneously loaded from the program memory and also by allowing the two operations *mul* (multiplication) and *add* (addition) to be simultaneously executed. As a result, the number of computing stages is reduced from eight to only four. The content for the OPQ for the parallel execution model is shown in Figure 3.2 (b).

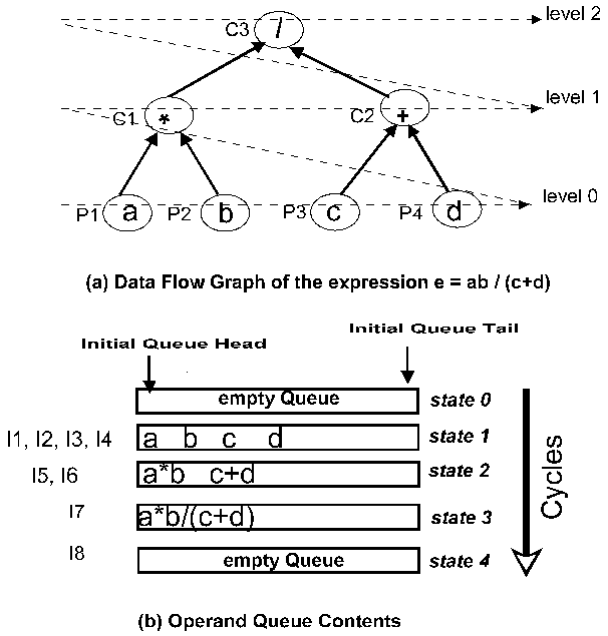

Figure 3.2 Basic Parallel Queue Computing

3.2.1 Queue Head and Tail Calculation

In order to have a correct execution, each instruction needs to know the values of the QHP and the QTP. The above values are easy to know in serial Queue execution model, since the QHP is always used to fetch instruction from the OPQ and the QTP 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 complete in order.

The QHP calculation for an instruction *I(i+1)* is given by:

$$QHP(i+1) = QHP(i) + \sum (Consumed\ Data)$$

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

The QTP calculation of an instruction (*i+1)* is calculated by:

$$QTP(i+1) = QTP(i) + \sum (Produced\ Data)$$

Where, *Produced Data* is the number of the results generated by an instruction.

3.2.2 Finding Executable Instructions

To avoid stalls and improve the whole processor performance, the next executable instructions should be determined in advance. We call this Executable Instructions Finding (EIF) scheme [14]. Because, the EIF is performed before the completion of previous instructions, some needed data within the OPQ may not be found. Therefore each instruction checks its validity data bit (VDB) in the OPQ.

The checking process is done during the issue stage. That is, before instructions are issued the issue unit has to check the validity of the data by checking the data availability bit in the corresponding queue entry.

4. System Architecture Overview

The PQP architecture, illustrated in Figure 4.1, consists mainly of a fetch unit (FU), an instruction extraction unit (IEU), a Queue computation unit (QCU), an executable instruction finding (EIF) unit, an issue unit (IU), and an execution unit (EU).

The PQP has five pipeline stages: (1) Fetch Stage, (2) Decode stage, (3) Issue stage, (4) Execute Stage and (5) Completion stage.

Instruction Fetch: The PQP processor fetches *n* bytes per cycle. The fetched instructions are buffered the instructions buffer (IB). Since the instruction length is not fixed within the PQP architecture, the number of fetched instruction depends on their sizes (one or three bytes). Therefore, the fetched instructions group may contain the first one or two bytes of an instruction. The PQP decoding circuitry handles this situation and detects uncompleted instructions.

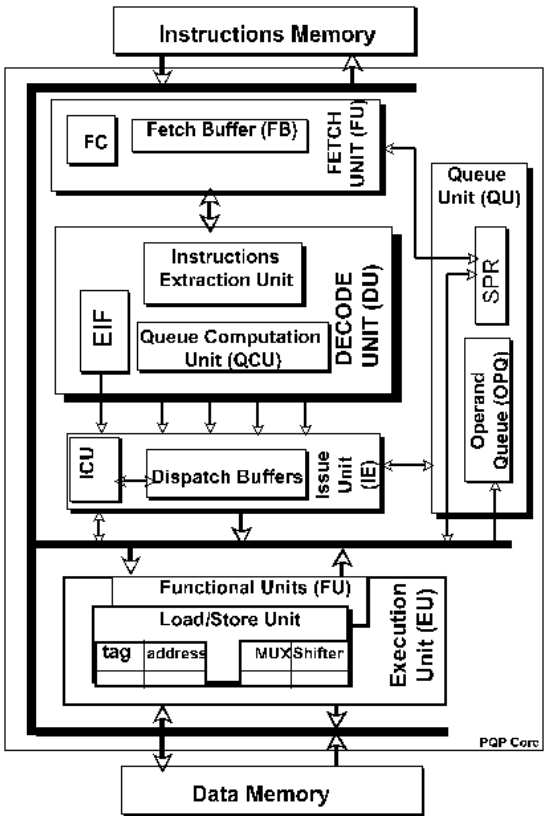

Figure 4.1 PQP System Architecture

Instruction Decode: In this stage instruction type and its appropriate functional unit are determined. The unaligned instructions are also checked and solved. For each operation, the queue head and tail values are determined.

Instruction Issue (IS): The *IS* unit collects information about ready instructions then issues independent instructions in out of program order (OOO) scheme. In this stage, the dispatch buffers are used to maintain the pool of instructions from which the instruction issuer issues instructions to the functional units (FU).
Instructions Execution: In this stage, instructions are executed and their results are written to the tail of the OPQ.
Instruction completion: In this sate the result of the execution stage is written first to the tail of the OPQ then the main memory is updated.

4.1 PQP Instruction Set Format

The PQP processor has a variable instructions length, with the more frequently instructions encoded at the shorter length. Indeed, the majorities are just one byte long. Instructions that are not one byte long are three bytes long. Instructions may operate on several data types, including byte, short, integer, double, and float. The full instruction set of the PQP can be found in [15].
The above instruction set is classified into tree classes (Figure 4.2) as follow:
(1) Basic class: consists of all arithmetic, shift, rotate, compare and logical operations.
(2) Mem class: consists of the memory, control, branch, and loop operations.
(3) MemIndex Class: consists of memory indexed address, control, branch and loop operations.

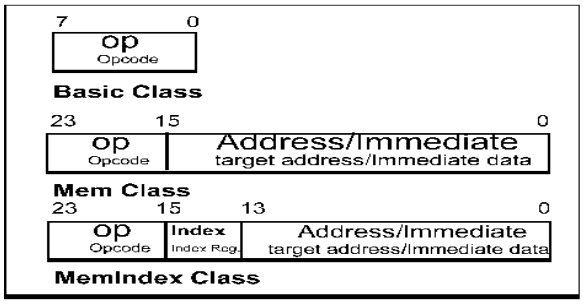

Figure 4.2 PQP Instruction Format

5. Preliminary Evaluation

5.1 Methodology

For the above PQP evaluation, we have developed a configurable simulator named PQPsim. We have also developed four Queue sample programs (*FFT, Newton, Prefix*, and *LU-Decompositions*). The *FFT32* benchmark is a 32-point Fast Fourier Transformation program. The *Newton* benchmark is a 4-degree Newton polynomial interpolation program. The *Prefix* benchmark is a prefix computation method of degree 8, which treats a given equation as a system of recurrences. The *LU-Decompositions* of degree 20 is a program to solve a sequence of matrix equations. The four sample programs characteristics are summarized in Table 1. The classification of the four sample programs is based on the maximum ILP and the average distance between producer instructions (PI) and consumer instructions (CI). As mentioned in the previous section, producer and consumer instructions appear reciprocally in executables instructions for the PQP processors.

Table 1. Benchmark Programs Characteristics.

Benchmark Name	***Max. ILP***	***Dist. Average***
Sample A (FFT32)	High	Long
Sample B (LU)	High	Short
Sample C (Prefix)	Low	Long
Sample D (Newton)	Low	Short

For the above preliminary evaluation we want to find the optimal number of FUs and the fetch width (FW), which are two major parameters of our PQP processor architecture. Therefore, and evaluated our PQP in terms of EILP (executed instruction level parallelism) over a range of FUS and FW.
The EILP is influenced by the distance between PI and CI. When the distance between PI and CI is short, the data dependency occurrence between PI and CI is high. Thus, the possibility of parallel execution within the above instruction segment ([PI, CI]) is low especially with the IO issue scheme configuration.
The maximum EILP becomes equal to the program's maximum ILP when we assume enough number of FUs and enough size of FW. For example, 64 load units, 32 arithmetic units, and 192 Bytes of FW is required to achieve the maximum ILP of sample program A. But, such a huge number of FUs and large size FW doesn't seem to be "implementable" in real hardware.
The PQP architecture has 6 types of functional units (FUs): (1) Load Unit (LU), (2) Store Unit (SU), (3) Integer Arithmetic Unit (IAU) for *add/sub* instructions, (4) Integer Arithmetic Unit for *mul/div*, (5) Floating-Point Arithmetic Unit (FPAU) for *add/sub* instructions, and (6) Floating-Point Arithmetic Unit for *mul/div* instructions.
The PQP processor assumes also that all instructions execute within one cycle.
In order to evaluate the effectiveness of the EIF, our evaluation was performed with two types of instruction issue policies: (1) In program order (IO) instruction issue and (2) in out of program order (OOO) instruction issue.
Note here that enough size of instruction buffer is assumed for the OOO instruction issue policy.
In order to study the effects of the number of functional units (FU) and the fetch width (FW) on the EILP, we varied the number of the processor's FUs from (2, 2, 2, 2) to (16, 16, 16, 16). Where, the first parameter, in (x, x, x, x), represents the number of load units (LU), the second parameter represents the numbers of store units (SU), the third parameter represents the number of arithmetic integer units for *add/sub* instructions and the last parameter represents the number of integer arithmetic

point unit for *mul/div* instructions. We also varied the FW from 6 to 18 Bytes (indicated in all Figures as FWx).

5.2 Simulation Results

Figure 5.1 shows the average of EILP with the IO instruction issue policy for sample program A. For example, this figure indicates that 12 Bytes FW is enough for the configuration (8,8,8,8) FUs because EIPC was not improved even if FW was increased to 15 Bytes and 18 Bytes.
Figure 5.2 shows the average of EILP with OOO instruction issue policy for the sample program A. From the above figure, we note that a fetch width of 15 Bytes is found to be suitable when the *PQPsim* is configured with (8, 8, 8, 8) FUs (same number of FUs in case of IO issue). Therefore, only additional 3 Bytes fetch width are required for OOO issue policy.
However, there is no considerable improvement of the EIPC average by applying the OOO issue policy for sample A. The average of EIPC with the IO issue policy is about 10 inst/cycle, while the average with OOO issue policy is about 11 inst/cycle. This is because the "contiguousness" of producer/consumer instructions beyond the FW.

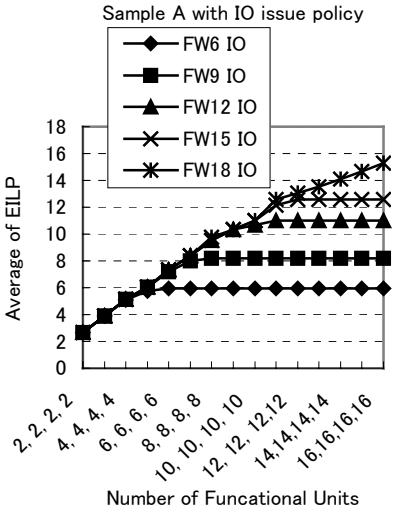

Figure 5. 1 EIPC for Sample A with IO Issue Policy

Figure 5.3 shows the average of EIPC with IO instruction issue policy for sample program B. In this case, there are no EILP improvements when the FW is increased. This result comes from the sample B's characteristic; that is, from the fact that the average of distance of PI and CI is short in sample B. Obviously, this characteristic prevents parallel execution of instructions.

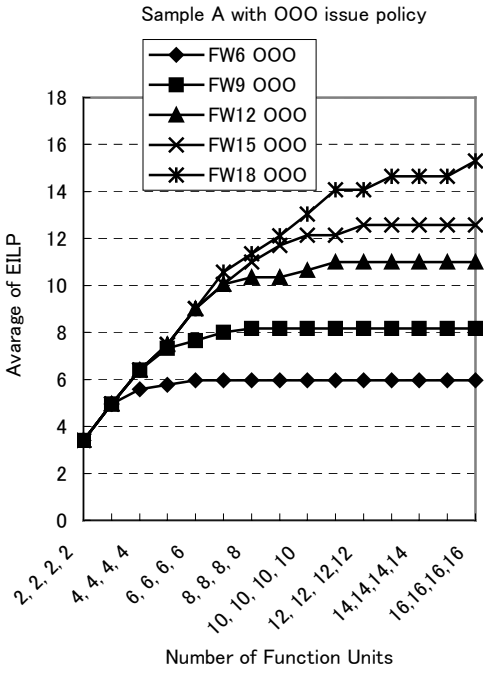

Figure 5. 2 EIPC for Sample A with OOO Issue Policy

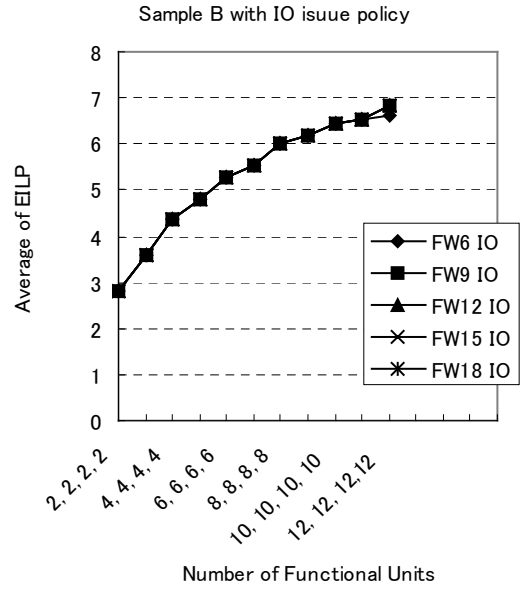

Figure 5. 3 EIPC for Sample B with IO Issue Policy

Figure 5.4 illustrates the average of EIPC with OOO instruction issue policy for sample B. In this case, the EIPC was improved effectively by applying the OOO instruction issue policy. For example for 12 Bytes FW and (8, 8, 8, 8) FUs configurations, the improvement is about 100% (the average of EIPC increases from 6 to 12 inst/cycle). However, with 18 Bytes FW and (12, 12, 12, 12) FUs configuration, the improvement is 143 % (the average of EIPC increases from 7 to 17 inst/cycle).

The OOO issue policy improves the EILP by executing following PI before the completion of CI. Moreover, the OOO issue policy can conceal the lack of FUs.
For a 6 Bytes FW configuration, for instance, only (4, 4, 4, 4) FUs are required to achieve a 6 inst/cycle EILP average for the OOO issue policy. However, (8, 8, 8, 8) FUs are required for the IO issue policy case.

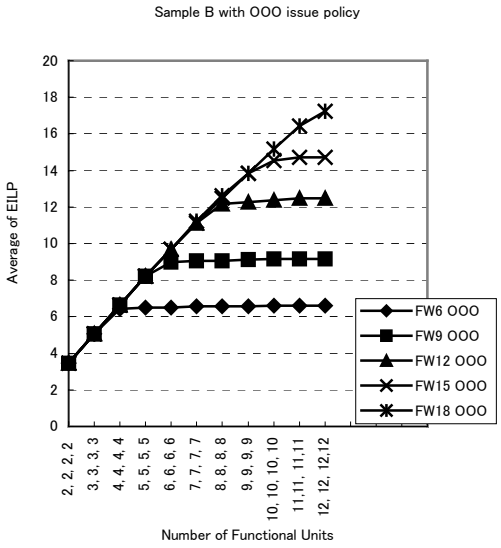

Figure 5.4 EIPC for Sample B with OOO Issue Policy

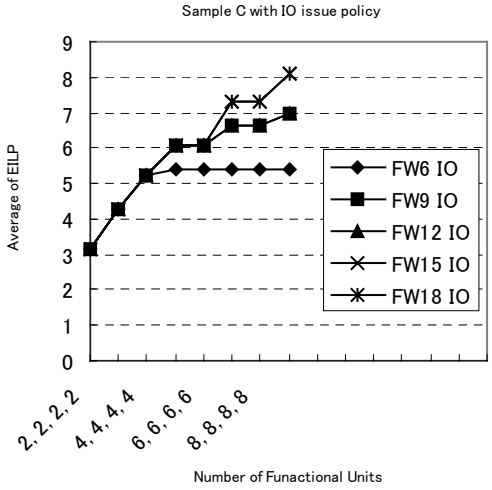

Figure 5. 5 EIPC for Sample C with IO Issue Policy

Figures 5.5 and 5.6 show the average of EIPC with IO and OOO instruction issue policy for sample C. These figures have the same tendency as figures 5.1 and 5.2 for sample A. However, the saturation of the EILP is faster than sample A, because Sample C's ILP is lower than sample A's ILP.

From Figure 6, we can also conclude that 12 Bytes FW is optimal when the PQP is configured with (8, 8, 8, 8) FUs. Figures 5.7 and 5.8 show the average of EIPC respectively with IO and OOO instruction issue policy for sample D. These figures also have the same tendency as Figures 5.2 and 5.3 for sample B. In this case, the EIPC was improved effectively by applying the OOO issue policy.

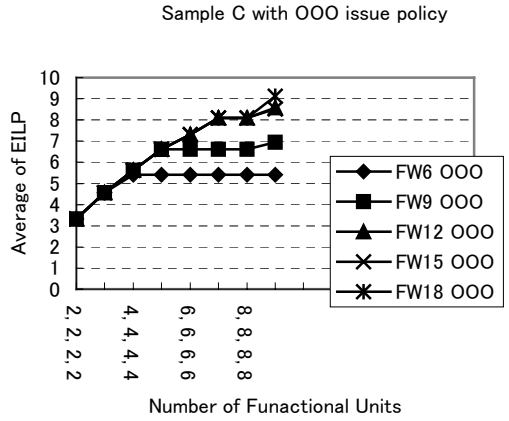

Figure 5. 6 EIPC for Sample C with OOO Issue Policy

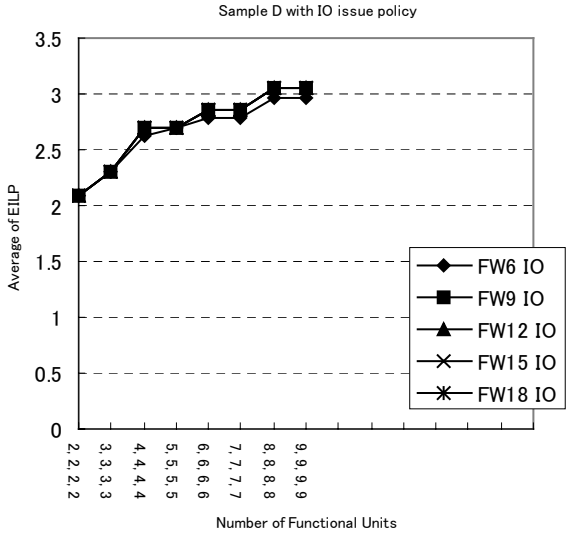

Figure 5. 7 EIPC for Sample D with IO Issue Policy

Table 2. Optimum PQP Design Parameters

Bench. Name	FUs Saturation (for 15 Bytes FW)		Opt.FW For (9,9,9,9) FUs
Sample A	IO	(12,12,12,12)	12
	OOO	(10,10,10,10)	15
Sample B	IO	(8,8,8,8)	6
	OOO	(9,9,9,9)	15
Sample C	IO	(7,7,7,7)	15
	OOO	(7,7,7,7)	18
Sample D	IO	(4,4,4,4)	9
	OOO	(7,7,7,7)	12

We have to note here that the EIPC, with 12 Bytes fetch width, saturates at (7, 7, 7, 7) FUs configuration.
As a summary, since the decision of the number of FUs, which is fixed to be (9, 9, 9, 9), is made according to our sample programs characteristics, the optimal FW (in term optimal EILP) is 15 Bytes. Table 2 shows that the above FW is feasible for almost all types of the used configurations.

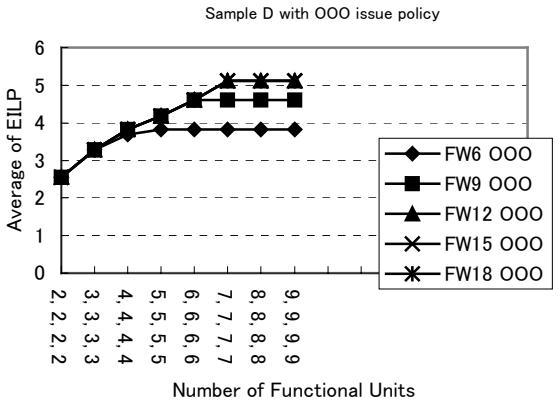

Figure 5. 8 EIPC for Sample D with OOO Issue Policy

6. Conclusions

In this research work we have proposed a parallel Queue processor architecture (PQP) based on Queue computing model. We have presented the novel aspects of the Queue execution model as well as the principle underlying the architecture and the constraints that must be met. Then, to characterize the behavior of the PQP architecture, we presented a preliminary evaluation results over a range of sample programs.
In order to evaluate the performance of the PQP system, the execution of sample programs that exhibit specialized forms of parallelism was simulated for two different execution schemes (in program order (IO) and out of program order (OOO)). Our goal is to estimate the optimal functional units FUs number and the fetch width (FW) of the PQP processor.
From our preliminary evaluations results, we conclude that the performance of the PQP processor linearly increases with the increase of functional units' number for a fixed fetch width. However, it saturates in a certain point. The optimal FUs and FW are (9, 9, 9, 9) and 15 Bytes respectively. The above parameters almost feat for all types of configurations we have used during our evaluation.
Finally, we conclude that our PQP processor can be implemented without considerable hardware complexity. Therefore, the total power consumption and the die area, which are still under investigation, are estimated to be satisfactory comparing to other conventional architectures.
Furthermore, our PQP processor is expected to have a bright feature especially for Internet applications and for new class of terminals requiring simple hardware, small memory footprints, and short programs sizes.
Our feature work is to compare the performance of our PQP with other Load/Store or/and Stack architectures. Further more, we expect to perform several optimizations to the *PQPcom* compiler.

References

[1] S. Okamoto, H. Suzuki, A. Maeda and M. Sowa, Design of a Superscalar Processor Based on Queue Machine Computation Model: IEEE PACRIM, 1999.

[2] G. Sohi, Instructions Issue logic for high-performance, interruptible, Multiple Functional Unit, Pipelined Computer, IEEE Trans. , Computers, 39(2), 1990, .349-359.

[3] D. Wall, Register Windows Vs Register allocation: Proc. SIGPLAN'88 conference on Programming Language Design and implementation, 1988, 234-242.

[4] B. A. Abderazek, N. Kirilka, 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, 2001, 939-944.

[5] K.M Michael and G.C. Harvey, Processor Implementation Using Queue, IEEE, Micro, 1995, 58-66.

[6] K. Philip, Stack Computers, the New Wave (Mountain View Press, 1989).

[7] Sowa Laboratory: http://www.sowa.is.uec.ac.jp

[8] R. Bruno and V. Carla: Data Flow on Queue Machines, 12th Int. IEEE Symposium on Computer Architecture, 1985, 342-351.

[9] H. Suzuki, O. Shusuke, A. Maeda and M. Sowa, Implementation and evaluation of a Superscalar Processor Based on Queue Machine Computation Model, IPSJ SIG, 99(21), 1999, 91-96.

[10] J. E. Smith, G. S. Sohi: The microarchitecture of Superscalar processors," Proceedings of the IEEE, 83(12), 1995, 609-1624.

[11] J. Silc, B. Robic, T. Ungerer: Processor Architecture: From Dataflow to Superscalar and Beyond (NY, Heidelberg: Springer-Verlag, 1999).

[12] B. R. Preiss: Data Flow on a Queue Machine (Doctoral thesis: Department of Electrical Engineering, University of Toronto, 1987).

[13] S. Palacharia, N. P Joupi, and J.E. Smith: Complexity-Effective Superscalar Processor (Ph.D. dissertation, Univ. of Wisconsin, 1998).

[14] M. Sowa, Fundamental of Queue machine, The Univ. of Electro-Communications, Sowa Laboratory, Technical Reports SLL97302, 2000.

[15] M. Sowa: Queue Processor Instruction Set Design, the Univ. of Electro-Communications, Sowa laboratory, Technical Report SLL 00303, 20000.

[16] R. Radhakrishnan, D. Talla and L. K. John: Allowing for ILP in an Embedded Java Processor, Proceedings of IEEE/ACM International Symposium on Computer Architecture, Vancouver, CA, 2000, 294-305.

[17] M. Sowa: Queue Processor Instruction Set Design, the Univ. of Electro-Communications, Sowa laboratory, Technical Report SLL97301, 1997.