Queue Processor Architecture for Novel Queue Computing Paradigm Based on Produced Order Scheme

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

Abstract

This paper proposes novel produced order parallel queue processor architecture. To store intermediate results, the proposed system uses a FIFO queue registers instead of random access registers. Datum is inserted in the queue in produced order scheme and can be reused. We will show that this feature has a profound implication in the areas of parallel execution, programs compactness, hardware simplicity and high execution speed. Our preliminary performance evaluations have shown a significant performance improvement (e.g., 10% to 26% decrease in program size and 6% to 46% decrease in execution time over a range of benchmark programs) when compared with the earlier proposed architecture.

1. Introduction

Nowadays, the shifts in hardware and software technology force designers and users to look at microarchitecture that process instructions stream with high performance, low power consumption, and short program length. In striving for high performance, microarchitecture researches have emphasized instruction-level parallelism processing, which has established in superscalar architectures without major changes to software. Since the program contains no explicit information about available ILP, it must be discovered by the hardware, which must then also construct a plan of actions for exploiting parallelism. In short, computers have far achieved this goal at the expense of tremendous hardware complexity – a complexity that has grown so large as to challenge the industry's ability to deliver ever-higher performance [1,2, 3, 4,12,13].

The possibility of designing a processor architecture that could offers simple hardware, high performance and small code size, while maintaining other characteristics is what led us at SOWA Laboratory to develop a parallel queue processor (PQP) architecture [5,6,8], which stores intermediate data in FIFO queue registers (operand queue (OPQ)) and exploits parallelism dynamically. In PQP, since the OPQ word(s) is/are designated implicitly within the instruction, the instruction length becomes short and independent from the number of physical OPQ words. Moreover, since each instruction removes the required number of data from the head of the OPQ and stores the result of the computations at the tail of the OPQ, the assignment of OPQ word obey to single assignment rule, where datum is stored at once into the OPQ word. As a result, WAR hazard does not occurs.

Earlier in [8,11,10], we proposed a consumed order parallel queue processor (PQPcv) architecture which offers much lower hardware complexity and execution speed than conventional architectures. PQPcv stores data in the OPQ in consumed order scheme. However, PQPcv has several restrictions, which lead to large program size and low execution speed. To overcome this and other architectural design problems we propose in this paper a high performance produced order parallel queue processor (PQPpv) architecture. The PQPpv adopts a new data manipulation technique named produced order scheme. In this scheme, data produced by instructions are inserted in the OPQ in their executions order and can be reused by other instructions. In Fig.2, we give an illustrative example showing data manipulation within the OPQ.
This paper is organized as follows: we first review related work to Queue architectures. Then, we give the consumed and produced order queue execution models’ principles. In section four, we describe the proposed architecture. We give the evaluation results in section five. Finally, we conclude the paper by some concluding remarks and future work.

2. Related Work

Historically, Queue computing idea is traced back to several decades ago. Sowa & al. [5,6,8,9,10] investigated the design constraints of a super scalar processor based on Queue computation model. Another research work was proposed by Bruno [7]. He investigated an indexed Queue machine architecture that uses a FIFO as the underlying mechanism for operands and results manipulations. At the execution stage, 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 produced order parallel Queue processor architecture (PQPpv), which efficiently executes data and control flow programs.

3. PQPcv and PQPPpv Execution Models

3.1. PQPcv Execution Model

Fig.1 (a) shows a data flow program graph for the expressions: “e=ab/c” and “f=ab(c+d)”. Datum is loaded by load instruction (ld), computed by multiply (*), add (+), and divide (/) instructions. Finally, the result is stored back to the memory by store instruction (st). In Fig.1 (b), producer and consumer nodes (instructions) are shown. For example, A2 is a producer node and A4 is a consumer node (A4 is also a producer to m6 node).

In PQPcv execution model, each instruction removes the required number of data from the head of the OPQ, performs some computations and stores the result into the OPQ with consideration for the next consumed order. That is, the result of a computation is inserted at one or more suitable locations in the OPQ according to the needed number of datum. However, items are only removed from the head of the OPQ. The execution order of instructions coincides with the order of the data in the OPQ. The above coincidence order is broken if, within a program: (1) instruction has more than two outgoing arcs, (2) there are arcs which cross each other, and (3) arc skips one or more level (case of “m3” to “A3” in Fig.1 (a)). Fig.1 is an
At "incremented by two and the QT is incremented by "mul" refers fifth and sixth instructions are also executed in parallel. "mul" refers a and b then inserts a*b into OPQ. QT is incremented by two and the QT is incremented by one. "add" refers c and b then inserts “a+b” into the OPQ. At this state, QT and LQH are incremented as shown in Fig.2 (b) (State 2). The eighth instruction (div -2) divides the datum pointed by QH (in this case a*b) by the datum located at ” -2”, negative offset, from QH (in this case c). The QH is incremented (it points to “c+d”). The ninth instruction (mul –1) multiplies the data pointed by QH (in this case “c+d”) with the data located at ” –1” from QH (in this case a*b). After parallel execution of these two instructions, OPQ contents becomes as shown in Fig.2 (b) (State 3). The last two instructions store the result back to the main memory. Since the OPQ becomes empty, LQH, QH and QT point to the same empty location within the OPQ (in Fig.2 (b) (State 4)). We have to notice here that the datum is inserted in produced order and some of them (c and a*b) are used twice. In PQPpv, data are basically referred (not consumed) then discarded when they are not needed. The given example is completed in four execution steps, whereas in PQPcv model it needs eight execution steps.

4. System Architecture
4.1 Instruction Set

The PQPpv processor has a variable instructions length. Zero-operand instructions are one byte wide and all single operand instructions are two bytes wide. While memory instructions, which are also single-
The full instruction set details can be found in [10]. The instruction set is classified into tree classes as follow:

**Basic class:** consists of all arithmetic, shift, rotate, compare and logical operations.

**Mem class:** consists of the memory, control, branch, and loop operations.

**MemIndex Class:** consists of memory indexed address, control, branch and loop operations.

### 4.2 Processing Stages

The architecture has six pipelining stages as described bellow: (1) **Fetch (FS):** 12 bytes are fetched from instruction memory and inserted into a fetch buffer; (2) **Extraction and Decode (EDS):** Extract each instruction and decode the function; (3) **Queue computation (QCS):** Calculate the QH and the QT values for each instruction; (4) **Issue (IS):** Find executable instructions and issue them; (5) **Execution (ES):** Execute instructions; (6) **Write back (WBS):** Write back the results to OPQ or/and data memory.

The EDS stage extracts each instruction from fetched instructions stream and then decode it. The QCS computes QH and QT values of each instruction. They are the values when each instruction is executed in serial. An instruction is ready to issue if all its data operand within the OPQ and a functional unit is available.

Instructions are executed at the execution stage and results are written back to the program memory at the WBS stage. The block diagram of the PQPpv is shown in Fig. 3. It consists of a fetch Unit (FU), an extract decode unit (EDU), a queue computing Unit (QCU), an issue unit (IU), and an execution unit (EU). EU has several types of functional units (FUs). The IU includes a computation instruction issue unit (CIU), a memory instruction issue unit (MIU) and a floating-point instruction issue unit (FIU). The MIU solves memory dependency problem for out of order (OOO) execution. For clarity, the CIU, MIU, and FIU units are not shown in Fig.3. The OPQ is a circular queue controlled by the queue register control unit (QCU).

The FU fetches 12bytes each cycle and sends them to the EDU, where they are extracted and decoded. The number of fetched instructions depends on the width of each instruction. Since instructions width is 1, 2, or 3 bytes the maximum number of fetched instructions can be 12 and the minimum can be 4 instructions. Because the computation is done on the datum in the OPQ, except RAW hazard no other hazards occur. This is due to the single assignment rule.

The IU unit issues instructions in OOO fashion after solving RAW hazard. The simultaneous appearing of continuous executable instructions and the no-hazard characteristics considerably simplifies the mechanism of OOO execution and dynamic parallelism extraction. In addition, since storage locations of intermediate results are implicitly designated, register renaming is not required as in superscalar architectures.

### 4.3 Queue Head and Tail Calculations

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 parallel execution scheme, these pointers (QHP and QTP) 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.

### 4.4. 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 [9]. 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 IS 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 operand queue entry.

### 5. Preliminary Evaluation

#### 5.1 Instruction Level Parallelism (EILP)

In this evaluation, we want to find the optimal number of FUs and the fetch width (FW), which are two major parameters of the proposed processor.
architecture. The EILP (executed instruction level parallelism), over a range of FUs and FW, is evaluated. We used four sample benchmark 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 (SA, SB, SC, and SD) 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.

The distance between PI and CI influences the EILP. 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 distance between PI and CI influences the EILP. 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.

### Table 1. Benchmark programs characteristics

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Max. ILP</th>
<th>Dist. Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA (FFT32)</td>
<td>High</td>
<td>Long</td>
</tr>
<tr>
<td>SB (LU)</td>
<td>High</td>
<td>Short</td>
</tr>
<tr>
<td>SC (Prefix)</td>
<td>Low</td>
<td>Long</td>
</tr>
<tr>
<td>SD (Newton)</td>
<td>Low</td>
<td>Short</td>
</tr>
</tbody>
</table>

The distance between PI and CI influences the EILP. 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 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.

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).

Figure 5 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.

![Fig. 11 EIPC for SD with IO Issue Policy](image1)

Figure 6 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 7 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 8 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.

![Fig. 12 EIPC for SD with OOO Issue Policy](image2)

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.

Figures 9 and 10 show the average of EIPC with IO and OOO instruction issue policy for sample C. These figures have the same tendency as Fig. 5 and 6 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 11 and 12 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 7 and 8 for sample B. In this case, the EIPC was improved effectively by applying the OOO issue policy.

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 12 Bytes.
5.2 Performance Comparison

For this evaluation, we simulated several benchmarks programs for both PQPcv and PQPpv systems on a configurable simulator (PQPsim). Both architectures have 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. We used six kinds of functional unit configurations. We evaluated the average speedup (for 12 bytes fetch width) and the program size for both architectures over a range of benchmark programs (Mtx1, Mtx2, Mtx3, lp1, lp2, lp2). Where, Mtx is a 4x4 matrix programs and lpx is loop programs. We used different kinds of functional unit (ALU) configurations. lp1, lp2 and lp3 are made respectively for 1 ALU (ALU1), 6 ALUs (ALU2) and ideal number of ALUs (ALU3) configurations. Mtx1, Mtx2 and Mtx3 are made for 1 ALU (ALU4), 64 ALUs (ALU5) and 128 ALUs (ALU6) respectively.

Table 3 shows a comparison between the size of different programs for PQPcv and PQPpv. From the above evaluation, we can notice that programs written for PQPcv are larger than programs written for PQPpv. In average, programs for PQPpv are 10 % to 26 % shorter than programs for PQPcv. This is due to the storing order of data and to the way data are consumed within the OPQ. In Fig. 13, the speedup over a range of benchmark programs and ALU configurations is shown. The base architecture is PQPcv. For “Mtx1”, the speedup gain is about 10 % for the three types of ALU configurations. While, for “Mtx2” is about 12%. The highest speedup (about 14%) is found with “Mtx3” benchmark. The speedup is more interesting with “lpx” benchmarks. The average speedup for “lp1” benchmark is about 46. The average speed up is almost similar with “lp2” and “lp3” benchmarks for different ALU configurations.

Table 3. Program Size Comparison

<table>
<thead>
<tr>
<th>Type</th>
<th>Benchmarks (program size given in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQPcv</td>
<td>Mtx1</td>
</tr>
<tr>
<td>645</td>
<td>733</td>
</tr>
<tr>
<td>PQPpv</td>
<td>645</td>
</tr>
</tbody>
</table>

6. Conclusion

In this paper we proposed parallel queue processor architecture (PQPpv) based on produced order queue computation model. We have presented the novel aspects of the PQPpv execution model as well as the principle underlying the architecture and the constraints that must be met.

First, we found that, in average, programs for PQPpv are 10 % to 26 % shorter than programs for PQPcv. PQPcv programs size were also found in our earlier research 66% smaller than programs for RISC (SPARC) code [14]. From the evaluation, we expect that programs for PQPpv are about 50% shorter than those for RISC code.

Second, to find the optimal fetch width and the Functional units number, we used several benchmark programs that exhibit specialized forms of parallelism was simulated for two different execution schemes (in program order (IO) and out of program order (OOO)). From the above evaluation, 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.

We finally compared the PQPpv performance with the PQPcv architecture over a range of benchmark programs. From this comparison, we conclude that the PQPpv achieves better performance than the earlier proposed architecture. When compared with PQPcv, the PQPpv achieves a speedup of 6% to 46%.

Finally, we conclude that the PQPpv, which is still under hardware evaluation [15], is expected to have simple hardware implementation when compared with conventional architectures. The total power
consumption and the die area, which are also still under investigation, are estimated to be satisfactory. Furthermore, the processor is expected to have a bright feature especially for new class of terminals requiring simple hardware, small memory footprints, and short code size.

Our feature work is to evaluate the hardware complexity and compare it with conventional RISC processors. A harder task would be to quantitatively compare PQPpv with GPR machines in some generalized setting. Finally, given the inherent elegance of PQPpv ISA (in term of conceptual economy) and the practical advantage (denser code, in-code parallelism, more natural code, etc.) we suggest that perhaps PQPpv’s ISA might be the ISA of choice for future microprocessors.

7. Acknowledgment

This work is partially supported by the graduate school of Information system at the University of Electro-Communications. The Author wish to thanks all anonymous reviewers for providing several useful comments.

REFERENCES