Graph Transformation Methods and Theoretical Performance Evaluation of Queue Computation Models

Masashi Masuda s1130209

Abstract

Queue is used to store intermediate calculation results into a First In First Out (FIFO) data structure. A Queue system can be classified into three main models according to the rules of enqueueing and dequeueing. These models are called: The Produced-Consumed Order Queue Computation Model, the Consumed Order Queue Computation Model, and the Produced Order Queue Computation Model. There are problems in making programming Queue, and these problems are named the Multiple Data Produced problem, the Cross Arc problem, and the Instruction Hole problem. This thesis presents solutions for these problems and a comparison of three queue computation models' fundamental characteristics (instruction number, instruction level parallelism, execution cycles, program size).

1 Introduction

A queue processor takes operands and stores the results in a First In First Out (FIFO) circular queue register (QREG) instead of the random access register. A given instruction always takes data from the head of the queue (QH) and the computed results of operations or memory loads are put in the tail of the queue (QT). Queue machines have several advantages over register based machines. First, queue programs have high instruction level parallelism (ILP), because they generate the instruction stream by breadth-first traversal. Then, they have short instruction length, because they do not need to specify operands. Data is implicitly taken from QH and the result is implicitly written in QT. Finally, instructions are free from false data dependencies. This eliminates the need for register renaming. However, there are some problems in making programming on a queue, these problems are named the Multiple Data Production problem (MDP), the Cross Arc problem (CA), and the Instruction Hole problem (IH). To solve these problems, the Consumed Order Queue Computing Model and the Produced Order Queue Computing Model have been proposed. Also, a traditional queue computing model named the Produced Consumed Order Queue Computing Model has been suggested. This research classify the solution of program generation problems, and compares each models’ fundamental characteristics (ILP, program size, execution cycle).

2 Queue Computation Models

Overview

Queue computation models can be classified into three main models according to the rules of enqueueing and dequeueing. This section presents three queue computation models’ fundamental features.

First, section 2.1 presents the fundamental features of the Produced Consumed Order Queue Computing Model (QPC). Second, section 2.2 presents the fundamental features of the Consumed Order Queue Computing Model (QC). Third, section 2.3 presents the fundamental features of the Produced Order Queue Computing Model (QP).

2.1 QPC Overview

A QPC is a traditional queue computing model. Fig. 1 shows the rule of QCP. A given instruction implicitly reads its first operand from the QH, its second operand from the next QH. The computed result is finally written into the QT. Fig. 2 shows Directed Acyclic Graph (DAG) for the expression: \( x = (a \times b) - (c/d) \) in which nodes represent instructions, arcs represent data dependency between instructions, and dashed lines represent that instructions traversed from left to right and from the highest level to the lowest level. Instruction streams of QPC are generated by breadth-first traversal.

Fig. 3 shows the state of the QREG at each execution stage of Fig. 2 on serial calculation. Initially, the QREG is empty at state 0: QH and QT point to the same location. The “load a” is produced “a” and written into QT. Therefore, QH stays at the same location, while QT
The “mul” is consumed “a” and “b” from QH, and produced the calculation result of “a*b” and written into QT. After that, QH is moved by two, while QT is moved by one at state 5. In the same way, QH and QT move at state 6 and 7. Finally, the “store” consumes the datum “(a*b)−(c/d)” from QH, and stores it to variable “x”; QH moves by one, while QT stays at the same location at state 8.

Fig.4 shows each execution stage of same expression on parallel queue calculation model. The “load a”, “load b”, “load c”, and “load d” can execute on parallel, because there are no data dependencies in the same level.

2.2 QC Overview

Fig.5 shows the rules of the QC. A given instruction implicitly reads its first operand from the QH, its second operand from the offset with QH. The computed result is finally written into the QT. [1] Therefore, this model can refer to past produced datum. In the QC model, instruction has the specification part of offset. Moreover, QC has “Live Queue Head (LQH)” pointer. It shows use and may be reused data. Data between QH and LQH are called live data.

2.3 QP Overview

Fig.6 shows the rules of the QP. A given instruction implicitly reads its first operand from the QH, its second operand from the offset with QH. The computed result is finally written into the QT. [1] Therefore, this model can refer to past produced datum. In the QP model, instruction has the specification part of offset. Moreover, QP has “Live Queue Head (LQH)” pointer. It shows use and may be reused data. Data between QH and LQH are called live data.

3 Program Generation Problems with Complex Directed Acyclic Graph

In section 2, simple DAG was discussed, using the pure queue computation model to generate valid results. However, executing arbitrary complex expressions may lead to wrong evaluation results. Consider for example, expression “x = (a + b)/(b * (a - c))”. The DAG for this expression is shown in Fig.7. An incorrect result is obtained if the executed instruction stream of Fig.7 is generated using breadth-first traversal. Comparing Fig.2 and Fig.7, defines the following problems:

1. In the DAG shown in Fig.2, all introductions always produce one datum. However, in Fig.7, there are some instructions that produce more than one datum (i.e. “I1” produces one data for “I4” and another data for “I5”). This problem is called the Mul-
4 Proposed Transformation Methods

In this section, the complex DAG transformation method for three Queue computation models (QPC, QC, and QP) is presented.

4.1 Transformation Methods for QPC

Fig.8 shows the proposed method for solving problems in Fig.7. The instruction is added to produce more data, because MDP occurred when the number of consumed data is more than the number of produced data. In Fig.8, this problem was solved by adding the “load” instruction. IH was solved by adding the “rot” instruction that moves datum from “QH” to “QT”. Finally, to eliminate the CA, “store” instruction was used to store datum from the queue register to memory and then “load” instruction was used to load it back into the queue register. These solutions lead to an increase of the number of instruction, program size, the number of execution cycles, and so on.

4.2 Transformation Methods for QC

Fig.9(a) and (b) shows the DAG, instruction stream and QREG at each execution stage of Fig.7. The “load a 3” produces the data “a” and written into “QT” and “QT + 3”. In this, “QT + 1” and “QT + 2” are still empty. As a result, “QH” stays at the same location, while “QT” moves “QT + 1” after carrying out the instruction at state 1. In the same way, “QH” stays at the same location, while “QT” moves “QT + 3”. After that, it is similar to QPC.

Compared to QPC and QC, QC can produce the two data and store them into a location and another location. This property provides a solution to the MDP, CA and IH. However, the instruction length becomes longer by adding the specification part of offset to the instruction. Moreover, it also wastes the queue memory space as it produces the data at the location away from QT.
4.3 Transformation Methods for QP

Fig.10(a) and (b) show the DAG, instruction stream, and QREG at each execution stage of Fig.7 in QP. State 1, 2, 3, 4 are similar to QPC. When the fifth instruction “subo -2" and the opposite instruction of “sub -2" is executed, it substracts “a" from “c" located at “QH - 2" and “QH". The result “a - c" is stored into QT. The sixth instruction “autqh" is moved the QH location from “QH" to “QH + 1". The seventh instruction “mul -3", performs multiplication on data pointed at by “QH" and “QH - 3" which are “a-c" and “b", and stores the result “(a-c)*b" into the location pointed at by QT.

Compared to the QPC, QP is able to refer the past produced data. Therefore, the QP can solve all the problems. However, QP instructions are longer than QPC instructions, because the instruction field is provided for offset. Another disadvantage of this model is that it needs the instructions of the queue pointer control.

5 Benchmark Programs Development

Comparing the QPC, QC, and QP, we selected three benchmark programs (Butterfly, LU decomposition, and Prefix). These programs have all the queue programming problems discussed in section 2. Also, they contain high degrees of parallelism that include many input and output data. This section presents a solution for queue programming problems for each model.

First, it is assumed that there are enough execution resources, and three benchmark programs for QPC, QC, QP (program size, the number of execution cycle, ILP) are compared.

Figure 10: Proposed transformation methods (a)DAG, (b)Execution cycle on QP

Figure 14: DAG of Butterfly program
(a) maximum ILP  
(b) average ILP  

Figure 11: Instruction Level Parallelism

(a) number of instructions  
(b) program size  

Figure 12: Program Size

Figure 13: Execution Cycle
5.1 Benchmark with Multiple Data Produced and Cross Arc Problem

Fig. 14(a) shows the original DAG for a Butterfly (simplified form of Fast Fourier Transformation (FFT)) calculation on 8 inputs data. It contains a lot of MDP and CA problems. Fig. 14(b), (c), (d) show the translated DAGs for QPC, QC, and QP. In QPC program (Fig. 14(b)), cross arc and multiple data produced problems are eliminated by using load/store instructions.

5.2 Benchmark with Multiple Data Produced, Instruction Hole, and Cross Arc Problem

I selected the LU decomposition and the Prefix. These benchmark programs include a lot of MDP, CA, and IH problems. QPC and QC are solved the MDP, CA, and IH problems by using load/store instructions and “rot” instruction. QP model is solved the MDP, CA, and IH problems by using queue pointer control instructions (“stpqh” and “autqh”).

6 Comparison Result

Fig. 11, 12, and 13 show a comparison result of the instruction level parallelism (max and average), program size, and the number of execution cycles for fundamental features of each models based on Butterfly, LU decomposition, and Prefix.

For the Butterfly benchmark, the QPC max ILP is 50% larger than in QC and QP. However, the QPC model used a lot of load/store instructions to solve the MDP and CA problems. Therefore, program size is 42%, and the execution cycle is 28% larger than in QC and QP. For the LU decomposition benchmark, the QP max ILP is 56% larger than in QPC and 67% larger than in QC. Therefore, QP used queue pointer control instructions instead of load/store instructions to solve the MDP and CA problems. Therefore, program size is 17% and execution cycle is 37% smaller than in QPC, and program size is 5% larger and the execution cycle is 11% smaller than in QC. For the Prefix benchmark, the QPC max ILP is 25% larger than in QC. However, QC can reduce the number of instructions because of features. Therefore, program size is 15% and execution cycle is 21% larger than in QC, and program size is 33% and execution cycle is 42% larger than in QP.

7 Conclusion

In this paper, I described the queue programming problems and solutions for MDP, CA, and IH. Also, classified three queue computation models’ fundamental characteristic by making three benchmark programs. As a result, it was shown that the QP model has several advantages over the other models, because it is able to refer to the past produced data, and adapt the program that contains high degrees of instruction level parallelism. For Produced Order Queue Computation Model, average ILP is almost same. Moreover, average program size is 42% smaller than QC, and 5% smaller than QC. Finally, average execution cycles is 56% smaller than QPC, and 14% smaller than QC.

8 Future Work

In this time, I manually wrote the assembly programs for all benchmark programs, because I have not developed tools for generating assembly programs for each queue computation model. My future work is to make compiler, and analyze its more general characteristic.

References
