A thesis submitted in partial satisfaction of the requirements for the degree of Master of Computer Science and Engineering in the Graduate School of the University of Aizu

Produced Order Queue Compiler Design

by

m5131145 Masashi Masuda

February 2011
The thesis titled

*Produced Order Queue Compiler Design*

by

m5131145 Masashi Masuda

is reviewed and approved by:

---

Main referee

*Assistant Professor* \[\text{Date}\]

*Ben Abdallah Abderazek*

*Professor* \[\text{Date}\]

*Kenichi Kuroda*

*Assistant Professor* \[\text{Date}\]

*Hiroshi Saito*

---

The University of Aizu

*February 2011*
Contents

Chapter 1 Introduction .............................................. 1
   1.1 Main Objectives ........................................... 4
   1.2 Organization of this Dissertation ......................... 5

Chapter 2 Target Architecture: QueueCore ..................... 6

Chapter 3 Queue Computation Model .............................. 8
   3.1 Produced Consumed Order Queue Computation Model ....... 8
      3.1.1 Example of Produced Consumed Order Queue Computation
          Model .................................................. 9
   3.2 Consumed Order Queue Computation Model ................ 10
      3.2.1 Example of Consumed Order Queue Computation Model ... 10
   3.3 Produced Order Queue Computation Model .................. 11
      3.3.1 Example of Produced Order Queue Computation Model ... 11

Chapter 4 Produced Order Queue Compiler Framework .......... 13
   4.1 Compiling for QueueCore Instruction Set .................. 13
      4.1.1 Queue Instructions and Format ...................... 18
   4.2 Compiler Framework Design ............................... 19
   4.3 1-offset P-Code Generation Phase ....................... 22
      4.3.1 Augmented LDAG Construction ....................... 23
      4.3.2 dup instruction assignment and ghost nodes elimination .. 26
   4.4 Offset Calculation Phase ................................ 26
   4.5 Instruction Scheduling Phase ............................ 28
   4.6 Statement Scheduling Transformation ..................... 31
   4.7 Assembly Generation Phase ............................... 33

Chapter 5 Compiler Evaluation ..................................... 35
   5.1 Methodology .............................................. 35
   5.2 Compiler Complexity ..................................... 35
   5.3 Queue Compiler Evaluation ............................... 36
   5.4 Comparison of Generated QueueCore Code with Optimized RISC Code ... 37

Chapter 6 Conclusion .............................................. 40

References ......................................................... 42
List of Figures

Figure 3.1  Produced consumed order queue computation model (PC-QCM) rules for queuing and dequeuing ................................. 9
Figure 3.2  PC-QCM’s data flow graph and assembly for LU decomposition 9
Figure 3.3  Consumed order queue computation model (C-QCM) rules for queuing and dequeuing ........................................ 10
Figure 3.4  C-QCM’s data flow graph and assembly for LU decomposition 10
Figure 3.5  Produced order queue computation model (P-QCM) rules for queuing and dequeuing ........................................ 11
Figure 3.6  P-QCM’s data flow graph and assembly for LU decomposition 12

Figure 4.1  Instruction sequence generation from the parse tree .......... 14
Figure 4.2  Instruction sequence generation from DAG ..................... 16
Figure 4.3  P-Code with offset references .................................. 17
Figure 4.4  Duplicate instruction generation ................................. 17
Figure 4.5  Instruction set format ............................................ 19
Figure 4.6  Instruction set format for memory instruction ................. 19
Figure 4.7  Instruction set format for control flow instruction .......... 19
Figure 4.8  Queue Compiler Framework .................................... 21
Figure 4.9  Queue compiler back-end transformations ...................... 24
Figure 4.10 QIR code fragment ............................................. 30
Figure 4.11 Statement scheduling transformation .......................... 32
Figure 4.12 Assembly output for QueueCore processor from C .......... 34

Figure 5.1  Instruction level parallelism improvement of queue compiler over optimizing compiler for a RISC machine .................. 39
Figure 5.2  Normalized code size for two embedded RISC processors and QueueCore ...................................................... 39
List of Tables

Table 4.1  Queue Instructions ................................................. 19
Table 4.2  Data type and sign .................................................. 20
Table 4.3  Queue Intermediate Representation (QIR) specification ...... 29
Table 5.1  Lines of C code for each phase of the queue compiler’s back-end 36
Table 5.2  Instruction category percentages for the compiled benchmarks for the QueueCore .................................................. 37
Table 5.3  QueueCore’s program maximum offset reference value ........ 37
Acknowledgement

I would like to thank professor Ben A. Abderazek for his support and encouragement. I wish to express my appreciation to professor Kenichi Kuroda and associate professor Junji Kitamichi, Yuichi Okuyama and Hiroshi Saito of the University of Aizu for their excellent advices and diligent efforts to guide me through this project. And, I would like to thank the members of Adaptive Systems Lab. at the University of Aizu. These peoples have encouraged and supported me.
Abstract

Queue processor arranges high-speed registers in a first-in-first-out queue. All read accesses are performed in the head of the queue and all writes accesses are performed in the tail of the queue. This characteristic allows the exploitation of maximum parallelism and improves code density. Compiling for the QueueCore requires a new approach since the concept of registers disappears. We propose a new efficient code generation algorithm for the QueueCore. Our queue compiler translates any programs written in C language into queue processor’s assembly codes. The queue compiler design is completely different from any other existing compiler, this due to the special characteristics of queue computing. First, compiler generates the data flow graphs from the input program. Then a set of custom analysis and transformations are performed to compute the offset reference values. Finally, the data flow graphs are scheduled in a level-order manner to generate the final instruction sequence for the target queue machine. The queue compiler can compile any program into queue object code. For a set of numerical benchmark programs our compiler extracts more parallelism than the optimizing compiler for a RISC machine by a factor of 1.38. Through the use of QueueCore’s reduced instruction set, we are able to generate 20% and 26% denser code than two embedded RISC processors.
Chapter 1

Introduction

Instruction level parallelism (ILP) is a key to improve the performance of modern processors. ILP allows the instructions of a sequential program to be executed in parallel on multiple functional units. Careful scheduling of instructions is crucial to achieve high performance. Effective scheduling for the exploitation of ILP greatly depends on two factors: the hardware features, and the compiler techniques. In super-scalar processors, the compiler exposes ILP by rearranging instructions. However, the final schedule is decided at run-time by the hardware [1]. In VLIW machines, the scheduling is decided at compile-time by aggressive static scheduling techniques [2, 3].

Sophisticated compiler optimization have been developed to expose high amounts of ILP in loop regions [4] where many scientific and multimedia programs spend most of their execution time. The purpose of some loop transformations such as loop unrolling is to enlarge basic blocks by combining instructions executed in multiple iterations to a single iteration. Basic block means the vertices or nodes in a data flow graph. A popular loop scheduling technique is modulo scheduling [5, 6] where the iterations of a loop are paralyzed in such a way that a new iteration initiates before the previous iteration has completed execution.
These static scheduling algorithms improve greatly the performance of the applications at the cost of increasing the register pressure [7]. When the schedule requires more registers than those available in the processor, the compiler must insert spill code to fit the application in the available number of architects registers [8]. Many high performance architectures born in the last decade [9, 10, 11] were designed on the assumption that applications could not make effective use of more than 32 registers [12]. Recent studies have shown that the register requirements for the same kind of applications using the current compiler technology demands more than 64 registers [13]. High ILP register requirements have direct impact in the processor performance as a large number of registers need to be accessed concurrently. The number of ports to access the register file affect the access time and the power consumption. In order to maintain clock speed and low power consumption, high performance embedded and digital signal processors have implemented partitioned register banks [14] instead of a large monolithic register file.

Several software solutions for the compiler have been proposed to reduce the register requirements of modulo schedules [15]. Other studies have focused on the compilation issues for partitioned register files [16, 17]. A hardware/compiler technique to alleviate register pressure is to provide more registers than those addressable by the instruction encoding. In [18, 19], the usage of queue register files has been proposed to store the live variables in a software pipelined loop schedule while minimizing the pressure on the architected registers. The work in [20] proposes the use of register windows to give the illusion of a large register file without affecting the instruction set bits.
An alternative to hide the registers from the instruction set encoding is by using a queue machine. A queue machine uses a first-in first-out structure, called the operand queue, as the intermediate storage location for computations. Instructions read and write the operand queue implicitly. Not having explicit operands in the instructions make instructions short improving code density. False dependencies disappear from programs eliminating the need for register renaming logic that reduces circuitry and improves power consumption [21]. Queue computers have been studied in several works.

Preiss [22] was the first who investigated the possibility of evaluating expression trees and highlighted the problems of evaluating directed acyclic graphs (DAG) in an abstract queue machine. In [23], Okamoto proposed the hardware design of a superscalar queue machine. Schmit et. al [24] use a queue machine as the execution layer for reconfigurable hardware. They transform the program’s data flow graph (DFG) into a spatial representation that can be executed in a simple queue machine. This transformation inserts extra special instructions to guarantee correct execution by allowing every variable to be produced and consumed only once. Their experiments show that the execution of programs in their queue machine have the potential of exploiting high levels of parallelism while keeping code size less than a RISC instruction set.

In our previous work [25, 26, 27], we proposed a novel 32-bit QueueCore processor with a 16-bit instruction set format. Our approach is to allow variables to be produced only once but consumed multiple times. We sacrifice some bits in the instruction set for an offset reference that indicates the location of a variable to be reused relative to the head of the queue. The goal is to allow DAGs to be executed without any transformations that increase the instruction count. Although this flexibility costs a
few bits in the instruction set, this design is able to produce programs with high ILP
and code density.

Compiling for queue machines still is an undecided art. Only few efforts have been
made to develop the code generation algorithms for the queue computation model.
A linear time algorithm to recognize the covering of a DAG in one queue has been
demonstrated in [28] together with the proof of NP-completeness for recognizing a
2-queue DAG. In [24], Schmit. et. al propose a heuristic algorithm to cover any
DAG in one queue by adding special instructions to the data flow graph. From their
experimental results a large amount of additional instructions is reported, making this
technique insufficient for achieving small code size.

Despite the large amount of extra instructions, the resulting size of their tested pro-
grams is smaller than RISC design. We tried to develop a queue compiler based on
a retargetable compiler for register machines [29]. A large amount of registers was
defined in the compiler to model the operand queue and to avoid spill code. Never-
theless, mapping register code into the queue computation model turns into low code
quality with excess of instructions making. This approach inappropriate for both, a
native compiler for our queue machines, and the generation of compact code. In this
article we present a new efficient code generation algorithm implemented in a com-
piler specifically designed for the Queue computation model. Our compiler generates
assembly code from C programs.

1.1 Main Objectives

The primary goal of this desertion is the research and development of the principles to
compile high-level languages into machine code for queue-based processors. As a re-
sult of this investigation, we present a queue compiler framework able to translate and accommodate any program in an actual queue processor. Second, we invent new techniques to generate code for specific objective functions such as code size, and queue register size control. Third, we aim to the generation of code with quality comparable to that of production and research compilers in terms of number of instructions, and instruction level parallelism.

1.2 Organization of this Dissertation

The queue compiler exposes natural ILP from the input programs to the QueueCore processor. Experimental results show that our compiler can extract more parallelism for the QueueCore than an ILP compiler for a RISC machine, and also generates programs with lower code size.

This dissertation is organized as follows: Chapter 2 presents the QueueCore characteristics. Chapter 3 presents the queue computing models. Chapter 4 presents the concepts, abstractions, and algorithms to develop queue compiler. Chapter 5 presents the evaluation of queue computing by means of compiling a set of benchmark programs with the presented queue compiler. Chapter 6 discuss several aspects on the development of a queue compiler and concludes.
Chapter 2

Target Architecture: QueueCore

The queue machine has several promising advantages. First, queue programs have higher instruction level parallelism (ILP) than random access register based machine. Second, instructions of queue processor are short. Finally, queue instructions are free from false dependencies. These advantages are come from the queue several characteristics. A queue processor takes operands and stores the results in first-in first-out (FIFO) queue register (QREG) instead of the random access register. A given instruction implicitly reads its first operand from the head of QREG, its second operand from a location explicitly addressed with an offset from the first operand location. The computed results of operations or memory loads are saved in the tail of the QREG.

The queue program is generated by the breadth first scheduling, or level-order scheduling, from a leveled directed acyclic graph (LDAG). The level-order scheduling can realize the queue processor architectures which does not need the false dependency. This no false dependency leads a nonnecessity of a register renaming and simple architecture. Moreover, the no false dependency makes the high ILP.

The architecture is a 32-bit processor with a 16-bit wide producer order queue computation model (QCM) instruction set architecture based on the produced order
parallel QCM [27]. The instruction format reserves 8-bit for the opcode and 8-bit for the operand. The operand field is used in binary operations to specify the offset reference value with respect of head of the queue from which the second source operand is dequeued. Unary operations have the freedom to dequeue their only source operand. Memory operations use the operand field to represent the offset and base register, or immediate value. For cases when 8-bit is not enough to represent an immediate value or an offset for a memory instruction, a special instruction named “covop” is inserted before the conflicting memory instruction. The “covop” instruction extends the operand field of the following instruction.

QueueCore defines a set of specific purpose registers available to programmers to be used as the frame pointer register ($fp), stack pointer register ($sp), and return address register ($ra). The frame pointer register serves as a base register to access local variables, incoming parameters, and saved registers. The stack pointer register is used as a base address for outgoing parameters to other functions.
Chapter 3

Queue Computation Model

Queue computation refers to the evaluation of expression using a first-in first-out data structure as the place where operands and intermediate results are saved. A first-in first-out (FIFO) data structure, or queue, is a well known data structure used in computer science. The queue data structure follows two simple rules for inserting and removing items from it. The first rule is that all items to be inserted in the queue are inserted through the rear, or tail, of the queue. The process of inserting items into the queue is also known as enqueue. The second rule establishes that all items that are removed from the queue are removed from the head of the queue. Similarly, the process of removing items from the queue is known as dequeue. Two pointers are required to keep track where the tails and the head of the queue are. These two pointers are known as Queue Head Pointer and Queue Tail Pointer. From now on these two pointer will be referred as QH and QT respectively.

3.1 Produced Consumed Order Queue Computation Model

The produced consumed order queue computation model strictly follows the rules of queuing and dequeuing. Insertions to the queue are done exclusively through QT. Removals from the queue are done exclusively through QH. This is the strictest model
available for queue computers. This queue computation model is also known as PC-QCM. Figure 3.1 shows the only places where queuing and dequeuing can be done.

Figure 3.1: Produced consumed order queue computation model (PC-QCM) rules for queuing and dequeuing

### 3.1.1 Example of Produced Consumed Order Queue Computation Model

Figure 3.2 shows the example of PC-QCM’s data flow graph and assembly code.

![Data Flow Graph for PC-QCM](image)

**Figure 3.2:** PC-QCM’s data flow graph and assembly for LU decomposition
3.2 Consumed Order Queue Computation Model

The consumed order queue computation model, or C-QCM, allows elements to be enqueued from a reference from QT. Figure 3.3 shows that enqueuing can be done through QT or a reference from QT as an offset QT\( - N \) and QT\( + M \). Where \( N \) and \( M \) is an arbitrary value that reaches the place where the new enqueued element will be placed. C-QCM restricts dequeuing to be exclusively at QH.

![Consumed order queue computation model (C-QCM) rules for queuing and dequeuing](image)

**Figure 3.3:** Consumed order queue computation model (C-QCM) rules for queuing and dequeuing

3.2.1 Example of Consumed Order Queue Computation Model

Figure 3.4 shows the example of C-QCM’s data flow graph and assembly code.

![Data flow graph and assembly for LU decomposition](image)

**Figure 3.4:** C-QCM’s data flow graph and assembly for LU decomposition
3.3 Produced Order Queue Computation Model

The produced order queue computation model, or P-QCM, allows elements to be dequeued from a reference from QH. Figure 3.5 shows that dequeuing can be done through QH or a reference from QH as an offset $\text{QH} - N$ and $\text{QH} + M$. Where $N$ and $M$ is an arbitrary value that reaches the element to be dequeued with respect of QH. Although the dequeuing has flexibility, this model restricts the queuing to be done exclusively through QT. Following this rule the elements from the queue, or data, is consumed from a reference from QH and elements or data is produced (enqueued) at a fixed point QT. This characteristic gives this model its name, data is always enqueued, or produced at QT.

![Figure 3.5: Produced order queue computation model (P-QCM) rules for queuing and dequeuing](image)

3.3.1 Example of Produced Order Queue Computation Model

Figure 3.6 shows the example of P-QCM’s data flow graph and assembly code.
Figure 3.6: P-QCM’s data flow graph and assembly for LU decomposition
Chapter 4

Produced Order Queue Compiler Framework

This chapter presents the main contribution of this dissertation which is the establishment of the techniques to develop a queue compiler for the produced order queue computation model. The structure of the queue compiler is different from traditional compilers since it integrates the concept of queue computing to all phases of code generation. We introduce a novel data structure that facilitates the compilation process and allows the calculation of offset references and level-order scheduling.

4.1 Compiling for QueueCore Instruction Set

The instruction sequence to correctly evaluate a given expression is generated from a level-order traversal of the expressions’ parse tree [22]. A level-order traversal visits all the nodes in the parse tree from left to right starting from the deepest level towards the root as shown in Figure 4.1.(a). The generated instruction sequence is shown in Figure 4.1.(b).

All nodes in every level are independent from each other and can be processed in parallel. Every node may consume and produce data. For example, a load operation
produces one datum and consumes none, a binary operation consumes two data and produces one. A QSTATE is the relationship between all the nodes in a level that can be processed in parallel and the total number of data consumed and produced by the operations in that level. Figure 4.1.(c) shows the production and consumption degrees of the QSTATEs for the sample expression.

Figure 4.1: Instruction sequence generation from the parse tree
Although the instruction sequence from a directed acyclic graph (DAG) is obtained also from a level-order traversal, there are some cases where the basic rules of enqueueing and dequeueing are not enough to guarantee correctness of the program [22]. Parse trees have the property that every produced element has one consumer therefore data cannot be used more than one. To reduce the size of the trees a DAG is constructed where elements have no limitation in the number of consumers, thus allowing reuse data. Figure 4.2.(a) shows the evaluation of an expression’s DAG that leads to incorrect results. In Figure 4.2.(c), notice that at QSTATE 1 there are three operands produced, and at QSTATE 2 the operations consume four operands. The add operation in Figure 4.2.(b) consumes two operands, \( a, b \), and produces one, the result of the addition \( a + b \). The sub operation consumes two operands that should be \( b, c \), instead it consumes operands \( c, a + b \).

In our previous work [27] we proposed a solution for this problem. We give flexibility to the dequeuing rule to get operands from any location in the operand queue. In other words, we allow operands to be consumed multiple times. The desired operand’s location is relative to the head of the queue and it is specified in the instruction as an offset reference, \( QH-N \). As the enqueuing rule, production of data, remains fixed at QT, we name this model the Producer Order Queue Computation Model.

Figure 4.3 shows the code for this model that solves the problems in Figure 4.2. Notice that add, sub, div instructions have offset references that indicate the place relative to QH where the operands should be taken. The “sub -1, 0” instruction now takes operand \( b \) from \( QH-1 \), and operand \( c \) from QH itself, \( QH+0 \).

We name the code for this model P-Code. This nontraditional computation model
Figure 4.2: Instruction sequence generation from DAG

requires new compiler support to statically determine the value of the offset references.

Using QueueCore’s single operand instruction set, the evaluation of binary instructions where both source operands are not in QH is not possible. To ensure correct evaluation of this case, a special instruction of \textit{dup} has been implemented in the processor. The \textit{dup} instruction takes a variable in the operand queue and places a copy in QT. The compiler is responsible of placing \textit{dup} instructions to guarantee that binary
ld   a
ld   b
ld   c
add 0, 1
sub -1, 0
div 0, 1
st   x

Figure 4.3: P-Code with offset references

(a) DAG
(b) DAG with dup instruction
(c) Instruction Sequence

ld   a
dup 0
neg 0
add 0, -1
div 0, 1
st   x

Figure 4.4: Duplicate instruction generation

instructions will have their first operand available always at QH. By placing a copy in QH the second operand can be taken from an arbitrary position in the operand queue by using QueueCore’s one operand instruction set. Let the expression $x = -a/(a + a)$
be evaluated using QueueCore’s one offset instruction set, its DAG is shown in Figure 4.4.(a).

Notice that the level $L_3$ produces only one operand, $a$, which is consumed by the following instruction, neg. The add instruction is constrained to take its first source operand directly from QH, and its second operand has freedom to be taken from QH–N. For this case, the dup instruction is inserted to make a copy of $a$ available as the first source operand of instruction add as indicated by the dashed line in Figure 4.4.(b). Notice that level $L_3$ in Figure 4.4.(b) produces two data instead of one. The instruction sequence using QueueCore’s one offset instruction set is shown in Figure 4.4.(c). This mechanism allows safe evaluation of binary operations in a DAG using one offset instruction set at the cost of the insertion of dup instructions.

The QueueCore’s instruction set format was decided from our design space exploration [30]. We found that binary operations that require the insertion of dup instructions are rare in program DAGs. We believe that one operand instruction set is a good design to keep a balance between compact instructions and program requirements.

### 4.1.1 Queue Instructions and Format

The target instruction set, or P-Code allows two offset references. P-Code instructions are classified in seven classes as shown in Table 4.1. Arithmetic & Logic, Memory, Comparison, Control Flow, Conversion, Special, and Queue. Table 4.2 shows the equivalence between C language data types and P-Code data types. P-Code’s instruction set format is shown in Figure 4.5.

For memory instructions that do not use offset references. Figure 4.6 shows the memory instruction’s format.
Figure 4.5: Instruction set format

Figure 4.6: Instruction set format for memory instruction

Figure 4.7 shows the control flow instruction’s format. Control flow instructions do not have a data type and the only operand is the target label symbol representing the target of jumps or function names.

Figure 4.7: Instruction set format for control flow instruction

Table 4.1: Queue Instructions

<table>
<thead>
<tr>
<th>Class</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic &amp; Logic</td>
<td>add, sub, div, mod, mul, neg, rsh, lsh, ior, xor, and, not, rrot, lrot, abs</td>
</tr>
<tr>
<td>Memory</td>
<td>ld, st, ldi, lea, sld, sst</td>
</tr>
<tr>
<td>Comparison</td>
<td>ceq, cne, clt, cle, cgt, cge</td>
</tr>
<tr>
<td>Control Flow</td>
<td>bt, bf, j, jal, ret</td>
</tr>
<tr>
<td>Conversion</td>
<td>conv</td>
</tr>
<tr>
<td>Special</td>
<td>dup, rot</td>
</tr>
<tr>
<td>Queue</td>
<td>moveqh, moveqt</td>
</tr>
</tbody>
</table>

4.2 Compiler Framework Design

We developed a C compiler for the QueueCore that uses GCC’s 4.0.2 consists of seven phases, including the front-end and middle-end. Figure 4.8 shows the phases and intermediate representations of the queue compiler infrastructure. The front-end parses C
program into abstract syntax tree (AST). Then the middle-end converts the ASTs into a language-and-machine-independent format called GIMPLE [31]. A set of tree transformations and optimizations to remove redundant codes and substitute sequences of code with more efficient sequences is optionally available from the GCC’s middle-end for this representation. Although these optimizations are available in our compiler, until this point our primary goal was to develop the basic compiler infrastructure for the QueueCore and we have not validated the correctness of programs compiled with these optimizations. We wrote a custom back-end that takes GIMPLE intermediate representation and generates assembly code for the QueueCore processor.

The queue compiler framework consists of the remaining five phases. *1-offset code generation* phase which takes QTrees as input and generates a hierarchical structure formed by levels according data dependencies among instructions. This hierarchical structure is called a leveled directed acyclic graph (LDAG) on which the compiler performs the offset calculation and tracking of the head of the queue. This structure facilitates also the scheduling of the programs as all nodes are organized into levels. The second phase is the *offset calculation*, consumes the LDAGs and for every instruc-

### Table 4.2: Data type and sign

<table>
<thead>
<tr>
<th>P-Code Description</th>
<th>Data Type</th>
<th>C-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Byte Signed</td>
<td>ibs</td>
<td>signed char</td>
</tr>
<tr>
<td>Integer Byte Unsigned</td>
<td>ibu</td>
<td>unsigned char</td>
</tr>
<tr>
<td>Integer Half-word Signed</td>
<td>ihs</td>
<td>signed short</td>
</tr>
<tr>
<td>Integer Half-word Unsigned</td>
<td>ihu</td>
<td>unsigned short</td>
</tr>
<tr>
<td>Integer Word Signed</td>
<td>iws</td>
<td>signed int,  long</td>
</tr>
<tr>
<td>Integer Word Unsigned</td>
<td>iwu</td>
<td>unsigned int, long</td>
</tr>
<tr>
<td>Integer Long Signed</td>
<td>ils</td>
<td>signed long</td>
</tr>
<tr>
<td>Integer Long Unsigned</td>
<td>ilu</td>
<td>unsigned long long</td>
</tr>
</tbody>
</table>
Figure 4.8: Queue Compiler Framework

tion it computes the offset reference values to reach it operands. Offset references are
annotated in the LDAGs.

The third phase, instruction scheduling traverses the LDAGs and produces a level-
order scheduling of instructions that compiles with the queue computing principle. As a result, this phase generates a low level intermediate representation called QIR. QIR is a linear representation designed to facilitate target dependent optimizations.

The fourth phase, statement scheduling, extract the natural instruction level parallelism (ILP).

Finally, the fifth phase, assembly generation, convert QIR into target assembly code.

The following sections describe in detail the phases, the algorithms, and the intermediate representations utilized by our queue compiler to generate assembly code from any C program.

4.3 1-offset P-Code Generation Phase

GIMPLE is a three address code intermediate representation used by GCC’s middle-end to perform optimizations. Three address code is a popular intermediate representation in compilers that expresses well the instructions for a register machine, but fails to express instructions for the queue computation model. The first task of our back-end is to expand the GIMPLE representation into QTrees. QTrees are ASTs without limitation in the number of operands and operations. GIMPLE’s high-level constructs for arrays, pointers, structures, unions, subroutine calls, are expressed in simpler GIMPLE constructs to match the instructions available in a generic queue hardware.

The task of the first phase of our back-end, 1-offset P-Code Generation, is to constrain the binary instructions in the program to have at most one offset reference. This phase detects the cases when \texttt{dup} instructions need to be inserted and it determines the correct place. The code generator takes as input QTrees and generates leveled directed
acyclic graphs (LDAGs) as output. A leveled DAG is a data structure that binds the nodes in a DAG to levels [28]. We chose LDAGs as data structure to model the data dependencies between instructions and QSTATEs. Figure 4.9 shows the transformations the C program suffers when it is converted to GIMPLE, QTrees, and LDAGs.

The algorithm works in two stages. The first stage converts QTrees to LDAGs augmented with ghost nodes. A ghost node is a node without operation that serves as a mark for the algorithm. The second stage takes the augmented LDAGs and remove all ghost nodes by deciding whether a ghost node becomes a \texttt{dup} instruction or is removed.

4.3.1 Augmented LDAG Construction

QTrees are transformed into LDAGs by a post-order depth-first recursive traversal over the QTree. All nodes are recorded in a lookup table when they first appear, and are created in the corresponding level of the LDAG together with its edge to the parent node. Two restrictions are imposed over the LDAGs for the 1-offset P-Code QCM.

**Definition 4.3.1.** A level is an ordered list of elements with at least one element.

**Definition 4.3.2.** The sink of an edge must be always in a deeper or same level than its source.

**Definition 4.3.3.** An edge to a ghost node spans only one level.

When an operand is found in the lookup table the Definition 4.3.2 must be kept. Line 5 in Algorithm 1 is reached when the operand is found in the lookup table and it has a shallower level (closer to the root) than the new level. The function
dag_ghost_move_node() moves the operand to the new level, updates the lookup table, converts the old node into a ghost node, and creates an edge from the ghost node to the new created node. The function insert_ghost_same_level() in Line 8 is reached when the level of the operand in the lookup table is the same to the new level. This
function creates a new ghost node in the new level, makes an edge from the parent node to the ghost node, and an edge from the ghost node to the element matched in the lookup table. These two functions build LDAGs augmented with ghost nodes that obey Definitions 4.3.2 and 4.3.3.

Algorithm 1 dag_levelize_ghost (tree $t$, level)
1: nextlevel $\leftarrow$ level + 1
2: match $\leftarrow$ lookup ($t$)
3: if match $\neq$ null then
4: if match.level < nextlevel then
5: relink $\leftarrow$ dag_ghost_move_node (nextlevel, $t$, match)
6: return relink
7: else if match.level = lookup ($t$) then
8: relink $\leftarrow$ insert_ghost_same_level (nextlevel, match)
9: return relink
10: else
11: return match
12: end if
13: end if
14: /* Insert the node to a new level or existing one */
15: if nextlevel > get_Last_Level() then
16: new $\leftarrow$ make_new_level ($t$, nextlevel)
17: record (new)
18: else
19: new $\leftarrow$ append_to_level ($t$, nextlevel)
20: record (new)
21: end if
22: /* Post-Order Depth First Recursion */
23: if $t$ is binary operation then
24: lhs $\leftarrow$ dag_levelize_ghost ($t$.left, nextlevel)
25: make_edge (new, lhs)
26: rhs $\leftarrow$ dag_levelize_ghost ($t$.right, nextlevel)
27: make_edge (new, rhs)
28: else if $t$ is unary operation then
29: child $\leftarrow$ dag_levelize_ghost ($t$.child, nextlevel)
30: make_edge (new, child)
31: end if
32: return new

25
4.3.2 dup instruction assignment and ghost nodes elimination

The second and final stage of the 1-offset P-Code generation algorithm takes the augmented LDAG and decides what ghost nodes are replaced by a dup node or eliminated from the LDAG. The only operations that need a dup instruction are those binary operations whose both operands are away from QH. The augmented LDAG with ghost nodes facilitates the task of identifying those instructions. All binary operations having ghost nodes as their left and right children need to be transformed as follows. The ghost node in the left children is substituted by a dup node, and the ghost node in the right children is eliminated from the LDAG. For those binary operations with only one ghost node as the left or right children, the ghost node is eliminated from the LDAG. Algorithm 2 describes the function dup_assignment().

Algorithm 2 dup_assignment (Node i)
1: if isBinary (i) then
2:     if isGhost (i.left) and isGhost (i.right) then
3:        dup_assign_node (i.left)
4:        dag_remove_node (i.right)
5:     else if isGhost (i.left) then
6:        dag_remove_node (i.left)
7:     else if isGhost (i.right) then
8:        dag_remove_node (i.right)
9:     end if
10: end if
11: return

4.4 Offset Calculation Phase

Once the LDAGs including dup instructions have been built, the next step is to calculate the offset reference values for the instructions. Following the definition of the producer order QCM, the offset reference value of an instruction represents the dis-
tance, in number of queue words, between the position of QH and the operand to be dequeued. The main challenge in the calculation of offset values is to determine the QH relative position with respect of every operation. We define the following properties to facilitate the description of the algorithm to find the position of QH with respect of any node in the LDAG.

**Definition 4.4.1.** An $\alpha$-node is the first element of a level.

**Definition 4.4.2.** The QH position with respect of the $\alpha$-node of Level-$j$ is always at the $\alpha$-node of the next level, Level-$(j+1)$.

**Definition 4.4.3.** A level-order traversal of a LDAG is a walk of all nodes in every level (from the deepest to the root) starting from the $\alpha$-node.

**Definition 4.4.4.** The distance between two nodes in a LDAG, $\delta(u, v)$, is the number of nodes found in a level-order traversal between $u$ and $v$ including $u$.

**Definition 4.4.5.** A hard edge is a dependency edge between two nodes that spans only one level.

Let $p_n$ be a node for which the QH position must be found. QH relative position with respect of $p_n$ is found after a node in a traversal $P_i$ from $p_{n-1}$ to $p_0$ ($\alpha$-node) meets one of two conditions. The first condition is that the node is the $\alpha$-node, $P_i = p_0$. From Definition 4.4.2, QH position is at $\alpha$-node of the next level $lev(p) + 1$. The second condition is that $P_i$ is a binary or unary operation and has a hard edge to one of its operands $q_m$. QH position is given by $q_m$'s following node as a result of a level-order traversal. Notice that $q_m$’s following node can be $q_{m+1}$, or the $\alpha$-node of $lev(q_m) + 1$ if $q_m$ is the last node in $lev(q_m)$. The proposed algorithm is described in Algorithm 3.
After the QH position with respect of \( p_n \) has been found, the only operation to calculate the offset reference value for each of \( p_n \)’s operands is to measure the distance \( \delta \) between QH’s position and the operand’s position as described in Algorithm 4.

In brief, for all nodes in a LDAG \( w \), the offset reference values to their operands are calculated by determining the position of QH with respect of every node, and measuring the distance to the operands. Every edge is annotated with its offset reference value.

Algorithm 3 \texttt{qh\_pos} (LDAG \( w \), Node \( u \))
1: \( I \leftarrow \text{getLevel} (u) \)
2: \textbf{for} \( i \leftarrow u.\text{prev} \) to \( I.\alpha\)-node \textbf{do}
3: \hspace{1em} \textbf{if} isOperation \( (i) \) \textbf{then}
4: \hspace{2em} \textbf{if} isHardEdge \( (i.\text{right}) \) \textbf{then}
5: \hspace{3em} \( v \leftarrow \text{BFS\_nextnode} (i.\text{right}) \)
6: \hspace{3em} \textbf{return} \( v \)
7: \hspace{2em} \textbf{end if}
8: \hspace{2em} \textbf{if} isHardEdge \( (i.\text{left}) \) \textbf{then}
9: \hspace{3em} \( v \leftarrow \text{BFS\_nextnode} (i.\text{left}) \)
10: \hspace{3em} \textbf{return} \( v \)
11: \hspace{2em} \textbf{end if}
12: \hspace{1em} \textbf{end if}
13: \hspace{1em} \textbf{end for}
14: \( L \leftarrow \text{getNextLevel} (u) \)
15: \( v \leftarrow L.\alpha\)-node
16: \textbf{return} \( v \)

Algorithm 4 \texttt{OpOffset} (LDAG \( w \), Node \( v \), Operand \( r \))
1: \( \text{offset} \leftarrow \delta (\text{qh\_pos} (w, v), r) \)
2: \textbf{return} \( \text{offset} \)

4.5 Instruction Scheduling Phase

The instruction scheduling algorithm of our compiler is a variation of basic block scheduling [3] where the only difference is that instructions are generated from a level-order traversal of the LDAGs. The input of the algorithm is an LDAG annotated with
offset reference values. For every level in the LDAG, from the deepest level to the root level, all nodes are traversed from left to right and an equivalent low level intermediate representation instruction is selected for every visited node. Instruction selection was simplified by having one low level instruction for every high level instruction in the LDAG representation. The output of the instruction scheduling is a Queue Intermediate Representation (QIR) list. QIR is a single operand low level intermediate representation capable of expressing the instruction set of the QueueCore. The only operand is used for memory operations and branch instructions. Offset reference values are encoded as attributes in the QIR instructions.

Table 4.3 shows the full QIR specifications. The Operands class identifies the
Figure 4.10: QIR code fragment

type of operand for the class One Operand, which is the only type of instructions that require an explicit operand. It includes immediate values, local variables, global variables, parameters, arguments, labels, return value, temporaries, symbols, and QH. The classes Binary, Unary, and Compare do not use any operand and offset values are encoded as attributes of the instruction itself. The Special class includes all instructions
where the semantics do not fall in the other categories and require special handling by
the code generator.

Figure 4.10 shows the QIR list for the LDAG in Figure 4.9.(d). The QIR includes
annotations depicted in Figure 4.10 with the prefix QMARK.*.

An extra responsibility of this phase is to check code correctness of the 1-offset
P-Code generation algorithm by comparing with zero the value of the offset reference
for the first operand of binary instructions based on the assumption that the 1-offset
P-Code generation algorithm constrains all instructions to have at most one offset ref-
erence. For every compiled function this phase also inserts the QIR instructions for the
function’s prologue and epilogue.

4.6 Statement Scheduling Transformation

Statement scheduling transformation reorders the instructions of a sequential program
in such a way that all independent instructions from different statements are in the
same level can be executed in parallel. This phase makes a dependency analysis on in-
dividual instructions of different statements looking for conflicts in memory locations.
Statements are considered as the transformation units. Whenever an instruction is re-
ordered, the entire data flow graph of the statement to where it belongs is reordered to
keep its original shape. In this way, all offsets computed by the offset calculation phase
remain the same, and the data flow graph is not altered. The data dependency analy-
ysis finds conflicting memory accesses whenever two instructions have the same offset
with respect of the base register. Instructions that may alias memory locations are
merged safely using a conservative approach to guarantee correctness of the program.
Statements with branch instructions and function calls are non-mergeable.
S1: x = a + b;
S2: y = x + (1 / y);
S3: z = 3;

(a) Statements

(b) Original data flow graph

(c) Data flow graph after statement merging transformation

Figure 4.11: Statement scheduling transformation
Figure 4.11.(a) shows a program with three statements $S_1, S_2, S_3$. The original sequential scheduling of this program is driven by a level-order scheduling as shown in Figure 4.11.(b). When the statement scheduling transformation is applied to this program a dependency analysis reveals a flow dependency for variable $x$ in $S_1, S_2$ in levels $L_4, L_3$. Instructions from $S_2$ can be moved one level down and the flow dependency on variable $x$ is kept as long the store to memory happens before the load. Statement $S_3$ is independent from the previous statements, this condition allows $S_3$ to be pushed to the bottom of the data flow graph. Figure 4.11.(c) shows the DFG for the sample program after the statement scheduling transformation. For this example, the number of levels in the DFG has been reduced from seven to five.

From the QCM principle, the QueueCore is able to execute the maximum parallelism found in DAGs as no false dependencies occur in the instructions. This transformation merges statements to expose all the available parallelism [32] within basic blocks. With the help of the compiler, QueueCore is able to execute natural instruction level parallelism as it appears in the programs. Statement scheduling is available in the queue compiler as an optimization flag which can be enabled upon user request.

### 4.7 Assembly Generation Phase

The last phase of the queue compiler is the assembly code generation for the QueueCore processor. It is done by a one-to-one translation from QIR code to assembly code. The assembly generator is in charge of inserting `covop` instructions to expand the operand field of those instructions that have operands beyond the limits of the operand field bits. Figure 4.12 shows the assembler output for the C program fragment in Figure 4.9.(a).

The code fragment in Figure 4.12 shows the assignment of an array element in-
Figure 4.12: Assembly output for QueueCore processor from C

dexed by variable to another variable, in C language “x = a[i]”. The first instruction loads the index variable into the queue, its operand specifies the base register and the offset to obtain the memory location of the variable. The operand in the second instruction specifies the immediate value to be loaded, if the value is greater than the instruction bits the assembly phase inserts a covop instruction to extend the immediate value. The operand in the third instruction works is used to compute the effective address of the first element of the array. The next two arithmetic instructions use their operand as the offset reference and help to compute the address of the array element indexed by a variable. For this example, both are binary instructions and take their first operand implicitly from QH, and the second operand from QH+1. The lds instruction loads into the queue the value of a computed address taken the operand queue as an offset reference given by its only operand. The last instruction stores the value pointed by QH to memory using base addressing.
Chapter 5

Compiler Evaluation

5.1 Methodology

In this chapter demonstrates the efficiency of our produced order queue compiler. For a set of recursive and iterative numerical computation benchmark programs, we evaluated the characteristics of the resulting queue compiler. We measured the effectiveness of statement merging optimization for improving ILP, we analyzed the quality of the generated code in terms of the distribution of instruction types, and we demonstrate the effectiveness of the queue compiler as a design space exploration tool for our QueueCore by analyzing the maximum offset value required by the chosen numerical benchmarks.

5.2 Compiler Complexity

The resulting back-end for the QueueCore consists of about 8000 lines of C code. Table 5.1 shows the number of lines for each phases of the back-end.
Table 5.1: Lines of C code for each phase of the queue compiler’s back-end

<table>
<thead>
<tr>
<th>Phase</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-offset P-Code Generation</td>
<td>3000</td>
</tr>
<tr>
<td>Offset Calculation</td>
<td>1500</td>
</tr>
<tr>
<td>Instruction Scheduling</td>
<td>1500</td>
</tr>
<tr>
<td>Statement Merging</td>
<td>1000</td>
</tr>
<tr>
<td>Assembly Generation</td>
<td>1000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>8000</strong></td>
</tr>
</tbody>
</table>

5.3 Queue Compiler Evaluation

To evaluate the quality of the generated code of our compiler, we organized the QueueCore instructions into five categories: memory, ALU, move data, control flow, and covop. Memory instructions are to load and store to main memory including loading immediate values; ALU includes comparison instructions, type conversions, integer and floating point arithmetic-logic instructions; move data includes all data transfer between special purpose registers; control flow includes conditional and unconditional jumps, and subroutine calls; and covop includes all covop instructions to extend memory accesses and immediate values. Table 5.2 shows the distribution of the instruction categories in percentages for the compiled programs. From the table we can observe that memory operations account for about 50% of the total number of instructions, ALU instructions about 40%, move data less than 1%, control flow less about 8%, and covop about 2%. These results point a place for future improvement of our compiler infrastructure.

The developed queue compiler is a valuable tool for the QueueCore’s architecture design space exploration since it gives us the ability to automatically generate assembly code and extract characteristics of the compiled programs that affect the processor’s pa-
rameters. To emphasize the usage of the queue compiler as a design tool, we measured the maximum offset value required by the compiled benchmarks. Table 5.3 shows the maximum offset value for the given programs. These compiling results show that the eight bits reserved in the QueueCore’s instruction format [26] for the offset reference value are enough to satisfy the demands of these numerical calculation programs.

Table 5.2: Instruction category percentages for the compiled benchmarks for the QueueCore

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Memory</th>
<th>ALU</th>
<th>Move Data</th>
<th>Control Flow</th>
<th>Covop</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft8g</td>
<td>48.60</td>
<td>47.55</td>
<td>0.32</td>
<td>2.90</td>
<td>0.63</td>
</tr>
<tr>
<td>livermore</td>
<td>58.55</td>
<td>33.29</td>
<td>0.20</td>
<td>5.95</td>
<td>4.01</td>
</tr>
<tr>
<td>whetstone</td>
<td>58.73</td>
<td>26.73</td>
<td>1.11</td>
<td>13.43</td>
<td>0.00</td>
</tr>
<tr>
<td>linpack</td>
<td>48.14</td>
<td>41.59</td>
<td>0.58</td>
<td>8.16</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Table 5.3: QueueCore’s program maximum offset reference value

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Maximum Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft8g</td>
<td>29</td>
</tr>
<tr>
<td>livermore</td>
<td>154</td>
</tr>
<tr>
<td>whetstone</td>
<td>31</td>
</tr>
<tr>
<td>linpack</td>
<td>174</td>
</tr>
</tbody>
</table>

5.4 Comparison of Generated QueueCore Code with Optimized RISC Code

The graph in Figure 5.1 compares the ILP improvement of the queue compiler over the optimizing compiler for MIPS processors. For all the analyzed programs, the queue compiler exposed more natural parallelism to the QueueCore than the optimizing compiler for the RISC machine. The improvement of parallelism comes from the natural parallelism found in the level-order scheduled data flow graph with merged statements.
QueueCore’s instruction set benefits from this transformation and scheduling as no register names are present in the instruction format. The RISC code, on the other hand, is limited by the architected registers. It depends on the good judgment of the compiler to make effective use of the registers to extract as much parallelism as possible, and whenever the register pressure exceeds the limit then spill registers to memory. The loop bodies in livermore, whetstone, and linpack benchmarks consist of one or few instructions with many operands and operations. The improvement of our technique in these programs comes mainly from the level-order scheduling of these “fat” statements since the statement merging has no effect across basic blocks. The greatest improvement on these benchmarks was for the fft8g program which is dominated by manually unrolled loop bodies where the statement merging takes full advantage. In average, our queue compiler is able to extract more parallelism than the optimizing compiler for a RISC machine by a factor of 1.38.

Figure 5.2 shows the normalized code size of the compiled benchmarks for MIPS16, ARM/Thumb and QueueCore using the MIPS I as the baseline. For most of the benchmarks our design achieves denser code than the baseline and the embedded RISC processors. A closer inspection to the object file revealed that the QueueCore program has about two times more instructions than the MIPS and MIPS16 code. This is due to the effect of local optimizations such as constant folding, common subexpression elimination, dead code removal, etc. that are applied in the RISC compilers and not in the queue compiler. In average, our design achieves 31% denser code than MIPS I, 20% denser code than the embedded MIPS16, and 26% denser code than the ARM/Thumb processor.
Figure 5.1: Instruction level parallelism improvement of queue compiler over optimizing compiler for a RISC machine

Figure 5.2: Normalized code size for two embedded RISC processors and QueueCore
Chapter 6

Conclusion

This paper presented the design and development of the queue compiler framework. The presented method integrates the queue computation principle into all stages of automatic compilation of high-level programming languages to executable code.

Our evaluation results highlight the benefits of the queue computation model as a viable alternative for high-performance computing. Our design eliminates the register pressure by hiding completely the register file from the instruction set while maintaining a short instruction format with one offset reference. The queue compiler exposes maximum natural parallelism available in the data flow graph by means of a statement merging transformation. We evaluated our design by comparing the compile-time extracted parallelism against an optimizing compiler for a conventional RISC processor for a set of numerical benchmarks. We estimated the overall quality of our compiler by compiling a set of multimedia and integer applications for the QueueCore processor. Our proposed architecture achieves higher instruction level parallelism (ILP) by an average factor of 1.38. Furthermore, the design of QueueCore processor allows the generation of compact code suitable for embedded systems. In average, QueueCore binaries are about 26% smaller than embedded RISC processors. Our future work in-
cludes the addition of classical optimizations in the queue compiler to reduce load/store instructions that account for about 50% of total generated instructions.

Our current compiler generates correct programs by traversing the data dependency graph of every expression and calculating the offset references to access their operands. This expression-by-expression compilation scheme generates correct code but ignores opportunity to generate very high performance code. Common-subexpression elimination (CSE) is a classical compiler optimization that reduces execution time by removing redundant computations. The program is analyzed for occurrences of identical computations and replaces the computations for the uses of a temporary that holds the computed value.

Further optimization will be implementation of the CSE optimization in the queue compiler that builds a DAG representation of the basic blocks rather than statements. As the compilation scope is larger, the characteristics of queue programs vary and may impose new requirements on the compiler, underlying hardware, and instruction set.
References


