New Code Generation Algorithm for QueueCore - An Embedded Processor with High ILP

Arquimedes Canedo, Ben A. Abderazek, Masahiro Sowa
University of Electro-Communications
1-5-1 Chofugaoka, Chofu-Shi 182-8585
Tokyo, Japan

Abstract

Modern architectures rely on exploiting parallelism found at the instruction level to achieve high performance. Aggressive ILP compilers expose high amounts of instruction level parallelism where, in some cases, the number of architectured registers is not enough to hold the results of potential parallel instructions. This paper presents a new code generation scheme for the QueueCore, a 32-bit queue-based architecture capable of executing high amounts of ILP. QueueCore’s instructions implicitly read their operands and write results. Compiling for the QueueCore requires that all instructions have at most one explicit operand represented as an offset calculated at compile-time. Additionally, the instructions must be scheduled in level-order manner. The proposed algorithm successfully restricts all instructions to have at most one offset reference, it computes the offset values, and makes a level-order scheduling of the program. To evaluate the effectiveness of the new code generation scheme we developed a queue compiler and compiled a set of benchmark programs. Our results show that the code has more parallelism than optimized RISC code by factors ranging from 1.12 to 2.30. QueueCore’s instruction set allows us to generate code about 40%-18% denser than optimized RISC code.

1. Introduction

Instruction level parallelism (ILP) is the key to improve the performance of modern architectures. ILP allows the instructions of a sequential program to be executed in parallel on multiple data paths and functional units. Data and control independent instructions determine the groups of instructions that can be issued together while keeping the program correctness [14]. Two scheduling approaches have been followed: superscalar machines using its hardware to schedule the program at run-time, and VLIW machines relying entirely on the compiler to schedule the program. For both technologies, the compiler algorithms are critical to facilitate the schedule. Sophisticated compiler scheduling algorithms are used to expose ILP on sequential code. This techniques concentrate mainly on loops where programs spend most of their running time. However, these aggressive optimizations have the side effect of increasing the register pressure. Traditional high performance processors [8, 9, 10] have a fixed number of architectured registers that are not enough for the current compiler technology [16]. Hardware designers and compiler writers have proposed ideas to handle the problem of high ILP while keeping register requirements low [22] to avoid spilling registers to memory. A hardware/compiler technique to alleviate register pressure is to provide more registers than allowed by the instruction encoding. In [6, 21] 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 [18] 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 making instructions short. Several researchers have investigated the use of queue machines for different purposes. Bruno [17] proposed an indexed queue machine to evaluate any expression, Okamoto [15] proposed a superscalar queue machine, Schmit et al [19] proposed a pure queue machine as the processing layer for reconfigurable hardware. None of these works have proposed the infrastructure or the algorithms to automate code generation from high-level languages. In [5] we explored the possibility of using a retargetable code generator for register machines to generate code for a queue machine. The resulting compiler
mapped the operand queue in terms of general purpose registers in the machine description file. This approach led to complex algorithms to reorder instructions, longer programs, poor parallelism, and poor code quality. On our previous work [1, 2, 20], we designed a 32-bit QueueCore processor with a 16-bit instruction set format. In this paper we present the development of the compiler framework for the QueueCore processor based on a custom compiler back-end. This new framework implements a new code generation algorithm and intends to provide the infrastructure for future machine-dependent transformations and optimizations. We show that the queue compiler extracts higher parallelism and generates denser programs than a traditional RISC-based processor and two popular reduced RISC-based instruction set processors.

In Section 2 the queue computation model is presented together with the challenges of generating code for the QueueCore, a hardware implementation of the 1-offset producer order queue computation model. The proposed code generation algorithm is described in Section 3. Section 4 describes briefly the structure of the developed queue compiler, and presents the evaluation results of our algorithm. Section 4 concludes.

2. Queue Computation Model

The Queue Computation Model (QCM) is the formal definition of a computer that uses a first-in first-out data structure to store temporary computations. A queue computer is similar to a stack computer [12] in the sense that operations implicitly access the operand stack. In the QCM, elements are inserted, or enqueued, through a write pointer named QT that references the rear of the queue. And elements are removed, or dequeued, through a read pointer named QH that references the head of the queue. All operations accessing the operand queue fall into one of three categories: read-write, write-only, and read-only. Read-write operations take some data from the queue, perform some computation, and write the result back to the queue. For example, add, implicitly takes two data from the queue, performs its addition, and writes the result back to the queue. Write-only operations just generate data in the operand queue, like ld operation, that loads a memory location into the queue. And read-only operations, that only take data from the operand queue like st operation, that takes a datum from the operand queue and writes it into a memory location. Depending on the instruction characteristics, the hardware of a queue computer manipulates the read and write pointers QH, and QT. Every time a datum is dequeued the QH pointer is updated to point to the next location QH+1, and every time a datum is enqueued the QT pointer is updated to point to the following location QT+1. Given these characteristics, no information about the location of the operands needs to be encoded in the instruction format allowing the instructions for a queue machine to be short.

2.1. Evaluating Expressions in the QCM

A correct evaluation of any expression using a queue machine can be derived directly from its parse tree by a level-order traversal [17]. A level-order traversal visits all the nodes of a tree from left to right starting from the deepest to the shallowest (root node) level as shown in Figure 1(a). Notice in Figure 1(b) that all the instructions from a same level are independent from each other and can be processed in parallel. For this reason, the QCM can potentially execute all the instructions in a level in parallel.

Figure 1. Level-order traversal of expression’s \( x = \frac{a + \frac{b}{c}}{b - c} \) parse tree and resulting queue program.

In the parse tree shown in Figure 1(a) the addition and subtraction nodes have \( b \) as one of their operands. One node exists in the tree for each use of \( b \). In order to minimize the number of redundant instructions it is desirable to evaluate expressions from their directed acyclic graphs (DAG) instead of parse trees. Figure 2(a) shows the DAG for the same expression where the node for operand \( b \) has been reduced from two to one, and it is shared by the two operations in level \( L_2 \). The queue program from a DAG is also obtained from a level-order traversal of the expression’s DAG [17]. However, by using the simple rules of enqueueing and dequeueing, the evaluation of a DAG may lead to incorrect results. To understand the problem, consider the queue program in Figure 2(b). First, operands \( a, b, c \) are loaded by the instructions in \( L_3 \). Figure 2(c) shows the queue contents before and after the add instruction is executed. The relative positions of QH and QT are shown, and the bold lines represent the dequeueing of operands \( a, b \), and the enqueueing of the result \( a + b \) back into the queue. So far, the partial evaluation of the expression is correct. The problem arises when the sub instruction dequeues its two operands. Given
the current contents of the queue and the \( Q_H \) position as shown in Figure 2(d), the \( \text{sub} \) instruction takes operands \( c, a + b \) instead of \( b, c \), leading to incorrect results. Clearly, this problem comes from the property of DAGs to have multiple uses of a single node and the insufficiency of simple enqueueing/dequeueing rules to express multiple uses of a variable.

![Diagram](image)

**Figure 2.** Level-order traversal of expression’s \( x = \frac{a + b}{b - c} \) DAG and incorrect evaluation from the basic rules of enqueueing and dequeueing.

### 2.2. Producer Order Queue Computation Model

In our previous work [1, 2, 20], we have proposed the **producer order QCM** to allow the evaluation of DAGs in a queue machine. The producer order QCM gives flexibility to the dequeueing rule allowing operands to be removed from the queue not only from \( Q_H \) but from a location relative to \( Q_H \) indicated by an offset reference in the instruction. For example, the binary instruction “\( \text{sub} -3, 0 \)” has two offset references: \(-3\) indicates that its first operand has to be dequeued three queue words before the current position of \( Q_H \), and \( 0 \) indicates that the second operand is dequeued from \( Q_H \) itself. A negative offset reference represents a use of an already used operand. Figure 3 shows how the producer order code allows correct evaluation of any expression DAG. The producer order program, or P-Code, shown in Figure 3(a) has arithmetic instructions with two offset references. Figure 3(b) shows the place with respect of \( Q_H \) from where the operands of “\( \text{sub} -1, 0 \)” instruction are dequeued. The gray operands on the left of \( Q_H \) are already used operands (by the previous “\( \text{add} 0, 1 \)” instruction). The subtraction instruction takes its first operand, \( b \), with an offset reference of \(-1\), and its second operand, \( c \), from \( Q_H \). P-Code correctly evaluates any expression from its DAG.

![Diagrams](image)

**Figure 3.** Dequeueing flexible semantics of producer order QCM code.

### 2.3. QueueCore: 1-offset Producer Order Architecture

Prior investigation [4] shown that programs rarely make effective use of two offsets in the producer order instructions. For most of cases, binary instructions in programs dequeue their both operands directly from \( Q_H \) or using only one offset reference for one of the operands and the other directly from \( Q_H \). The case when two operands are dequeued using an offset reference is very rare. In order to reduce the producer order instruction bits and satisfy the requirements of applications, we proposed a producer order instruction set that encodes at most one offset reference [20], we call this model 1-offset producer order queue computation model. To preserve correctness with a one offset reference instruction set, QueueCore provides a “\( \text{dup} N \)” that reads an operand in the operand queue addressed by the offset reference \( N \) and generates a copy in \( Q_T \). The \( \text{dup} \) instruction reads an operand from the queue but does not dequeue it making it a write-only instruction that affects \( Q_T \) pointer exclusively.

QueueCore is a 1-offset producer order QCM 32-bit machine with 16-bit instruction set. 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 VH from which the second source operand is dequeued, VH−N. Unary operations have the freedom to dequeue their only source operand from VH−N. Memory operations use the operand field to represent the offset and base register, or immediate value. QueueCore defines a set of specific purpose registers available to the programmer to be used as the frame pointer register (\$fp), stack pointer register (\$sp), and return address register (\$ra). Frame pointer register serves as base register to access local variables, incoming parameters, and saved registers. Stack pointer register is used as the base address for outgoing parameters to other functions. The QueueCore architecture is capable of executing instructions in parallel. A special unit, called the Queue Computation Unit [1], decides serially the correct source operands and the destination for every instruction. Once the sources and destination have been decided, the instructions are executed safely in parallel.

3. Code Generation Algorithm for QueueCore

When a binary instruction requires two offsets for its operands and a one offset producer order instruction set is available, selectively inserting \texttt{dup} instructions to create a copy of one of the operands to be available at VH ensures correctness. The code generator for the QueueCore must accomplish three tasks that make it different from traditional code generation schemes [3, 14]: (1) guarantee correctness of the program by constraining all instructions to have at most one offset reference, (2) compute the offset reference values for all 1-offset P-code instructions, (3) schedule the program by making a level-order traversal of the data flow graph. Each one of the previous points opens new problems. How to determine the case when a binary operation requires two offset references; once detected, where in the DAG is the correct place to insert the \texttt{dup} instructions; how to compute the offset reference values directly from a DAG; how to reorder instructions in a level-order schedule to best exploit the characteristics of the QueueCore. In this Section we present a new code generation algorithm for the 1-offset producer order QCM instruction set implemented in the QueueCore architecture [1] that gives a solution for all above mentioned problems. The proposed algorithm consists of three parts: limiting instructions to one offset by means of \texttt{dup} instruction insertion, offset reference calculation, and level-order scheduling. The input of the algorithm is a parse tree that is converted to a leveled DAG (LDAG). A LDAG [7] is a data structure that maps all nodes in a DAG to levels identified by integers such that a node \( v \) is level \(- j \) if \( \text{lev}(v) = j \). Correct manipulation of this data structure is the key of the code generation as it fully expresses the data dependences between operations and operands, levels, offset references, and VH relative position. The output of the presented algorithm is a one-operand low level intermediate representation.

3.1. 1-offset P-Code Generation by \texttt{dup} Instruction Insertion

The purpose of inserting \texttt{dup} instructions is to create copies of operands in order to guarantee that all binary instructions have at most one offset reference. Let the expression \( x = \frac{-a}{(a+a)} \) be evaluated using QueueCore's one offset instruction set, its DAG is shown in Figure 4.(a). Notice that the level \( L_3 \) produces only one operand, \( a \), that is consumed by the following instruction, neg. The 1-offset producer order QCM restricts add instruction to take its first source operand directly from VH, and gives freedom for its second operand to be taken with an offset reference VH−N. For this case, the \texttt{dup} instruction is inserted to make a copy of \( a \) available as the first source operand of instruction add as shown with the dashed line in Figure 4.(b). Notice that level \( L_3 \) in Figure 4.(b) produces two data instead of one. The instruction sequence using QueueCore's one offset instruction set is shown in Figure 4.(c). By these means, helped by the code generator, QueueCore provides a mechanism to execute any program in a constrained 1-offset producer order QCM.

![Figure 4. QueueCore one offset instruction sequence generation for expression \( x = \frac{-a}{a+a} \) including \texttt{dup} instruction.](image)

The first task the code generator does is building LDAGs from the input parse trees. The insertion of \texttt{dup} instructions is performed during this stage. We impose the following characteristics in the LDAGs:

\textbf{Definition 3.1.} A level is an ordered list of elements with at least one element.

\textbf{Definition 3.2.} An \( \alpha \)-node is the first element of a level.

\textbf{Definition 3.3.} The root node of the LDAG is the only node in Level-0.
**Definition 3.4.** The sink of an edge must be always in a deeper or same level than its source.

The construction of an LDAG is done by a recursive post-order depth-first traversal of the parse tree together with a lookup table that saves information about every node and its assigned level. The levelizing of parse trees to LDAGs works as follows. For every visited node in the parse tree a level is assigned and the lookup table is searched. If the node is not found in the table then it is its first appearance and a new entry is recorded in the table and the node is created in the LDAG. If the node is found in the table then a decision must be taken in order to satisfy the property in Definition 3.4. If the level of the new node is greater than the one in the lookup table, then the node already in the LDAG is substituted by a dup instruction and the new node is created in the corresponding level updating also the lookup table with the new level. A dependence edge with source at the dup node and sink in the new node is created to satisfy the semantics of dup instruction. The algorithm to build LDAG and insert dup instructions is listed in Algorithm 1.

### 3.2. Offset Reference Calculation

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 distance, 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 3.5.** The QH position with respect of the α-node of Level-j is always at the α-node of the next level, Level-(j+1).

**Definition 3.6.** 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 α-node.

**Definition 3.7.** 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 3.8.** A hard edge is a dependence 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 \( P_i \) in a traversal from \( p_{n-1} \) to \( p_n \) (α-node) meets one of two conditions. The first 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 neighbor node as a result of a level-order traversal. Notice that from a level-order traversal, \( q_m \)’s following node can be \( q_{m+1} \), or the α-node of lev(\( q_m \)) + 1 if \( q_m \) is the last node in lev(\( q_m \)). The second condition is that the node is the α-node, \( P_i = p_0 \). From Definition 3.5, QH position is at α-node of the next level lev(\( p \)) + 1. The dotted lines in Figure 5 show the QH relative position with respect of every binary and unary operations. The QH position with respect of add, mul, and div operations is given by the second condition as they are α-nodes of their respective levels. The QH position for the two neg and sub operations is found by the above explained rules of the first condition. The proposed algorithm is listed in Algorithm 2.

![Figure 5. QH relative position for all binary and unary operations in a LDAG](image)

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 \( P_i \) as described in Algorithm 3. 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.

### 3.3. Level-Order Scheduling

Traditionally, the main goal of code scheduling is to minimize the execution time by rearranging instructions to make an effective use of the underlying hardware resources [14]. The non-random access characteristics of the QCM redefine the main goal of code scheduling for a queue processor. The code scheduler for the QCM should maximize the number of instructions that can be executed in parallel. We propose a level-order scheduling algorithm...
The dependence analysis between $S_1$ and $S_2$ reveals a true data dependence for operand $x$, and an antidependence for operand $a$. The first merging point candidate, for the true dependence, is in $S_1$'s level-0 and $S_2$'s level-3. And the second merging point candidate is in $S_1$'s level-1 and $S_2$'s level-0. The merging point is chosen to be the one that preserves all dependences in the list. For the given example, the second candidate is chosen, and the result is shown in Figure 6. For the case when no dependences between two expressions are found, the merging point can be any level and the deepest level is chosen for maximizing the number of nodes in the same level. The importance of merging statements is to reduce the number of levels, increasing the number of operations in the same level as a consequence.

![Figure 6. Statement merging using statements as transformation units for reducing number of levels.](image-url)
level intermediate representation. One-operand intermediate representation is suitable for the producer order QCM.

4. Results

4.1. Queue Compiler Construction

To evaluate the effectiveness and correctness of our method, we developed a compiler back-end that implements the proposed code generation algorithm for the QueueCore processor. Figure 7 shows the block diagram of our queue compiler. The front-end is based on GCC 4.0.2 and converts C language programs into abstract syntax trees in three-address form called GIMPLE representation [13]. Each GIMPLE quadruple is limited to have at most two operands and one operator. GIMPLE trees are reconstructed into QTrees. QTrees are GIMPLE trees without limitation in the number of operands and operations. This characteristic makes them suitable for expressing entire expressions and simplify the following stages. The first phase of our back-end is the 1-offset P-Code generator which takes QTrees and builds LDAGs inserting dup nodes when necessary. The second phase computes the offset references for every binary and unary operations in the LDAGs. The third phase schedules in level-order the LDAGs into QIR, a one-operand low level intermediate representation. As statements are treated as units, the fourth phase safely reorders the statements to reduce the number of levels in a basic block. The fifth phase, generates assembly code for the QueueCore processor from the QIR representation.

Figure 7. Queue compiler block diagram

4.2. Exposed ILP and Code Size Evaluation

In this paper, to measure the quality of the code generated by the proposed code generation scheme, we concentrate on two aspects: instruction level parallelism (ILP), and code size. We chose a set of benchmark programs: quicksort, N-queens, SHA encryption algorithm, whetstone benchmark, livermore loops, and radix-8 fast fourier transform. Figure 8(a) shows the improvement factor of our compiler in extracted ILP compared to two RISC processors. For the RISC machines we chose MIPS I [9] as the baseline code and MIPS16 [11]. The MIPS16, in addition to its 32-bit ISA it includes a 16-bit ISA with the goal of reducing code size. GCC 4.0.2 compiler with full optimizations (-O3) was used for both RISC machines. The queue compiler at this stage does not have any optimization, therefore, it generates unoptimized code. For all the analyzed programs, the queue compiler exposed more parallelism ranging from 1.12 to 2.30 times when compared to the MIPS I code. The improvement of parallelism comes from the natural parallelism found by the level-order scheduling. In average, the queue compiler extracts 1.78 times more parallelism than an optimizing compiler for a RISC machine.

Figure 8(b) shows the normalized code size using the MIPS I as the baseline. For all benchmarks, our compiler generates denser code than the full 32-bit RISC ISA, and the embedded RISC processor. In average, the code for the QueueCore is 40% denser than MIPS I, and 18% denser than the MIPS16 architecture.

5. Conclusion

In this paper a new code generation scheme for the 1-offset producer order QCM has been presented. This computation model imposes new restrictions to the generated code which are successfully addressed by our code generation method. Our algorithm models the relationships
between operations, operands, and the queue computation model. Based on this model the algorithm restricts all operations to have at most one offset reference, it computes the offset values, and schedules the program in level-order manner. To demonstrate the effectiveness of our method, we implemented the algorithm in a queue compiler. For a set of benchmark programs, the generated non-optimized code exposes more parallelism than the optimized code for a RISC processor from 1.12 to 2.30 times. We also shown that the QueueCore’s code is in average 40% denser than a MIPS I, and 18% denser than the embedded MIPS16 code. Our future work will extend the queue compiler infrastructure to include ILP optimizations.

References


