Queue Register File Optimization Algorithm for QueueCore Processor

Arquimedes Canedo, Ben Abderazek, Masahiro Sowa
Graduate School of Information Systems
University of Electro-Communications
Chofugaoka 1-5-1, Chofu-Shi, Tokyo, Japan, 182-8585
{canedo,ben,sowa}@sowa.is.uec.ac.jp

Abstract

The queue computation model offers an attractive alternative for high-performance embedded computing given its characteristics of short instructions and high instruction level parallelism. A queue-based processor uses a FIFO queue to read and write operands through hardware pointers located at the head and tail of the queue. Queue length is the number of elements stored between the head and the tail pointers during computations. We have found that 95% of the statements in integer applications require a queue length of less than 32 words. The remaining 5% requires larger queue length sizes up to 230 queue words. In this paper we propose a compiler technique to optimize the queue utilization for the hungry statements that require a large amount of queue. We show that for SPEC CINT95 benchmarks, our technique optimizes the queue length without decreasing parallelism. However, our optimization has a penalty of a slight increase in code size.

1. Introduction

Queue-based computers are a novel alternative for embedded architectures. Several queue computation models have been proposed in the literature [1, 2, 24, 22, 23, 18, 27, 20]. The features that make these architectures suitable for embedded hardware are their compact instruction set that improves code density, high instruction level parallelism (ILP) capabilities that delivers good performance, and simple hardware that reduces the chip area and power consumption.

Queue Computation Model (QCM) refers to the evaluation of expressions using a first-in first-out queue. Elements are read, or dequeued, at the head of the queue. And elements are written, or enqueued, at the tail of the queue. Two hardware pointers are required to keep track of the head and tail positions, these pointers are referred as $\text{QH}$ for the head, and $\text{QT}$ for the tail. At any point of execution of a program, the queue length is the number of elements between $\text{QH}$ and $\text{QT}$. Every statement in a program may have different queue length requirements. Queue length requirements is tightly related to the way queue programs are generated. To evaluate any expression, the directed acyclic graph (DAG) of the expression should be scheduled in level-order manner [20]. A level-order scheduling visits all nodes in a DAG from the deepest level towards the root as shown in Figure 1(a). All nodes belonging to the same level $L_i$ are data independent and can be executed in parallel. The queue length at any point of execution of a program can be measured by the number of elements in level $L_i$. Figure 1(c) shows the position of $\text{QH}$ and $\text{QT}$ for each level in the expression. The queue length requirements are shown in the Figure. For $L_3$ the queue length requirements is three, for $L_2$ is two, for $L_1$ is one, and for $L_0$ is zero.

In our previous work [5] we have developed the compiler infrastructure for the QueueCore processor [1]. The compiler translates C programs into QueueCore assembly. An important task of the queue compiler is to schedule the program in level-order manner and correctly compute the offset references of instructions. Arithmetic instructions in Figure 1(b) have two offset references as operands. Each offset indicates the location with respect of $\text{QH}$ from where the operation should read the corresponding operand. However, instructions always write the result at $\text{QT}$. For example, the fourth instruction “add 0, 1” reads its first operand from $\text{QH}+0$ (directly from the head) and the second operand from the location right after the head, $\text{QH}+1$. After the addition is computed the result is written to $\text{QT}$. At this stage the queue contains only $\{c, (a+b)\}$. The next instruction, subtraction, requires its first operand, $b$, to be read from one queue word in the left of head, $\text{QH}-1$. Its second operand, $c$, is taken directly from the head $\text{QH}+0$. The rest of the program is executed in similar fashion as shown in Figure 1(c).

Using our queue compiler infrastructure we compiled
In this paper we propose an algorithm to optimize the queue length for the QueueCore processor. The presented algorithm breaks the conflicting large statements into clusters suitable to be executed by the QueueCore. A cluster is a set of operations and operands that preserve the semantics of the queue computation model and their queue length demands fit into the queue register file available in the underlying hardware. The clusterization of large statements incurs into the addition of spill code to communicate intermediate computations. We implemented the algorithm in the queue compiler and we analyze its effect in code size increase over the baseline code size for SPEC CINT95 benchmarks. Experiments show that the algorithm effectively deals with the 5% of statements with a increase of less than 2% in code size.

2. Queue Compiler Infrastructure

An important role of the compiler for the QueueCore processor is to produce a level-order scheduling of the DAGs in the programs and correctly calculate the offset reference values for every instruction. Figure 3 shows the block diagram of the queue compiler including the proposed clusterization pass. The front-end of the compiler is based on GCC 4.0.2 and it parses C files into a abstract syntax trees (AST), a high level intermediate representation called GIMPLE [16]. GIMPLE representation is a three-address code suitable for code generation for register machines but not for queue machines. As the level-order scheduling traverses the full DAG of an expression, we reconstruct GIMPLE trees into trees of arbitrary depth and width called QTrees. During the expansion of GIMPLE into QTrees we also translate the high-level constructs such as aggregate types into their low level representation in QueueCore instructions. After having fully expanded DAGs as QTrees, we build the core data structure, the leveled DAGs (LDAG) [9]. LDAGs allow the code generation algorithm of the queue compiler to determine the offset references for every instruction. The offset calculation phase consists of determining, for every instruction, the current location of the QH and measuring the distance to the instruction’s operands. The measured distance is the offset reference value and it is annotated in the LDAG. Then the LDAGs are level-order scheduled and a linear low level
The intermediate representation (QIR) is emitted. The last phase of the compiler is to generate assembly code from the QIR representation.

Under the queue computation model, the queue utilization requirements are given by the number of nodes in every computation level. In the following section we propose an algorithm that partitions the expressions into clusters to reduce queue length in such a way that the rules of queue computing are preserved. We chose LDAGs as the input of the algorithm as all dependency edges between operations and operands are determined.

Figure 3. Queue Compiler’s block diagram including the phases and intermediate representations.

3. Queue Length Optimization

Queue length refers to the number of elements stored between $Q_H$ and $Q_T$ at some computation point. We have introduced how queue length can be measured from the number of elements in an independent queue computation level. Nevertheless, DAGs often present a case when this assumption is not enough to estimate the queue length requirements of an expression. Consider the DAG shown in Figure 4 for expression “$x = (-a * b) / c$”. Notice that the edge with source at node “$a$” and sink at node “$/c$” spans three levels. Edges that span more than one level (soft edges) increase the queue length requirements as the sink node must be kept in the queue until the time when the source node is executed. For the given example, the queue length requirement of the DAG is four queue words. The algorithm must deal with two different conditions that directly affect the queue length requirements of an expression. One is the length of computation levels, and the second is the length of soft edges. The former can be solved by splitting the level into blocks of manageable size. The later can be solved by re-scheduling the child’s subtree. The order on which these actions are performed affects the quality of the output DAGs and therefore the quality of the generated code.

If the levels are splitted first and then subtrees re-scheduled, the second action affects the length of the levels in the final DAG. The first transformation should be performed one more time to guarantee that all levels comply with the target queue length. If the order of the actions is inverted, then the DAG will be expanded into a tree and all subexpressions have to be recomputed as all subtrees are completely expanded affecting the performance and code size. We propose an algorithm that deals with the above problems in a single step. Our integrated solution reduces the subexpression re-scheduling and minimizes the insertion of spill code.

Figure 4. Queue length is determined by the width of levels and length of soft edges.

3.1. Clusterization Algorithm

The main task of the clusterization algorithm is to reduce a DAG’s queue length requirements by splitting it into clusters of realizable size. The algorithm must partition the DAG in such a way that every cluster is semantically correct in terms of the queue computing model. Partitioning involves the addition of extra code to communicate the intermediate values computed in different clusters. Our algorithm uses memory to communicate intermediate values
between clusters. The input of the algorithm is a LDAG data structure. For the queue compiler, a cluster is defined as a LDAG with spill code that communicates intermediate values to other clusters through memory. Keeping clusters as LDAGs allows the implementation to use the same infrastructure and the later phases of the queue compiler remain without any modification.

The algorithm is divided into two phases: labeling, and spill insertion. The labeling phase is in charge of grouping subtrees of the DAG into clusters in order to preserve the rules of queue computing. For any given DAG or subtree \( W \) rooted at node \( R \), the width of \( W \) is verified to be smaller than the threshold. If the condition is true then all nodes in \( W \) are labeled with a unique identifier called the cluster ID. In case the width of the DAG exceeds the threshold then the DAG must be recursively partitioned starting from the left child and then the right child of \( R \). The output of the labeling phase is a labeled DAG as shown in Figure 5. Two clusters are shown in the Figure, the first cluster has its root node at \((-)\) node, and the second at \((+)\) node. For simplicity, the threshold selected for the example in the figure is two. Therefore, the labeling algorithm compares the width of the DAG against two. To measure the width of a subtree \( W \), the DAG is traversed as a tree and the level with more elements is considered the width of \( W \). Notice that in Figure 5, the width of the subtree rooted at \((/)\) is four as the right child \((+)\) is traversed as a tree and its children (nodes \( c, + \)) visited twice. Therefore, the level \( L_3 \) has four elements. The labeling algorithm is listed in Algorithm 1.

**Algorithm 1** labelize (LDAG \( W \))

**Require:** global \( \text{Threshold} \)

**Require:** global \( \text{cluster\_id} = 0 \)

1. \( \text{root} \leftarrow W\text{'s root node} \)
2. \( \text{if SubTree\_Width (root) > Threshold} \) then
3. \( \quad \text{lhs} \leftarrow \text{labelize (root.lhs)} \)
4. \( \text{if isBinary (root)} \) then
5. \( \quad \text{if SubTree\_Width (root.rhs) > Threshold} \) then
6. \( \quad \quad \text{rhs} \leftarrow \text{labelize (root.rhs)} \)
7. \( \quad \quad \text{root} \leftarrow \text{Assign\_ID\_to\_node (rhs.id)} \)
8. \( \quad \quad \text{return root} \)
9. \( \quad \text{else} \)
10. \( \quad \quad \text{root.rhs} \leftarrow \text{Assign\_ID\_to\_subtree (cluster\_id++)} \)
11. \( \quad \quad \text{root} \leftarrow \text{Assign\_ID\_to\_node (root.rhs.id)} \)
12. \( \quad \quad \text{return root} \)
13. \( \quad \text{end if} \)
14. \( \text{end if} \)
15. \( \text{else} \)
16. \( \quad \text{root} \leftarrow \text{Assign\_ID\_to\_subtree (cluster\_id++)} \)
17. \( \quad \text{return root} \)
18. \( \text{end if} \)

Spill insertion phase is the second and last phase of the algorithm. The annotated LDAG from the previous phase is processed and a list of \( N \) number of clusters (a cluster set) is generated as the output. The input annotated DAG is traversed in post-order manner. For every visited node in the traversal, a set of actions are performed: (1) assign the node to the corresponding cluster, (2) insert reload operations to retrieve temporaries computed in a different cluster, (3) insert operations to spill temporaries used by different clusters.

Assigning nodes to the corresponding cluster involves the creation of the LDAG data structures, node information, and data dependency edges. Using the queue compiler’s LDAG infrastructure [5] allows the clusterization algorithm to be implemented in a clean and simple manner. The addition of a list of length \( N \) is required in the compiler to generate the clusters. The value of \( N \) is the number of clusters discovered by the labeling phase.

Spill code is inserted in two situations, to deal with intermediate results used by different clusters, and to solve the problem of soft edges that span more than one level and violate the queue length given by the threshold. Only subexpressions are spilled to memory and reloaded. Variables and constants that are used by multiple nodes are only reloaded since spill/reload would require an extra instruction and extra memory space for temporaries. For every node, the algorithm detects which operation \( u \) needs operands \( v \) to be reloaded whenever the cluster identifier of the node and the operand are different, \( \text{ID}(u) \neq \text{ID}(v) \). After reloading detection, the node \( u \) is analyzed for spilling as follows. If the analyzed node \( u \) is a subexpression and has more than one parent node then a spill operation is inserted. Figure 6 shows the generated clusters for the example in

![Figure 5. Output of the labeling phase of the clusterization algorithm](image-url)
Figure 5. Two clusters are generated after the spill code is inserted. The gray nodes in the first cluster represent the nodes that are spilled to memory. Notice that the node (c) has two parents but no temporary is generated as it is not a subexpression. For the second cluster, three reload operations represented by the rectangles are required, two for the temporaries computed in the first cluster, and one for the variable.

Algorithm 2 clusterize (node u, LDAG W)

Require: An empty cluster set C of N elements
1: /* Traverse as a DAG */
2: if AlreadyVisited (u) then
3:    return NIL
4: end if
5: /* Action 1: add to corresponding cluster */
6: ClusterSet_Add (C, ID(u), u)
7: /* Action 2: generate reloads */
8: for all children v of u do
9:    if ID(v) ≠ ID(u) then
10:       GenReload (C, ID(u), v)
11:   else if isViolatingSoftEdge (u, v) then
12:      GenReload (C, ID(u), v)
13:   else
14:    /* Post-Order traversal */
15:       clusterize (v, W)
16: end if
17: end for
18: /* Action 3: generate spills */
19: if Parents (u) > 1 AND isSubexpression (u) then
20:   GenSpill (C, ID(u), u)
21: end if
22: /* Mark visited and return */
23: MarkVisited (u)
24: return u

The target architecture of the queue compiler is QueueCore [1], an embedded 32-bit queue-based ILP processor. The instructions of QueueCore are 16-bit wide. QueueCore is capable of executing instructions in parallel. A special unit, called the Queue Computation Unit [1] bookkeeps QH and QT references. This mechanism decides the correct source operands and the destination for every instruction. The correct location of operands is found by adding the offset reference of the instructions to the current location of QH reference. For all experiments the queue compiler was configured only with the presented queue-length optimization tuned for a queue length of 32 to match the size of the QueueCore’s queue register file size. Our ongoing work is concentrated on classical optimizations [17, 3] and none of these are available in the queue compiler infrastructure.

Figure 6. Output of the clusterization algorithm. Spill nodes marked in gray circles and reload operations in rectangles.

Algorithm 2 lists the actions performed over the annotated LDAG to generate a set of clusters. As clusters have the same shape as LDAGs, we can use the queue compiler infrastructure to generate code directly from the cluster set. Each cluster is treated as a LDAG and the code generator [5] calculates the offset references for all instructions, including spill code. Besides from the described clusterization algorithm, the queue compiler internals remained untouched and the compilation flow remains the same as the original compiler.

4. Results

4.1. Methodology

We implemented the queue register file optimization algorithm in the queue compiler infrastructure. We evaluated the effectiveness of our technique by analyzing the output code of the queue compiler and measuring the code size and extracted ILP for the SPEC CINT95 benchmark.
4.2. Effects in Code Size

Our algorithm effectively deals with the statements in a program that demand a large number of queue words. Table 4.2 shows the number of extra spill instructions generated by the queue-length optimization for the QueueCore processor. The second column shows the total number of instructions generated including the spill code. From these results we observe that the optimization presented in this paper increases the length of the programs by about 1.46%. The program which presents less extra spill code is 147.vortex with the addition of 0.66% more instructions. And the program with more spill code is 130.li with 2.23% more instructions. The third column shows the number of instructions generated for a conventional 8-way issue machine using global ILP scheduling techniques [29]. The queue compiler generates about 22% larger programs, in terms of number of instructions, than the optimizing compiler [6,8] for the 8-way universal machine in [29]. There is a room for improvement in our queue code since our compiler does not perform any classical optimizations that remove redundant computations [7, 15, 26]. The compact instruction set of the QueueCore allows programs to present high code density. Figure 7 compares the code size of the text segment of the generated programs for QueueCore and the 8-way issue machine. We assume two byte instructions for the QueueCore and four byte instructions for a typical register-based multiple-issue machine [10]. The results are normalized to one using the QueueCore program without queue-length optimization as the baseline. Our optimization increases the code size of QueueCore programs less than 2%. Compared to the conventional register machine code, ours is from 27% to 47% denser.

4.3. Effects in Instruction Level Parallelism

Queue computing relies on the level-order scheduling for the generation of correct programs. The level-order scheduling naturally exposes all available parallelism in an expression. First, we analyze the effects of the presented optimization on ILP. We measured the compile-time extracted parallelism of the queue compiler without and with the queue length optimization presented in this paper. We also compared our code against the optimized code for the 8-way universal machine. In [29], the compiled code is classically optimized and scheduled using a global scheduling technique aided by tail duplication. Figure 8 shows the parallelism exposed by the compiler for the 8-way universal machine, for the QueueCore without queue-length optimization, and for the QueueCore with queue-length optimization enabled. From the graph we observed that the impact of the queue length optimization for QueueCore increases ILP for all programs about 1.3%. The raise represents artificial ILP given by the insertion of extra spill code. Compared to the ILP compiler for a 8-way universal machine, the level-order scheduling of the non-optimizing queue compiler can extract about 13.67% more parallelism.

5. Related Work

The register file is tightly related to the overall performance of any computer architecture. High-performance applications demand a large number of registers [19, 15, 11]. Accessing a large number of registers costs time and power [4]. A number of hardware and compiler techniques have been deployed to optimize the register file as much as possible [14, 28, 13, 12, 21] to deliver high-performance at low power.

Several queue-based computers have been proposed. In [20] the principle of executing directed acyclic graphs
in a queue machine is explored. The mechanisms to allow superscalar execution on a queue machine were proposed in [18]. Other researches have used a queue register file as an additional unit in a register-based machine to execute loops more efficiently [27, 25]. In [24, 2, 1], a realizable hardware for a queue machine is proposed. None of the above mentioned works have proposed hardware or software mechanisms to optimize the queue register file. Given the suitable characteristics of the QueueCore for an embedded processor, we propose the exploration of mechanisms to keep the queue register file optimized for the demands of high performance applications.

6. Conclusion

In this paper we presented an algorithm implemented in the queue compiler infrastructure to optimize the queue register file for the QueueCore processor. Statements in programs that violate the length of the queue register file are broken into clusters that are semantically correct for the queue computing model. The clustering algorithm effectively deals with the two conditions that make programs consume a large number of queue words, the width of the statement, and the edges spanning across several levels. The insertion of spill code to communicate intermediate values is necessary but we demonstrated that for the SPEC CINT95 applications our queue compiler optimizes the queue register file with an increase of about 1.36% in the code size. Additionally, we have shown the code density and high ILP characteristics of the queue computing and the QueueCore processor against a conventional multiple-issue register machine. In average, our non-optimized code has 13.67% more parallelism than the code optimized for a 8-issue conventional processor.

References


