Efficient Code Generation Algorithm for Natural Instruction Level Parallelism-aware Queue Architecture

Masashi Masuda and Abderazek Ben Abdallah  
The University of Aizu, Adaptive System Laboratory  
Graduate School of Computer Science and Engineering  
Aizu-Wakamatsu 965-8580, Fukushima, Japan

Arquimedes Canedo  
IBM, Tokyo Research Laboratory  
1623-14 Shimotsuruma, Yamato-Shi  
Kanagawa-Ken 242-8502, Japan

Abstract: Queue based instruction set architecture processor offers an attractive option in the design of embedded systems. In our previous work, we proposed a novel queue processor architecture as a starting point for hardware/software design space exploration for embedded applications. In this paper, we present a high performance 32-bit QueueCore architecture. This work presents novel code generation algorithm for the QueueCore architecture. Compiling for the QueueCore requires a new approach since the concept of registers disappears. The compiler extracts more parallelism than the optimizing compiler for a RISC machine over a set of various numerical benchmark programs. In addition, we are able to generate in average about 23% denser code than two embedded RISC processors.

Keywords: QueueCore; Code generation; Produced Order; Parallel;

1 INTRODUCTION

Queue based instruction set architecture processor offers an attractive option in the design of embedded systems. In our previous work, we proposed a novel queue processor architecture as a starting point for hardware/software design space exploration for embedded applications [1, 10]. ILP is the key to improve the performance of modern architectures. It 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 [8]. A good scheduling is crucial to achieve high performance. An effective scheduling for the exploitation of ILP depends greatly on two factors: the processor features, and the compiler techniques. In superscalar processors, the compiler exposes ILP by re-arranging instructions. However, the final schedule is decided at run-time by the hardware [6]. In VLIW machines, the scheduling is decided at compile-time by aggressive static scheduling techniques [3, 8].

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. Also, false dependencies disappear from programs eliminating the need for register renaming logic that reduces circuitry and improves power consumption [7]. We earlier designed a 32-bit QueueCore processor with a 16-bit instruction set format. Our approach is to allow variables to be produced only once but can be consumed multiple times. We sacrifice some bits in the instruction set for an offset reference to indicate the relative location of a variable to be reused. The goal is to allow DAGs to be executed without transformations that increase the instruction count while keeping reduced instructions that generate dense programs [1]. Ideas about compiling for queue machines have been discussed in the previous work in an abstract way. Some problems have been clearly identified but no algorithms have been proposed. Before, we explored the possibility of using a retargetable code generator for register machines to map register code into the queue computation model [4]. The resulting compiler mapped the operand queue in terms of a large number general purpose registers in the machine description file that is used by the code generator in order to avoid spill code. This approach led to complex algorithms to map register programs into queue programs, excessively long programs, poor parallelism, and poor code quality. In this paper we present a new code generation scheme implemented in a compiler for the QueueCore processor. Our compiler generates assembly code from C programs. This compiler is, for the best of our knowledge, the first automated code generation tool designed for the queue computation model. 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.

2 Compiler Framework

There are three tasks the queue compiler must do that make it different from traditional compilers for register machines: (1) constrain all instructions to have at most one offset reference, (2) compute offset reference values, and (3) schedule the program expressions in level-order manner. We developed a C compiler for the QueueCore that uses GCC’s 4.0.2 front-end and middle-end. The C program is transformed into abstract syntax tree (AST) by the front-end. Then the middle-end converts the ASTs into a language and machine independent format called GIMPLE [9]. Fig.1 shows the phases and intermediate
represents the queue compiler infrastructure.

2.1 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 the 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 [5]. We chose LDAGs as data structure to model the data dependencies between instructions and QSTATEs. Fig. 2 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 dup instruction or is removed.

2.1.1 Augmented LDAG Construction

QTrees are transformed into LDAGs by a post-order depth-first recursive traversal over the QTrees. 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. 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.

2.1.2 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 assigned to be 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 facilitate 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
Algorithm 1 dag_levelize_ghost (tree t, level)
1: nextlevel ← level + 1
2: match ← lookup (t)
3: if match ≠ null then
4: if match.level < nextlevel then
5: relink ← dag_ghost_move_node (nextlevel, t, match)
6: return relink
7: else if match.level = lookup (t) then
8: relink ← 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 ← make_new_level (t, nextlevel)
17: record (new)
18: else
19: new ← 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 ← dag_levelize_ghost (t.left, nextlevel)
25: make_edge (new, lhs)
26: rhs ← dag_levelize_ghost (t.right, nextlevel)
27: make_edge (new, rhs)
28: else if t is unary operation then
29: child ← dag_levelize_ghost (t.child, nextlevel)
30: make_edge (new, child)
31: end if
32: return new

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: return
11: end if

2.2 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 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. Let pn be a node for which the QH position must be found. QH relative position with respect of pn is found after a node in a traversal P from pn−1 to pn (α-node) meets one of two conditions. The first condition is that the node is the α-node, Pn = p0. QH position is at α-node of the next level lev(p) + 1. The second condition is that Pn is a binary or unary operation and has a hard edge to one of its operands qn. QH position is given by qn’s following node as a result of a level-order traversal. Notice that qm’s following node can be qm+1, or the α-node of lev(qm) + 1 if qm is the last node in lev(qm). The proposed algorithm is described in Algorithm 3. After the QH position with respect of pn has been found, the only operation to calculate the offset reference value for each of pn’s operands is to measure the distance δ 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.

2.3 Instruction Scheduling Phase

The instruction scheduling algorithm of our compiler is a variation of basic block scheduling [8] where the only difference is that instructions are generated from a level-order topological order 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. In-
Algorithm 3 $\text{qh\_pos (LDAG } w, \text{ Node } u)$

1: $I \leftarrow \text{getLevel (} u \text{)}$
2: for $i = u.\text{prev}$ to $I.a$-node do
3: if isOperation ($i$) then
4: if isHardEdge ($i$.right) then
5: $v \leftarrow \text{BFS\_nextnode (} i$.right$)
6: return $v$
7: end if
8: if isHardEdge ($i$.left) then
9: $v \leftarrow \text{BFS\_nextnode (} i$.left$)
10: return $v$
11: end if
12: end if
13: end for
14: $L \leftarrow \text{getNextLevel (} u \text{)}$
15: $v \leftarrow L.a$-node
16: return $v$

Algorithm 4 $\text{OpOffset (LDAG } w, \text{ Node } v, \text{ Operand } r)$

1: offset $\leftarrow \delta(qh\_pos(} w, v\text{),} r \text{)$
2: return offset

struction 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 QIR list. QIR is a single operand low level intermediate representation capable to express 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. A 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 reference. For every compiled function this phase also inserts the QIR instructions for the function’s prologue and epilogue.

2.4 Statement Merging Transformation

Statement merging transformation in for natural ILP extraction. It reorders the instructions of a sequential program in such a way that all independent instructions from different statements are in the same level and can be executed in parallel following the principle of the QCM. This phase makes a dependence analysis on individual instructions of different statements looking for conflicts in memory locations. Statements are considered the transformation unit. Whenever an instruction is reordered, 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 dependence analysis looks for two accesses to the same memory location 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. 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 the available parallelism 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 merging is available in the queue compiler as an optimization flag which can be enabled upon user request.

2.5 Assembly Generation Phase

The last stage of the queue compiler is the assembly code generation for the QueueCore processor. It is done by a one-to-one translation from QIR to assembly code. The assembly generator is in charge of inserting $\text{covop}$ instructions to expand the operand field of those instructions that have operands beyond the limits of the operand field bits. Fig.3.(a) shows the generated assembly code and Fig.3.(b) shows the assembly code with natural parallelism exposed for the C program in Fig.2.(a). Notice that the original assembly code and the assembly code after statement merging contain exactly the same instructions with the only difference that the order the the instructions change. All instructions have one operand. Depending on the instruction type the only operand has different meaning. The highlighted code fragment in Fig.3.(a) shows the assignment of an array element indexed by variable to another variable, in C language “$x = a[1]$”. 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 $\text{covop}$ 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 $\text{QH}$ and the second operand from $\text{QH}+1$. The $\text{ld}$ 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 $\text{QH}$ to memory using base addressing.

3 Evaluation

To demonstrate the efficiency of our one-offset queue computation model, we developed a C compiler that targets the QueueCore processor. For a set of 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
Table 1: 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>

3.1 Results

The resulting back-end for the QueueCore consists of about 8000 lines of C code.

3.1.1 Queue Compiler Evaluation

Fig.4 shows the improvement factor of the compiled code with statement merging transformation over the original code without statement merging, both scheduled using the level-order traversal. All benchmarks show an improvement gain ranging from 1.73 to 4.25. The largest ILP improvement is for the fft8g program because it contains very large loop bodies without control flow where the statement merging transformation can work most effectively. Statement merging is a code motion transformation and does not insert or eliminate instructions.

The graph in Fig.5 compares the ILP improvement of the queue compiler over the optimizing compiler for MIPS processor. 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 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. Fig.6 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...
4 Conclusion

In this paper we presented code generation for QueueCore compiler. The queue compiler takes advantage of this design and it is capable to expose the 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 traditional RISC machine for a set of numerical benchmarks. Our proposed architecture achieved higher ILP by an average factor of 1.38. Furthermore, the QueueCore processor is also a good candidate for an embedded processor from its reduced instruction set achieving more compact code than embedded RISC-like processors by 26%.

References


31% denser code than MIPS I, 20% denser code than the embedded MIPS16, and 26% denser code than the ARM/Thumb processor.