Compiler Support for Code Size Reduction using a Queue-based Processor

Arquimedes Canedo[1], Ben Abderazek[1], and Masahiro Sowa[1]

Graduate School of Information Systems, University of Electro-Communications,
Chofugaoka 1-5-1 Chofu-Shi 182-8585, Japan

Abstract. Queue computing delivers an attractive alternative for embedded systems. The main features of a queue-based processor are a dense instruction set, high-parallelism capabilities, and low hardware complexity. This paper presents the design of a code generation algorithm implemented in the queue compiler infrastructure to achieve high code density by using a queue-based instruction set processor. We present the efficiency of our code generation technique by comparing the code size and extracted parallelism for a set of embedded applications against a set of conventional embedded processors. The compiled code is, in average, 12.03% more compact than MIPS16 code, and 45.1% more compact than ARM/Thumb code. In addition, we show that the queue compiler, without optimizations, can deliver about 1.16 times more parallelism than fully optimized code for a register machine.

Keywords: Code Generation, Code Size Reduction, Reduced bit-width Instruction Set, Queue Computation Model

1 Introduction

One of the major concerns in the design of an embedded processor is code density. Code size is directly related to the size of the memory and therefore the cost of the system [1, 2]. Compact instructions improve memory bandwidth, fetching time, and power consumption [3].

0-operand instruction set computers are the best alternative for achieving high code density. Stack machines have demonstrated their capacity to generate compact programs [4] but their sequential nature [5, 6] excludes them from the high-performance arena. Queue machines use a FIFO data structure to perform computations [7–9]. Opposite to the stack machines, the queue machines offer a natural parallel computation model as two different locations are used for reading and writing, thus avoiding a bottleneck. All read/write accesses to the FIFO queue are done at the head and rear of the queue respectively. Thus, instructions implicitly reference their operands. This characteristic allows the instruction set to be short and free of false dependencies freeing the hardware from the complexity of register renaming mechanism. Compiling for queue machines presents serious challenges as the problem of laying out a directed

acyclic graph in a queue is known to be NP-complete [10]. Additional hardware support alleviates this task to a certain extent [21]. In [7–9], we proposed a Parallel Queue Processor (PQP), an architecture based on the producer order queue computation model. The PQP allows the execution of any data flow graph by allowing instructions to read their operands from a location different than the head of the queue, specified as an explicit offset reference in the instruction. Although the place to read operands can be specified, the place to write operands remains fixed at the rear of the queue. As the rule of writing data (producing data) remains fixed, this model is known as the producer order model. However, in real applications, instructions that require both of their operands to be read from a place different than the head of the queue are scarce [11]. Therefore, we present the possibility of reducing the instruction set to encode at most one operand offset reference placing additional burden on the compiler. This modification has the purpose of reducing the bits in the instruction and thus improving the code size of compiled programs. The PQP is a 32-bit processor with 16-bit instructions capable of executing up to four instructions in parallel. Instructions allow one of its operands to be read from anywhere in the queue by adding the offset reference provided in the instruction to the current position of the head of the queue. A special unit called Queue Computation Unit is in charge of finding the physical location of source operands and destination within the queue register file enabling parallel execution. For cases when there are insufficient bits to express large constants and memory offsets, a `covop` instruction is inserted. This special instruction extends the operand field of the following instruction by concatenating it to its operand. We have developed the queue compiler infrastructure for the producer order model [11] as part of the design space exploration chain for the PQP processor that includes a functional simulator, cycle accurate simulator, and CPU described in RTL level. In this compiler, the code generation algorithm produces code for an abstract PQP architecture that is insufficient for our present goals. Another serious limitation of the queue compiler is its inability to compile programs with system header files restricting its input only to simple programs.

In this paper, we propose a code size-aware optimizing compiler infrastructure that efficiently generates compact code using a queue-based instruction set. The compiler deals with the potential increase of instructions by inserting a special queue instruction that creates a duplicate of a datum in the queue. The presented code generation algorithm selectively inserts these special instructions to constrain all instructions in the program to have at most one explicit operand. We analyze the compiling results for a set of embedded applications to show the potential of our technique highlighting the code size and parallelism at the instruction level. In summary, the contributions of this paper are:

- To demonstrate that a compiler for the PQP processor is able to produce compact and efficient code.
- The methods and algorithms to build the code size-aware compiling infrastructure for a producer order queue based processor.

The remainder of the paper is as follows: Section 2 gives a summary of the related work. In Section 3 we give an overview of the queue computation model, the problems related to compilation of programs, and we motivate our target of using the queue computation model to reduce code size in embedded applications. In Section 4 we describe the code generation algorithm developed for restricting directed acyclic graphs into a suitable form for executing in a constrained queue instruction set. Section 5 reports our experimental results. Section 6 we open discussion, and Section 7 concludes.

2 Related Work

Improving code density in CISC and RISC architectures has been a thoroughly studied problem. A popular architecture enhancement for RISC is to have two different instruction sets [12, 13] in the same processor. These dual instruction set architectures provide a 32-bit instruction set, and a *reduced instruction set* of 16-bit. The premise is to provide a reduced 16-bit instruction set for the operations most frequently executed in applications so two instructions are fetched instead of one. The improvement in code density comes with a performance degradation since more 16-bit instructions are required to execute the same task when compared to 32-bit instructions. The ARM/Thumb [12] and MIPS16 [14] are examples of dual instruction sets. In [12], a 30% of code size reduction with a 15% of performance degradation is reported for the ARM/Thumb processor. Compiler support for dual instruction set architectures is crucial to maintain a balance between code size reduction and performance degradation. Different approaches have been proposed to cope with this problem [15–20].

Queue Computation Model (QCM) refers to the evaluation of expressions using a first-in first-out queue, called *operand queue*. This model establishes two rules for the insertion and removal of elements from the operand queue. Operands are inserted, or enqueued, at the rear of the queue. And operands are removed, or dequeued, from the head of the queue. Two references are needed to track the location of the head and the rear of the queue. The *Queue Head*, or `QH`, points to the head of the queue. And *Queue Tail*, or `QT`, points to the rear of the queue. Only a handful of queue machine hardware designs have been proposed. In 1985, Bruno [21] established that a level-order scheduling of an expression's parse tree generates the sequence of instructions for correct evaluation. A level-order scheduling of directed acyclic graphs (DAG) still delivers the correct sequence of instructions but requires additional hardware support. This hardware solution is called an *Indexed Queue Machine*. The basic idea is to specify, for each instruction, the location with respect of the head of the queue (an index) where the result of the instruction will be used. An instruction may include several indexes if it has multiple parent nodes. All these ideas were in the form of an abstract queue machine until the hardware mechanisms of a superscalar queue machine were proposed by Okamoto [22].

The operand queue has been used as a supporting hardware for two register based processors for the efficient execution of loops. The WM architecture [23]

is a register machine that reserves one of its registers to access the queue and demonstrates high streaming processing capabilities. Although the compiler support for the WM Architecture has been reported in [23], details about code generation are not discussed. In [24, 25] the use of Register Queues (RQs) is demonstrated to effectively reduce the register pressure on software pipelined loops. The compiler techniques described in this processor do not present a significant contribution for queue compilation since the RQs are treated as special registers. In our previous work [7, 8], we have designed a parallel queue processor (PQP) capable of executing any data flow graph. The PQP breaks the rule of dequeueing by allowing operands to be read from a location different than the head of the queue. This location is specified as an offset in the instruction. The fundamental difference with the indexed queue machine is that our design specifies an offset reference in the instruction for reading operands instead of specifying an index to write operands. In the PQP's instruction set, the writing location at the rear of the queue remains fixed for all instructions. 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 [10] together with the proof of NP-completeness for recognizing a 2-queue DAG. In [26], 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 programs is smaller when compared to RISC code.

In [27], we developed a queue compiler based on a conventional retargetable compiler for register machines. To provide a clean model of the queue, a very large amount of general purpose registers were defined in the machine definition files of the compiler to avoid the spillage of registers by the register allocator. Nevertheless, mapping register code into the queue computation model turned 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 part of the initiative to deliver a compiler technology designed specifically for the queue computation model.

3 Queue Computing Overview

Correct queue programs are obtained from a level-order traversal of the parse trees of the expressions [21]. Figure 1(a) shows the parse tree for a simple expression and the obtained pseudo-program. The first four executed instructions (L_3) place four operands in the queue and `QH` points at the first loaded operand and `QT` to an empty location after the last loaded operand. The contents of the queue are the following: $\{a, b, a, b, \epsilon\}$. The next two binary instructions from level L_2 consume the four operands leaving the queue in the

following status: $(a+b), (a-b), \epsilon$. The only instruction in level L_1 consumes the previously computed subexpressions and generates a new intermediate value: $\{(a+b)/(a-b), \epsilon\}$. The last instruction in level L_0 consumes the only operand in the queue and stores it into memory leaving an empty queue with `QH` and `QT` pointing at the same empty location: $\{\epsilon\}$.

Notice that in the previous example, the operands a, b are loaded twice in level L_3. Complications arise when the program is scheduled from its DAG. Figure 1(b) shows the DAG for the same expression. Notice that operands a, b are loaded once and are shared by two operations $(+, -)$. The addition correctly obtains its operands from `QH` and writes back the result into `QT`. By the time the subtraction is executed, `QH` does not point to any of its operands. If the basic enqueueing/dequeueing is kept it leads to incorrect results. Our PQP solves this problem by allowing the operands to be dequeued from a location different than `QH`. The location from where to dequeue the operands is represented as an offset in the instruction as shown in Figure 1(b). The offset represent the relative distance with respect of `QH` from where the operand must be read. We classify all binary and unary instructions into three categories: 2-offset, 1-offset, and 0-offset instructions. 2-offset instructions read both operands from a place different than `QH`, e.g. "`sub -2, -1`". 1-offset instructions read one operand from `QH` and the other from a different location (not shown in the Figure). And 0-offset instructions read both operands directly from `QH`, e.g. "`add 0,1`". The PQP updates the `QH` position automatically every time an operand is read directly from `QH` [7]. For 0-offset binary instructions `QH` is moved two positions after its execution. For 1-offset instructions `QH` is moved only one position, and for 0-offset instructions `QH` is not updated. This mechanism guarantees correct evaluation of any data flow graph.

To have an insight of the demands of applications on PQP instruction set we compiled several applications and obtained the distribution of offsetted instructions. Table 3 shows the distribution of offsetted PQP instructions for a set of embedded and numerical applications. Notice that the 2-offset instructions represent from 0.1% to 2.6%. 1-offset instructions represent from 2.9% to 18.2%. And 0-offset instructions represent the majority of instructions in the applications. Restricting PQP's instructions to encode at most one offset makes instructions shorter and covers the great majority of instructions in programs. This has direct effect over the code size of the compiled programs since only a single operand is encoded in the instruction format. In the following section we discuss the compiler technique required to deal with the correct evaluation of 2-offset instructions in programs on a constrained 1-offset instruction set.

4 Code Generation Algorithm

Figure 2 shows a 4-point fourier transform DAG and its evaluation using the PQP instruction set [7]. 0-offset instructions are represented by the white nodes, 1-offset instructions by the gray nodes, and 2-offset instructions by the black nodes. It is responsibility of the compiler to transform the DAG into a suitable

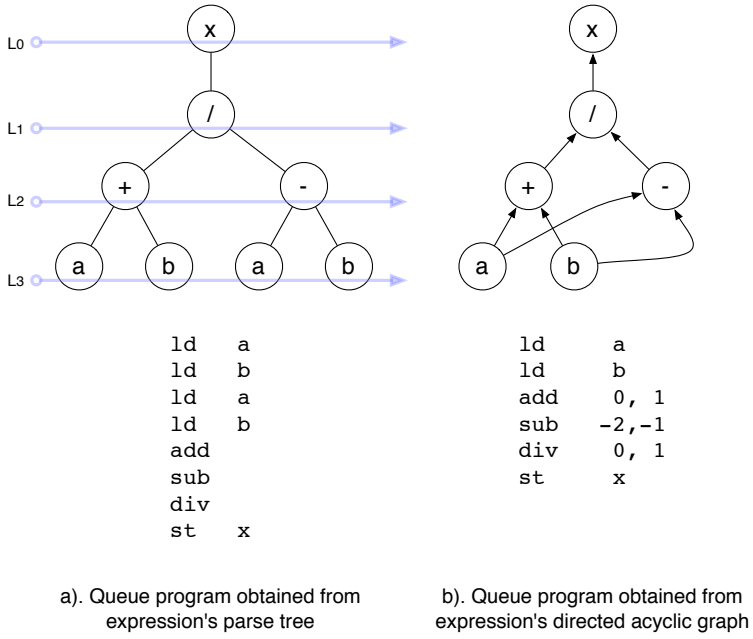


Fig. 1. Evaluation of expressions in the queue computation model. (a) queue code from a parse tree. (b) hardware support for evaluation of any directed acyclic graph.

form to be executed by our reduced PQP instruction set. One approach to deal with this problem is to re-schedule the instructions to execute a subexpression while reading the other intermediate result with the available offset reference. While this approach generates correct programs, it has the overhead of extra instructions. In order to efficiently generate compact programs we propose the utilization of a special queue instruction called `dup` instruction. The purpose of this instruction is to create a copy, *duplicate*, a datum in the queue. The `dup` instruction has one operand which is an offset reference that indicates the location with respect of `QH` from where to copy the datum into `QT`. Figure 3 shows the transformed DAG with extra `dup` instructions. These two `dup` instructions place a copy of the left hand operand for nodes $+_2, +_3$ transforming them into 1-offset instructions.

4.1 Queue Compiler Infrastructure

For a better comprehension of our algorithm we introduce the queue compiler infrastructure [11]. The queue compiler parses C files using GCC 4.0.2 front-end. Our back-end takes GCC's GIMPLE trees [28, 29] as input and generates assembly code for the PQP processor. The first task of the back-end is to expand three-address code GIMPLE representation into unrestricted trees of arbitrary

Table 1. Distribution of PQP offsetted instructions for a set of embedded and numerical applications.

Application	0-offset	1-offset	2-offset
MPEG2	90.0%	9.3%	0.7%
H263	86.7%	11.4%	1.9%
Susan	97.0%	2.9%	0.1%
FFT8G	93.9%	5.9%	0.2%
Livermore	82.2%	15.6%	2.2%
Linpack	80.8%	18.2%	1.0%
Equake	85.5%	11.9%	2.6%

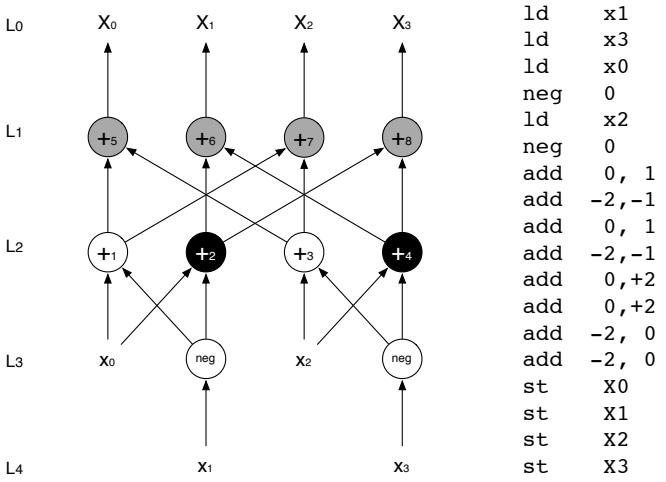


Fig. 2. 4-point Fourier transform directed acyclic graph.

depth and width called QTrees. QTrees help in the construction of our core data structures, the *level DAGs* (LDAG) [10]. A LDAG is a data structure that assigns all nodes in a DAG into levels and expresses well the interaction between operations and operands under the queue computation model. The code generation algorithm takes LDAGs to perform the offset calculation for every instruction, the most important feature of the queue compiler. After all offset references have been computed, the code generator schedules the program in level-order manner and crafts a linear low level intermediate representation called QIR. The QIR inherits the characteristics of the queue computation model making it a single operand intermediate representation. The operand is used by memory and control flow operations to represent memory locations, immediate values, target label, and function names. Offset references for binary and unary instructions are considered attributes of the instructions rather than operands.

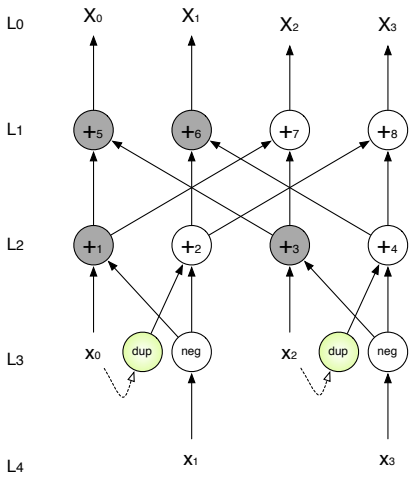


Fig. 3. Fourier transform's directed acyclic graph with `dup` instructions.

The last phase of the compiler takes QIR and generates the final assembly code for the PQP. Figure 4 shows the phases of the queue compiler.

4.2 Code Generation Algorithm for Code Size reduction

We implemented the algorithm into the queue compiler infrastructure. The main task of this algorithm is to determine the correct location of `dup` instructions in the programs' data flow graph. The algorithm accomplishes its task in two stages during code generation. The first stage converts QTrees to LDAGs augmented with *ghost nodes*. A ghost node is a node without operation that serves as placeholder for `dup` instructions. This first stage gathers information about what instructions violate the 1-offset instruction restriction. The second stage decides which ghost nodes are turned into `dup` nodes or are eliminated from the flow graph. Finally, a level-order traversal of the augmented LDAGs computes the offset references for all instructions and generates QIR as output.

Augmented LDAG construction Algorithm 1 presents the leveling function that transforms QTrees into ghost nodes augmented LDAGs. The algorithm makes 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 1. *The sink of an edge must be always in a deeper or same level than its source.*

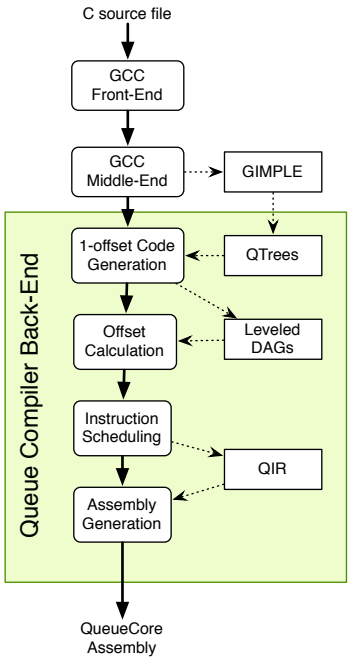


Fig. 4. Queue Compiler Infrastructure

Definition 2. *An edge to a ghost node spans only one level.*

When an operand is found in the lookup table the Definition 1 must be kept. Line 5 in Algorithm 1 is reached when the operand is found in the lookup table and it has a shallow level compared to 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 as 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 1 and 2. Figure 5 illustrates the result of leveling the QTree for the expression $x = (a * a)/(-a + (b - a))$. Figure 5.b shows the resulting LDAG augmented with ghost nodes.

`dup` instruction assignment and ghost nodes elimination The second stage of the algorithm works in two passes as shown in Lines 4 and 7 in Algorithm 2. Function `dup_assignment()` decides whether ghost nodes are substituted by `dup` nodes or eliminated from the LDAG. Once all the ghost

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
```

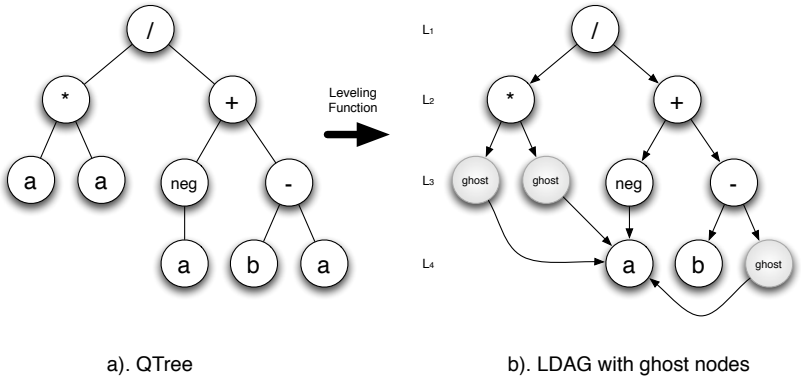


Fig. 5. Leveling of QTree into augmented LDAG for expression $x = \frac{a \cdot a}{-a+(b-a)}$

nodes have been transformed or eliminated, the second pass performs a level order traversal of the LDAG, and for every instruction the offset references with respect of `QH` are computed in the same way as in [11]. The output of the code generation algorithm is QIR where all instructions use at most one offset reference.

Algorithm 2 codegen ()

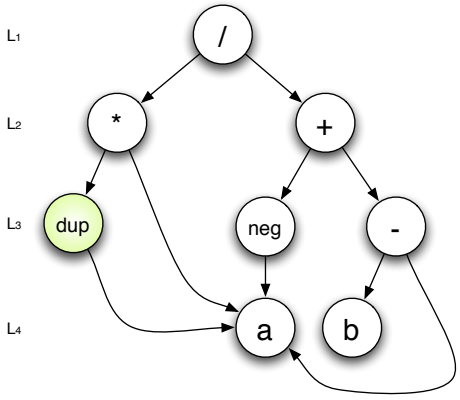
```
1: for all basic blocks BB do
2:    for all expressions W_k in BB do
3:       for all instructions I_j in TopBottom (W_k) do
4:          dup_assignment (I_j)
5:       end for
6:       for all instructions I_j in LevelOrder (W_k) do
7:          p_qcm_compute_offsets (W_k, I_j)
8:       end for
9:    end for
10: end for
```

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 transformed into 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 3 describes the function `dup_assignment()`. The effect of Algorithm 3 is illustrated in Figure 6. The algorithm takes as input the

LDAG with ghost nodes shown in Figure 5.b and performs the steps described in Algorithm 3 to finally obtain the LDAG with `dup` instructions as shown in Figure 6.a. The last step in the code generation is to perform a level-order traversal of the LDAG with `dup` nodes and compute for every operation, the offset value with respect of `QH`. `dup` instructions are treated as unary instructions by the offset calculation algorithm. The final constrained 1-offset QIR for the expression $x = (a * a)/(-a + (b - a))$ is given in Figure 6.b.

Algorithm 3 dup_assignment (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
```



a). LDAG with `dup` instructions

```
ld     a
ld     b
dup    0
neg    0
sub   -1
mul   -2
add    0
div    0
st     x
```

b). 1-offset P-Code program with `dup` instruction

Fig. 6. 1-offset constrained code generation from a LDAG

5 Experimental Results

5.1 Code Size Comparison

We selected ten applications commonly used in embedded systems from MiBench and MediaBench suites [30, 31]. This selection includes video compression applications (H263, MPEG2), signal processing (FFT, Adpcm), image recognition (Susan), encryption (SHA, Blowfish, Rijndael), and graph processing (Dijkstra, Patricia). We compiled these applications using our queue compiler infrastructure with the presented code generation algorithm. The resulting code is 1-offset PQP assembly code where every instruction is 16-bit long. We compare our result with the code of two dual-instruction embedded RISC processors: MIPS16 [13], ARM/Thumb [12]. With two traditional RISC machines: MIPS I [14], ARM [32]. And with a traditional CISC architecture: Pentium processor [33]. We prepared GCC 4.0.2 compiler for the other five architectures and measured the code size from the text segment of the object files. All compilers, including our compiler, were configured without optimizations in order to compare the density of the baseline code. Figure 7 shows the normalized code size for all applications with respect of MIPS code. These results confirm the higher code density of the embedded RISC processors over their original 32-bit versions. Our PQP code is, in average, 12.03% denser than MIPS16 code, and 45.1% denser than ARM/Thumb code. Compared to a traditional variable length instruction set CISC machine, our PQP achieves 12.58% denser code.

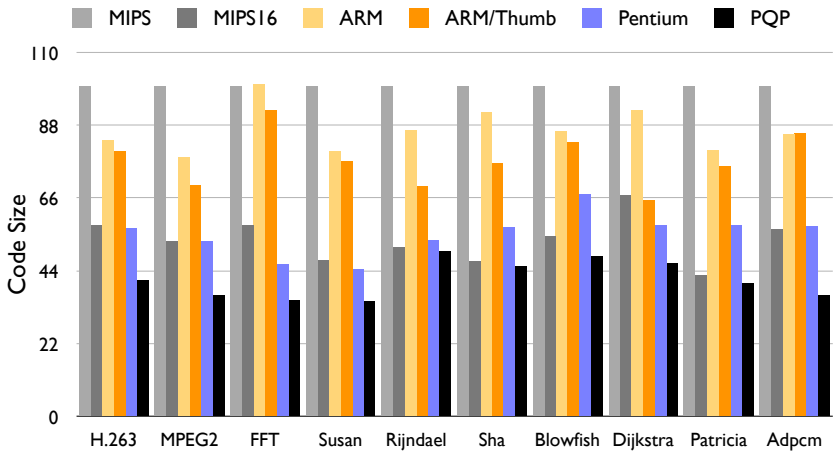


Fig. 7. Code size comparison

5.2 Effect of `dup` instructions on Code Size

Our algorithm inserts `dup` instructions in the program's data flow graph to constrain all instructions to at most one offset reference. Table 5.2 shows the extra `dup` instructions inserted over the original 2-offset PQP code. The increase in number of instructions is below 1% for the chosen embedded applications. This confirms that for embedded applications the 2-offset instructions are rare and our technique can take advantage of this characteristic to improve code size by reducing the bits in the instruction to encode at most one offset reference.

Table 2. Number of inserted `dup` instructions for the compiled embedded applications

Application	`dup`	2-offset PQP
H263	751	39376
MPEG2	299	42016
FFT	18	9127
Susan	11	11177
Rijndael	12	821
Sha	5	711
Blowfish	16	5377
Dijkstra	0	910
Patricia	1	1260
Adpcm	1	1213

5.3 Instruction Level Parallelism Analysis

The queue compiler exposes *natural* parallelism found in the programs from the level-order scheduling. All instructions belonging to the same level in the LDAG are independent from each other and can be executed in parallel by the underlying queue machine. We compare the parallelism extracted by our compiler against the parallelism extracted by the MIPS I compiler. Our compiler was set with all optimizations turned off, and the MIPS-GCC compiler was configured with maximum optimization level (-O3). The extracted parallelism for the MIPS architecture was measured from the assembly output code using a conservative analysis by instruction inspection [34] to detect the data dependencies between registers and grouping those instructions that can be executed in parallel. For the PQP code, data dependences are given by the levels in the LDAG and are not expressed in the instructions. The only information available in the instructions is their offset and it cannot be used to determine dependences as it is relative to `QH`. To measure the parallelism of our code the compiler emits marks at the beginning of every level in the LDAGs grouping all parallel instructions. Figure 8 shows the extracted parallelism by the two compilers. Our compiler extracts about 1.16 times more parallelism than fully optimized RISC code.

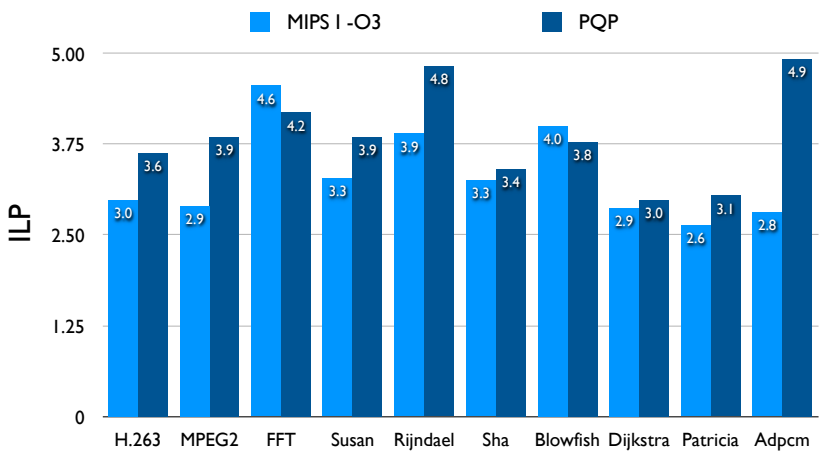


Fig. 8. Compile-time extracted instruction level parallelism

6 Discussion

The above results show that embedded applications require a small amount of 2-offset instructions. This motivates the idea of shortening the PQP instruction set to support at most one offset reference. The compiler is responsible for preserving the data flow graph to fit the program into the constrained instruction set by the addition of a single `dup` instruction. We believe that there is a chance to reduce even more the code size of PQP programs by using a variable length instruction set. Instructions that read their operands directly from `QH` can be reduced to 0-operand instructions without wasting the field to encode their *dead* offset. Another possibility is to extend the presented algorithm to constrain all instructions to 0-offset format with the penalty of a larger increase of extra `dup` instructions.

With our modification and 16-bit instructions configuration, our compiler generates denser code than embedded RISC processors, and a CISC processor. Also we demonstrated that the level-order scheduling naturally exposes more parallelism than fully optimized RISC code. We believe that the addition of classical and ILP optimizations, our compiler can generate higher quality code.

7 Conclusion

In this paper we presented an improved queue compiler infrastructure to reduce code size by using a reduced queue-based instruction set of the PQP processor. The algorithm handles the correct transformation of the data flow graph to evaluate the programs using a reduced queue instruction set. The algorithm was successfully implemented in the queue compiler infrastructure. We have presented the potential of our technique by compiling a set of embedded

applications and measuring the code size against a variety of embedded RISC processors, and a CISC processor. The compiled code is about 12.03% and 45.1% denser than MIPS16 and ARM/Thumb architectures. The efficiency of our code generation technique is not only limited to code size but also the generation of parallel code. Without any optimization, our compiler achieves, in average, 1.16 times more parallelism than fully optimized code for a RISC machine. Queue architectures are a viable alternative for executing applications that require small code size footprint and high performance.

References

1. Liao, S.Y., Devadas, S., Keutzer, K.: Code density optimization for embedded DSP processors using data compression techniques. In: Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95). (1995) 272
2. Wolfe, A., Chanin, A.: Executing compressed programs on an embedded RISC architecture. In: Proceedings of the 25th annual international symposium on Microarchitecture. (1992) 81–91
3. Gordon-Ross, A., Cotterell, S., Vahid, F.: Tiny instruction caches for low power embedded systems. ACM Transactions on Embedded Computing Systems (TECS) **2**(4) (November 2003) 449–481
4. Koopman, P.J.: Stack Computers: the new wave. Ellis Horwood (1989)
5. Vijaykrishnan, N.: Issues in the Design of a Java Processor Architecture. PhD thesis, University of South Florida (1998)
6. Shi, H., Bailey, C.: Investigating Available Instruction Level Parallelism for Stack Based Machine Architectures. In: Proceedings of the Digital System Design, EUROMICRO Systems on (DSD'04). (2004) 112–120
7. Sowa, M., Abderazek, B., Yoshinaga, T.: Parallel Queue Processor Architecture Based on Produced Order Computation Model. Journal of Supercomputing **32**(3) (June 2005) 217–229
8. Abderazek, B., Yoshinaga, T., Sowa, M.: High-Level Modeling and FPGA Prototyping of Produced Order Parallel Queue Processor Core. Journal of Supercomputing **38**(1) (October 2006) 3–15
9. Abderazek, B., Kawata, S., Sowa, M.: Design and Architecture for an Embedded 32-bit QueueCore. Journal of Embedded Computing **2**(2) (2006) 191–205
10. Heath, L.S., Pemmaraju, S.V.: Stack and Queue Layouts of Directed Acyclic Graphs: Part I. SIAM Journal on Computing **28**(4) (1999) 1510–1539
11. Canedo, A.: Code Generation Algorithms for Consumed and Produced Order Queue Machines. Master's thesis, University of Electro-Communications, Tokyo, Japan (September 2006)
12. Goudge, L., Segars, S.: Thumb: Reducing the Cost of 32-bit RISC Performance in Portable and Consumer Applications. In: Proceedings of COMPCON '96. (1996) 176–181
13. Kissel, K.: MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group (1997)
14. Kane, G., Heinrich, J.: MIPS RISC Architecture. Prentice Hall (1992)
15. Krishnaswamy, A., Gupta, R.: Profile Guided Selection of ARM and Thumb Instructions. In: ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems. (2002) 56–64

16. Halambi, A., Shrivastava, A., Biswas, P., Dutt, N., Nicolau, A.: An Efficient Compiler Technique for Code Size Reduction using Reduced Bit-width ISAs. In: Proceedings of the Conference on Design, Automation and Test in Europe. (2002) 402
17. Sheayun, L., Jaejin, L., Min, S.: Code Generation for a Dual Instruction Processor Based on Selective Code Transformation. In: Lectures in Computer Science. (2003) 33–48
18. Kwon, Y., Ma, X., Lee, H.J.: Pare: instruction set architecture for efficient code size reduction. Electronics Letters (1999) 2098–2099
19. Krishnaswamy, A., Gupta, R.: Enhancing the Performance of 16-bit Code Using Augmenting Instructions. In: Proceedings of the 2003 SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems. (2003) 254–264
20. Krishnaswamy, A.: Microarchitecture and Compiler Techniques for Dual Width ISA Processors. PhD thesis, University of Arizona (September 2006)
21. Preiss, B., Hamacher, C.: Data Flow on Queue Machines. In: 12th Int. IEEE Symposium on computer Architecture. (1985) 342–351
22. Okamoto, S.: Design of a Superscalar Processor Based on Queue Machine Computation Model. In: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing. (1999) 151–154
23. Wulf, W.: Evaluation of the WM Architecture. In: Proceedings of the 19th annual international symposium on Computer architecture. (1992) 382–390
24. Smelyanskiy, M.G., Tyson, S., Davidson, E.S.: Register queues: a new hardware/software approach to efficientsoftware pipelining. In: Proceedings of Parallel Architectures and Compilation Techniques. (2000) 3–12
25. Fernandes, M.: Using Queues for Register File Organization in VLIW Architectures. Technical Report ECS-CSG-29-97, University of Edinburgh (1997)
26. Schmit, H., Levine, B., Ylvisaker, B.: Queue Machines: Hardware Computation in Hardware. In: 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. (2002) 152
27. Canedo, A., Abderazek, B., Sowa, M.: A GCC-based Compiler for the Queue Register Processor. In: Proceedings of International Workshop on Modern Science and Technology. (May 2006) 250–255
28. Merrill, J.: GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In: Proceedings of GCC Developers Summit. (2003) 171–180
29. Novillo, D.: Design and Implementation of Tree SSA. In: Proceedings of GCC Developers Summit. (2004) 119–130
30. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization. (2001) 3–14
31. Lee, C., Potkonjak, M., Mangione-Smith, W.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: 30th Annual International Symposium on Microarchitecture (Micro '97). (1997) 330
32. Patankar, V., Jain, A., Bryant, R.: Formal verification of an ARM processor. In: Twelfth International Conference On VLSI Design. (1999) 282–287
33. Alpert, D., Avnon, D.: Architecture of the Pentium microprocessor. Micro, IEEE **13**(3) (June 1993) 11–21
34. Debray, S., Muth, R., Weippert, M.: Alias Analysis of Executable Code. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. (1998) 12–24