An Efficient Code Generation Algorithm for Code Size Reduction Using 1-Offset P-Code Queue Computation Model

Arquimedes Canedo, Ben A. Abderazek, and Masahiro Sowa

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

Abstract. Embedded systems very often demand small memory footprint code. A popular architectural modification to improve code density in RISC embedded processors is to use a dual instruction set. This approach reduces code size at the cost of performance degradation due to the greater number of reduced width instructions required to execute the same task. We propose a novel alternative for reducing code size by using a single reduced instruction set queue machine. We present a efficient code generation algorithm to insert additional instructions to be able to execute programs in the reduced instruction set. Our experiments show that the insertion of additional instructions is minimal and we demonstrate improved code size reduction of 16% over MIPS16, 26% over Thumb, and 50% over MIPS32 code. Furthermore, we show that our compiler without any optimization is able to extract about the same parallelism than fully optimized RISC code.

1 Introduction

One of the major concerns in the design of an embedded RISC processor is the code density. Code density is tightly related to performance, energy consumption, and ROM memory utilization. A well known design strategy to reduce code size in RISC architectures is allowing the processor to execute two different instruction sets [5,8,17]. These dual instruction set architectures provide a 32-bit instruction set, and a *reduced instruction set* of 16-bit. The idea is to shorten the instruction length to improve the code size as well as fetching efforts as two short instructions can be read instead of one at the same cost. As the reduced instruction set has width limitation, more 16-bit instructions are needed to execute a given task. Since the 16-bit code requires more instructions than the 32-bit code to execute the same task, the performance of the reduced instruction set code is lower than the 32-bit code. The ARM-Thumb [5] and MIPS16 [8] are examples of dual instruction sets. In [5], 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 due to the

T.-W. Kuo et al. (Eds.): EUC 2007, LNCS 4808, pp. 196–208, 2007.

increase in number of instructions. Different researches have been proposed to cope with this problem. Profile guided compiler heuristics at function level granularity have been proposed in [9] to determine which code, from the two available, maximizes the code size reduction without causing a significant loss in performance. Halambi et.al. proposed in [6] a compiler algorithm at instruction level granularity based on a profitability analysis function discerning between code size and performance. A similar approach based on a path-based profitability analysis is proposed in [15]. Kwon et.al. in [12], proposed a compiler-architecture interaction scheme to compress even more the ARM-Thumb instructions by using a partitioned register file and an efficient register allocation algorithm. In [10,11], Krishnaswamy and Gupta proposed a compiler-architecture approach where the compiler identifies pairs of 16-bit instructions that can be combined safely into a single 32-bit instruction and replaces it with augmenting instructions that are later coalesced by the hardware at zero performance penalty.

An attractive alternative to reduce code size is using a queue-based computation model. The queue computation model uses a FIFO data structure, analogous to a register file, to perform computations [3]. All accesses to the FIFO queue, named operand queue, follow the rules of enqueueing and dequeueing. These accesses are done at the rear, and head of the queue, respectively. Thus, instructions are free of location names from where to dequeue or enqueue their operands. Having only the instruction itself allows the instruction set to be short.

In our previous work we have investigated and designed a Parallel Queue Processor (PQP) based on the Queue Computation Model [16,1,2]. The PQP breaks the rule of dequeueing to make a better utilization of the data in the operand queue. This feature leads to better performance and shorter programs. PQP instructions allows operands to be dequeued from a specified position with respect of the head of the queue. Thus, PQP instruction set format requires the encoding of the offset references making it longer than the instruction set for a traditional queue machine. We found in [4], that for a set of scientific programs the instructions that require two offsets are almost nonexistent. Below 10% of the instructions require only one offset reference, and above 90% of remaining instructions require no offsets to be specified. As queue programs require at most one offset, we propose to reduce the number of offsets encoded in the instructions of the PQP from two to one. Having only one offset reference to be encoded in the instruction set saves bits in the instruction format making instructions shorter. In this paper, we present a compiler and architecture support to reduce code size using a reduced PQP instruction set. An efficient code generation algorithm for 1-offset P-Code QCM is described in detail. Our algorithm is able to reduce code size while keeping the instruction count increase very low since the queue instructions are free of false dependencies. To our best knowledge, code generation algorithms for queue machines have not been proposed by previous published work.

The remainder of this paper is organized as follows: Section 2 gives an overview of the queue computation model, the queue compiler infrastructure,

and describes the 1-offset P-Code QCM. Section 3 describes in detail the phases of the code generation algorithm for 1-offset P-Code QCM. We present the results of our approach in Section 4. We discuss our results in Section 5, and conclude in Section 6.

2 1-Offset P-Code Queue Computation Model

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.

A program for the QCM is obtained from a breadth-first traversal of the expressions' parse tree [3]. Figure 1 shows the parse tree of expression $x = (a + b)/(a - b)$ and the resulting *queue program* after a breadth-first traversal of the parse tree. The first four instructions enqueue the operands a, b, a, b from memory into the operand queue. At this stage `QH` points to the first loaded operand (a), and `QT` points to an empty location after the last operand (b). The addition instruction `add` requires two operands to be dequeued (a and b). `QH` now points to the third operand originally loaded (a). After the addition is computed the result is written to `QT`. The rest of the instructions have the same behavior until the expression has been completely evaluated and the result (x) is dequeued to memory.

In our early work [4], we have proposed a *2-offset P-Code* QCM to allow programs to be generated directly from their directed acyclic graphs (DAGs). 2-offset P-Code QCM, strictly follows enqueueing rule but has flexibility in the dequeueing rule of operands. Operands can be taken from `QH` or from a different position other than `QH`. The desired position of the datum that should be dequeued is specified as an offset with respect of the position of `QH`.

Figure 2 shows the DAG and the 2-offset P-Code program for the same expression shown in Figure 1, $x = (a + b)/(a - b)$. Notice that the program

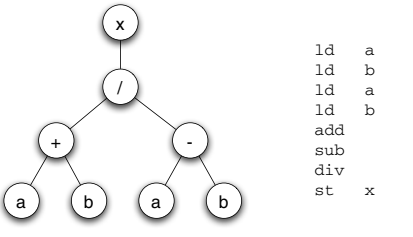

Fig. 1. Queue program generation from an expression's parse tree

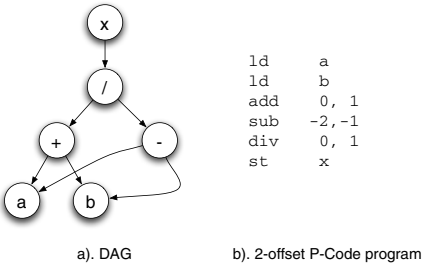

Fig. 2. 2-offset P-Code program generation from a directed acyclic graph

in Figure 2.b contains only one enqueue operation for operand a, and one for operand b instead of two as in Figure 1. After operands have been enqueued the queue contents are: a, b. The "`add 0, 1`" instruction has two offset references, one for each of its operands. The first offset, `0`, indicates that the first operand should be taken from a zero distance from `QH`, or `QH`+0. That is, from `QH` itself. The second offset reference, `1`, indicates that the second operand should be taken from a distance one from `QH`, or `QH`+1. After the `add` instruction dequeues its first operand, a, the `QH` is updated and points to the next queue location, in this case, to the operand b. After the second operand for instruction `add` is dequeued, operand b, `QH` is updated and points to an empty location after operand b. The `add` instruction is performed with operands a, b and the result, $a + b$, is written back to `QT`, and `QT` is updated. At this stage, `QH` points to $a + b$, and `QT` to an empty location after $a + b$. Next, the "`sub -2, -1`" instruction takes its first operand from a distance of `-2` from `QH`, or `QH`-2. A negative offset indicates that the operand should be taken from the data in the operand queue that has been used before by other operations. For this case `QH`-2 points to the already used operand a. The second operand for `sub` is taken from `QH`-1, the used operand b. This instruction takes its two operands from a different location than `QH`, therefore, `QH` pointer is not updated and remains pointing to $a + b$ after the `sub` instruction is executed. At this point, the contents of the queue are: $a + b, a - b$. The following instructions are executed in similar fashion until the final expression (x) has been computed entirely.

Together with the 2-offset P-Code QCM we developed the compiler infrastructure and code generation algorithms [4]. Our queue compiler uses GCC 4.0.2 front-end. The custom back-end of our compiler takes GIMPLE trees [13,14] as input and generates assembly code for the PQP. GIMPLE intermediate representation is first expanded into an unrestricted trees named QTrees. The difference between GIMPLE and QTrees is that QTrees do not have limitation in the number operands an expression can hold [13]. QTrees are very similar to GENERIC trees [13] but QTrees are generated after all GCC's tree optimizations have been applied to GIMPLE form. QTrees are then passed to a leveling function that creates a *leveled DAG* (LDAG) [7]. A LDAG is a data structure chosen to be the input to the code generation algorithms for the queue compiler due to its expressiveness in the relations hip between operations, operands, and

the queue computation model. The code generation algorithm takes LDAGs to perform the offset calculations for the operations in the program. After all P-Code offset references are computed, the code generation algorithm produces a linear low level intermediate representation named QIR. QIR is a one operand intermediate representation suitable for generating final assembly code for the PQP. The last phase of the compiler takes QIR and generates the final assembly code for the PQP. Figure 3 shows the structure of the queue compiler.

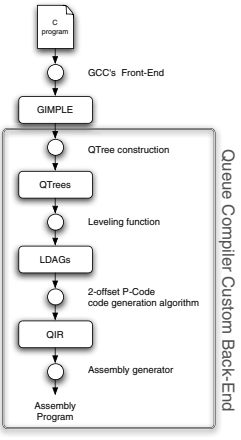

Fig. 3. Queue Compiler Infrastructure

We conducted experiments to measure the distribution of P-Code instructions required by some scientific test programs and the 2-offset P-Code QCM [4]. We found that the instructions that require the two operands to be dequeued from a different position than `QH` (2-offset instructions) appear rarely in programs. Instructions that require only one operand to be dequeued from an offset reference from `QH` (1-offset instructions), and instructions that take their operands directly from `QH` (0-offset instructions) are the most common instructions in programs. Based on these experiment results, we propose a 1-offset P-Code QCM. 1-offset P-Code QCM restricts all operations to have at most one offset reference. This limitation forces binary operations to take one of the operands always from `QH`, and the other operand can be taken from a location different from `QH`. Unary operations do not suffer from this limitation since they follow the rule to have at most one offset reference and the only operand can be taken with an offset reference.

It is impossible for the 1-offset P-Code QCM to execute an instruction where both operands are away from `QH`. To solve this problem, a special operation named `dup` instruction is added to the 1-offset P-Code's instruction set. The semantics of the `dup` instruction is to copy the value of an operand in the operand queue to `QT`. `dup` instruction takes one operand which is an offset reference. The offset reference is used to indicate the position with respect of `QH` of the operand

to be copied to QT. Figure 4.a shows the DAG for the expression $x = (a+b)/(a*a)$. 1-offset P-Code QCM cannot execute the multiplication node since both of its operands are away from QH by the time the operation is evaluated. Figure 4.b shows the DAG for the same expression augmented with dup instruction node. The effect of the dup node is to create a copy of operand a to the place where the dup node appears as shown with the dashed arrow. Figure 4.c lists the resulting 1-offset P-Code program. Operands a, b are enqueued. The offset for the dup instruction is zero indicating that the operand that has to be copied is placed in QH itself. After dup is executed, the contents of the operand queue are: $a, b, a\prime$. Where $a\prime$ is the copy of a produced by dup. QH points to the first operand, a, and QT points to an empty location after the operand $a\prime$. add 0 instruction takes both operands from QH. Then mul instruction takes its first operand from QHthat at this stage points to $a\prime$, and its second operand, a, from a distance of -2 with respect of QH. The rest of the program is executed similarly.

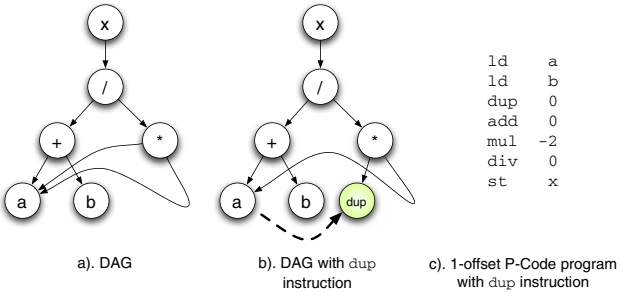

Fig. 4. 1-offset P-code program generation from a directed acyclic graph

1-offset P-Code QCM requires a new algorithm capable to determine the correct location of dup instructions. In the next section we will describe in detail the code generation algorithm for 1-offset P-Code.

3 Code Generation Algorithm

The algorithm works 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 a mark for the algorithm. The second stage takes the augmented LDAGs and assigns dup instructions to the ghost nodes when necessary. Finally, a breadth-first traversal of the LDAGs with dup nodes computes the offset references for all instructions and generates QIR as output. Figure 5 shows the two stages for the proposed code generation algorithm represented by the numbered circles. The elements inside the dashed area are the modified modules with respect of the original queue compiler [4] shown in Figure 3.

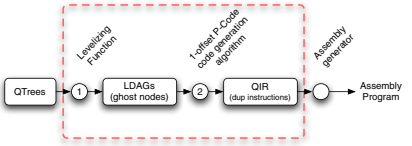

Fig. 5. 1-offset P-Code code generation compiler phases

3.1 Augmented LDAG Construction

QTrees are transformed into LDAGs by the leveling function. Algorithm 1 lists the leveling function that outputs an augmented LDAG with ghost nodes. 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.*

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 to the new level. This function creates a new ghost node in the new level, makes an edge from the parent node to the ghost node, and an edge from the ghost node to the element matched in the lookup table. These two functions build LDAGs augmented with ghost nodes that obey Definitions 1 and 2. Figure 6 illustrates the result of leveling the QTree for the expression $x = (a * a)/(-a + (b - a))$. Figure 6.b shows the resulting LDAG augmented with ghost nodes.

3.2 `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 nodes have been transformed or eliminated, the second pass performs a breadth-first traversal of the LDAG, and for every instruction the offset references with respect of `QH` are computed in the same way as in [4]. The output of the code generation algorithm is QIR for 1-offset P-Code.

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

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
```

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 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 7. The algorithm takes as input the LDAG with ghost nodes shown in Figure 6.b and performs the steps described in Algorithm 3 to finally obtain the LDAG with `dup` instructions as shown in Figure 7.a. The last step in the code generation is to perform a breadth-first 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

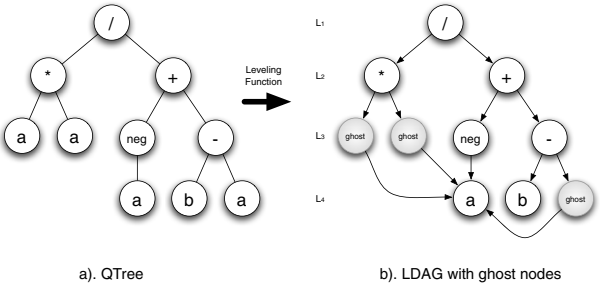

Fig. 6. Leveling of QTree into augmented LDAG for expression $x = \frac{a \cdot a}{-a+(b-a)}$

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 BreadthFirst (W_k) do
7:          p_qcm_compute_offsets (W_k, I_j)
8:       end for
9:    end for
10: end for
```

the offset calculation algorithm. The final 1-offset P-Code QIR for the expression $x = (a * a)/(-a + (b - a))$ is given in Figure 7.b.

3.3 Increase in Number of Instructions

A single `dup` instruction is inserted for every binary operation whose both operands are away from `QH` (β).

$$dup_i = \beta_i \tag{1}$$

The increase in number of instructions (Δ) for the 1-offset P-Code compared to 2-offset P-Code is given by the addition of `dup` instructions in the program.

$$\Delta = \sum_{i=1}^{n} (dup_i) \tag{2}$$

Thus, the total number of instructions for 1-offset P-Code ($Total$) is given by the total number of instructions for 2-offset P-Code (T_old) plus the inserted `dup` instructions (Δ):

$$Total = T\_old + \Delta \tag{3}$$

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
```

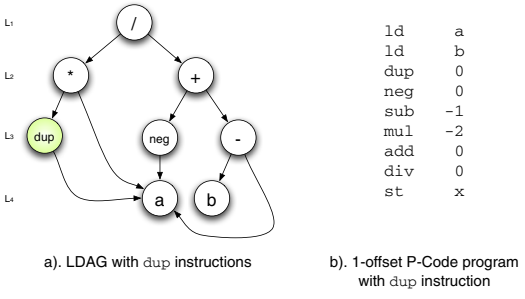

Fig. 7. 1-offset P-Code code generation from a LDAG

The length of the instruction set of the PQP is 2 byte [1]. The PQP has a special instruction, `covop`, which extends the value of the operand of the following instruction. The `covop` instructions are used to extend immediate values that are not representable with a single PQP 16-bit instruction. Thus, the code size for a 1-offset P-Code is obtained from the Equation 3 as:

$$Code\_Size = 2 * (Total + covop) \tag{4}$$

4 Experiments

We measured the code size for some benchmarks using our queue compiler for the PQP, and using GCC version 4.0.2 for MIPS32, MIPS16, 32-bit ARM, and Thumb architectures. We chose a set of recursive and iterative numerical computation benchmarks including the fast fourier transform, livermore loops, linpack, matrix multiplication, Rijndael encryption algorithm, etc. All programs were compiled without code reduction optimizations to estimate the real overhead of using a reduced instruction set architecture to reduce code size and compare it with our solution. Compiler based optimization techniques for improving code size on reduced instruction set architectures remains out of the scope of this paper.

Table 1. Code size comparison using GCC for different RISC architectures and the queue compiler for the PQP

Benchmark	ARM32	MIPS16	Thumb	PQP
quicksort.c	0.95	0.40	0.63	0.43
nqueens.c	0.85	0.53	0.78	0.52
md5.c	0.74	0.39	0.76	0.48
matrix.c	0.93	0.42	0.63	0.68
fft8g.c	1.02	0.92	0.60	0.54
livermore.c	1.16	0.74	0.80	0.58
vegas.c	1.11	0.89	0.73	0.51
whetstone.c	1.15	0.73	0.73	0.34
linpack.c	0.97	0.58	0.81	0.52
aes.c	0.83	0.51	0.67	0.38

For each benchmark we take the code size for MIPS32 as the baseline. Table 1 shows the normalized code size for all compiled benchmarks for ARM32 in column 2, MIPS16 in column 3, Thumb in column 4, and PQP in column 5 as compared to the baseline code size. GCC generates about the same code size for MIPS32 and ARM32 architectures with an average difference of 3%. For the MIPS16 and Thumb architectures, gcc reduces the code size for all benchmarks compared to the baseline MIPS32 code size. In average, for the MIPS16 it produces 42% smaller code size. For the Thumb it produces 24% smaller code size. We compiled the set of benchmarks for the PQP processor using our queue compiler. The presented results for the PQP take into account the extra `dup` instructions. Most of the programs require zero or one extra `dup` instruction except for `linkpack.c` which required six extra `dup` instructions. In average, our compiler technique produces 51% smaller code than the baseline code size. Our compiler is able to generate in average 16% smaller code than gcc for MIPS16, and 36% smaller code than gcc for Thumb architecture. For three of the benchmarks, `quicksort.c`, `md5.c`, and `matrix.c`, our compiler generated larger code compared to MIPS16. An inspection to the source of the programs revealed that these programs have a common characteristic of having functions with arguments passed by value. Our queue compiler handles these arguments sub-optimally as they are passed in memory. Therefore, additional instructions are required to copy the values to local temporary variables.

To compare the effect of breadth-first scheduling on parallelism, we compiled the benchmark programs and analyzed the compile-time parallelism exposed by the compiler. Our compiler at the moment does not include any optimization of any kind. For the RISC code we selected GCC-MIPS compiler and we enable all optimizations (-O3). Figure 8 shows the results of the experiment. Our compiler is able to extract about the same parallelism than fully optimized RISC code, in average, 1.07 times more. Our current and future work include the addition of optimization phases on the queue compiler infrastructure.

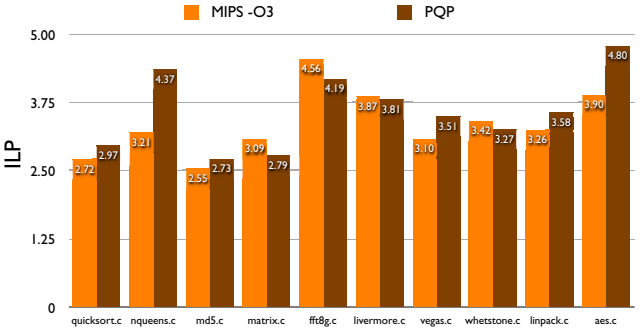

Fig. 8. Compile-time extracted instruction level parallelism

5 Discussion

The presented code generation algorithm efficiently reduces code size by using a 1-offset P-Code queue computation model processor. From the presented results, we observed that our technique reduces code size while keeping the instruction number increase very low. Techniques to reduce code size by using a dual instruction set [5,8] have the tradeoff that the increase in number of instructions, to relieve register pressure, leads to performance degradation of about 15%. In our technique, the increase in number of instructions comes from the insertion of `dup` instructions by the code generation algorithm. As presented in the experiments, and as found on our previous work [4], the additional instruction count is very low. The instructions of the PQP do not have register references making the programs free of false dependencies and, as a consequence, the need of spill code disappears. As future work, we will conduct experiments to measure the performance, in terms of execution time, of our technique. We shown here that the code for 1-offset P-Code QCM is about 50% denser than a full 32-bit RISC processor, and we expect it to be about the same performance since the increase of instructions is minimal. The benefits of 1-offset P-Code QCM are not just limited to code size. Since our technique introduces very small number of additional instructions, and the width of the instruction set is 16-bit, we expect less power consumption while accessing the memory to fetch instructions when compared to a fixed width 32-bit instruction set such as MIPS32.

6 Conclusion

In this paper we presented a code generation algorithm together with a 1-offset P-Code QCM for reducing code size. Our code generation algorithm has been integrated to the Queue compiler and the presented results have demonstrated the efficiency of the algorithm. The contributions of this paper can be summarized as follows: (1) the development of a new code generation algorithm for a 1-offset P-Code QCM using `dup` instructions and its integration to the queue compiler infrastructure; (2) the utilization of a 1-offset P-Code QCM to reduce

code size; (3) evidence that the queue-based computers are a practical alternative for systems demanding small code size and high performance. Our technique is able to generate in average 16% denser code than MIPS16, 26% denser code than Thumb, and 50% denser code than MIPS32 and ARM architectures. Without optimizations, the queue compiler is able to extract about the same parallelism than fully optimized code for a RISC machine.

References

1. Abderazek, B., Yoshinaga, T., Sowa, M.: High-Level Modeling and FPGA Prototyping of Produced Order Parallel Queue Processor Core. Journal of Supercomputing, 3–15 (2006)
2. Abderazek, B., Kawata, S., Sowa, M.: Design and Architecture for an Embedded 32-bit QueueCore. Journal of Embedded Computing 2(2), 191–205 (2006)
3. Bruno, R., Carla, V.: Data Flow on Queue machines. In: 12th Int. IEEE Symposium on computer Architecture, pp. 342–351 (1985)
4. Canedo, A.: Code Generation Algorithms for Consumed and Produced Order Queue Machines, University of Electro-Communications, Master Thesis (2006), http://www2.sowa.is.uec.ac.jp/~canedo/master_thesis.pdf
5. Goudge, L., Segars, S.: Thumb: Reducing the Cost of 32-bit RISC Performance in Portable and Consumer Applications. In: Proceedings of COMPCON 1996, pp. 176–181 (1996)
6. 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, p. 402 (2002)
7. Heath, L., Pemmaraju, S., Trenk, A.: Stack and Queue Layouts of Directed Acyclic Graphs. SIAM Journal of Computing 28(4), 1510–1539 (1999)
8. Kissel, K.: MIPS16: High-density MIPS for the embedded market, Technical report, Silicon Graphics MIPS Group (1997)
9. Krishnaswamy, A., Gupta, R.: Profile Guided Selection of ARM and Thumb Instructions. In: ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems, pp. 56–64 (2002)
10. Krishnaswamy, A., Gupta, R.: Enhancing the Performance of 16-bit Code Using Augmenting Instructions. In: Proceedings of the, SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems, 2003, pp. 254–264 (2003)
11. Krishnaswamy, A.: Microarchitecture and Compiler Techniques for Dual Width ISA Processors, University of Arizona, Ph.D Dissertation (2006), http://cs.arizona.edu/~gupta/Thesis/arvind.pdf
12. Kwon, Y., Ma, X., Jae Lee, H.: PARE: instruction set architecture for efficient code size reduction. Electronics Letters, 2098–2099 (1999)
13. Merrill, J.: GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In: Proceedings of GCC Developers Summit, pp. 171–180 (2003)
14. Novillo, D.: Design and Implementation of Tree SSA. In: Proceedings of GCC Developers Summit, pp. 119–130 (2004)
15. Sheayun, L., Jaejin, L., Min, S.: Code Generation for a Dual Instruction Processor Based on Selective Code Transformation. Lectures in Computer Science, pp. 33–48 (2003)
16. Sowa, M., Abderazek, B., Yoshinaga, T.: Parallel Queue Processor Architecture Based on Produced Order Computation Model. Journal of Supercomputing, 217–229 (2005)
17. SuperH RISC Engine, http://www.superh.com