

Optimizing Reaching Definitions Overhead in Queue Processors

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

Abstract

Queue computers are a viable option for embedded systems design. Queue computers feature a dense instruction set, high parallelism, low hardware complexity. In this paper we propose an optimization technique to reduce the overhead of long reaching definitions of variables in queue processors. Long reaching definitions have direct relationship with the queue register file utilization of the processor, and also to the bits in the instruction set reserved to reference operands. Using integer and embedded benchmarks, we demonstrate that our technique effectively reduces the length of reaching definitions up to 90%.

1. Introduction

Queue computing [14, 2, 12, 13] is a novel computer architecture alternative for embedded systems. Instructions of a queue processor implicitly reference their operands making instructions short in length and free of false dependencies. Queue programs are generated from a level-order scheduling that exposes maximum parallelism found in the problem itself [5, 12].

Generating a queue program from a directed acyclic graph (DAG) of a basic block generates complications for the fundamental queue computing [7, 12, 8]. In our previous work we have proposed a producer order queue computer [14, 2] which allows operand to be read from anywhere in the queue with an offset reference relative to the head of the queue. The length of the offset determines the lifetime, in queue words, of a reaching definition. Long offsets have a negative effect on the proper utilization of the available queue register file and demand more bits to be reserved in the instruction format. In this paper we propose an algorithm implemented in the queue compiler

infrastructure that effectively reduces the overhead of lengthy reaching definitions of variables. We demonstrate the effectiveness of our algorithm by compiling a set of well known benchmark programs and measuring the reduction of maximum offset length of memory store instructions.

2. Queue Computing

The Queue Computation Model (QCM) is the abstract definition of a computer that uses a first-in first-out queue data structure as the storage space to perform operations. Elements are written through the tail of the queue (QT) and elements are read through the head of the queue (QH). For correct evaluation of any expression the instruction sequence is given by a level-order traversal of the expressions' parse tree [12] as shown in Figure 1. Starting from the deepest level towards the root level, all nodes are visited from left to right.

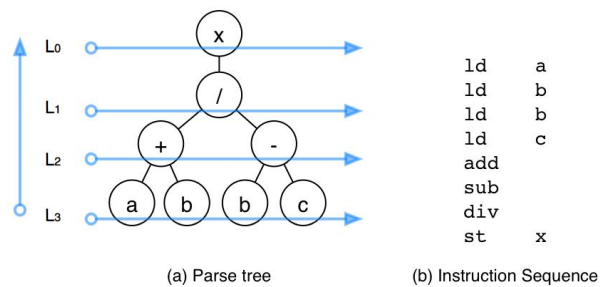


Figure 1. Level-order scheduling of parse tree

Although evaluating an expression from its directed acyclic graph is done in the same manner, it introduces some complications that require heuristic algorithms to level-planarize the data flow graph [13, 7, 8] or the enhancement of the instruction set of a queue processor. In our previous work [14, 2], we have proposed the Producer Order QCM. Such model allows the operands to be read from any location in the queue specified by

an offset reference relative to the QH position. The hardware calculates the physical location of the operand by adding the QH with the offset reference. The rule to write operands remains fixed at the QT. Figure 2(b) shows the generated producer order queue code to evaluate the expression's DAG. Notice that arithmetic operations (*add*, *sub*, *div*) have two operands representing the offset reference from where to read each operand. And Figure 2(c) shows the queue contents after executing the marked instructions. For example, the "add 0,1" instruction reads its first operand, a, from QH+0. And its second operand, b, from QH+1. Graphically, for the "sub -1, 0," its first operand b is on the left of QH since it was utilized by a previous instruction. Thus, the offset reference for its first operand is QH-1, and for its second operand c is QH+0.

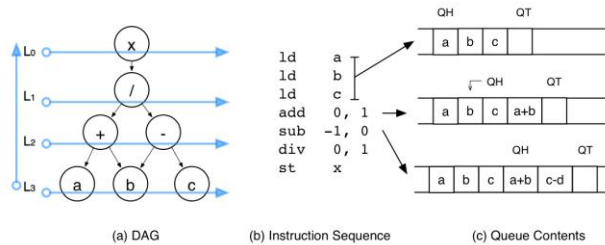


Figure 2. Level-order scheduling of DAG with producer order instructions

Our previous work also includes the development of the Queue Compiler Infrastructure [5]. The queue compiler is responsible of scheduling the program in level-order manner and computing the offset references for every instruction.

3. Reducing the Effects of Redundancy Elimination in Queue Processors

Common-subexpression elimination (CSE) is a classical compiler optimization [3, 11] that reduces execution time by removing redundant computations. The program is analyzed for occurrences of identical computations and replaces the second and later occurrences of them with uses of a temporary holding the common-subexpression.

The goal of common-subexpression elimination is to improve execution time by replacing redundant operations with uses of stored temporaries. Let the following expressions form a basic block:

Basic Block

```

S1: x = a + b
S2: y = 1

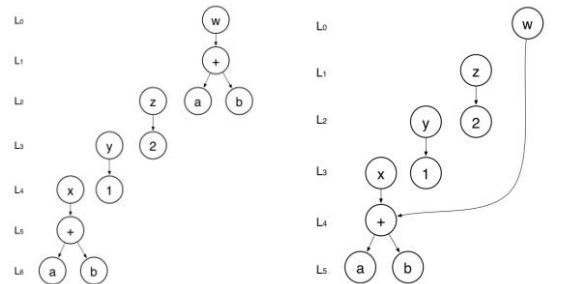
```

```

S3: z = 2
S3: w = a + b

```

CSE finds that the subexpression "a + b" is common for statements S1, S3 in the basic block. A temporary T is generated holding the common subexpression "a + b" and the common subexpression in S1, S3 is replaced by uses of T as shown in Figure 3(b). However, the last instruction that stores w requires an offset reference of -3 to read the value of the common subexpression. The lifetime of the subexpression "a+b" is three queue words since all data between the definition and the last use of a variable cannot be discarded. Therefore, the offset references, or lifetime of reaching definitions, have direct effect over the queue utilization of a queue processor. Since a queue processor potentially features a reduced bit-width instruction set, the cost of reserving bits for offset references in memory instructions is high. Therefore, optimizing the reaching definitions of variables by reducing the offset length contributes to an efficient utilization of the queue register file and allows the instruction set to remain short.



<pre> ld a ld b add 0, 1 st x ldi 1 st y ldi 2 st z ld a ld b add 0, 1 st w </pre>	<pre> ld a ld b add 0, 1 st x ldi 1 st y ldi 2 st z st w, -3 </pre>
--	---

(a) Non-Optimized Code (b) CSE optimized code

Figure 3. Redundancy elimination enlarges the reaching definitions of variables

The basic idea to reduce the offset length in a queue program is by moving the memory store instructions close to the definition of the variable. This can be done by moving the instructions to deeper levels as shown in Figure 3. The offset for the second definition of “x” in level L_0 is $-n$, after relocating the variable to level L_{n+3} the offset is reduced to -1 . Nevertheless, the relocation of instructions in the data flow graph must be carefully done in order to keep all data dependencies including the aliased variables as illustrated with the following two basic blocks:

Basic Block 1	Basic Block2
$S_1: x_1 = a + b$	$p_0 = \&x_0$
$S_2: i = 1$	$x_1 = a + b$
$S_3: x_2 = 1$	$*p_1 = c$
$S_4: c = 1$	$d = 1$
$S_5: x_3 = a + b$	$x_2 = a + b$

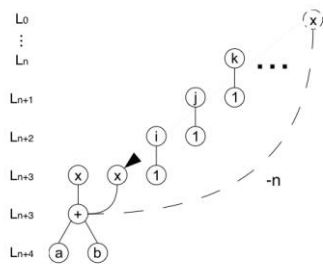


Figure 4. Reducing the length of reaching definitins by relocating the memory store into a deeper level.

The CSE algorithm finds the “a+b” subexpression common for statements S_1 , S_5 in basic block 1. Ideally the resulting long offset of S_5 could be reduced to -1 but there is a write-after-write data dependency between x_2 and x_3 . Therefore, the deepest level where x_3 can be relocated without altering the semantics of the program is to the level of x_2 as shown in Figure 5(a). The basic block 2 shows the case when a memory alias limits our optimization. In this case the CSE discovers the common subexpression for statements S_2 , S_5 , our optimization attempts to shorten the offset reference for x_2 by relocating to a deeper level. A more careful analysis of the basic block shows that there is a data dependency between p_1 and x_2 since p dereferences q . Figure 5(b) shows the place on which x_2 can be safely relocated.

Our proposed algorithm is listed in Algorithm 1. The input is a linear intermediate representation of the program. The algorithm scans the instruction list for memory store operations. If the offset of a store operation is not zero then the proper actions are taken.

If no dependency is found the store can be relocated next to the last use of the right hand side of the assignment. If a data dependency is found then the new position is determined by the dependency analysis. For either case, the new offset must be recalculated with the δ function [5].

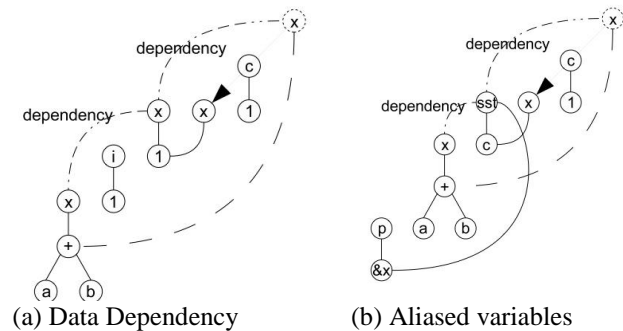


Figure 5. Instruction relocation with data dependency and alias analysis

Algorithm 1 shorten stores (ProgramList p)

```

Require: ProgramList  $p \neq \text{nil}$ 
1: repeat
2:   if isStore ( $p$ ) AND Offset ( $p$ ) > 0 then
3:     child  $\leftarrow p.\text{child}$ 
4:     newpos  $\leftarrow \text{getLastUse}(\text{child})$ 
5:     if !Dependency ( $p$ , &newpos)
       AND !AliasedDependency ( $p$ , &newpos)
       then
6:       Relocate (newpos,  $p$ )
7:       Offset ( $p$ )  $\leftarrow \delta(\text{newpos}, \text{child})$ 
8:     else
9:       RelocateSafe (newpos,  $p$ )
10:      Offset ( $p$ )  $\leftarrow \delta(\text{newpos}, \text{child})$ 
11:    end if
12:  end if
13: until EndOfProgram  $p$ 

```

4. Result

We implemented the proposed algorithm to shorten the offset references of memory store instructions in the Queue Compiler Infrastructure [5]. To evaluate the effectiveness of our technique we selected twelve benchmarks from SPEC CINT95 [6] and MediaBench [10] suites and measured the reduction of the maximum offset reference for store operations.

Figure 4 shows the reduction of offset reference length for the selected benchmarks. The left column

represents the maximum offset value for the original code optimized with common-subexpression elimination. The right column represents the maximum offset value for the code optimized with the transformation presented in this paper. From these results we can observe that our transformation significantly reduces the offset length of the memory store instructions. For example, the 147.vortex program the maximum offset is reduced from 230 to 13. For SPEC benchmarks, after the offset references are reduced, the maximum values are kept under 25. For all MediaBench programs, our optimization effectively reduces the offset length by more than 50%.

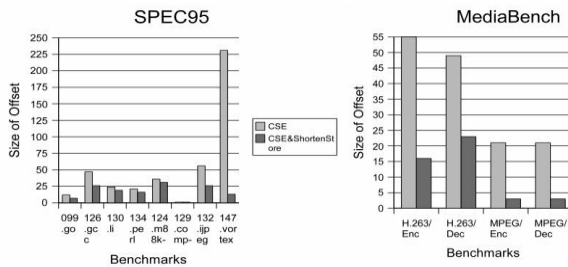


Figure 6. Maximum offset length reduction

To show the potential of the queue computing for the embedded domain, we compiled a set of embedded benchmark programs targeting the QueueCore processor [1]. The QueueCore is a 32-bit processor with 16-bit instructions. We compare the resulting code size with the code generated for MIPS I [9] and for Pentium [4]. The results are normalized to one using the MIPS I code size as the baseline. The density of the programs for the QueueCore processor are, in average, about 50% more compact than RISC code. And about 15% more compact than CISC code.

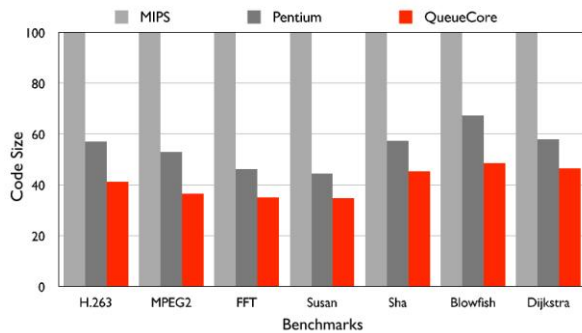


Figure 7. Normalized code size for a set of embedded applications

5. Conclusions

In this paper we presented a method to reduce the lifetime of reaching definitions in queue processors. Long reaching definitions have detrimental effect on the queue utilization and the instruction set requirements of a producer order queue processor. We developed and implemented the algorithm to reduce the offset references of memory store instructions and measured its effectiveness with a set of benchmark programs. Our experimental results show that the length of the offset references can be reduced up to 90% for some applications. Furthermore, we demonstrated the potential of the reduced bit-width instruction set of queue processors for the embedded world by comparing the code size against two conventional processors. In average, a queue processor achieves about 50% denser code than a conventional RISC machine, and about 15% denser code than a conventional CISC machine.

6. References

- [1] B. Abderazek, S. Kawata, and M. Sowa. Design and Architecture for an Embedded 32-bit QueueCore. *Journal of Embedded Computing*, 2(2):191–205, 2006.
- [2] B. Abderazek, T. Yoshinaga, and M. Sowa. High-Level Modeling and FPGA Prototyping of Produced Order Parallel Queue Processor Core. *Journal of Supercomputing*, 38(1):3–15, October 2006.
- [3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [4] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *Micro, IEEE*, 13(3):11–21, June 1993.
- [5] A. Canedo, B. Abderazek, and M. Sowa. A New Code Generation Algorithm for 2-offset Producer Order Queue Computation Model. *Journal of Computer Languages, Systems & Structures*, Accepted for publication 2007.
- [6] J. J. Dujmovic and I. Dujmovic. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):2–9, December 1998.
- [7] L. S. Heath and S. V. Pemmaraju. Stack and Queue Layouts of Directed Acyclic Graphs: Part I. *SIAM Journal on Computing*, 28(4):1510–1539, 1999.
- [8] L. S. Heath and A. L. Rosenberg. Laying Out Graphs using Queues. *SIAM Journal on Computing*, 21(5):927–958, September 1991.
- [9] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[10] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In 30th Annual International Symposium on Microarchitecture (Micro '97), page 330, 1997.

[11] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[12] B. Preiss and C. Hamacher. Data Flow on Queue Machines. In 12th Int. IEEE Symposium on computer Architecture, pages 342–351, 1985.

[13] H. Schmit, B. Levine, and B. Ylvisaker. Queue Machines: Hardware Compilation in Hardware. In 10th Annual IEEE Symposium on Field-Programmable Custom ComputingMachines, page 152, 2002.

[14] M. Sowa, B. Abderazek, and T. Yoshinaga. Parallel Queue Processor Architecture Based on Produced Order Computation Model. *Journal of Supercomputing*, 32(3):217–229, June 2005