Parallelizing Queue Compiler

by

Arquimedes Canedo

Submitted in Partial Fulfillment of the Requirements for the Degree of
Doctor of Engineering

Graduate School of Information Systems
University of Electro-Communications
Tokyo, Japan

September 2008

Parallelizing Queue Compiler

by

Arquimedes Canedo

Approved by Supervisory Committee:

Member: Prof. Masahiro Sowa
Prof. Masanori Idesawa
Prof. Toshihiko Kato
Asoc. Prof. Tsutomu Yoshinaga
Asoc. Prof. Tsuneyasu Komiya
Asoc. Prof. Masaaki Kondo

To Petty Monster

Abstract

キュープロセッサは、中間結果格納用にFIFOレジスタを持っており、キュー内のデータはキューの先頭から取り出され、データの入力は、キューの末尾にされる。この計算モデルは、従来のランダムアクセスを用いるレジスタマシンやスタックマシンとは異なるものであるため、現用のコンパイラと同様のコード生成方法を適用する事は不可能である。本論文では、初めて実用に近いコードをコンパイルすることができるようになったキュープロセッサ用並列コンパイラの設計及び開発手法について述べるものである。

本キュープロセッサ用並列コンパイラは、C言語で書かれた任意のプログラムを、キュープロセッサの機械語に変換する。キューコンパイラでは、全てのコード生成段階においてキュー計算モデルに沿ってコードを生成しなければならないので、従来のコンパイラとは異なる構造と手続きを必要とする。そのため内部構造は、従来のコンパイラとは大幅に異なる構造になっている。

本コンパイラではプログラムのコントロールフロー及びデータフローグラフを構成し、それをキューコード生成に適したQTreeに変換し、そこから幅優先トラバース用のLeveled DAGを生成する。次に、オフセットの計算を行うため、グラフの解析と変換を行い、最後に、ターゲットとなるキューマシンへのコード生成を幅優先トラバースによって行う。これらの手続きの多くは未知のものであったため、様々な問題を解決するために新しく効率的なアルゴリズムとデータ構造の開発を行った。

本コンパイラでは目的のプロセッサの要求にあうように、コードサイズ、並列性、性能、オフセット最適化、キューサイズといった目的を指定したコードを生成する方法を同時に開発した。

本コンパイラは成功裏に完成し、自身のコードを含め、任意のプログラムをコンパイルしキューオブジェクトコードを出力するコンパイラとなった。出力されたコードの質は、従来のコンパイラの命令数、命令種の分布と似たようなものになり、並列性、コードサイズに関しては従来の結果を上回るものとなった。

Abstract

Queue processors arrange high-speed registers in a first-in first-out queue. All read accesses are performed in the head of the queue and all writes at the tail. This computation model is substantially different from conventional random access register machines and stack machines. Traditional code generation methods cannot be applied to queue machines, therefore, this doctoral dissertation presents the design and development of a parallelizing queue compiler framework. The queue compiler translates any program written in C language into queue processor's machine code. The most important characteristic of the queue compiler is the integration of the queue computing principle in all stages of code generation, making it the first available true queue compiler. Novel and efficient algorithms and data structures have been developed to facilitate and solve all the problems related to compilation for the queue model. This work is the result of two years of research and development, and it provides efficient solutions to a broad range of problems on queue computing including the principles of compiler development, automatic code generation, offset calculation, scheduling, parallelization, optimization, data allocation, and constrained compilation. Internally, the queue compiler is completely different to any other existing compiler, this due to the special characteristics of queue computing. First, control flow and data flow graphs of the input program are generated. Then a set of custom analyses and transformations are performed to compute the offset reference values. Finally, the data flow graph is scheduled in a level-order manner to generate the final instruction sequence for the target queue machine. The compiler is also capable of producing code for a set of objective functions such as code size, high parallelism, high performance, offset reference control, and queue control. The queue compiler has been successfully completed and any program, including itself, can be compiled into queue object code. Quantitatively and qualitatively, the generated queue programs are similar to the ones generated by conventional register compilers in terms of number of instructions, parallelism, code size, and instruction distribution.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A compiler is a computer program that translates one computer language (source language) into another computer language (target language). Most of compilers translate a high-level programming language into machine language program called the object code. The goal of high-level programming languages is to hide the details of the microprocessor in a set of abstract, easy to use concepts to make complex programming simpler. Sophisticated programs such as operating systems and modern applications rely on high-level programming languages to facilitate their development, to reduce implementation time, and to avoid error-prone assembly programming. Compilers are a very important layer in the computer systems stack as they translate application code into machine code with comparable, or better, performance than hand-coded assembly.

Queue computing is a computation model that has not received much attention since the invention of microprocessors. A queue-based processor uses a first-in first-out queue to store and retrieve values for data processing. This model is analogous to conventional computers based on the concept of random access registers, or stack. In principle, register, stack, and queue architectures are implementations of sequential von Neumann architectures. To increase the performance of these kind of processors engineers have found ways to execute instructions in parallel. Microarchitectural techniques such as pipelining and multiple execution units have been developed to extract instruction-level parallelism in a single processor. However, the fundamental characteristics of each model complicate or facilitate the parallelization of instructions in different degrees. The stack model, for example, uses the top of the stack to perform reads and writes

and makes parallelization a challenging problem in the hardware and in the compiler. Compilers for superscalar and VLIW register must perform complex and sophisticated transformations to extract parallelism from the compiled programs. On the other hand, queue computing model allows parallelism to be easily extracted, yet no previous research has been conducted to understand the automatic compilation of programs for queue machines, and no compiler has been developed. Some previous attempts were made to use a conventional compiler and translate register-based code into queue code. These works made clear that conventional compilation techniques for register machines are not applicable to queue processors and, in order to obtain practical and competent code we needed to conduct original research to establish the principles of compilation for queue machines and to develop the first queue compiler.

Compiler development is a major engineering undertaking that requires significant time and efforts. Historically, the development time of an optimizing compiler takes longer than the development of the microarchitecture. Each computation model imposes distinct, yet very challenging problems to the compiler technology. For example, the recent trend in computer architecture design to achieve higher performance with identical VLSI technology is towards multi-core microprocessors. Although compiler technology is well understood for parallelizing programs for single processing elements, the introduction of many cores has brought major challenges for the compiler writers [75, 6, 53, 63, 83]. Compiling for queue computing paradigm requires a new and different approach. This dissertation presents the design and development of a compiler framework for queue machines. We tackle the fundamental problems of code generation and compiler development for queue machines with novel methods. We also introduce techniques to generate code for specific hardware constraints and program transformations dependent on the target queue machine.

At Sowa Laboratory [85, 86] we are researching and developing the first parallel queue-based processor. The queue compiler plays a very important role in the research, implementation, design, and testing of new ideas. Having the ability to generate queue code for actual applications allows the designers to identify areas of improvement on the microprocessor and the compiler itself. Once the actual queue processor has been completed, the compiler remains as a valuable development tool that facilitates the

deployment of operating systems, system libraries, assemblers, and any application in general.

1.1 Main Objectives

The primary goal of this dissertation is the research and development of the principles to compile high-level languages into machine code for queue-based processors. As a result of this investigation, we present a queue compiler framework able to translate and accommodate any program in an actual parallel queue processor. Second, we invented new techniques to generate code for specific objective functions such as code size, high parallelism, queue size control, offset reduction. Third, we aimed to the generation of code with quality comparable to that of production and research compilers in terms of number of instructions, code size, and compile-time instruction level parallelism. Fourth, to deliver a complete framework that contributes and promotes the study and progress of queue computing.

1.2 Related Work to Queue Machines

The concept of a queue machine was first proposed by Feller and Ercegovac in [24], they present and highlight some of the properties of queue machines for parallel processing such as fast instruction issue, fault-tolerance, and simple interconnection properties. In [71], Preiss discusses the fundamental techniques to generate programs for a queue machine by traversing the parse trees in level-order manner, and the complications [35] of generating code from a directed acyclic graph (DAG). Okamoto [67] proposed the actual design of a superscalar queue machine able to execute instructions in parallel. In [76], Schmit et al. proposed a queue machine as the execution engine for reconfigurable hardware with high parallelism properties and simple hardware complexity. In [87], Sowa et al. established a method to execute arbitrarily complex DAGs in a single queue without modifications to the original graph. Instead, this *producer order* model relies on allowing instructions to randomly access any operand in the queue with offset references indicating the place, relative to the head of the queue, from where to read the operands. Based on this model

the QueueCore parallel processor was developed [1] together with a custom compiler [13]. QueueCore programs expose similar parallelism and are smaller than embedded RISC processors. With the same idea of high flexibility of queue computing, research has been conducted on *consumer order* [86] and *multidimensional* [29] queue computing. Where the consumer order model only gives flexibility when writing operands anywhere in the queue. And the multidimensional queue computing introduces the concept of multiple queues to reuse data as much as possible and thus avoiding long latency memory accesses.

Another class of computers such as the Astronautics ZS-1 [84] and the WM machine [96] decouple memory accesses from execution using visible queues for communication. Since execution runs asynchronously from memory accesses and instructions can read/write operands from/to the queues, streamlined processing can be effectively exploited. Inspired by the decoupled architectures, a VLIW processor using a queue register file was proposed to boost the execution of software pipelined loops [25]. The idea is extending the architected register file by connecting registers to queues. Thus, queues can hold more values than the architected registers. Every write access to register connected to a queue places the element at the tail of the connected queue, and every read access to a connected register dequeues on element. A modified register allocator emits rq_connect instructions to map specific registers to specific queues, allowing values of different iterations to reside in the queues. This technique has also been demonstrated to be effective in solving register pressure problem in conventional superscalar machines [89].

Register machines using queues for extending the architected register file provided compiler support [25, 89]. Their approach is to use a traditional register compiler and modifying the conventional register allocator phase [16] by making visible a larger number of registers. The modified register allocator attempts to place the most important variables in the architected registers, while storing the less important variables in the queues or memory. After register allocator is complete, the compiler is required to emit the *connect* instructions to map registers to queues. This process emulates the register mapping table at compile time [44] instead of run time as the register renaming process [49]. Clearly, the compiler treats queues as additional low priority registers as the main computation is done by explicit referencing the architected registers. This simple approach is able to employ the queues as additional registers, however, it does not

contribute to the design of code generation techniques for formal queue machines.

In [35], it was established that some directed acyclic graphs need more than one queue to be laid out, however, level-planar graphs can be always laid out in a single queue. Based on this property, Schmit et al. [76] proposed a heuristic to convert any non level-planar graph into level-planar. This conversion is achieved through inserting special instructions to duplicate and exchange position of data. However, no compiler with such functionality was developed.

Some efforts have been done to improve the performance of conventional high-level languages by the utilization of the queue computation model. In [58], the idea of using a queue machine for the implementation of a parallel functional language is discussed. An initiative to develop a queue-based Java Virtual Machine for parallelizing the execution of Java bytecode [55] was proposed in [80, 81]. This project developed a Java compiler capable of rearranging the abstract syntax trees of programs and extracting parallelism for direct execution on the queue-based Java virtual machine. This compiler generates bytecodes from abstract syntax trees and their method does not handle code generation from directed acyclic graphs. Based on this work, [42] presents an improved compiler able to calculate the correct location of operands relative to the head of the queue and thus generating offset references for instructions. The latter Queue-Java compiler is able to handle simple directed acyclic graphs and generate correct bytecode.

From 2000 to 2005, several attempts were made to develop a queue compiler based on a retargetable register compiler [68, 27, 37, 8, 11]. The idea was to map three-address register code into queue code. First, the data flow graph of the program had to be reconstructed and re-scheduled for the queue computation model. During this process register references had to be eliminated since queue machines do not have explicit operands. This mapping technique led to very complicated mapping algorithms and very poor output programs. Furthermore, these compilers were never completed and could only compile toy programs of less than a hundred lines of code with very simple operations and control flow. Such approach and results made clear that the fundamental problem lies on the compilation strategy: to be able to generate clean, correct, and complete queue code, the compiler must be blind to the concept of registers. Since all back-ends of conventional compilers are based on the concept of registers, the need of a compiler crafted specifically

for the queue computation model became clear.

1.3 Organization of this Dissertation

This dissertation is organized as follows: Chapter 2 presents the principles of queue computing. Chapter 3 establishes the concepts, abstractions, and algorithms to develop a queue compiler. Chapter 4 provides novel techniques to cope with the code generation for constrained hardware. Chapter 5 presents a technique to improve data allocation in the queue while reducing memory traffic. Chapter 6 presents the evaluation of queue computing by means of compiling a set of benchmark programs with the presented queue compiler. Chapter 7 discusses several aspects on the development of a queue compiler and concludes.

Chapter 2

Queue Computation Model

A queue-based computer employs a first-in first-out (FIFO) queue to evaluate expressions. To avoid high-latency memory accesses, the queue is implemented with high-speed registers arranged and accessed in special manner. The physical implementation of the queue is called the Queue Register File. Reading operation from the queue is done always through the head of the queue, and writing operation is done always through the tail of the queue. Therefore, the hardware must provide two pointers to track the head and tail of the queue. Such pointers are implemented as special registers, `QH` and `QT` to track the head and the tail positions of the queue. The Queue Computation Model (QCM) is the set of rules and conventions that allow programs to be executed in a queue processor. In this Chapter, the hardware and software aspects of the QCM are discussed.

2.1 Differences with Conventional Computation Models

The goal of any computer is to perform correct operations on data items. The underlying principles of a computer vary from one to another. Conventional register machines, such as RISC[1], employ a set of random access registers to hold data and perform operations. A typical RISC instruction consists of a quadruple that specifies an operator, and three operands: a destination operand, and two source operands. For example, the instruction

[1]Reduced Instruction Set Computer

"`add R1, R2, R3`" adds (operator) the contents of registers R3 and R2 (source operands), and places the result of the addition in register R1 (destination operand). An important characteristic of random access register machines is that all operands must be explicitly referenced by name (register number).

A stack machine employs a set of registers organized as a last-in first-out (LIFO) stack to perform computations. All accesses, read and write, to the stack are done through the *top of the stack*, or TOS. Since all accesses to the stack are fixed at TOS, operands can be omitted from the instruction allowing 0-operand instructions. For example, the stack instruction "`sub`" reads (pop) two elements from TOS, performs the subtraction, and writes (push) the result to TOS. Compared to RISC-based 3-operand instructions, 0-operand instructions are smaller and require less memory bandwidth for fetching, less hardware to decode, and improved instruction cache performance. Generally, stack-based computers require less hardware and have better power consumption characteristics than register machines. However, the performance of stack machines is fundamentally limited by the bottleneck created at the TOS [91, 73, 79].

The QCM inherits all characteristics of 0-operand computers and, contrary to the stack model, it is a fundamentally parallel computation model. Queue machines can easily and effectively exploit parallelism as two different locations are used for reading and writing. One important advantage of queue computers over conventional register computers is that queue programs are free of false dependencies and the hardware and power consumption can be greatly improved with the absence of register renaming techniques [49]. Table 2.1 summarizes the differences between the queue computation model and conventional random access register and stack models.

2.2 Code Generation for Queue Machines

Queue processors employ a different arrangement of registers with well established rules for accessing its elements and, therefore, conventional code generation techniques for register machines cannot be applied for the queue model. As the queue obeys the rule that the first inserted element is the element that will be read first, the generation of correct queue programs is guaranteed by a *level-order* traversal of the expression trees [71].

Table 2.1: Characteristics of queue computation model compared to conventional register and stack models

	Queue	**Register**	**Stack**
Principle	FIFO	Random Access	LIFO
Operands	0	3	0
Instruction Encoding	small	large	small
False Dependencies	no	yes	no
Hardware Complexity	low	high	low
Power Consumption	low	high	low
Performance	high	high	low

Expression trees have the property that every produced element has only one consumer and therefore data cannot be used more than once. To reduce the size of the trees a directed acyclic graph (DAG) is constructed where elements have no limitation in the number of consumers, thus allowing to reuse data. Despite this property, queue programs are still generated by a level-order traversal of the expression DAGs [71]. However, a data ordering problem between producers and consumers is introduced and it should be solved by the hardware and the software.

2.2.1 Consumers-Producers Data Ordering Problem

To illustrate the queue code generation from DAGs and the data ordering problem consider the expression in Figure 2.1. The Figure 2.1(a) shows the DAG of expression "$x = (a + b)/(a - c)$". The horizontal dotted lines show the level-order traversal starting at the deepest level (L_0) and finishing at the root level (L_3). For each level (L_0, L_1, L_2, L_3) all nodes are visited from left to right. The level-order traversal generates the instruction sequence to evaluate the expression in a queue machine. Figure 2.1(b) shows the pseudo-instructions of the queue program. Figure 2.1(c) shows the contents of the queue during the execution of the program. For every stage, the head of the queue is represented by the left-most element and the tail of the queue by the right-most end of the queue.

Firstly, operands a, b, c are loaded into the queue. The next instruction, "*add*", is a binary operation that takes its two operands a, b from the head of the queue and writes the addition $a + b$ into the tail. Notice that after the addition is performed the elements "a" and "b" were consumed and no longer present in the queue. The problem appears when any further instruction has any of these consumed data as operand. For example, the following subtraction "*sub*" requires operands "b" and "c" but the head of the queue contains operand "c" and the contiguous element is the result of the addition "a+b". If the subtraction is executed the result would be "$c + (a - b)$" as shown in the shaded element in the Figure rather than "$a - c$". This situation leads to incorrect evaluation of the expression.

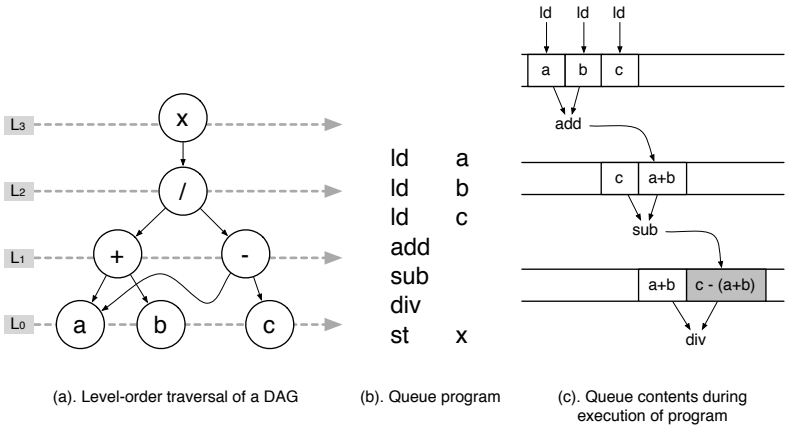


Figure 2.1: Code generation for queue machines consist of: (a) traversing the DAG in level-order manner, (b) obtain the instruction sequence of the queue program, and (c) executing the program in the queue.

2.3 Queue Computation Model Taxonomy

We identify three approaches that can be taken in the hardware and/or software to solve the problem of consumers-producers. These models constitute a variation of the queue computation model. In the first solution, called *producer-consumer order model*, the software is in charge of modifying the DAGs and the hardware remains intact as a pure queue machine. The other two solutions, called *consumer order model* and *producer order*

model, add hardware support for the execution of DAGs balancing the complexity between the software and the hardware.

2.3.1 Producer-Consumer Order Queue Computation Model (PC-QCM)

For this model, the rules of enqueueing and dequeueing remain intact. Reading is strictly performed at `QH`, and writing at `QT` as shown in Figure 2.2. During code generation, the compiler analyzes the data flow DAGs and determines whether any node incurs in a consumers-producers data ordering problem. This can be identified by dependency edges that intersect and by edges spanning across more than one level. In [76], a heuristic algorithm is proposed where the original DAG is verified to be *level-planar* [35]. A level-planar DAG has the properties that all edges span only one level and there are no crossing arcs, therefore can be executed as is in a pure queue machine. Level-planarity can be tested in linear time [40]. For non level-planar DAGs, the algorithm selectively places `swap` instructions to exchange position of data and remove crossing arcs, and `fill`[2] instructions to fill empty slots of edges spanning multiple levels. In this way any DAG can be converted into a level-planar DAG for direct execution in a pure queue machine.

Figure 2.3(a) shows a sample expression whose straightforward representation in a DAG is non level-planar. In Figure 2.3(b), a crossing arc between (>> → `16`) and (− → `sum`) exists. Furthermore, the edge spanning from (`*` → `4`) has two empty levels. Figure 2.3(c) shows the transformed DAG into a level-planar DAG using `fill` (squared nodes) and `swap` instructions (diamond shaped nodes). Notice that the `fill` nodes shaded in gray produce two data. The `swap` node exchanges the position of a copy of `sum` with `16`. And the chain of `fill` instructions in the right-most column converts the empty instruction slots into data. Executing the level-planar graph in the PC-QCM model is straightforward as no consumers-producers data ordering problem appears. However, the overhead of level-planarazing a DAG is the insertion of extra instructions and extra levels. For this example, the original DAG has 8 nodes and 5 levels, and the level-planar graph 16 nodes and 7 levels.

[2]The original name of `fill` instruction in [76] is `dup`. The name has been changed not to confuse the reader the duplicate (`dup`) instruction used in this thesis.

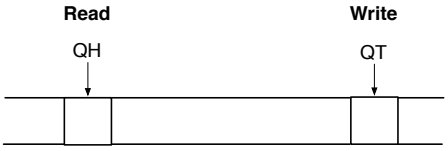


Figure 2.2: Producer-consumer model (PC-QCM) strictly uses `QH` and `QT` for reading and writing.

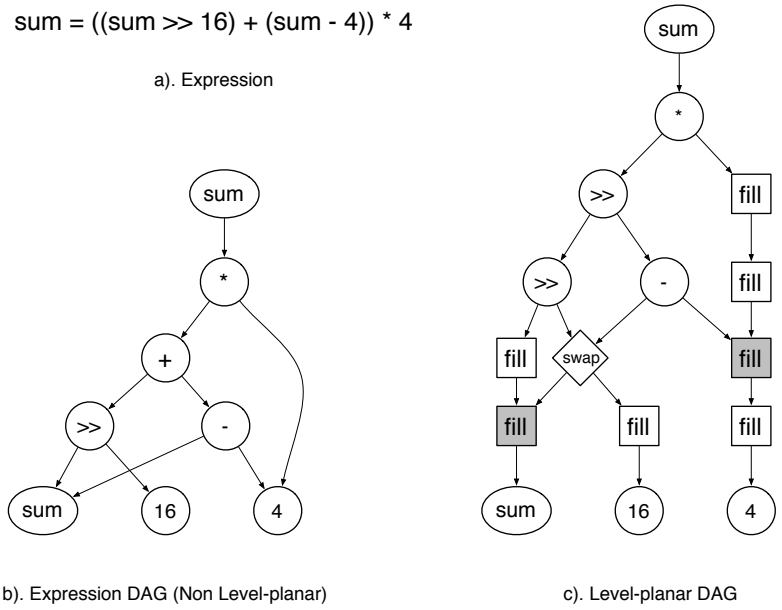


Figure 2.3: Transforming a DAG into a Level-planar DAG

2.3.2 Consumer Order Queue Computation Model (C-QCM)

The Consumer order model, or C-QCM, was named after the strict rule that reading, or *consuming*, elements is done always through `QH`. The rule for writing is broken to give flexibility and solve the consumers-producers data ordering problem. The instructions in a C-QCM program, apart from producing always a datum in `QT`, have the ability to specify a location relative to the `QT` where to produce a copy of the produced data as shown in Figure 2.4. The hardware must be modified to enable the execution of such special cases of instructions, and the compiler must calculate the correct locations where to place copies of data. This is a hardware/software approach to solve the consumers-producers data ordering problem.

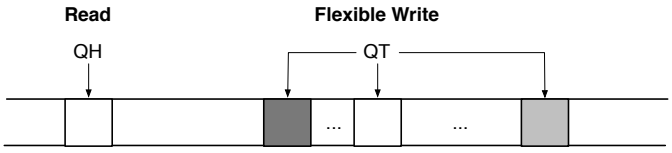


Figure 2.4: Consumer model (C-QCM) gives flexibility in writing but the reading location remains fixed at `QH`.

Figure 2.5(a) shows the C-QCM program generated to evaluate the expression "`sum = ((sum >> 16) + (sum - 4)) * 4`". The program is obtained by a level-order traversal of the expression's DAG. Figure 2.5(b) shows the queue contents during the execution of the program. The first instruction loads `sum` into the queue and places a copy two locations after the current `QT` location. The second operand in the first instruction, 2, is shown by the dashed line in the queue contents figure. After the second instruction loads `16`, the third instruction loads `4`, and produces a copy four locations away to `QH` current position. Notice that the hardware must be capable of identifying the data that was produced ahead. The elements marked with a question mark, ?, represent queue registers that can be used for computation. However, the rightmost 4, should be not overwritten. The following instructions are executed and the ?-nodes are filled with computations to evaluate the expression. For clarity of tracking the progress of computations, the consumed data is shown by the light gray queue elements. This hardware/sofware mechanism solves the consumers-producers problem keeping instruction count same as the original problem and adding extra complexity to the instructions and the hardware.

The Indexed Queue Machine proposed in [71] follows this principle. The only difference is that the location to produce the extra copy is relative to `QH` rather than `QT` as in our method. Although same principle, we believe our method is superior than the Indexed Queue Machine since the offset reference used to specify the location is generally shorter when relative to the `QT`. Let the number of elements between `QH` and `QT` is N, and a C-QCM instruction needs to produce a copy M locations away from `QT`. Then, for our method the offset reference relative to `QT`is M, and for the Indexed Queue Machine is $N + M$.

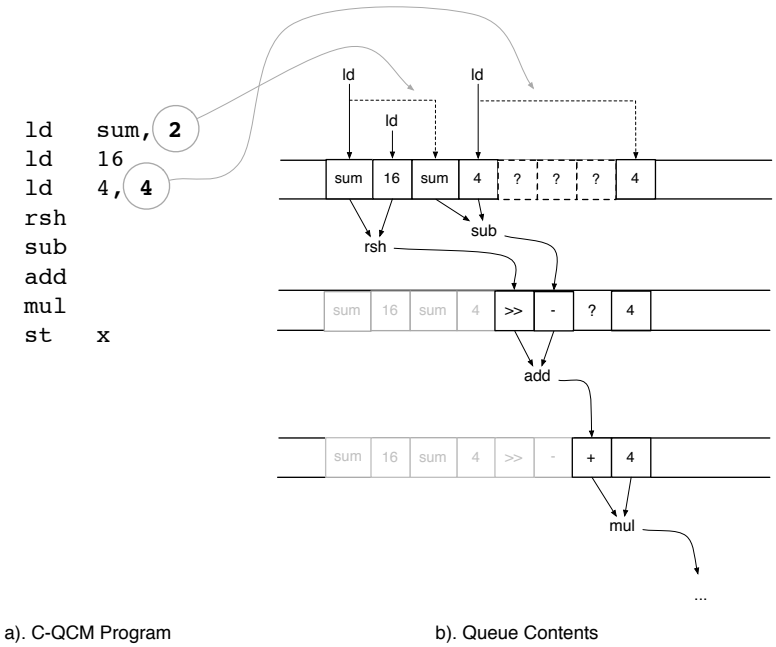


Figure 2.5: Consumer model (C-QCM) execution

2.3.3 Producer Order Queue Computation Model (P-QCM)

The Producer Order model, P-QCM, is a hardware/software approach to solve the consumers-producers data ordering problem. The unique characteristic is that the rule for writing, *producing*, data remains fixed at `QT` and the rule for reading data has flexibility. Figure 2.6 shows the rationale behind the P-QCM model.

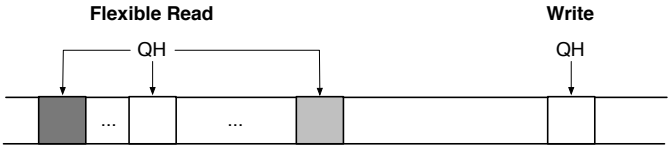


Figure 2.6: Producer order model (P-QCM) gives flexibility in reading data but the writing location remains fixed at `QT`.

Figure 2.7(a) shows the P-QCM program generated for the same sample expression "`sum = ((sum >> 16) + (sum - 4)) * 4`". The execution of the program and its queue contents are displayed in Figure 2.7. Execution flows normally up to the fourth instruction. The subtraction operation needs operands `sum` and `4`. Notice that `sum` was consumed by

the `rsh` instruction in the first queue state. Therefore, the subtraction instruction must specify a location relative to `QH` to read its operand, for this case −2. The negative value denotes the access of a value that was already consumed by a previous operation. Graphically, it represents the access to a value on the left of `QH`. The addition consumes its two operands directly from `QH`, and the multiplication finds the same problem. It requires the access to operand `4` which is three queue registers away from `QH`, hence the offset reference is −3.

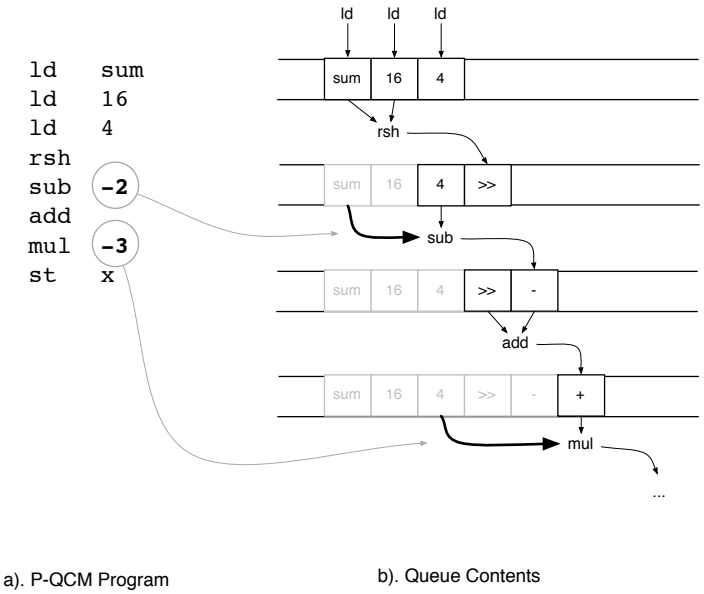


Figure 2.7: Producer Order model (P-QCM) execution

We believe the P-QCM offers the best alternative over the other two queue computation models. It provides great flexibility for executing scientific and conventional codes. The C-QCM programs have the tendency to be longer than P-QCM programs. Table 2.2 shows the fundamental characteristics of C-QCM and P-QCM for the Livermore loops program. The C-QCM needs 22% more instructions than the P-QCM program. The available parallelism at the instruction level is higher for the C-QCM but the execution time is the same for both models. This means that the C-QCM requires about 20% more hardware to execute the program in the same time as the P-QCM. The maximum queue utilization represents the amount of queue registers needed to execute the most demanding part of the program. For this benchmark program, the C-QCM needs 60%

Table 2.2: C-QCM and P-QCM program characteristics for Livermore loops

	C-QCM	**P-QCM**
Generated Instructions	4094	3350
Instruction Level Parallelism	3.82	3.13
Execution time	1542	1542
Max. Queue Utilization	227	94

more registers than the P-QCM.

To illustrate the advantages of P-QCM over the PC-QCM consider the example in Figure 2.8(a), it shows the directed acyclic graph for a 4-point Fourier Transform. Using the method proposed in [76] the original DAG is transformed into a level-planar graph. Basically, edges spanning more than one level are eliminated by a *fill operator*, and crossing arcs are eliminated by a *swap operator*. By employing these two special instructions [76] the original DAG is transformed into a level-planar graph as shown in Figure 2.8(b). Duplicate operators are represented by the squared nodes, and swap operators are represented by the diamonds. As it is reported by its authors, this method increases the number of instructions and the depth of the graph, for the given example from 18 to 49 instructions and from 5 to 12 levels. Figure 2.8(c) shows the producer order graph. White circles represent instructions that read operands directly from `QH` gray circles represent instructions that need one operand to be taken by an offset reference, and black circles represent instructions that take all its operands using the flexible reading rule. Compared to the PC-QCM model, the P-QCM model does not increase the number of instructions nor the depth of the graph.

Table 2.3 compares the code size and depth of data flow graphs (DFG) for the PC-QCM model and the P-QCM model for a set of benchmark programs. The P-QCM programs are from 55% to 80% smaller than PC-QCM programs. From the fact that all instructions belonging to the same level can be potentially executed in parallel, shallow data flow graphs are preferred over deep ones. The table shows that P-QCM data flow graphs are about 1.46 to 5.85 times shallower than the PC-QCM. For the aforementioned reasons, the P-QCM model is preferred over the other two queue computation models and

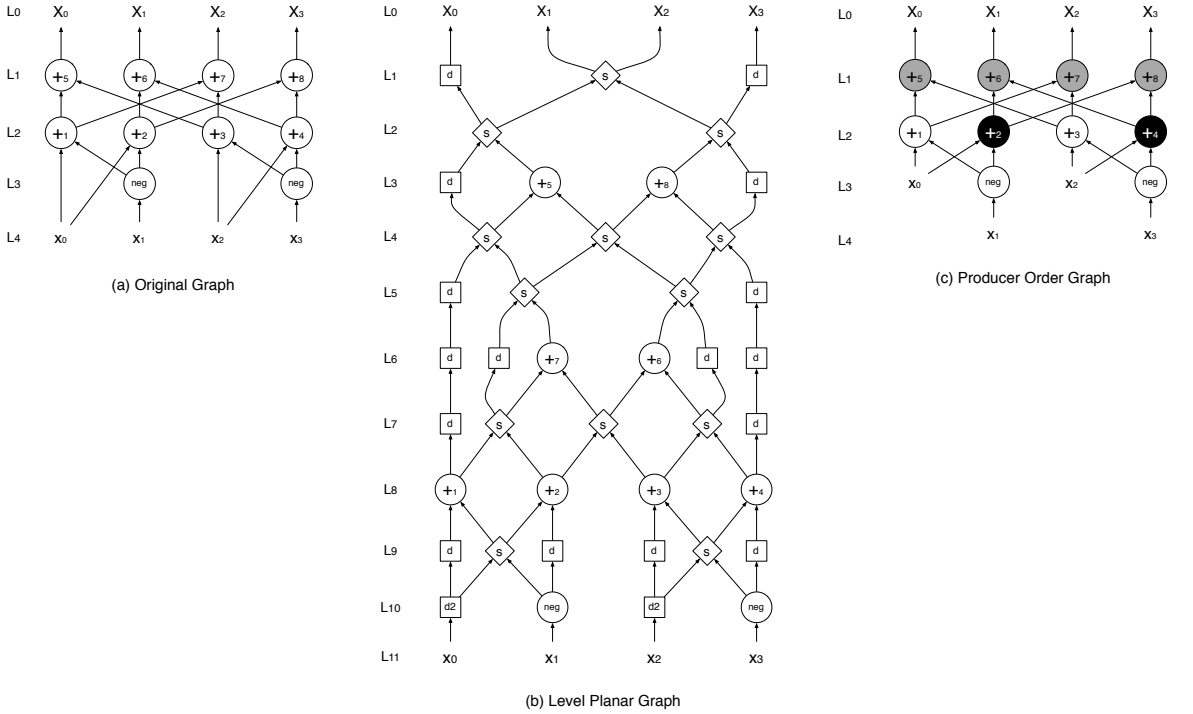


Figure 2.8: Comparison between (a) original directed acyclic graph, (b) PC-QCM level-planar model, (c) P-QCM model

Table 2.3: Code Size and Depth of Application Graphs Comparison

	PC-QCM Graph		**P-QCM Graph**	
Application	Code Size	Depth	Code Size	Depth
dct1	537	49	244	18
fft8_iterative	909	41	176	7
haar16	918	17	248	6
rc6	330	42	148	23
idea	1462	235	606	160
popcount	229	24	62	5

all efforts of this thesis were made to develop a P-QCM compiler framework.

Chapter 3

Producer Order Queue Compiler Framework

Among all queue computation flavors, the producer-order model offers the highest flexibility for the software and the hardware. Programs generated for the producer-order model are characterized by being small and having two offset references that specify the location with respect of the head of the queue from where to read the source operands. This chapter presents the main contribution of this dissertation which is the establishment of the techniques to develop a queue compiler for the producer order queue computation model. The structure of the queue compiler is different from traditional compilers since it integrates the concept of queue computing to all phases of code generation. We introduce a novel data structure that facilitates the compilation process and allows the calculation of offset references and level-order scheduling. The functionality of the actual queue compiler implementation is presented and we analyze the complexity of the algorithms in terms of lines of lines of code and compile time against conventional compilers for a set of standard applications.

3.1 Target Queue-based Architecture

To avoid compilation complexities introduced by specific hardware implementations, the queue compiler generates code for a generic producer-order instruction set architecture inspired by the Parallel Queue Processor (PQP) [87]. The target instruction set, or P-

Table 3.1: Generic queue instructions

Class	**Instructions**
Arithmetic & Logic	`add, sub, div, mod, mul, neg, rsh, lsh, ior, xor, and, not rrot, lrot, abs`
Memory	`ld, st, ldi, lea, sld, sst`
Comparison	`ceq, cne, clt, cle, cgt, cge`
Ctl. Flow	`bt, bf, j, jal, ret`
Conversion	`conv`
Special	`copyp, dup, rot`
Queue	`moveqh, moveqt`

Code, allows two offset references to be encoded without length restrictions. Hardware details such as type and number of functional units, memory hierarchy, queue register file size, instruction length, are parametrized in the compiler to allow flexibility and to gain the ability to target multiple implementations of queue processors. P-Code instructions are classified in seven classes as shown in Table 3.1: Arithmetic & Logic, Memory, Comparison, Control Flow, Special, and Queue. To differentiate between integer and floating point operations, each instruction defines its data type and sign. Table 3.2 shows the equivalence between C language data types and P-Code data types. For full description of P-Code instructions refer to Appendix A.

P-Code's instruction set format is as follows. Instructions using offset references such as arithmetic and logic, comparison and special groups, and `sst` and `sld` instructions the encoding is as follows:

opcode	data type	offset1	offset2

For memory instructions that do not use offset references the encoding is as follows:

opcode	data type	memory address

Control flow instructions do not have a data type and the only operand is the target label symbol representing the target of jumps or function names.

Table 3.2: Data type and sign information

P-Code type		C-Type
Description	**Data Type**	
Integer Byte Signed	ibs	signed char
Integer Byte Unsigned	ibu	unsigned char
Integer Half-word Signed	ihs	signed short
Integer Half-word Unsigned	ihu	unsigned short
Integer Word Signed	iws	signed {int, long}
Integer Word Unsigned	iwu	unsigned {int, long}
Integer Long Signed	ils	signed long long
Integer Long Unsigned	ilu	unsigned long long
Floating Point Single Precision	fps	float
Double Precision Floating Point	fpd	double
Long Double Precision Floating Point	fpD	long double

opcode	target label

3.1.1 Offset Referenced Instructions Classification

P-Code instructions are classified in three categories according the number of operands read by an offset reference. We say that "an operand is read by an offset reference" when it is not directly accessed by the current `QH`, in other words, `QH`+N where N$\leq$0. 2-offset instructions read both operands by offset references, 1-offset instructions read only one operand by an offset reference, and 0-offset instructions do not use offset references. Thus, the binary instructions can be 2-offset, 1-offset, or 0-offset. Unary instructions can be 1-offset, and 0-offset. By notation, the two offsets values in binary instructions are explicitly given by a pair (N, M). Operand location is given by `QH`+N and `QH`+M for the first and second operands, respectively. We use the following convention to specify 0-operand binary instructions:

Definition 3.1. The pair of offset references (0, 1) defines a 0-operand binary instruction,

Table 3.3: Examples of producer-order instructions

Type	Binary	Unary
0-offset	`mul 0, 1`	`not 0`

1-offset	`add -3, 0`
	`sub 0, -2`
	`div 1, 0`
		`neg -3`

2-offset	`rsh -3, -1`	N/A

e.g. "`div 0, 1`".

Table 3.3 shows examples of binary and unary instructions according their Producer Order classification.

3.2 Compiler Framework Design and Implementation

The queue compiler infrastructure consists of six phases, including the front-end. Figure 3.1 shows the block diagram of the queue compiler. The front-end parses C files into language independent abstract syntax trees (AST) which are a high level intermediate representation (HIR). These language independent ASTs facilitate the addition of any other high-level language parsers and permit the reutilization of the back-end of the compiler. The queue compiler framework consists of the remaining five phases. *QTree generation* phase lowers the high level constructs in ASTs into a generic low level sequence of queue instructions (P-Code instruction set). The resulting representation is a tree-like structure called QTrees and its main purpose is representing all program in queue-like instructions. QTrees completely ignore the concept of registers as the queue machine has none, instead, they represent a pure data flow graph model of the basic blocks. The second phase is the *Queue Code Generator* which takes QTrees as input where redundancies are eliminated and all nodes are assigned to a hierarchical structure formed by *levels* according data dependencies among instructions. This hierarchical structure is called a *leveled*

directed acyclic graph (LDAG) on which the compiler performs the offset calculation and tracking of the head of the queue. This structure facilitates also the scheduling of the program as all nodes are organized into levels. Although direct translation from ASTs to LDAGs is possible, we introduce the QTrees to separate instruction selection from redundancy elimination and to simplify the algorithms. The third phase, *offset calculation*, consumes the LDAGs and for every instruction it computes the offset reference values to reach its operands. Offset references are annotated in the LDAGs. The fourth phase, *instruction scheduling* traverses the LDAGs and produces a level-order scheduling of instructions that complies with the queue computing principle. As a result, this phase generates a low level intermediate representation called QIR. QIR is a linear representation designed to facilitate target dependent optimizations. The QIR is tightly coupled to the DFG which allows the retrieval of data dependency information among instructions. The fifth phase, *assembly generation*, converts QIR into target assembly code.

Throughout all the compilation process, a common symbol table is created and maintained by the compiler. This symbol table includes information about local variables, global variables, function parameters and arguments, function names, read-only data such as constants and strings, and target labels for jumps. A control flow graph (CFG) consisting of basic blocks of the compiled program is also built and maintained by the compiler. These two structures are global and accessible to any phase of the compiler.

One of the goals in our design was to keep the compiler implementation independent from the target architecture. Although the presented compiler generates code for a specific target architecture, the Parallel Queue Processor, all the algorithms and data structures are machine independent and can be applied for the queue computation model in general.

3.2.1 QTree Generation

The front-end of the compiler is based on GCC 4.0.2 and it parses C files into a language and target independent high-level intermediate representation called GIMPLE [64, 66]. Although our queue compiler is capable of generating code from GCC's optimized GIMPLE code, at this point we have not validated the results and correctness of such programs. For the rest of this section, the GIMPLE code used as input for the queue compiler is non-optimized code. GIMPLE representation is a tree-based three-address

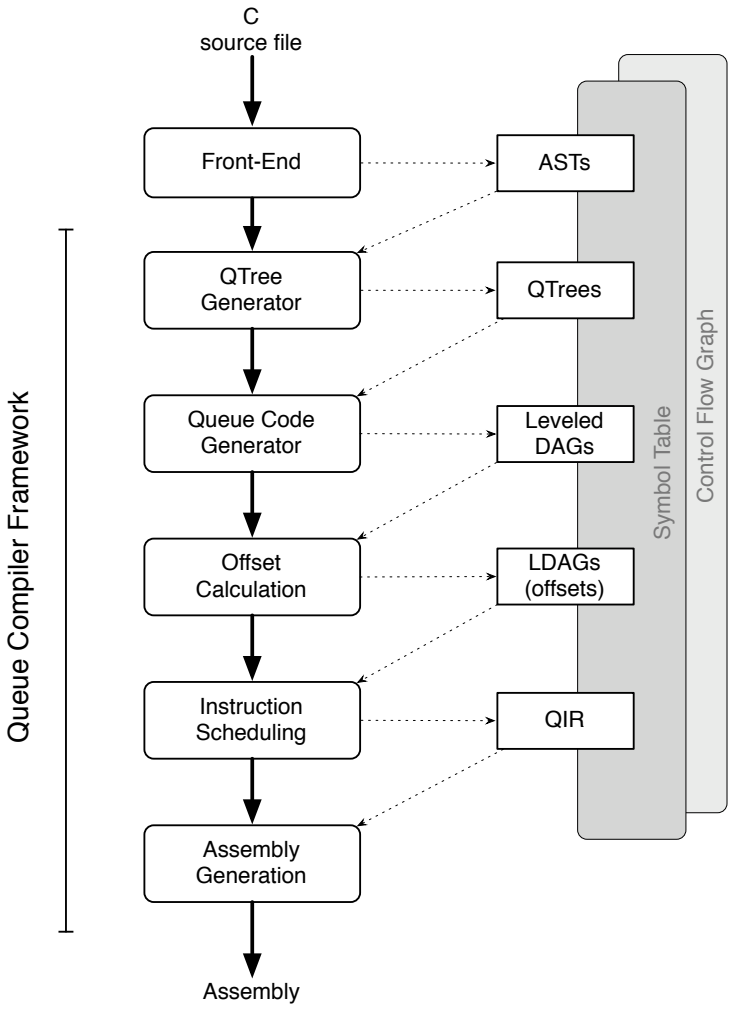


Figure 3.1: Queue compiler block diagram

code suitable for code generation for register machines but not for queue machines. As the level-order scheduling traverses the full DAG of an expression, we facilitate the job of the scheduler by reconstructing GIMPLE trees into trees of arbitrary depth and width called QTrees. During the expansion of GIMPLE into QTrees we also translate the high-level constructs such as aggregate types into their low level representation in generic queue instructions. The instruction selection phase is in charge of expressing any high level statement into generic queue instructions equivalent. Figure 3.2 shows a fragment of a C program and its GIMPLE representation. Notice that in the C-like GIMPLE representation in Figure 3.2(b), the statement inside the conditional is split into three-address statements with the help of compiler generated temporaries (i.e. D1098). To illustrate the high level nature of GIMPLE representation we show in Figure 3.2(c) the same program but using GIMPLE language. As shown in Figure 3.3(a), QTrees are unrestricted in the number of operands and operations. For the given example, the QTree representation is shown in Figure 3.3(b). Now the program is expressed through low level generic queue instructions.

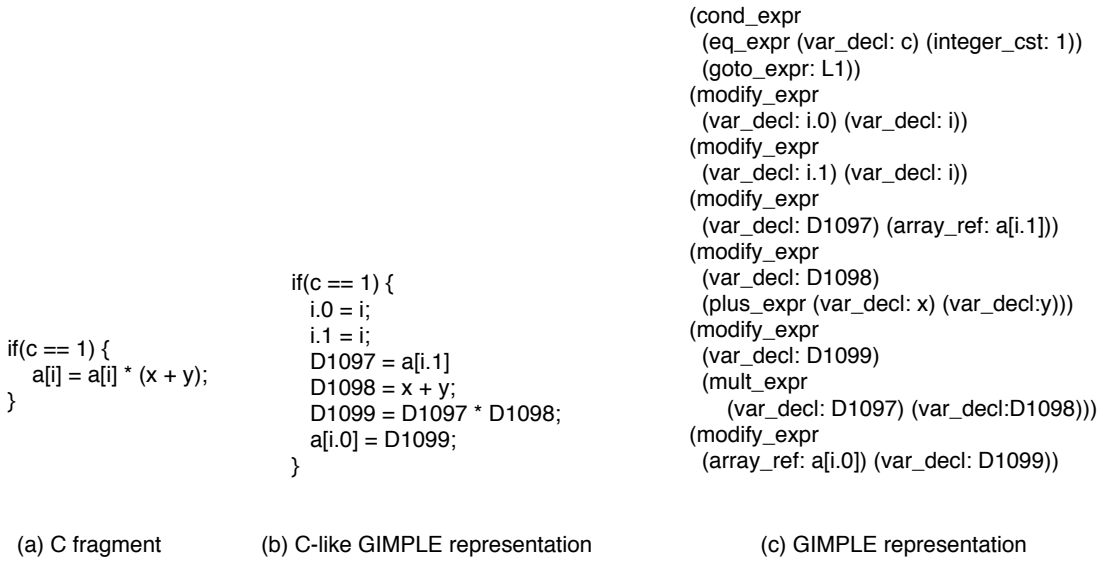


Figure 3.2: High-level intermediate representation. (a) C fragment, (b) C-like GIMPLE representation, (c) GIMPLE representation

```
if(c == 1) {
   a[i] = &a + (i * sizeof(a)) * (x + y);
}
```

(a) C-like QTree representation

```
(bt: L1
  (cne
    (ld: c)
    (ld: #1)))
(sst: a[i]
  (mul
    (sld: *a[i]
      (add
        (lea: &a)
        (mul
          (ld: i)
          (ld: #4))))
    (add
      (ld: x)
      (ld: y))
  )
)
```

(b) QTree representation

Figure 3.3: QTrees. (a) C-like Qtree representation, (b) QTree representation using low level generic queue instructions

3.2.2 Queue Code Generation

The most important task of the code generator is to shape the program in a suitable format that facilitates offset reference calculation and level-order scheduling. Leveled DAGs offer such characteristics where scalar calculations such as distances between nodes and levels is simple. In this section we establish the rules and techniques to build LDAGs from ASTs.

Leveled Directed Acyclic Graph (LDAG) Construction

Formally, a LDAG $\vec{G} = (V, \vec{E})$ is the mapping of the nodes to integers such that if there is an edge from u to v, then $lev(v) = lev(u) + N, N > 0$ for all edges in the graph. A node v is a *level-j node* if $lev(v) = j$. The hierarchical organization of the nodes in the DAG according their true data dependencies is what makes LDAGs the most suitable representation for finding offset references. Notice that all instructions in the same level are independent from each other. Therefore, the queue compiler exposes maximum natural

instruction level parallelism to the queue processor. The compiler builds a list of *slots* that bind together all nodes of every level. Figure 3.4 shows a LDAG for expression "$a[i] = (\&a + (i * sizeof(a))) * (x + y)$". The squared nodes in the left of the figure are the slots that indicate the levels of the DAG.

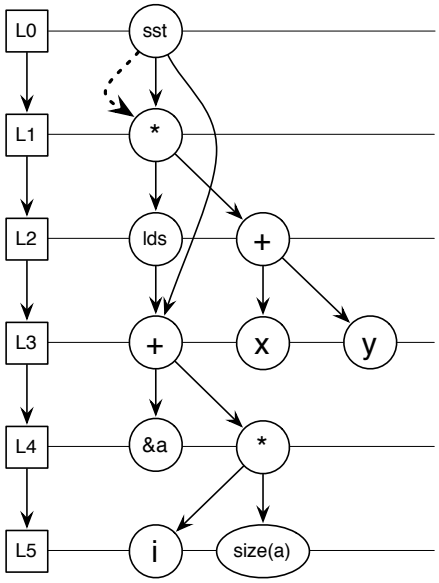


Figure 3.4: Leveled DAG for expression $a[i] = (\&a + (i * sizeof(a))) * (x + y)$

For describing the algorithm to build LDAGs from ASTs consider the following definitions:

Definition 3.2. A level is a non-empty list of elements.

Definition 3.3. An α-node is the first element of a level.

Definition 3.4. The root node of the LDAG is the only node in Level-0.

Definition 3.5. The sink of an edge must always be in a deeper or same level than its source.

Algorithm 1 shows how the construction of a LDAG is performed by a recursive post-order depth-first traversal of the parse tree together with a lookup table that saves information about every node and its assigned level. The *levelizing* of parse trees to LDAGs works as follows. For every visited node in the parse tree a level is assigned

and the lookup table is searched. If the node is not found in the table then it is its first appearance and a new entry is recorded in the table and the node is created in the LDAG. If the node is found in the table then a decision must be taken in order to satisfy the property in Definition 3.5. If the level of the new node is greater than the one in the lookup table, then the following steps should be performed: (1) create the new node with in its corresponding level in the LDAG, (2) the sink of all incoming dependency edges to the node in the table is replaced by the new node, (3) the old node is eliminated from the LDAG, and (4) the new level for the node is updated in the lookup table.

```
   Input: Expression tree, t
   Input: level
   Output: LDAG node, new
 1 begin
 2 |  nextlevel ← level + 1
 3 |  match ← lookup (t)
 4 |  /* Satisfy Definition 3.5 */
 5 |  if match ≠ null then
 6 |  |  if match.level < nextlevel then
 7 |  |  |  relink ← dag_move_node (nextlevel, match)
 8 |  |  |  return relink
 9 |  |  else
10 |  |  |  return match
11 |  |  end
12 |  end
13 |  /* Insert the node to a new level or existing one */
14 |  if nextlevel > get_Last_Level() then
15 |  |  new ← new_level_with_alpha (t, nextlevel)
16 |  |  record (new)
17 |  else
18 |  |  new ← append_to_level (t, nextlevel)
19 |  |  record (new)
20 |  end
21 |  /* Post-Order Depth First Recursion */
22 |  if t is binary operation then
23 |  |  lhs ← dag_levelize (t.right, nextlevel)
24 |  |  make_edge (new, lhs)
25 |  |  rhs ← dag_levelize (t.right, nextlevel)
26 |  |  make_edge (new, rhs)
27 |  else if t is unary operation then
28 |  |  child ← dag_levelize (t.child, nextlevel)
29 |  |  make_edge (new, child)
30 |  end
31 |  return new
32 end
```

Algorithm 1: dag_levelize (tree t, level)

3.2.3 Offset Calculation

The offset calculation phase calculates, for every binary and unary instruction in the program, the offset reference values to access their operands. Two steps are required to obtain the offset reference value for any operation u. First, the `QH` position relative

to u must be determined. And second, the distance δ from `QH` to the operand must be calculated. For any operand m of u, the offset reference value is given by the following equation:

$$offset(m) = \delta(qh\_pos(u), m) \tag{3.1}$$

The distance between two nodes (u, v) in a LDAG is given by the number of nodes found in the level-order traversal from u to v. For example, the `QH` relative position with respect of `sst` node in level L_0 of Figure 3.4 is in the only node of level L_1, shown by the dotted line. For its first operand, the `QH` points exactly to the same location in the LDAG, making an offset reference value of zero. For its second operand, the first node in level L_3, the distance between `QH` and the second operand is five as five nodes are visited in a reversed level-order traversal (`+, lds, y, x, +`). The output of this phase is LADGs with offset reference values computed for every operand of every instruction.

`QH` Position Algorithm

Once the LDAGs 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.

Definition 3.6. The `QH` position with respect of the α-node of Level-j is always at the α-node of the next level, Level-(j+1).

Definition 3.7. A level-order traversal of a LDAG is a walk of all nodes in every level (from the deepest to the root) starting from the α-node.

Definition 3.8. A hard edge is a dependence edge between two nodes that spans only one level.

Let p_n be a node for which the `QH` position must be found. `QH` relative position with respect of p_n is found after a node P_i in a traversal from p_{n-1} to p_0 (α-node) meets one of two conditions. The first condition is that P_i is a binary or unary operation and has a hard edge to one of its operands q_m. `QH` position is given by q_m's neighbor node as a result of a level-order traversal. Notice that from a level-order traversal, q_m's following node can be q_{m+1}, or the α-node of $lev(q_m) + 1$ if q_m is the last node in $lev(q_m)$. The second condition is that the node is the α-node, $P_i = p_0$. From Definition 3.6, `QH` position is at α-node of the next level $lev(p) + 1$. The dotted lines in Figure 3.5 show the `QH` relative position with respect of every binary and unary operations. The `QH` position with respect of `add`, `mul`, and `div` operations is given by the second condition as they are α-nodes of their respective levels. The `QH` position for the two `neg` and `sub` operations is found by the above explained rules of the first condition. The proposed algorithm is listed in Algorithm 2.

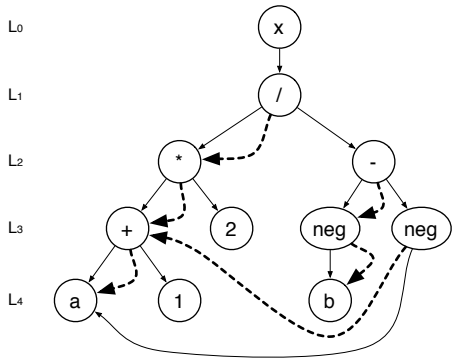


Figure 3.5: `QH` relative position for all binary and unary operations in a LDAG

```
   Input: LDAG, w
   Input: Node, u
   Output: QH position node, v
 1 begin
 2 |  I ← getLevel (u)
 3 |  for i ← u.prev to I.α-node do
 4 |  |  if isOperation (i) then
 5 |  |  |  if isHardEdge (i.right) then
 6 |  |  |  |  v ← BFS_nextnode (i.right)
 7 |  |  |  |  return v
 8 |  |  |  end
 9 |  |  |  if isHardEdge (i.left) then
10 |  |  |  |  v ← BFS_nextnode (i.left)
11 |  |  |  |  return v
12 |  |  |  end
13 |  |  end
14 |  end
15 |  L ← getNextLevel (u)
16 |  v ← L.α-node
17 |  return v
18 end
```

Algorithm 2: qh_pos (LDAG w, node u)

Offset Reference Calculation Algorithm

Definition 3.9. The distance between two nodes in a LDAG, $\delta(u, v)$, is the number of nodes found in a level-order traversal between u and v including u.

After the QH position with respect of p_n has been found, the only operation to calculate the offset reference value for each of p_n's operands is to measure the distance δ between QH's position and P_i as described in Algorithm 3. 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.

```
   Input: LDAG, W
   Output: Edge set with computed offset references, E
 1 begin
 2   forall nodes u in W do
 3     if u.lhs then
 4       u.lhs.offset ← δ (qh_pos(W, u), u.lhs)
 5     end
 6     if u.rhs then
 7       u.rhs.offset ← δ (qh_pos(W, u), u.rhs)
 8     end
 9   end
10 end
```

Algorithm 3: OffsetCalculation (LDAG W)

The result is 2-offset instructions where binary instructions require two source operands. The destination operand is omitted since it is fixed at `QT` following the enqueueing rule of producer-order model. The semantics of the 2-offset producer-order binary operations is the following:

operator	QH$\pm M$	QH$\pm N$

Where M, N are integer numbers representing the offset reference from where the operand will be dequeued. Similarly, the semantics for the unary operations is the following:

operator	QH$\pm M$

3.2.4 Instruction Scheduling

The instruction scheduling algorithm of our compiler [12] is a variation of basic block scheduling [65] where the only difference is that instructions are generated from a level-order topological order of the LDAGs. The input of the algorithm is a 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. The output of the algorithm is QIR, a linear low level intermediate representation of the program implemented as a double linked list. To match the nature of the queue computation model as much as

possible, we include in the QIR nodes a single operand. The only operand is exclusively used by memory, and branch instructions to specify the memory location, an immediate value, or the target label of the jump. All binary and unary instructions resemble the queue computation model having zero operands. We consider offset references attributes of the instructions. Additionally, we insert annotations in QIR to facilitate machine dependent transformations. Table 3.4 shows the full QIR specification. The *Operands* class identifies the type of operand for the class *One Operand*, which is the only type of instructions that require an explicit operand. It includes immediate values, local variables, global variables, parameters, arguments, labels, return value, temporaries, symbols, and `QH`. The classes *Binary*, *Unary*, and *Compare* do not use any operand and offset values are encoded as attributes of the instruction itself. The *Special* class includes all instructions where the semantics do not fall in the other categories and require special handling by the code generator.

Figure 3.6 shows the QIR debugging representation of the sample C program in Figure 3.2. The instructions have one to one equivalence to the generic queue instructions defined for the queue compiler. Annotations that mark the beginning of levels are shown and every level has a unique identifier. All instructions show the data type attribute as [iws]. The offset reference values are indicated for binary operations. Notice that memory operations (`PUSH_Q, LOADI_Q`) have an explicit operand that represents the memory location of local variables or numeric constants.

3.2.5 Assembly Generation

The last phase translates the QIR program list into assembly code for the QueueCore processor. Figure 3.7 shows the assembler output for the C program fragment in Figure 3.2. All instructions in the assembly language consist of the opcode followed by the data type on which the instruction should be executed. Depending on the instruction type, there maybe be from one to three extra operands. For example, the instruction "`j iws, L2`" has only one extra operand that indicates the target label to where the jump instruction should pass control. Binary instructions such as "`add iws, qt, qh, qh+1`" have three extra operands. The `qh` indicates that the first source operand should be taken from `QH`(the zero offset is omitted), and the second source operand from `QH`+1. Although

Table 3.4: QIR specification

Class	**Name**
Operands	NULL_Q, INTEGER_Q, LOCAL_VAR_Q, GLOBAL_VAR_Q, PARM_VAR_Q, ARG_VAR_Q, LABEL_Q, RETVAL_Q, TEMP_Q, SYMBOL_Q, QH_Q
Binary	ADD_Q, SUB_Q, DIV_Q, MUL_Q, NEG_Q, RSHIFT_Q, LSHIFT_Q, RROT_Q, LROT_Q, RDIV_Q, BIT_IOR_Q, BIT_XOR_Q, BIT_AND_Q, MOD_Q,
Unary	NEG_Q, ABS_Q, BIT_NOT_Q, ROTATE_Q
One Operand	PUSH_Q, POPQ_Q, LOADI_Q, RETURN_Q
Compare	EQ_Q, NE_Q, LT_Q, LE_Q, GT_Q, GE_Q
Special	LOAD_ADDR_Q, STORE_Q, SLOAD_Q, CONVERT_Q, CALL_Q, GOTO_Q, TLABEL_Q, COPY_P_Q, MOVEQH_Q, MOVEQT_Q, DUP_Q
Annotations	QMARK_LEVEL, QMARK_STMT, QMARK_BBSTART, QMARK_BBEND, QMARK_FUNCSTART, QMARK_FUNCEND

```
(QMARK_LEVEL [id=0])
(PUSH_Q [iws] (LOCAL_VAR_Q: 8($fp) )
(LOADI_Q [iws] (INTEGER_Q: 1 )
(QMARK_LEVEL [id=1])
(NE_Q [iws] )
(GOTO_Q [iws] (LABEL_Q: L1 )

(TLABEL_Q [iws] (LABEL_Q: L0 )
(QMARK_LEVEL [id=2])
(PUSH_Q [iws] (LOCAL_VAR_Q: 12($fp) )
(LOADI_Q [iwu] (INTEGER_Q: 4 )
(QMARK_LEVEL [id=3])
(LOAD_ADDR_Q [iws] (LOCAL_VAR_Q: 16($fp) )
(MUL_Q [iws]  ( QT, QH+0, QH+1 ))
(QMARK_LEVEL [id=4])
(ADD_Q [iws]  ( QT, QH+0, QH+1 ))
(PUSH_Q [iws] (LOCAL_VAR_Q: 4($fp) )
(PUSH_Q [iws] (LOCAL_VAR_Q: 0($fp) )
(QMARK_LEVEL  [id=5])
(SLOAD_Q [iws]  ( QT, QH+0, QH+0 ))
(ADD_Q [iws]  ( QT, QH+0, QH+1 ))
(QMARK_LEVEL  [id=6])
(MUL_Q [iws]  ( QT, QH+0, QH+1 ))
(QMARK_LEVEL  [id=7])
(STORE_Q [iws]  (QH-5, QH+0 ))
(GOTO_Q [iws] (LABEL_Q: L2 )
```

Figure 3.6: QIR representation

the QueueCore implicitly writes to the queue always to the `QT`, we include `qt` operand in the assembly for readability. The assembler is in charge of removing unnecessary fields from the code when generating object code, therefore, the `qt` field is removed from the instructions in the object code. Memory operations such as "`ld iws, qt, ($fp)12`" have two extra operands, the destination `qt` and the memory location where to access the operand. The `$fp` represents the frame pointer, a special purpose register to access local variables.

3.2.6 Application Binary Interface (ABI)

Stack Frame Layout

A function that calls another function is the *caller* function. The function that has been called is named the *callee* function. The compiler is responsible for setting the correct environment so the callee function is executed correctly. The compiler is also responsible of restoring the status of the processor when the call was made, this is, restoring the environment of the caller function to continue its execution.

Every function, including leaf functions, have their own stack. The stack frame grows downwards, from high memory to low memory. The beginning of the stack (from top to

```
        ld      iws, qt, ($fp)8
        ldi     iws, qt, 1
        cne     iws, iws, $cc, qh, qh+1
        bt      iws, L1, $cc
L0:
        ld      iws, qt, ($fp)12
        ldi     iws, qt, 4
        lea     iws, qt, $fp, 16
        mul     iws, qt, qh, qh+1
        add     iws, qt, qh, qh+1
        ld      iws, qt, ($fp)4
        ld      iws, qt, ($fp)0
        sld     iws, qt, qh
        add     iws, qt, qh, qh+1
        mul     iws, qt, qh, qh+1
        sst     iws, qh-5, qh
        j       iws, L2
```

Figure 3.7: QueueCore assembly output

bottom) contains the saved registers: return address and frame pointer. The next area in the stack is the one reserved for local variables. The last area is used for temporaries such as parameters to other functions and return values from other functions. Figure 3.8 shows the stack frame layout for the Queue Compiler. Notice that frame pointer register points to the beginning of local variable area. All local variables are accessible through an offset from FP. Stack pointer register points to the end of the stack frame for the current function. All outgoing parameters to other functions and return values are accessible by an offset from SP. Stack frames are aligned to 128 bits to match the natural alignment of the long double data type, the largest data type in P-Code.

Calling Conventions

Variables that are passed to other functions are called *outgoing parameters*. Variables that are passed from other functions are called *incoming parameters*. Outgoing parameters are placed in the temporaries area. The first outgoing parameter is placed in zero offset from the Stack Pointer of the caller function and all other outgoing parameters, if any, are placed from an offset from stack pointer:

$$outgoing\_parameters = \texttt{SP} + \texttt{offset} \tag{3.2}$$

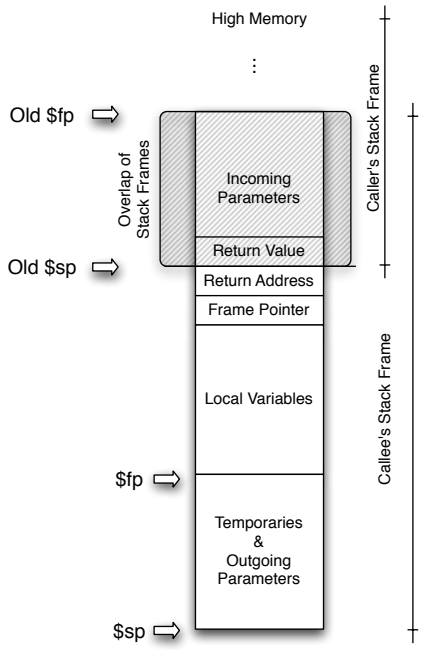


Figure 3.8: Stack Frame Layout

Incoming parameters are available from an offset from frame pointer:

$$incoming\_parameters = \texttt{FP} + Size\_of\_Local\_Variable\_Space + \texttt{offset} \tag{3.3}$$

This mechanism allows that outgoing parameters of the caller function become the input parameters for the callee function. The outgoing parameters area (temporaries area) for the caller function becomes the incoming parameters area for the callee function. After the callee function returns, if it returns any value, the value is put after the local variables area:

$$return\_value_{callee} = \texttt{FP} + Size\_of\_Local\_Variable\_Space \tag{3.4}$$

Thus, the return value for the caller function after the callee function returns is

available at zero offset from stack pointer as shown in Figure 3.8:

$$return\_value_{caller} = \texttt{SP} \tag{3.5}$$

Local Variable Space Calculation

Only two registers are needed to restore the status of the processor after the callee function returns: the return address register and the frame pointer register. Let `function1` be the caller function and `function2` the callee function invoked from the first. Let the Program Counter (PC) point the instruction which calls `function2`. Before the processor sets the effective address of the callee function to the PC, the compiler saves the contents of the return address register into the stack frame of the callee function. This value is needed for the return of the callee to restore the execution to the point where the callee was invoked.

The total space occupied by local variables is know at compile time. Thus, the total amount of space required to hold the local variable space is the total space occupied by the local variables plus the two saved registers: RA and FP. The local variable space is 64 bit aligned.

Temporaries Area Calculation

The temporaries area is a space allocated to hold all outgoing parameters to other functions. Let f_c be the current function for which the temporaries area is being calculated. Let σ be the sum of the size of outgoing parameters for a given function. The size of the temporaries area for f_c is given by the biggest σ of all functions called from f_c. Notice that if the current function has no calls to other functions, a *leaf function*, then the size of the temporaries area is zero. The temporaries area is aligned to 64 bits.

Prologue and Epilogue

The prologue and epilogue code is fixed for all functions compiled by the Queue Compiler. The prologue saves the return address and frame pointer registers, allocates space for the

saved registers, local variables, and the temporaries area. The epilogue code restores the status of the processor after the callee function was invoked.

```
1 begin
2     SP ← SP - (Size_of_Local_Variable_Space + Saved_Registers_Size)
3     save_registers ()
4     FP ← SP
5     SP ← SP - σ
6 end
```

Algorithm 4: prologue()

```
1 begin
2     FP + Size_of_Local_Variable_Space ← return_value
3     SP ← FP
4     restore_saved_registers ()
5     SP ← SP + Size_of_Local_Variable_Space + Saved_Registers_Size
6 end
```

Algorithm 5: epilogue()

3.3 Functionality

As we presented in the previous sections, the core of the Queue Compiler Framework is driven by fundamental algorithms crafted for the queue computation model. Having a completed compiler allowed the further development of advanced compilation techniques. Figure 3.9 shows all functionality of the queue compiler. Related functionality is grouped, for example, code generation for specific hardware is formed by "High-ILP, Code Size, Constant Parallelism, Queue Register File Control, and Reduced bit-width ISA compilation". This functionality group describes more sophisticated algorithms that allow the generation of code for specific hardware implementations. We have made the compiler aware of the target architecture to accomplish better code. As described by the figure, the compiler becomes a powerful tool that delivers compact code with high instruction level parallelism, and it allows fine grained control of the hardware resources. The next chapter, Chapter 4, is dedicated to show the advanced compilation techniques for such constrained compilation functionality.

Optimization group is formed by the three top circles: "Optimization, Classic CSE, Loop Unrolling". Using the queue compiler we have explored the optimization space

for queue processors including data flow optimizations such as common subexpression elimination, dead code elimination, dead store elimination; and ILP optimizations such as loop unrolling. We also have developed custom optimizations for queue computation model. Chapter 4 discusses the main findings in queue code optimization.

Memory traffic optimization group is shown in the bottom: "Memory Traffic Reduction, Inside BB, BB Communication". This group represents a novel technique to reduce memory traffic by using the queue for holding temporaries and propagating them along computation not to saturate the physical queue register file. Chapter 5 introduces and evaluates this idea.

Performance models group is shown in the left "Dynamic Evaluation LBET" and for the rest of this Chapter we will concentrate on the functionality and properties of the compiler itself and the performance models.

3.3.1 Self-Hosted Compiler

As for today, there is no queue processor hardware available nor system software such as operating system libraries, linkers, and loaders. Technically, the queue compiler sits on top of working systems and it produces assembly files for the queue processor making a cross-compiler. However, the cross-compiler is able to compile itself into queue assembly code, making it a self-hosted compiler ready for deployment in a queue-based system. As a cross-compiler it has been bootstrapped in a variety of systems such as Linux, Mac OSX, Solaris, and Windows; and different hardware platforms: x86, PowerPC, and SPARC. The portability of the queue compiler framework is a good characteristic. The queue compiler does not allow statically-linked programs. The reason is that system libraries for queue system have not been generated as they heavily depend on the target operating system and hardware implementation. During cross-compilation, system definitions are taken from the libraries of the host system. This compiler configuration allows the compilation of any program including standard benchmarks from SPEC CINT95 [21], MiBench [32], MediaBench [52], DSP Stones [90], and other reference programs. Figure 3.10 shows the cross-compilation setup of the queue compiler. The flow related to the profiler and run-time statistics is discussed in the following section.

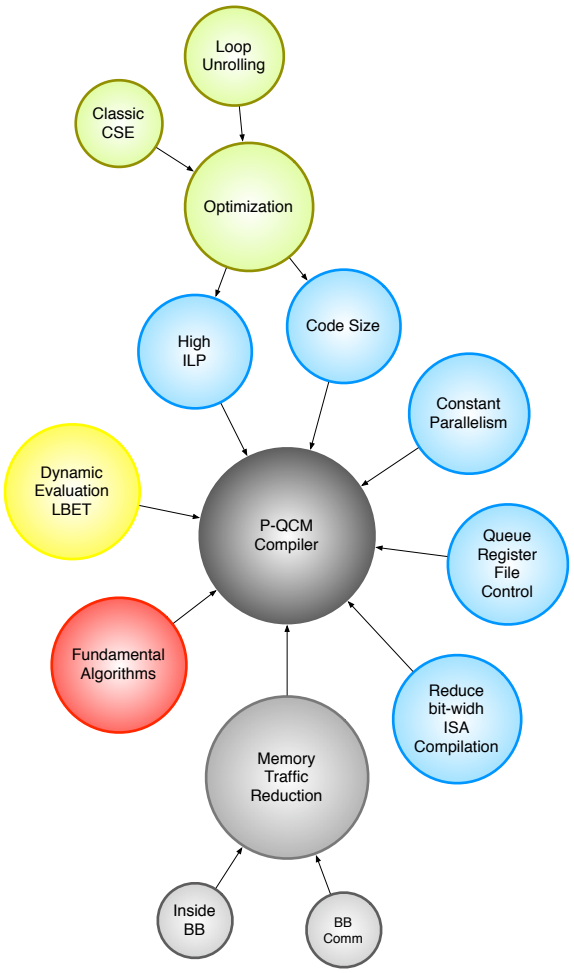


Figure 3.9: Framework's functionality. Related techniques grouped by color.

3.3.2 Lower Bound Execution Time (LBET)

In [99, 100], the Lower Bound Execution Time (LBET), is defined as the best performance that can be achieved by a microarchitecture for a given workload. Performance bounds models have been used as a way to evaluate the effects of compiler optimizations or architectural innovations in performance. The addition of LBET in the queue compiler framework provides a valuable design space exploration tool for the development of specific code generation optimizations. We use a LBET equation similar to [100]:

$$LBET = \sum LBET_{block_i} * freq_{block_i} \tag{3.6}$$

$$= \sum FIT\left(data\_dependence\_bound_{block_i}, resource\_bound_{block_i}\right) * freq_{block_i} \tag{3.7}$$

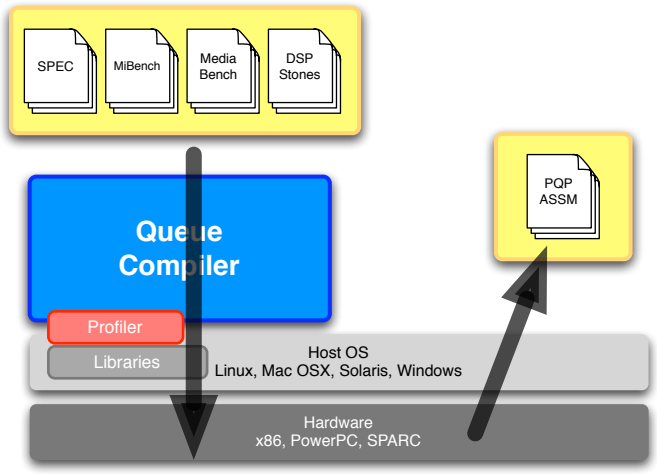


Figure 3.10: Cross-compiler configuration

The LBET of a program is the sum of all LBET computed for each basic block multiplied by the block's execution frequency. The execution frequency is obtained by a profiled execution of the program. The compiler instruments the output code using the profiler as shown in Figure 3.10. The instrumented code is executed by the host system and generates a profiled execution statistics which contains the execution frequency of every basic block in the program.

In equation 3.6, the LBET of a basic block is the relationship (FIT) between the *data dependence bound* and the *resource bound* of the block. Since the level-order scheduling exposes all parallelism in the data flow graph (DFG), the data dependence bound is given by number of levels in the DFG, in other words, the height of the DFG. The resource bound is given by the number and type of functional units in the target queue processor. Therefore, FIT is the result of constraining a basic block's DFG into the available hardware resources. Figure 3.11 shows the procedure to obtain the performance estimation using LBET. The frequency bound is obtained by profiling the input program and characterizing the execution for a given workload, the data dependence bound is obtained by the queue compiler, and the resource bound is given by a machine description file (MD file) that contains the available hardware resources including functional units and instruction latencies. Then performance is calculated by LBET as shown in Equation 3.7.

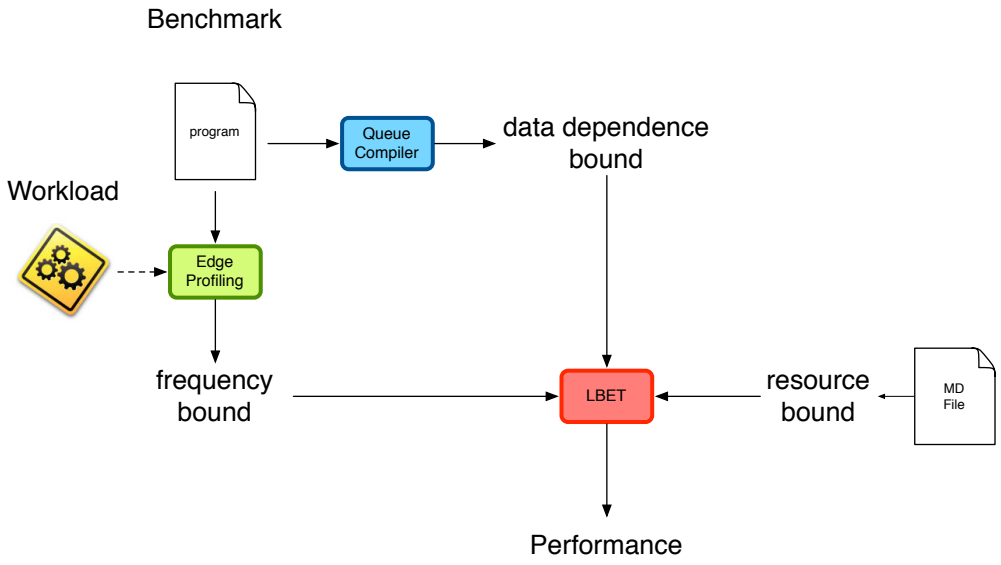


Figure 3.11: Lower Bound Execution Time (LBET) model

3.4 Framework Complexity Evaluation

3.4.1 Lines of Code Complexity

The Queue Compiler was developed by the author of this thesis from March 2005 to March 2008, 3 years. The approximate number of coding lines is 15,000. To provide an estimation of the programming complexity of the Queue Compiler we compare it against other opensource research and production compilers. Figure 3.12 shows a histogram comparing the number of lines of code for five compiler including the Queue Compiler. For all compilers, except LCC 4.2, the given number of lines belongs only to the back-end. We ignore the front-ends as the Queue Compiler uses a conventional front-end. The LCC 4.2 [51, 26] compiler is also known as *Little C Compiler* and it is the smallest of all. The Trimaran 4.0 [88] is a VLIW research infrastructure. And LLVM 2.2 [57] and GCC 4.2.0 [28] are two production compilers used and developed by large communities. For all register compiles except GCC 4.2.0, the back-end is clearly identified in the source code tree. For GCC 4.2.0 we measured the files related to register allocation and instruction scheduling only.

From this comparison we observe that the implementation complexity of the Queue Compiler infrastructure is similar to that of conventional compilers. Although the

compiling phases are completely different, the final result suggests that the man power required to implement a queue compiler is significant.

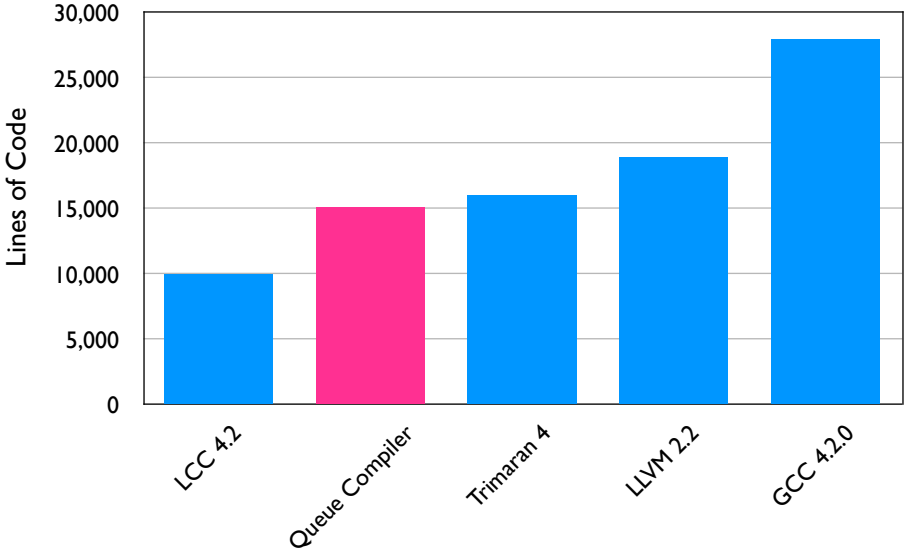


Figure 3.12: Lines of Code complexity of five compiler back-ends

3.4.2 Compile-time Complexity

To analyze the complexity of the queue code generation algorithms implemented in the compiler, we measure and compare the *real* time taken to compile a set of benchmark programs. Table 3.5 shows the lines of code (LOC) for each SPEC CINT95 benchmark [21]. The host system is a dual 3.2 GHz Xeon processor computer running Linux 2.6.20 kernel. We selected GCC 4.0.2 compiler as the native compiler for the system. Since the Queue Compiler uses GCC's 4.0.2 front-end, the compilation time is fair and can be compared without considerations. Both the Queue Compiler and the native compiler were bootstrapped without optimizations (-O0). Figure 3.13 shows the time, in seconds, spent to compile the benchmark programs. For all programs the compilation time is similar for the queue compiler and the native compiler. For the largest benchmark, 126.gcc, the queue compiler is able to compile faster by a factor of 2.

Table 3.5: Compiler complexity by compile-time analysis

Benchmark	LOC
099.go	28547
124.m88ksim	17939
126.gcc	193752
129.compress	1420
130.li	6916
132.ijpeg	27852
134.perl	23678
147.vortex	52633

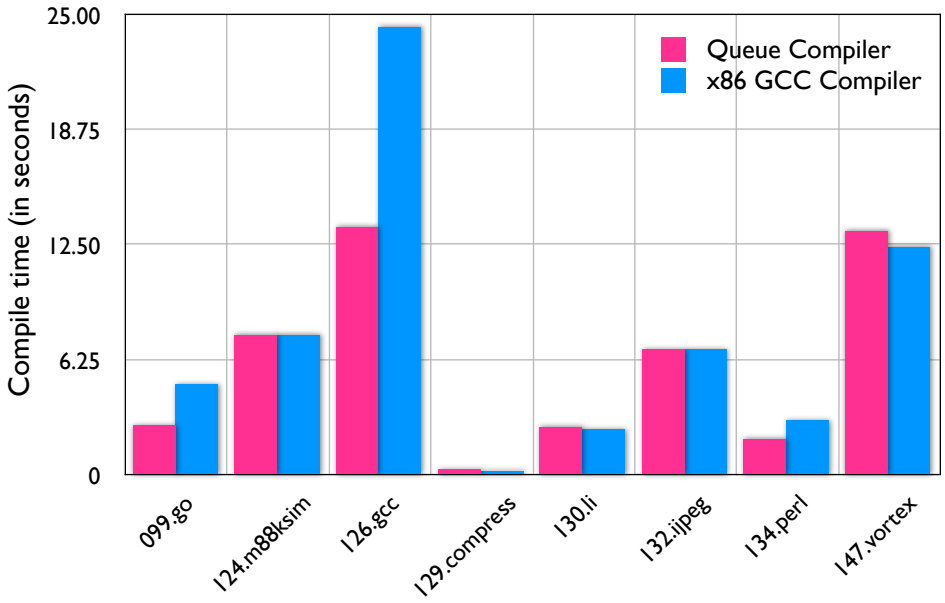


Figure 3.13: Compile Time Compiler

Chapter 4

Constraint-Driven Compilation

This chapter presents novel techniques to drive the code generator to meet specific objective functions such as small code size, queue register file control, optimization for execution time and instruction count reduction, parallelism extraction. To accomplish such objectives, the compiler must be aware of the available hardware resources, and make best efforts to generate the most suitable code. This chapter is organized in four sections, each section presents a compiling technique for different objective. Every section explains and motivates the need for such functionality, it describes the necessary methods to accomplish the task, it provides an evaluation for the presented method, and it opens further discussion and gives conclusions.

4.1 Code Size-aware Compilation

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 [31, 43] 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 [31] and MIPS16 [41] are examples of dual instruction sets. In [31], 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 [47, 33, 78, 50, 48, 46].

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 [54, 95]. Compact instructions improve memory bandwidth, fetching time, and power consumption [30]. In this section, 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. Therefore, this section presents compilation for a 1-offset P-Code instruction set. 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 technique are as follows:

- 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.

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 4.1 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.

Table 4.1: 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%

4.1.1 1-offset P-Code

Figure 4.1 shows a 4-point fourier transform DAG and its evaluation using the PQP instruction set [87]. 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 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 4.2 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.

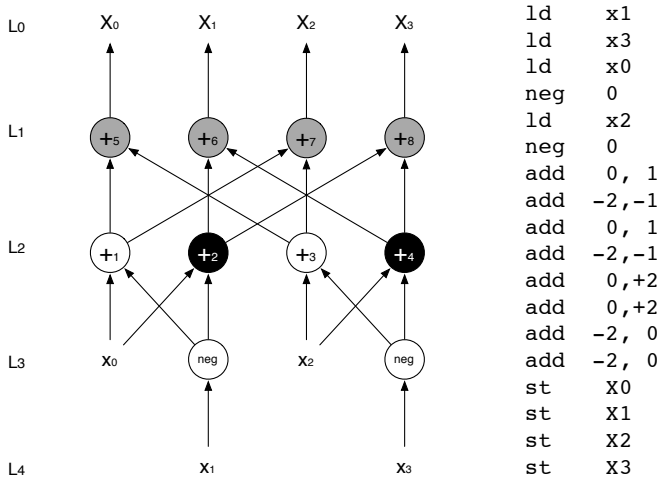


Figure 4.1: 4-point Fourier transform directed acyclic graph.

4.1.2 Code Size Reduction-aware Code Generation

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

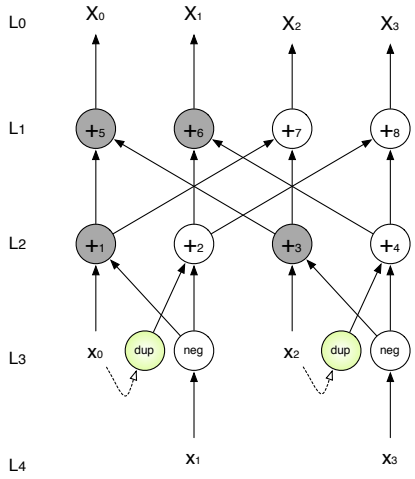


Figure 4.2: Fourier transform's directed acyclic graph with `dup` instructions.

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 6 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 4.1. The sink of an edge must be always in a deeper or same level than its source.

Definition 4.2. An edge to a ghost node spans only one level.

When an operand is found in the lookup table the Definition 4.1 must be kept. Line 6 in Algorithm 6 is reached when the operand is found in the lookup table and it has a shallower level when 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 6 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 4.1 and 4.2. Figure 4.3 illustrates the result of leveling the QTree for the expression $x = (a * a)/(-a + (b - a))$. Figure 4.3(b) shows the resulting LDAG augmented with ghost nodes.

```
   Input: Tree, t
   Input: level
   Output: Augmented LDAG
 1 begin
 2 |  nextlevel ← level + 1
 3 |  match ← lookup (t)
 4 |  if match ≠ null then
 5 |  |  if match.level < nextlevel then
 6 |  |  |  relink ← dag_ghost_move_node (nextlevel, t, match)
 7 |  |  |  return relink
 8 |  |  else if match.level = lookup (t) then
 9 |  |  |  relink ← insert_ghost_same_level (nextlevel, match)
10 |  |  |  return relink
11 |  |  else
12 |  |  |  return match
13 |  |  end
14 |  end
15 |  /* Insert the node to a new level or existing one */
16 |  if nextlevel > get_Last_Level() then
17 |  |  new ← make_new_level (t, nextlevel)
18 |  |  record (new)
19 |  else
20 |  |  new ← append_to_level (t, nextlevel)
21 |  |  record (new)
22 |  end
23 |  /* Post-Order Depth First Recursion */
24 |  if t is binary operation then
25 |  |  lhs ← dag_levelize_ghost (t.left, nextlevel)
26 |  |  make_edge (new, lhs)
27 |  |  rhs ← dag_levelize_ghost (t.right, nextlevel)
28 |  |  make_edge (new, rhs)
29 |  else if t is unary operation then
30 |  |  child ← dag_levelize_ghost (t.child, nextlevel)
31 |  |  make_edge (new, child)
32 |  end
33 |  return new
34 end
```

Algorithm 6: dag_levelize_ghost (tree t, level)

`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 7. Function `dup_assignment()` decides whether ghost nodes are substituted by

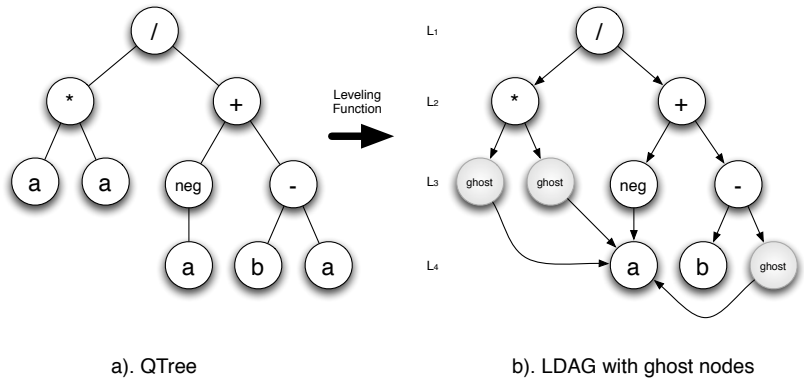


Figure 4.3: Leveling of QTree into augmented LDAG for expression $x = \frac{a \cdot a}{-a+(b-a)}$

`dup` nodes or eliminated from the LDAG. Once all the ghost 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 [10]. The output of the code generation algorithm is QIR where all instructions use at most one offset reference.

```
1  begin
2  |  forall basic blocks BB do
3  |  |  forall expressions W_k in BB do
4  |  |  |  forall instructions I_j in TopBottom (W_k) do
5  |  |  |  |  dup_assignment (I_j)
6  |  |  |  end
7  |  |  |  forall instructions I_j in LevelOrder (W_k) do
8  |  |  |  |  p_qcm_compute_offsets (W_k, I_j)
9  |  |  |  end
10 |  |  end
11 |  end
12 end
```

Algorithm 7: 1offset_codegen ()

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 8 describes the function `dup_assignment()`. The effect of Algorithm 8 is illustrated in Figure 4.4. The algorithm takes as input the LDAG with ghost nodes shown in Figure 4.3(b) and performs the steps described in Algorithm 8 to finally obtain the LDAG with `dup` instructions as shown in Figure 4.4(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 4.4(b).

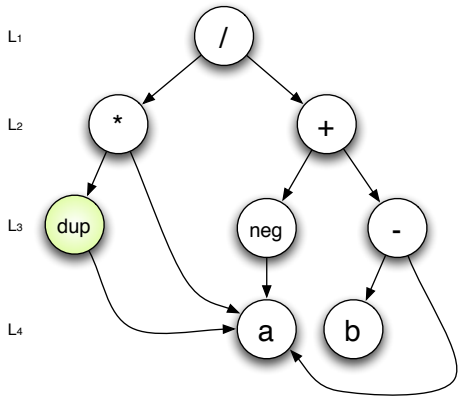
Input: LDAG node, i
Output: Modified LDAG

```
1  begin
2      if isBinary (i) then
3          if isGhost (i.left) and isGhost (i.right) then
4              dup_assign_node (i.left)
5              dag_remove_node (i.right)
6          else if isGhost (i.left) then
7              dag_remove_node (i.left)
8          else if isGhost (i.right) then
9              dag_remove_node (i.right)
10         end
11     end
12 end
```

Algorithm 8: dup_assignment (i)

4.1.3 Code Size Reduction Evaluation

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. We compiled these applications using our queue compiler infrastructure with the presented code generation algorithm for code size reduction. 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 conventional RISC processors and their embedded versions:



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

Figure 4.4: 1-offset constrained code generation from a LDAG

MIPS I [41], ARM32 [69], MIPS16 [43], and ARM/Thumb [31]. We prepared GCC 4.0.2 compiler for the register-based 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 4.5 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 compiler technique produces 51% smaller code than the baseline code size, in average. These results include the extra `dup` instructions. 1-offset queue code is, in average, 16% smaller code than gcc for MIPS16, and 36% smaller code than gcc for ARM/Thumb architecture. For three of the benchmarks, `quicksort.c`, `md5.c`, and `matrix.c`, our compiler generated larger code compared to MIPS16. 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.

4.1.4 Effect of `dup` instructions on Code Size

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

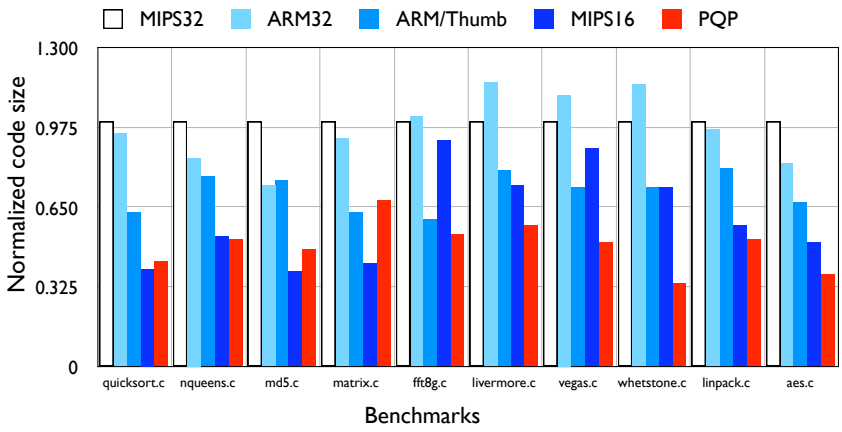


Figure 4.5: Code size evaluation of 1-offset P-Code technique

$$dup_i = \beta_i \tag{4.1}$$

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.

$$\Delta = \sum_{i=1}^{n} (dup_i) \tag{4.2}$$

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):

$$Total = T\_old + \Delta \tag{4.3}$$

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 4.3 as:

$$Code\_Size = 2 * (Total + covop) \tag{4.4}$$

Figure 4.6 shows the overhead of `dup` instructions inserted over the original 2-offset PQP code. The increase in number of instructions is below 5% for all applications. These results confirm that our technique can effectively exploit the characteristic of applications having only a few 2-offset instructions to improve code density by compiling for a reduced bit-width instruction set.

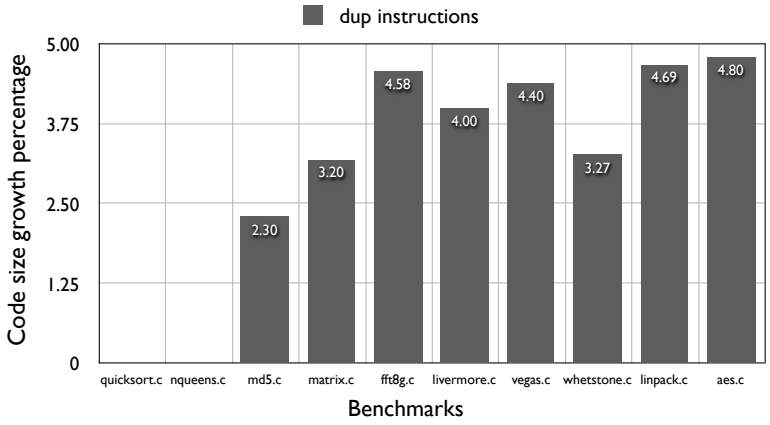


Figure 4.6: Overhead of `dup` instructions.

4.1.5 Discussion on Variable-length Instruction Set

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 encoded in 8-bits as 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.

4.1.6 Conclusion

In this Section 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 16% and 36% denser than MIPS16 and ARM/Thumb architectures. Queue architectures are a viable alternative for executing applications that require small code size footprint and high performance.

4.2 Queue Register File Optimization

At any point of execution of a program, the *queue length* is the number of elements between `QH` and `QT`. Every statement in a program may have different queue length requirements and the hardware should provide enough words in the FIFO queue to hold the values and evaluate the expression. We developed the Queue Compiler Infrastructure [12] as part of the design space exploration tool-chain for the QueueCore processor [2]. The original queue compiler targets an abstract queue machine with unlimited resources including an infinite queue register file. On this assumption we measured the queue length requirements of SPEC CINT95 applications. Figure 4.7 shows that 95% of the statements in the programs require less than 32 queue words for their evaluation, and the remaining 5% demand a queue size between 32 and 363 words. In our previous work [15] we gained insight on how queue length is mainly affected by two characteristics of programs: parallelism and *soft edges*. Soft edges represent the lifetime, in queue words, of reaching definitions of variables. Graphically a soft edge is an edge that spans across more than one level in the data flow graph. Table 4.2 shows the maximum queue requirements for the peak parallelism and maximum def-use length in SPEC CINT95 programs compiled for infinite queue. This table demonstrates that a reasonable and realizable amount of queue is needed in queue processors to execute the programs without performance penalty. However, assistance of the compiler is required to schedule the programs in such a way that parallelism and soft edges comply with the queue register file size in a realistic queue processor.

This Section presents an optimizing compiler to partition the data flow graphs of programs into clusters of constant parallelism and limited length of soft edges that can be executed in a queue processor with a limited queue register file. The compiler is also responsible for generating clusters that obey the semantics of the queue computation model. The proposed algorithm was implemented in the queue compiler infrastructure [12, 10] affecting compile-time by a negligible amount. The goal of this Section is to estimate how the characteristics of the output code are affected when the available queue is constrained. We estimate how the critical path, the available parallelism, and the program length of SPEC CINT95 benchmarks is affected for different

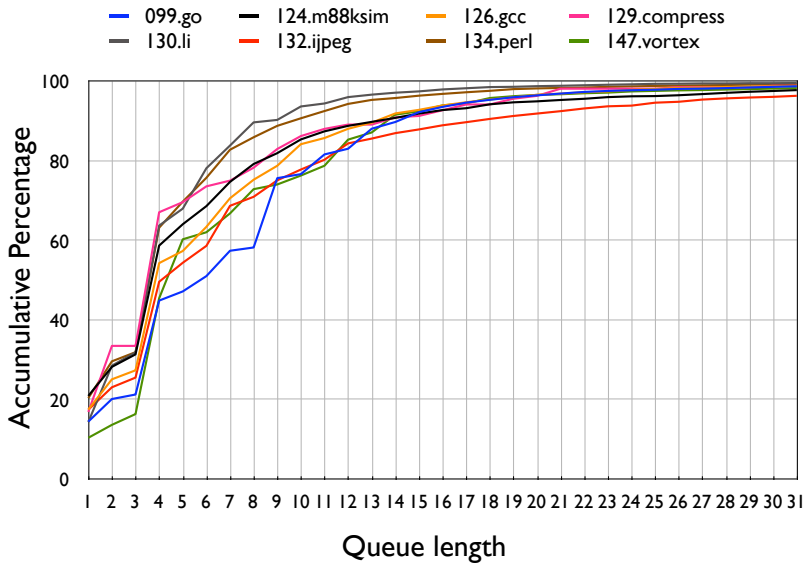


Figure 4.7: Queue size requirements. The graph quantifies the amount of queue required to execute statements in SPEC CINT95 benchmarks. A point, (x, y), denotes that y% of the statements in the program require x, or less, queue words to evaluate the expression.

size configurations of the queue register file. The contributions of this work are:

- The first study, for the best of our knowledge, that estimates the performance of a queue processor with limited amount of queue words.
- An efficient compiler algorithm that partitions the data flow graph into clusters that demand no more queue than the available in the underlying architecture. This is achieved by limiting the parallelism and the length of reaching definitions in the data flow graph.

4.2.1 Related Work

In our previous work [87, 3, 2], we have investigated and designed a producer order parallel queue processor, QueueCore, capable of executing any data flow graph. Our model 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

Table 4.2: Characteristics of programs that affect the queue length in queue-based computers

Benchmark	Peak Parallelism	Max. def-use
099.go	20	19
124.m88ksim	29	19
126.gcc	35	56
129.compress	9	10
130.li	17	18
132.ijpeg	26	24
134.perl	15	15
147.vortex	49	14

write operands. In the QueueCore's instruction set, the writing location at the rear of the queue remains fixed for all instructions. To realize QueueCore as an actual processor we must explore how the queue register file size affects performance. None of the previous work related to queue computers has considered a constrained queue register file. We know by the experience of more than fifty years in register machines, that the size of the register file directly affects the overall performance of a computer system. Many works have proposed the optimization of the register file for the improvement of execution time [98, 56, 23], parallelism [93, 70, 59], power consumption [7, 72, 49, 82], hardware complexity [39, 38], etc [50, 99].

4.2.2 Target Architecture: QueueCore processor

The QueueCore processor [2] implements a producer order instruction set architecture. Each instruction can encode at most two operands that specify the location in the queue register file from where to read the operands. The processor determines for each instruction the physical location of the operands by adding the offset reference in the instruction to the current position of `QH` pointer. 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 allowing parallel execution of instructions. Every instruction of the QueueCore is 16-bit wide. For cases when there are insufficient bits to express large constants, memory offsets, or offset references, a `covop` instruction is inserted. This special instruction extends the operand field of the following instruction by concatenating it to its operand. The queue register file size of the QueueCore processor is set to 256 words. The Queue Compiler Framework is able of generating code for the QueueCore processor.

4.2.3 Algorithm for Queue Register File Constrained Compilation

Queue length refers to the number of elements stored between `QH` and `QT` at some computation point. We have introduced how queue length can be determined by counting the number of elements in a computation level. Nevertheless, DAGs often present a case when this assumption is not enough to estimate the queue length requirements of an expression. Consider the DAG shown in Figure 4.8 for multiply-accumulate operation commonly used in signal processing "y[n] = y[n] + x[i] * b[i]". Notice that some edges span more than one level (soft edges), for example the edge with source at node "sst" and sink at node '+' of L_3 spans three levels. Soft edges increase the queue length requirements as the sink node must be kept in the queue until the time when the source node is executed. For the given example, the maximum queue length requirement of the DAG is five queue words and it is imposed by the longest soft edge. The algorithm must deal with two different conditions that directly affect the queue length requirements of an expression. One is the length of computation levels, and the second is the length of soft edges. The former can be solved by splitting the level into blocks of manageable size. The later can be solved by re-scheduling the child's subtree. The order on which these actions are performed affects the quality of the output DAGs and therefore the quality of the generated code.

If the levels are split first and then subtrees re-scheduled, the second action affects the length of the levels in the final DAG. The first transformation should be performed one more time to guarantee that all levels comply with the target queue length. If the order of

the actions is inverted, then the DAG will be expanded into a tree and all subexpressions have to be recomputed as all subtrees are completely expanded affecting the performance and code size due to the extra redundant instructions. We propose an algorithm that deals with the above problems in a unified manner. Our integrated solution reduces the subexpression re-scheduling and minimizes the insertion of spill code.

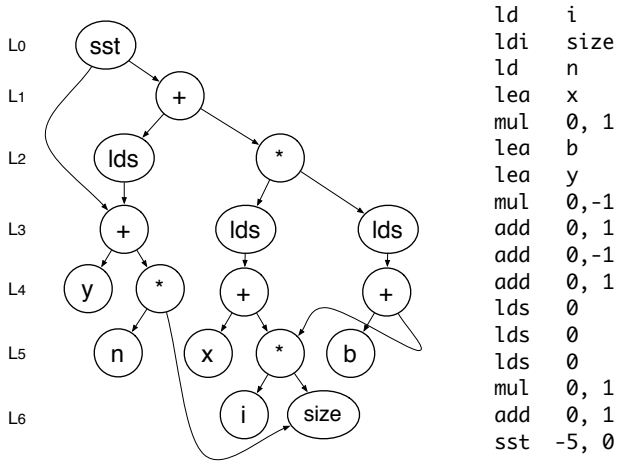


Figure 4.8: Queue length is determined by the width of levels and length of soft edges.

Data Flow Graph Clusterization

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

The algorithm is divided into two phases: labeling, and spill insertion. The labeling phase is in charge of grouping subtrees of the DAG into clusters in order to preserve the

rules of queue computing. For any given DAG or subtree W rooted at node R, the width of W is verified to be smaller than the threshold. The threshold is the size of queue register file for which the compiler should generate constrained code. If the condition is true then all nodes in W are labeled with a unique identifier called the *cluster ID*. In case the width of the DAG exceeds the threshold then the DAG must be recursively partitioned in post-order manner, this is, starting from the left child and then the right child of R. The labeling algorithm is listed in Algorithm 9. To measure the width of a subtree W, the DAG is traversed as a tree and the level with more elements is considered the width of W. Notice that when the DAG rooted at "sst" in Figure 4.8 is traversed as a tree, the maximum width is encountered in level L_5 with six elements corresponding to nodes $\{n, size, x, *, b, *\}$. For simplicity of the explanation, assume the threshold to be equal to 2. Since $SubTree\_Width(sst) > Threshold$, the partitioning algorithm recurses on the left hand side node "+" at L_3. The width of "+" subtree is 2, equal to the threshold. Thus, all nodes belonging to the subtree rooted at "+" node at L_3 are marked with $cluster\_ID = 1$ by line 9 of Algorithm 9. The algorithm continues with the rest of the DAG until all nodes have been traversed and assigned to a cluster. The output of the labeling phase is a labeled DAG as shown in Figure 4.9. Four clusters are shown in the Figure, the first cluster has its root node at (+) node in L_3, the second cluster is rooted at (lds) node in L_2, the third cluster at (lds) node in L_3, and the fourth cluster at (sst) node in L_0.

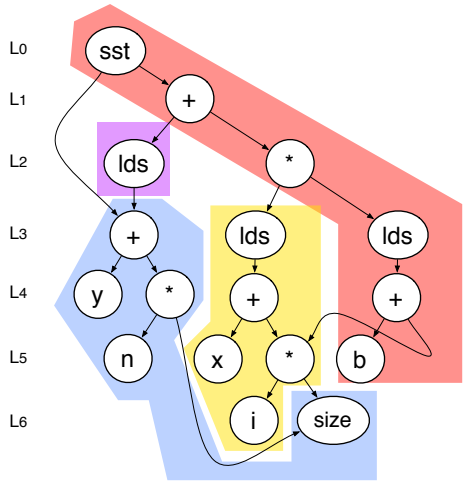


Figure 4.9: Output of the labeling phase of the clusterization algorithm

Input: Threshold
Input: LDAG, *W*
Output: LDAG annotated with labels

```
1  begin
2      root ← W's root node
3      if SubTree_Width (root) > Threshold then
4          lhs ← labelize (root.lhs)
5          if SubTree_Width (root.rhs) > Threshold then
6              rhs ← labelize (root.rhs)
7              root ← Assign_ID_to_node (rhs.id)
8              return root
9          else
10             root.rhs ← Assign_ID_to_subtree (root.rhs);
11             root ← Assign_ID_to_node (root.rhs);
12             return root
13         end
14     else
15         root ← Assign_ID_to_subtree (root);
16         return root
17     end
18 end
```

Algorithm 9: labelize (LDAG *W*)

Spill insertion phase is the second and last phase of the algorithm. The annotated LDAG from the previous phase is processed and a list of N number of clusters (a cluster set) is generated as the output. The input annotated DAG is traversed in post-order manner. For every visited node in the traversal, a set of actions are performed to: (1)

assign the node to the corresponding cluster, (2) insert reload operations to retrieve temporaries computed in a different cluster, (3) insert operations to spill temporaries used by different clusters.

Assigning nodes to the corresponding cluster involves the creation the LDAG data structures, node information, and data dependency edges. Using the queue compiler's LDAG infrastructure [10] allows the clusterization algorithm to be implemented in a clean and simple manner. In terms of memory complexity, the addition of a list of length N is required in the compiler to generate the clusters. The value of N is the number of clusters discovered by the labeling phase.

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

Figure 4.10 shows the generated clusters for the example in Figure 4.9. Four clusters are generated after the spill code is inserted. The gray nodes in the figure represent the nodes that are spilled to memory. Notice that the node (*size*) has two parents in Figure 4.9 but no temporary is generated as it is not a subexpression but a constant known at compile time. The rectangle nodes represent the reload operations needed to retrieve the computed subexpressions from other clusters, variables, and constants. All four clusters in the figure comply with the requirement of not exceeding a queue utilization greater than two. For this example, the penalty of compiling for a queue register file size of two words is the insertion of ten extra instructions: four spills and six reloads.

Algorithm 10 lists the actions performed over the annotated LDAG to generate a set of clusters. As clusters have the same shape as LDAGs, we can use the queue compiler

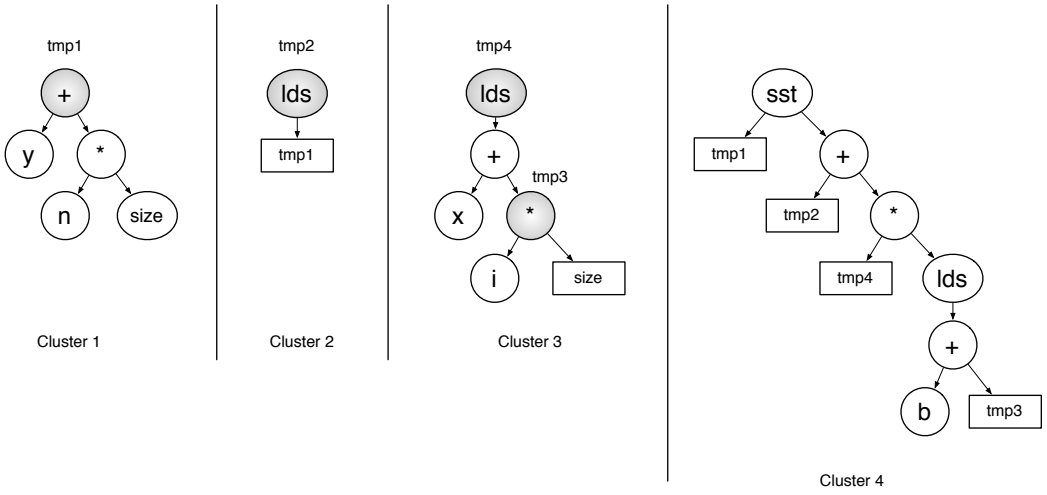


Figure 4.10: Output of the clusterization algorithm. Spill nodes marked in gray circles and reload operations in rectangles.

infrastructure to generate code directly from the cluster set. Each cluster is treated as a LDAG and the code generator [10] calculates the offset references for all instructions, including spill code. Besides from the described clusterization algorithm, the queue compiler internals remained untouched and the compilation flow remains the same as the original compiler.

Clusters are connected to each other by data dependency edges. The order on which the clusters are scheduled is very important to preserve correctness of the program. We build a *cluster dependence graph* (CDG) to facilitate the code generation. The CDG for the above given example is shown in Figure 4.11. At first the cluster 1 must be scheduled for execution, followed by clusters 2 and 3, to finally schedule cluster 4. Some clusters are independent from each other, like clusters 2 and 3, and can be scheduled in any order. In this algorithm we schedule the clusters in the same order as they are discovered by the labeling algorithm. However, we notice here that this may present an opportunity for further optimizations.

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

Algorithm 10: clusterize (node u, LDAG W)

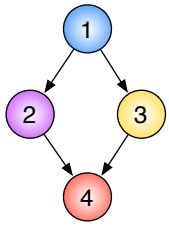


Figure 4.11: Cluster Dependence Graph (CDG)

4.2.4 Evaluation of Queue Register File Constrained Compilation

The primary concern of this study was to analyze how the quality of the generated programs is affected when the program is constrained to run with a limited amount of queue words. We concentrate on three aspects of the output programs: (1) instruction count, (2) critical path, and (3) instruction level parallelism. Instruction count is the number of generated instructions including spill code and reloads. The critical path refers to the height of the program's data flow graph given by the number of queue computation levels. This metric provides a compile-time estimation of the execution time of the program in a parallel queue processor. The instruction level parallelism in a queue system is estimated as the average number of instructions on every computation level of the data flow graph of a program.

The methodology followed to perform the experiments is as follows. We successfully implemented the presented algorithm in the queue compiler infrastructure [10]. The threshold input value for the algorithm is given as an compiler option. For all experiments, the compiler was configured only with the presented clusterization algorithm. No other optimizations are currently available in the queue compiler. We compiled SPEC CINT95 benchmark programs [21] with threshold values of 2, 4, 8, 16, 32, and infinity.

Table 4.3 quantifies the compilation time cost of the presented algorithm. The second column (LOC) shows the lines of C code for the input programs. The third column (Baseline) shows the compilation time with constrained compilation disabled. The rightmost column (Constrained) shows the compilation time taken by the queue compiler with constrained compilation enable with threshold set to 2. This threshold

Table 4.3: Estimation of constrained compilation complexity measured as compile-time for the SPEC CINT 95 benchmark programs with threshold set to two.

Benchmark	LOC	Baseline	Constrained
099.go	28547	9.34s	9.35s
124.m88ksim	17939	9.58s	9.69s
126.gcc	193752	42.67s	43.39s
129.compress	1420	0.37s	0.38s
130.li	6916	3.20s	3.27s
132.ijpeg	27852	8.88s	9.10s
134.perl	23678	6.92s	7.26s
147.vortex	52633	18.73s	19.02s

value is the worst-case configuration for the algorithm as the available queue is only two words. The table demonstrates that the penalty of this optimization negligibly affects the complexity of the queue compiler. The compilation time is the real-time of a dual 3.2 GHz Xeon computer running GNU/Linux 2.6.20. The compiler was bootstrapped with debugging facilities, and no optimizations.

Instruction Count

The most evident effect of the clusterization algorithm in the output code is in the instruction count. Spill code is inserted whenever the width of a level or a soft edge exceeds the threshold value. Table 4.12 shows the normalized instruction count for the benchmark programs for different lengths of queue. The baseline is the programs compiled for infinite resources (INFTY), where the clusterization is not present. We selected various lengths of queue for the following reasons. The most restrictive configuration for a queue processor is a queue length of 2. This configuration estimates the worst case conditions for compilation that may strongly affect the quality of the programs. The other three chosen queue lengths ($threshold = 4, 8, 16$) are values above the average available parallelism in non-optimized SPEC CINT95 programs. The relationship between queue length and available parallelism is that N parallel instructions consume at most $2N$ queue length.

However, peak parallelism and some soft edges are beyond these values and our algorithm finds clusterization opportunities. The last length of queue is set to the infinity to compare the previous constrained configurations against an ideal hardware.

As we expected, the most restrictive queue length configuration incurs in the most substantial insertion of spill code ranging from 3% to 11% more instructions. The clusterization algorithm works on the premise that the width of the data flow graph, or degree of parallelism, must be partitioned in case its queue requirements violate the available queue. Therefore, compiling for a queue length of two queue words forces a large number of partitions of the original data flow graph inserting a substantial amount of spill code. For queue lengths of 4 and 8 the increase in number of instructions is about 2% and 1% respectively. Compiling for a queue length of 4 words exceeds the average queue requirements of SPEC CINT95 programs which is about 3.5 queue words per level. Compiling for queue length of 16, the insertion of spill code is insignificant for most of the programs. These few cases that demand more than 16 queue words are the bursts of peak parallelism and long soft edges.

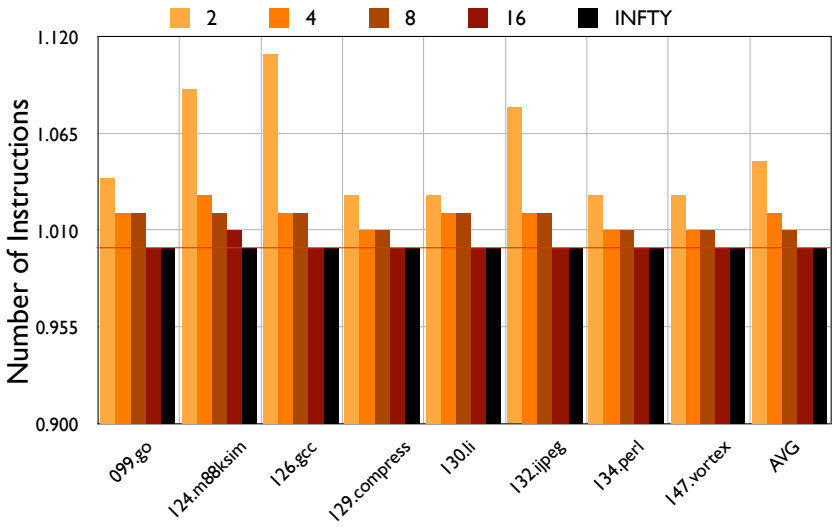


Figure 4.12: Normalized instruction count measurement for different lengths of queue, $threshold = 2, 4, 8, 16, INFTY$.

Spill Code Distribution

We separated the inserted spill code into three components as shown in Figure 4.13: parallelism, soft edges, and reloads. The parallelism accounts for all spill instructions inserted to constrain the width of the data flow graph to the available queue. Soft edges represent all spilled temporaries generated to constrain the soft edges that exceed the available queue. And reloads are the instructions to read the spilled temporaries and uses of shared constants and variables of other clusters. The Figure quantifies the contribution of each component of spill code into the total number of extra instructions for the 124.m88ksim benchmark. Considering only the two components that contribute with spill code (parallelism, and soft edges) and ignoring the reload instructions. Notice that for a queue size of 2 the parallelism component dominates with 89% of extra code. The other 11% is from the soft edges component. As explained above, the parallelism component contributes with most of the code since a large number of partitions must be made for this given configuration and these kind of benchmarks. When a larger queue size configuration is imposed and exceeds the average queue utilization of the compiled program the distribution changes, in average, the parallelism component contributes with 40%, and soft edges component with the remaining 60% of spill code.

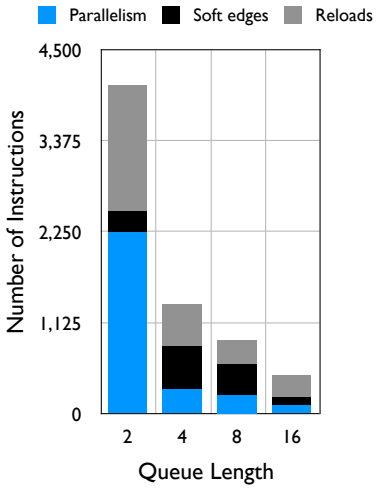


Figure 4.13: Spill code distribution of 124.m88ksim benchmark.

Critical Path

We define the critical path of a program as the number of computation levels in its data flow graph. These levels represent the true data dependencies of the program's data flow graph and the limits of a queue processor. Assuming that all instructions in every level are executed in parallel by the queue processor, the execution time is bounded by the number of levels in the program. We use the critical path to estimate the static execution time of the compiled programs. Since partitioning the data flow graph into clusters increases the number of levels in the data flow graph, we were interested in determining how the static execution time is affected when compiling for a constrained queue register file. Figure 4.14 shows the experimental results for different sizes of queue. For queue sizes of 4, 8, 16 the performance degradation of the static execution time is less than 1% when compared to the unrestricted (INFTY) configuration. Compiling these programs for a queue size of 2, the performance degradation is up to 16% of static execution time. These results demonstrate that constraining the compilation to a queue size value larger than the average queue utilization maintains the critical path almost unaffected as clusterization is needed for only few cases.

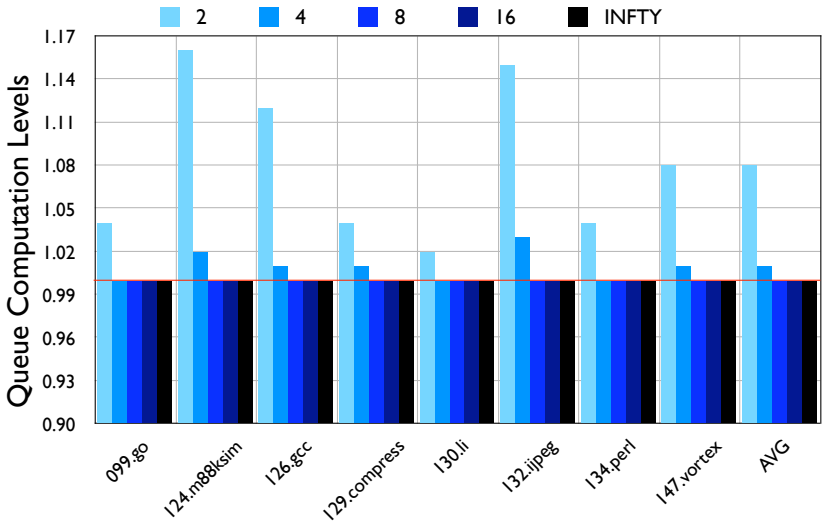


Figure 4.14: Queue computation levels in the programs' data flow graph as an estimation of static execution time.

Instruction Level Parallelism

The instruction level parallelism (ILP) in queue programs is given by the number of instructions per level of the data flow graph. The presented algorithm increases the number of instructions and the number of computation levels, hence affecting the degree of ILP. Figure 4.15 shows the normalized degree of ILP for different configurations of queue size using infinite queue as the baseline. From the Figure two cases are observed, decrease of ILP and raise of ILP. The former case happens when the number of extra computation levels inserted as a consequence of the data flow partitioning is proportionally larger than the extra spill code. Therefore, it results in less instructions per level when compared to the baseline. This situation is observed clearly for the queue size of 2 where the critical path is affected more than the instruction count. The latter case occurs when the number of extra instructions is proportionally larger than the extra computation levels. In the Figure, the compilation for queue lengths of 4 and 8 show this behavior since the partitioning of the data flow graph is modest compared to the insertion of spill code. For a queue size configuration of 16 the ILP is about the same as the baseline. For this case the clusterization algorithm finds only few levels and soft edges exceeding the threshold value. Although there is a raise of ILP for some of the queue size configurations, this parallelism represents artificial ILP introduced by the spill code.

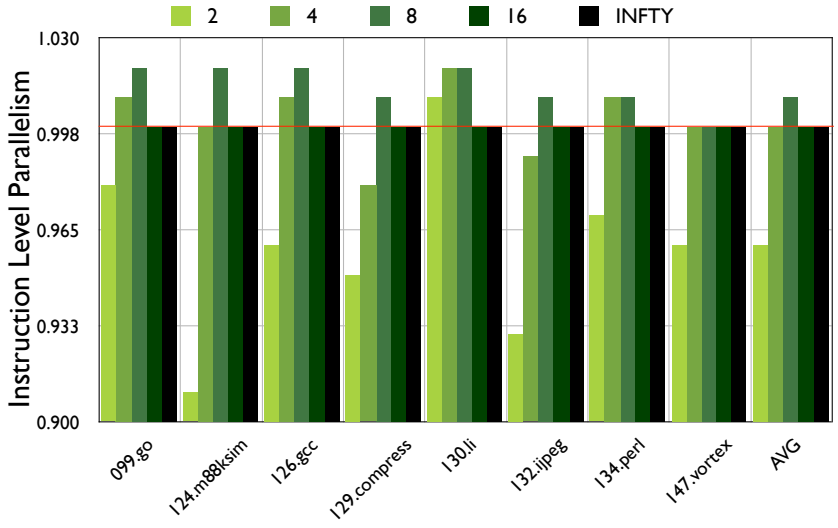


Figure 4.15: Degree of instruction level parallelism for constrained compilation for different sizes of queue register file.

4.2.5 QueueCore Processor Evaluation

When targeting a real hardware, the queue compiler is configured to generate code for the QueueCore [2] processor, an embedded 32-bit queue-based ILP processor. The instructions of QueueCore are 16-bit wide. QueueCore is capable of executing instructions in parallel. A special unit, called the Queue Computation Unit [2] bookkeeps `QH` and `QT` references. This mechanism decides the correct source operands and the destination for every instruction. The correct location of operands is found by adding the offset reference of the instructions to the current location of `QH` reference. For all experiments the queue compiler was configured only with the presented queue-length optimization tuned for a queue length of 32 to match the size of the QueueCore's queue register file size.

Furthermore, to highlight the features of our queue-based processor and the queue computing in general, we compare our results with the results obtained in [99, 100] for a 8-way universal issue machine. In [99], the authors propose a model to find an optimal compromise between region-enlarging optimizations that affects code size and global scheduling that affects ILP. We concentrate solely on their experimental results as they give a realistic estimation on the characteristics of the code compiled and optimized for a traditional register machine.

Code Size

Our algorithm effectively deals with the statements in a program that demand a large number of queue words. Table 4.4 shows the number of extra spill instructions generated by the queue-length optimization for the QueueCore processor. The second column shows the total number of instructions generated including the spill code. From these results we observe that the optimization presented in this paper increases the length of the programs by about 1.46%. The program which presents less extra spill code is 147.vortex with the addition of 0.66% more instructions. And the program with more spill code is 130.li with 2.23% more instructions. The third column shows the number of instructions generated for a conventional 8-way issue machine using global ILP scheduling techniques [99]. The queue compiler generates about 22% larger programs, in terms of number of instructions, than the optimizing compiler [17, 34] for the 8-way universal machine in [99]. There is a room for improvement in our queue code since our compiler does not perform any

Table 4.4: Extra spill instructions and total number of instructions for QueueCore and a conventional 8-way issue machine.

Benchmark	Spill insn	Total insn	8-way issue
099.go	1462	98071	66436
124.m88ksim	882	45542	33965
126.gcc	7700	418963	387408
129.compress	16	2047	1626
130.li	462	21143	14530
132.ijpg	863	56112	41080
134.perl	955	90497	73897
147.vortex	1198	182317	155741

classical optimizations that remove redundant computations [20, 59, 92]. The compact instruction set of the QueueCore allow programs to present high code density. Figure 4.16 compares the code size of the text segment of the generated programs for QueueCore and the 8-way issue machine. We assume two byte instructions for the QueueCore and four byte instructions for a typical register-based multiple-issue machine [36]. The results are normalized to one using the QueueCore program without queue-length optimization as the baseline. Our optimization increases the code size of QueueCore programs less than 2%. Compared to the conventional register machine code, ours is from 27% to 47% denser.

Instruction Level Parallelism

Queue computing relies on the level-order scheduling for the generation of correct programs. The level-order scheduling naturally exposes all available parallelism in an expression. First, we analyze the effects of the presented optimization on ILP. We measured the compile-time extracted parallelism of the queue compiler without and with the queue length optimization presented in this paper. We also compared our code against the optimized code for the 8-way universal machine. In [99], the compiled code is classically optimized and scheduled using a global scheduling technique aided by tail duplication. Figure 4.17 shows the parallelism exposed by the compiler for the 8-way

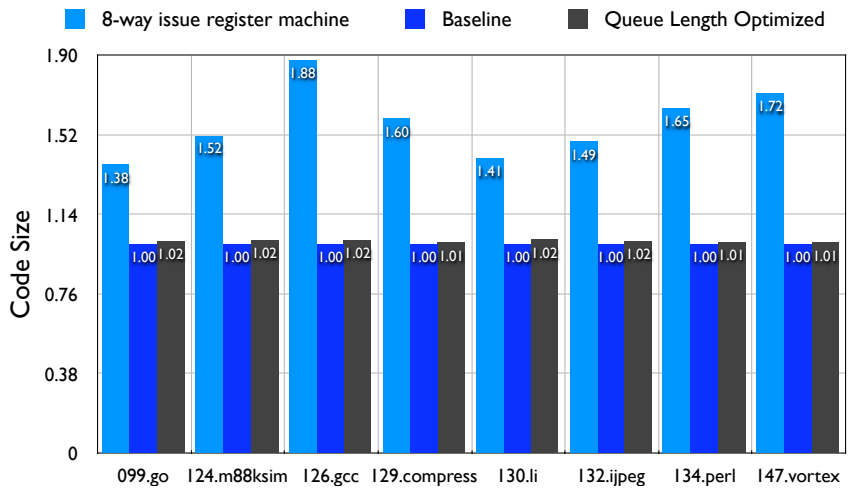


Figure 4.16: Normalized size of the text segment for a conventional register machine and the QueueCore.

universal machine, for the QueueCore without queue-length optimization, and for the QueueCore with queue-length optimization enabled. From the graph we observed that the impact of the queue length optimization for QueueCore increases ILP for all programs about 1.3%. The raise represents artificial ILP given by the insertion of extra spill code. Compared to the ILP compiler for a 8-way universal machine, the level-order scheduling of the non-optimizing queue compiler can extract about 13.67% more parallelism.

4.2.6 Conclusion

In this Section we developed a queue register file size-aware compiler that effectively handles the queue utilization by: controlling the degree of parallelism, or width of computation levels; and limiting the soft edges, or lifetime of reaching definitions of variables. The proposed algorithm partitions the data flow graph of programs into clusters in such a way that spill code and redundant computation insertion is minimized. Our algorithm negligibly affects the complexity of the queue compiler infrastructure. To measure the effectiveness and overall effects of our proposal, we compiled SPEC CINT95 applications for different sizes of queue register file. For the most restrictive queue size with two words the instruction count raises up to 11%, the critical path increases up to 16%, and parallelism decreases up to 7% when compared to code utilizing infinite

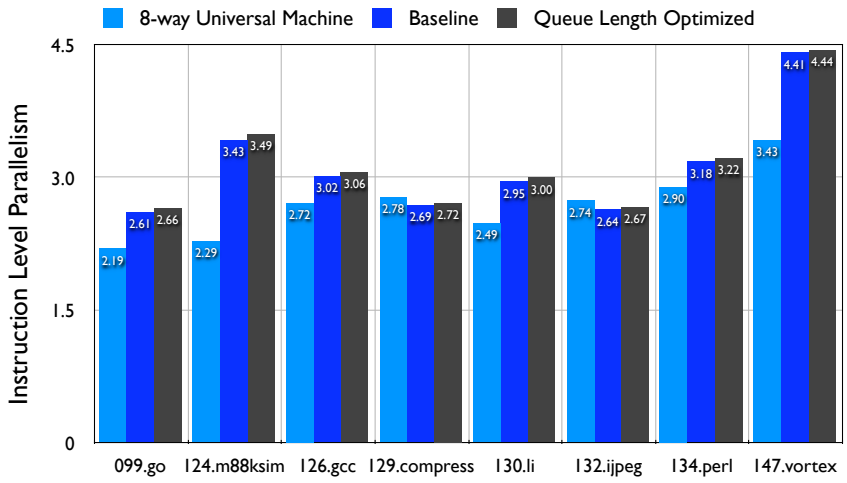


Figure 4.17: Exposed instruction level parallelism by an ILP compiler for a conventional multiple issue machine, and for the QueueCore without and with queue-length optimization.

queue. For a queue size of four words, which is a number greater than the average queue utilization and parallelism for these kind of applications, the instruction count increases about 2%, the critical path increases about 1%, and available parallelism remains about the same. For queue sizes of eight and sixteen words the increase in number of instructions, critical path, and available parallelism is insignificant. Our experiments show that queue lengths from 8 to 16 words suffice to execute unoptimized integer applications and the code characteristics are nearly the same as code compiled for infinite resources.

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

4.3 Classic Optimization: Common Subexpression Elimination

In this section we introduce some problems encountered when optimizing code for queue machines. Common-subexpression elimination (CSE) is a widely used optimization to improve execution time. We analyze how this optimization affects the characteristics of queue programs. We have found that in average, 28% of instructions are eliminated, and 15% of the critical path is reduced. We determine how enlarging the scope of compilation from expressions to basic blocks affects the distribution of offsetted instructions and therefore producing extra pressure on the queue utilization.

The queue compiler generates correct programs by traversing the data dependency graph of every expression and calculating the offset references to access their operands. This expression-by-expression compilation scheme generates correct code but ignores any opportunity to generate high performance code. Common-subexpression elimination (CSE) is a classical compiler optimization [4, 65] that reduces execution time by removing redundant computations. The program is analyzed for occurrences of identical computations and replaces the computations for the uses of a temporary that holds the computed value. We implemented the CSE optimization in the queue compiler that builds a DAG representation of the basic blocks rather than statements. As the compilation scope is larger, the characteristics of queue programs vary and may impose new requirements on the compiler, underlying hardware, and instruction set. The contributions of this section are as follows:

- Given the unique features of the queue computation model, we introduce the ways common-subexpression elimination affects the quality and characteristics of programs.
- We quantify the effects of CSE optimization in number of generated instructions, reduction in computation steps, and distribution of offsetted instructions for a set of scalar and numerical benchmark programs. Furthermore, we give a specific example on how the CSE transformation affects the binaries of QueueCore processor [2].

4.3.1 Implementation of CSE in Queue Compiler

The fundamental idea of queue code generation is to produce a level-order scheduling of the DAGs and correctly calculate the offset reference values of every instruction. This work is part of the initiative to develop an optimizing queue compiler. Therefore, we must enlarge the sections of code where to look for optimization opportunities rather than compiling statement by statement as in the baseline queue compiler. Figure 4.18 shows the block diagram of the queue compiler including the local common subexpression elimination phase depicted by the shaded block.

We implemented the common subexpression elimination pass after instruction scheduling since the scheduler is tightly related with the offset calculation of instructions [10], and the offset calculation algorithm works for single statements. The output of CSE is a DAG representing all statements within a basic block and may have multiple roots. However, the expressiveness of our QIR representation eases the implementation of code transformations such as CSE. The CSE pass is based a well known local CSE algorithm [65].

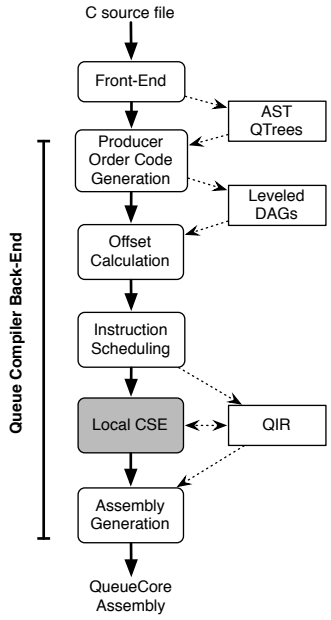


Figure 4.18: Queue Compiler Block Diagram

4.3.2 Effects of CSE on Queue Programs

Common subexpression elimination identifies computations that are performed more than once and generates an equivalent program by eliminating the second and later occurrences of them. The goal of common-subexpression elimination is to improve execution time by replacing redundant operations with uses of stored temporaries. The effect of classical optimizations on conventional architectures has been studied [60]. However, these findings cannot be generalized to a queue-based instruction set computer since the underlying computation model is substantially different. To illustrate the implications of common-subexpression elimination in queue programs consider the following statements constitute a basic block:

$$
\begin{aligned}
S_1: &\quad x = (a + b)/(c - d) \\
S_2: &\quad y = 1 \\
S_3: &\quad w = 2 \\
S_4: &\quad z = a + b
\end{aligned}
$$

The queue compiler's unoptimized representation of the basic block is shown in Figure 4.19(a). The semantics of the queue computation model allow each level in the data flow graph of the basic block to be processed in parallel. Therefore, the execution time for a parallel queue machine can be estimated by the number of levels in the flow graph. The given example has eight levels, from L_0 to L_7, and sixteen operations. The available parallelism is 16/8. Notice that the operation "$a + b$" is computed twice in the block and the second occurrence can be eliminated by the common-subexpression elimination algorithm. The resulting optimized data flow graph is shown in Figure 4.19(b). The number of levels has been reduced from eight to six, and the number of instructions to thirteen. For this particular example, the available parallelism has been raised to 13/6.

This transformation improves execution time for queue machines not only from code removal, but also by reducing the number of levels in the data flow graph. Reducing both number of instructions and execution levels, the available parallelism might remain about the same as in the original program, unlike conventional machines where available parallelism is typically reduced [60]. However, optimizing code for a producer order

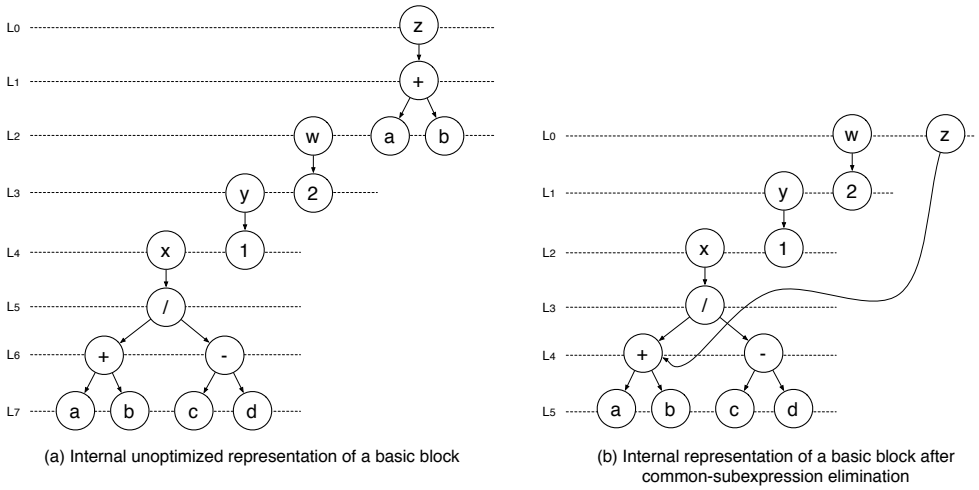


Figure 4.19: Queue compiler's representation of basic block. (a) Original representation. (b) After common-subexpression elimination the redundant computation is removed, the number of execution levels decreases, and an edge is stretched.

instruction set [2] raises some problems that the compiler must solve to correctly execute the program on a system determined by the hardware and instruction set limitations, for example, given a 1-offset instruction set the compiler must transform the program DAGs in such a way that 2-offset instructions are able to execute using a single offset instruction set [14]. Eliminating subexpressions and substituting them with uses of already computed temporaries enlarges the edges in the data flow graph as shown in Figure 4.19(b). The edge with source at node "z" and sink in "$a + b$" was stretched from one level in the unoptimized program to four levels in the optimized version.

A producer order instruction set such as the one supported by the QueueCore processor, allows operands to be read from any location in the queue register file specified as an explicit offset reference with respect of `QH`. Stretched edges signify that temporary values must be kept alive longer, therefore the offset values to access these temporaries are larger. If a temporary is kept alive for too long we may exhaust the available queue. Assuming an infinite queue, if the offset reference to access an operand is too long the bits to encode the operand in the instruction may be insufficient. Both cases can be effectively solved by giving to the compiler knowledge about the size of the queue and the characteristics of the target instruction set [15].

Previous research on unoptimized queue programs [10] shown that the frequency of instructions that need one or two operands to be read away from `QH` is about 10% of total offsetted instructions, the other 90% read their operands directly from `QH`. As the common-subexpression elimination stretches the edges in the data flow graph of the programs, it changes the characteristics of the queue programs by affecting the distribution of offsetted instructions.

4.3.3 Evaluation

For all experiments, we consider the unoptimized code produced by the queue compiler as the baseline [10]. We enabled the common-subexpression elimination in the queue compiler to measure its effect on the output code. The output code is a generic producer order instruction set where every instruction can explicitly specify at most two offset references to read its operands [12]. We selected twelve benchmark programs, eight scalar programs from SPEC CINT95 suite [21] and four numerical intensive benchmarks including 8-radix fast fourier transform, livermore loops, linpack, and equake.

We start our analysis by measuring the expected reduction in number of instructions. Then we analyze the exclusive effects of common-subexpression elimination on the queue computation model. First, we analyze how this transformation reduces the execution time by reducing the computation levels in the data dependence graph and we estimate the overall performance gain. Second, we explore how the elimination of redundant instructions, and the reduction of computation levels affect the available parallelism in the programs. Third and finally, we analyze how the offsetted instruction distribution gets affected and we give a concrete estimation on how this change affects the binaries of the 16-bit embedded QueueCore processor.

Redundancy Elimination

Figure 4.20 shows the reduction in number of instructions for all the benchmark programs after common-subexpression elimination is performed. The results are normalized using the unoptimized output of the compiler as the baseline. For all programs there is a reduction in number of instructions between 9% to 55%. The effectiveness of common-subexpression elimination algorithm depends on the basic block characteristics of the

input program. Large basic blocks present more opportunities to find and eliminate common subexpressions than small basic blocks. In average, the common-subexpression elimination pass of the queue compiler, reduces the number of instructions by 28%.

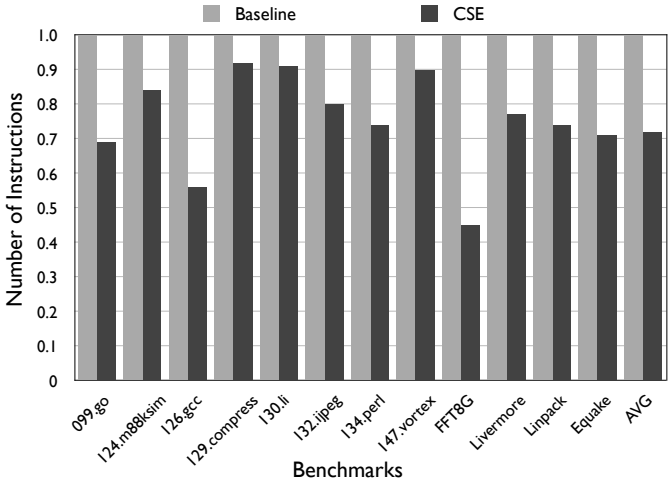


Figure 4.20: Instruction count reduction

Execution Time

The queue computation model requires a level-order scheduling of the data flow graph to correctly evaluate expressions. The level-order traversal breaks the data flow graph of any expression into groups of independent instructions that can be safely executed in parallel. Visually, these groups of independent instructions are the levels L_n of the data flow graph. For a parallel queue machine, the execution time is bounded by the total number of levels in the program. On the queue computation model the common-subexpression elimination may reduce the number of levels of the data flow graph, therefore reducing execution time. The graph in Figure 4.21 shows the percentage of levels reduced by the common-subexpression elimination. The reduction in levels ranges from 3% to 55%, or speeding up the program from 1.03 to 1.81 times. In average, the levels are reduced by 15% speeding up the queue programs by 1.17 times.

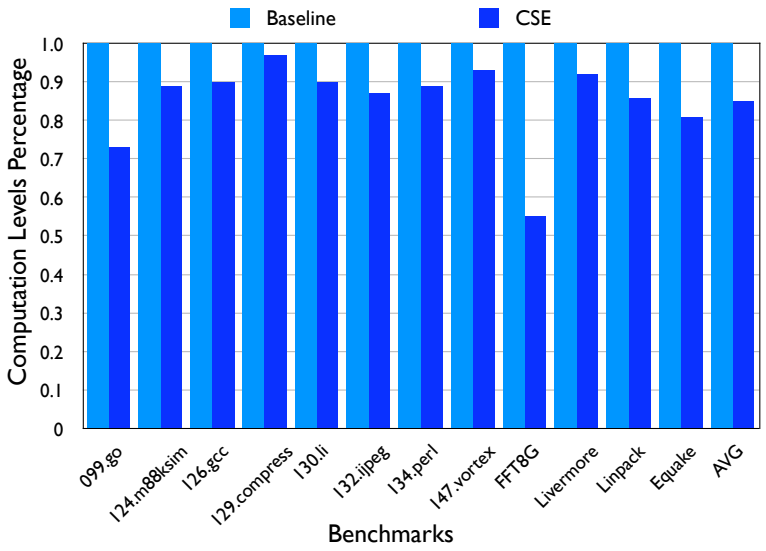


Figure 4.21: Computation levels reduction.

Available ILP

The instruction level parallelism (ILP) in queue programs is given by the number of instructions on each level of the data flow graph. In this study we consider the compile-time available parallelism rather than run-time parallelism since our target is to quantify the effects of the common-subexpression elimination in the output programs. Figure 4.22 shows the compile-time extracted parallelism for the benchmark programs. Since the elimination of common-subexpressions from queue programs affects the number of instructions and the number of levels, the available parallelism depends on the proportion on which instructions and levels are reduced. For applications where level elimination is proportionally more than instruction elimination the ILP raises. In contrast, applications where instruction elimination is proportionally more than level elimination the ILP decreases. However, from the experimental results we can observe that the available ILP remains about the same after common-subexpression elimination on queue programs is performed.

Producer Order Offsetted Instruction Distribution

As discussed in Section 4.3.2, the elimination of subexpressions enlarge the dependency edges in the data flow graph. The length of edges is directly related to the offset references

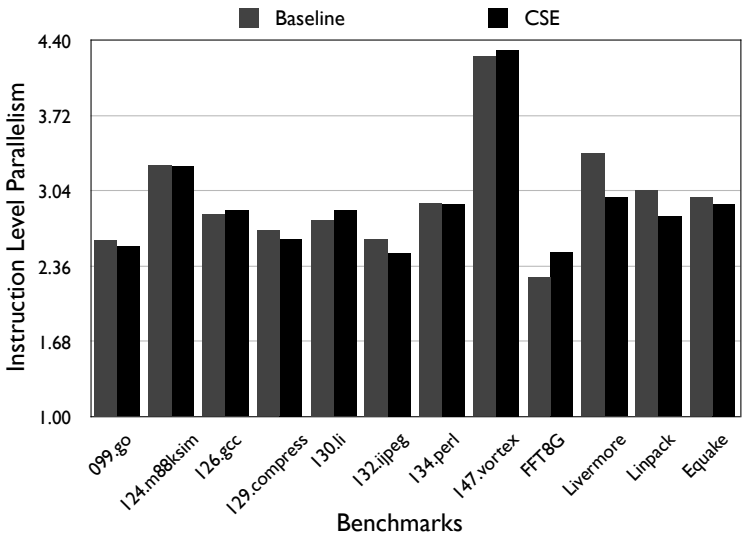


Figure 4.22: Instruction level parallelism

in a producer order queue instruction set. Knowing the offset characteristics of programs is crucial to determine the hardware and instruction set features for a queue processor. Figure 4.23 shows the distribution of offsetted instructions for a generic producer order queue-based instruction set. The programs are separated into two groups: scalar and numerical benchmarks. For each benchmark, two measurements are presented: non optimized distribution is represented by the benchmark name in the x-axis, and CSE optimized distribution represented by the benchmark name with *CSE* suffix in the x-axis. For the unoptimized scalar benchmarks, the frequency of 2-offset and 1-offset instructions is about 10% as suggested by our previous research [10]. After common-subexpression elimination, the distribution of 2-offset instruction changes from less than 1% to 3%, and 1-offset instruction density changes from 9% to 28%. Together, 1-offset and 2-offset instructions represent the 31% of offsetted instructions of scalar programs. For the numerical benchmarks the change is more significant. After common-subexpression elimination 2-offset instructions represent 12% and 1-offset instructions represent 44% of all offsetted instructions, together, 56% of all offsetted instructions in programs.

In [2], we implemented a producer order queue-based 32-bit processor, the QueueCore. The QueueCore features 16-bit instructions to favor code density and hardware simplicity. The bit limitations in the instruction set of the QueueCore allow at most one offset reference to be encoded. This feature restricts the QueueCore processor to execute 0-

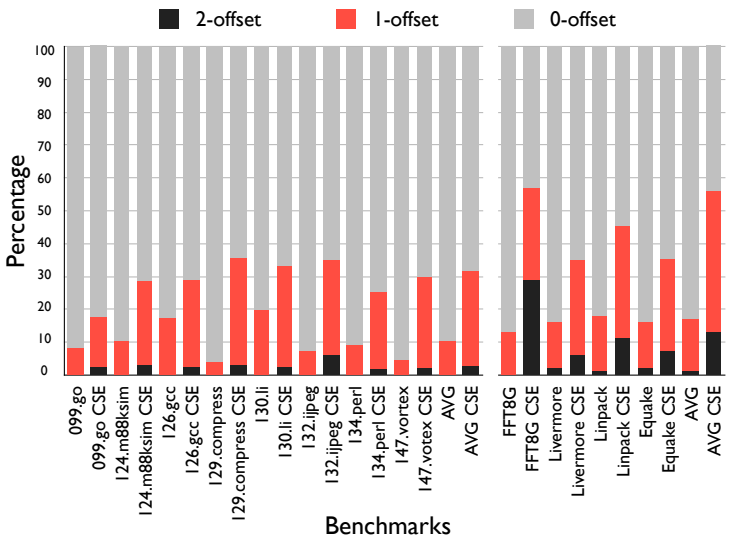


Figure 4.23: Offsetted instructions distribution for scalar and numerical benchmarks.

offset and 1-offset instructions, excluding 2-offset instructions. The queue compiler is responsible of constraining the programs to use the 1-offset QueueCore instruction set. Compiling for QueueCore requires the conversion of 2-offset instructions into 1-offset instructions and an additional special instruction called `dup` instruction [10]. Constraining the programs to use at most 1-offset reference incurs in a penalty of inserting an extra `dup` instruction for every converted 2-offset instruction. Although common-subexpression elimination increases the amount of 2-offset instructions, and an additional `dup` instruction is inserted for QueueCore binaries, the generated programs are in average 24% shorter than unoptimized code. These results suggest that a 1-offset instruction set computer such as the QueueCore processor, is a good architectural decision that provides flexibility and bit-reduced instructions.

4.3.4 Conclusion

This section presents a quantitative analysis on the effects of common-subexpression elimination in queue machines. Queue programs benefit from this optimization in the same manner as traditional architectures. However, there are some inherent characteristics of queue computation model that push the effects of common-subexpression elimination further. Our results show that for a set of scalar and numerical programs the number of

instructions is reduced by 28%. As consequence of CSE, the edges in the data flow graph are enlarged, and the distribution of offsetted instructions in queue program changes. For scalar programs, the sum of 2-offset and 1-offset instruction raises from 10% to 32%. For numerical programs from 16% to 56%. Although the offsetted instruction raises about three times the number of 2-offset instruction remains low.

4.4 ILP Optimization: Statement Merging Transformation

Statement merging transformation reorders the instructions of a sequential program in such a way that all independent instructions from different statements are in the same level an can be executed in parallel. This phase makes a dependency 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 dependency analysis finds conflicting memory accesses 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.

Figure 4.24(a) shows a program with three statements S_1, S_2, S_3. The original sequential scheduling of this program is driven by a level-order scheduling as shown in Figure 4.24(b). When the statement merging transformation is applied to this program a dependency analysis reveals a flow dependency for variable x in S_1, S_2 in levels L_4, L_3. Instructions from S_2 can be moved one level down and the flow dependency on variable x is kept as long the store to memory happens before the load. Statement S_3 is independent from the previous statements, this condition allows S_3 to be pushed to the bottom of the data flow graph. Figure 4.24(c) shows the DFG for the sample program after the statement merging transformation. For this example, the number of levels in the DFG has been reduced from seven to five.

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 all the available parallelism [93] in basic block boundaries. 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.

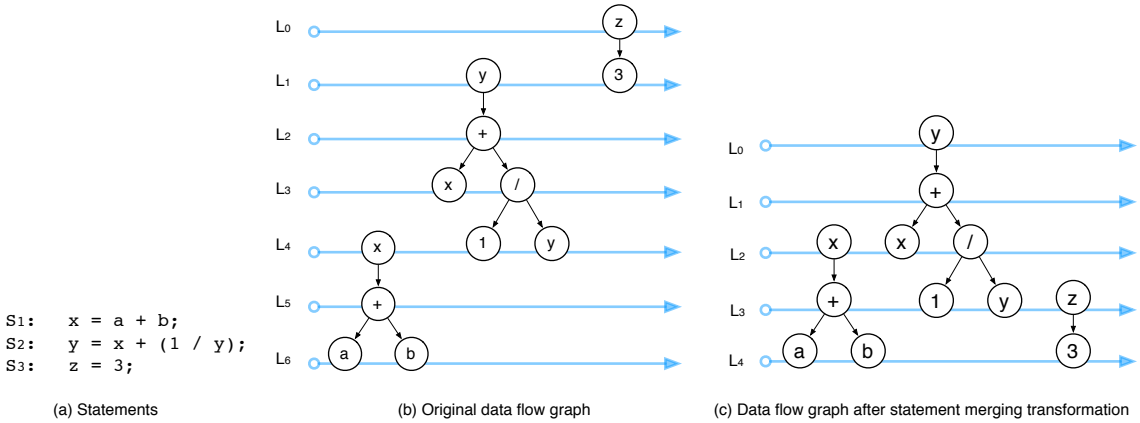


Figure 4.24: Statement merging transformation

4.4.1 Algorithm

The algorithm, shown in Algorithm 11, scans all statements in a basic block. For each statement i, a dependency and alias analysis is performed against the following statement, j. If a data dependency exists between elements of the two statements, the `dependency_and_alias ()` function in Line 11 returns a tuple that contains the respective levels of each statement where the dependency exists. Control flow statements such as conditional, unconditional, and function calls are marked as `NON_MERGEABLE` and the algorithm skips to the next statement.

The basic block in Figure 4.25(a) consists of two statements S1, S2. Data dependency analysis returns the tuple (L_4, L_6) as there is a *read-after-write* (RAW) data dependency in "y". Notice that although the dependency in S_2 exists in level L_7, the returned level is L_6 to maintain data flow correctness that all dependencies grow downwards. The next step is to compute, for each statement, the distance from the conflicting level to the statement's root level $(a1, a2)$, and the distance from the conflicting level to the statement's sink level $(b1, b2)$. The distance to the sink level is the number of levels between the conflicting level, including itself, and the sink level. The distance to the root level is the number of levels between the conflicting level, excluding it, to the sink level. For this example $\{a1, b1\} = \{0, 5\}$, and $\{a2, b2\} = \{3, 2\}$. Figure 4.25(b) shows the conflicting levels as shaded levels (L4, L6), and the diamond and bold arrows show the distance between the conflicting and the root/sink levels. With this information the two statements are merged

into a new one and substituted in the instruction list.

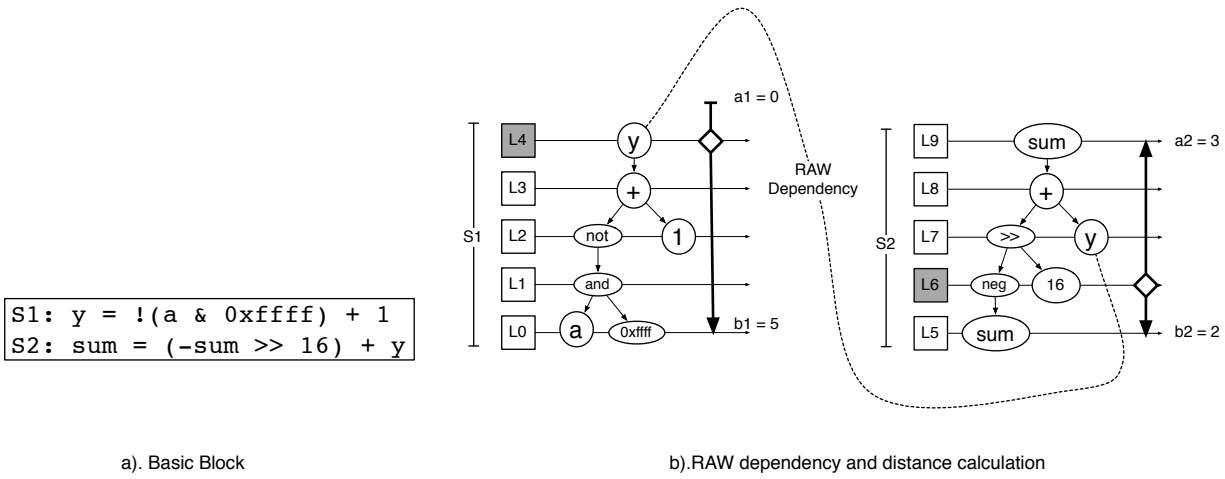


Figure 4.25: Statement merging example.

Input: Block's instruction list, B

```
1  begin
2      forall statements i in list B do
3          j ← i.next
4          if i or j is NON_MERGEABLE then
5              continue;
6          else
7              (L_i, L_j) ← dependency_and_alias (i, j);
8              if (L_i, L_j) ≠ NIL then
9                  a1 ← distance (L_i− root_level (i))
10                 b1 ← distance (L_i− root_level (i))
11                 a2 ← distance (L_j− sink_level (j))
12                 b2 ← distance (L_j− sink_level (j))
13                 S_new ← stmt_merge (i, j, a1, b1, a2, b2);
14                 replace (i, j, S_new, B)
15             end
16         end
17     end
18 end
```

Algorithm 11: stmt_merge (B)

If the tuple exists then the pair of statements are merged by the `merge ()` algorithm. The merging algorithm is shown in Algorithm 12. It takes as input the two statements and the computed distances $a1, b1, a2, b2$. A new statement with height $MAX(a1, a2) + MAX(b1, b2) = 5 + 3 = 8$ is created. The for loop in Algorithm 12 performs a level-order traversal over the new statement and fills every level with the corresponding levels of

the two input statements. It uses two flags (triggers) to determine which levels of the old statements correspond the levels of the new statement. The result of merging the statements in Figure 4.25 is shown in Figure 4.26.

```
   Input: Statement, S1, S2
   Input: Height, a1, b1, a2, b2
   Output: Statement, S_new
 1 begin
 2 |  maxdepth ← MAX(a1, a2) + MAX(b1, b2);
 3 |  trigger1 ← MAX(b1, b2) - b1;
 4 |  trigger2 ← MAX(b1, b2) - b2;
 5 |  S_new ← newstmt();
 6 |  for i = 0 to i < maxdepth do
 7 |  |  L_new ← newlevel();
 8 |  |  if trigger1 ≤ i then
 9 |  |  |  L_new.append ← S1.level(i).insns;
10 |  |  end
11 |  |  if trigger2 ≤ i then
12 |  |  |  L_new.append ← S2.level(i).insns;
13 |  |  end
14 |  |  S_new.append ← Lnew
15 |  end
16 |  return S_new
17 end
```

Algorithm 12: merge (S1, S2, a1, b1, a2, b2)

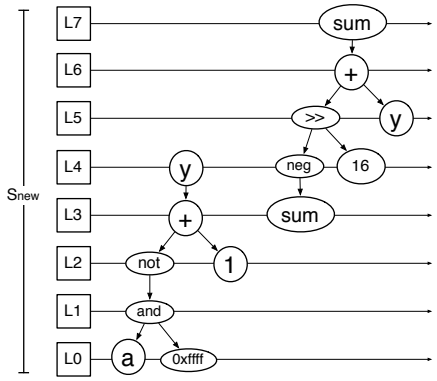


Figure 4.26: Merged statement with a height of 5+3 = 8

4.4.2 Evaluation

Figure 4.27 shows the improvement of ILP by using the statement merging transformation for a set of scalar and numerical programs. The exposed parallelism increases with statement merging by factors ranging from 4.11 to 1.08 times. The effectiveness of this transformation depends on the characteristics of the basic blocks of the compiled program. The improvement on programs with small basic blocks is limited like in scalar programs from SPEC: 099.go, 126.gcc, and 129.compress. Programs with larger blocks allow the statement merging to work most effectively like in the case of fft8g.c with a gain factor of 4.11 times. This program contains very large unrolled loop bodies without control flow. Statement merging is a code motion transformation and does not insert or eliminate instructions.

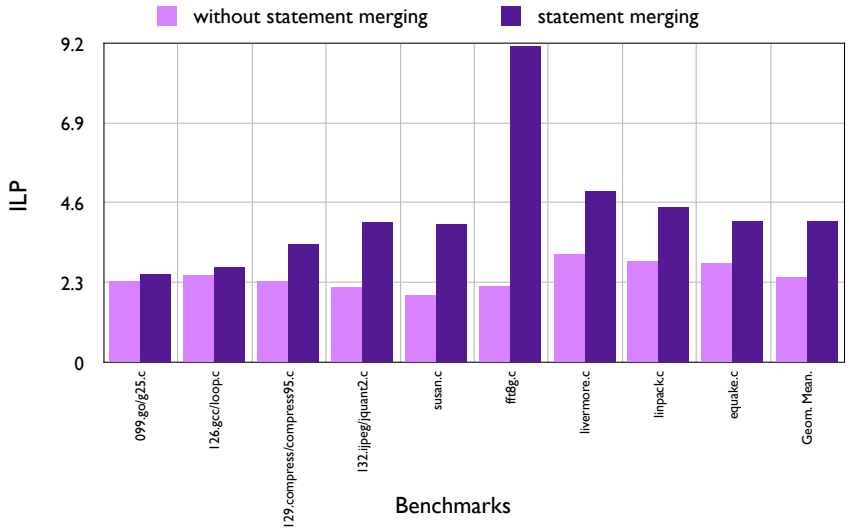


Figure 4.27: Effects of statement merging transformation on compile-time ILP

As it has been discussed in Section 4.2, parallelism is tightly related to queue utilization. Figure 4.28 shows the peak queue utilization of the programs. In other words, this graph shows how statement merging transformation, by increasing parallelism, it also increases the queue utilization of the programs. The extra pressure in queue utilization depends on how wide the new merged statements become. For scalar code the raise in queue utilization ranges from 1.44 to 3.46 times. And for numerical code with larger blocks and fat statements the queue utilization pressure increases up to 18.2 times more.

However, using the optimization presented in Section 4.2 to control queue utilization this peak parallelism can be controlled by the insertion of very small amount of extra code.

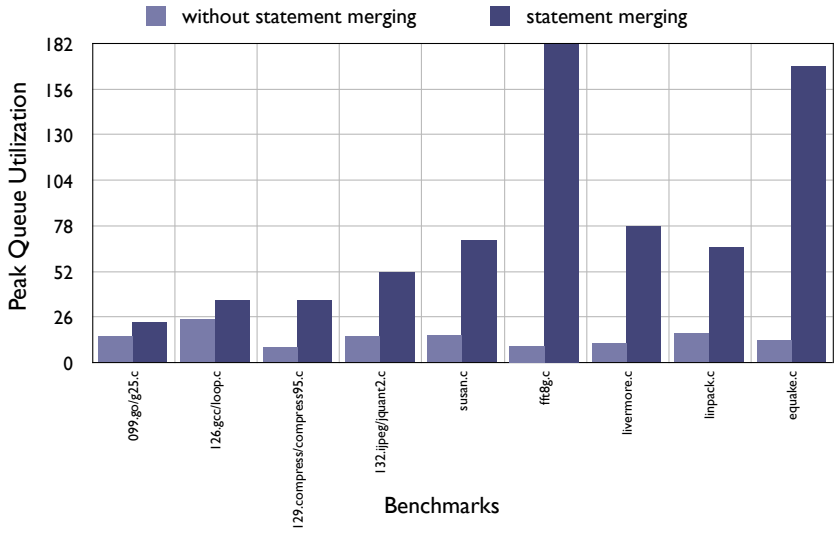


Figure 4.28: Queue utilization on peak parallelism

4.4.3 Conclusion

In this section we presented the *statement merging* transformation that increases compile-time parallelism by combining independent statements in the same level. The idea behind this transformation is to maximize the benefits of the level-order scheduling on which queue programs are generated. We have presented an algorithm implemented in the queue compiler framework that achieves statement merging. As shown by the experimental evaluation, this transformation extracts, in average, about 1.66 times more parallelism than the baseline code generation. However, raising parallelism is associated with extra pressure in the queue utilization.

Chapter 5

Queue Allocation: reducing memory traffic in producer order queue machines

The speed disparity between processors and main-memory is one of primary concerns in modern computer systems as it directly affects performance. Conventional architectures use a set of high-speed random access registers for holding values and avoiding long-latency memory accesses. On the other hand, queue processors employ a different arrangement of registers organized as a first-in first-out queue, with well established rules for accessing its elements where all reads are performed at the head of the queue and all writes at the tail. Therefore, conventional register allocation techniques for register machines cannot be applied to the queue computation model. In this chapter we propose a technique for reducing memory traffic and controlling queue utilization and instruction complexity by the insertion of few instructions that copy and propagate data. To analyze the effectiveness of our method, we implement a novel compiler algorithm in the Queue Compiler and measured the load/store instruction reduction for a set of benchmark programs. From our experimental results we observe that our technique effectively reduces the load/store instructions by an average of 20%, with a small overhead of around 6.47% and 1.69% extra data propagation instructions.

Main-memory technology has been unable to improve at the pace of microprocessor technology. This performance gap that keeps increasing is known as the *memory wall* [97].

Processors employ a hierarchy of high-speed memory organized as registers and cache memories to reduce the long-latency accesses to main-memory and thus achieving high-performance. Since registers are limited, the problem of allocating data items to registers such that memory traffic is minimized has attracted the attention of many researchers [16, 18, 9, 94]. From the observation that subroutine calls are a common and expensive operation due to registers save and restore, interprocedural register allocation techniques have also been proposed [74, 72].

Register allocation for stack-machines is fundamentally different than conventional register machines. Stack-machines have a different arrangement and rules for accessing the registers. To much lesser extent than conventional register machines, register allocation for stack machines has been studied locally to basic blocks [45], across block boundaries [61], and globally [77]. A straight-forward stack machine implements the stack in memory, therefore for every item accessed in the stack a memory access is required. The idea of a stack cache is to keep the stack items, or part of it, in hardware registers to improve performance [22]. PicoJava processor uses a stack cache of 64 entries implemented in hardware as a circular register file to unleash high performance [62]. To overcome the overhead of stack manipulation instructions and the serialization created at the top of the stack [91] a technique called instruction folding [73] allows the stack cache to be accessed as a random access register file and instructions to be parallelized as in RISC machines.

This problem has not been studied for the queue computation model. The primary objective of this chapter is to introduce a new method and the corresponding efficient compiler techniques to reduce memory traffic in queue programs to increase performance. We identify the main causes that produce memory traffic overhead in queue processors, we discuss the drawbacks of the shared main memory communication method, and propose the new technique that reuses data in the queue and reduces memory operations within blocks and across block boundaries. The main principle of the proposed general method for queue processors with a single queue is to copy data that is consumed more than once and keep it in the queue long enough to reach its uses without incurring in excessive pressure on the queue register file and the instruction's offset references.

5.1 Shared Main Memory Communication Method

A straightforward method to solve the long offset references and its associated problems is to employ main memory for data communication. The main idea is to control the maximum size of offsets and therefore the queue utilization. Whenever an offset reference spans more than given threshold the datum is spilled to memory and filled back to the queue when a use is reached. This mechanism guarantees that queue utilization will never exceed certain limit and the offset references never violate the instruction set constraints.

The random access nature of the memory alleviates the problem of long offset references at the cost of increasing the memory traffic. A more sophisticated implementation that avoids memory might include an on-chip high speed registers to hold the spilled values. However, the addition of such functionality would change the nature of a queue-based processor into a hybrid queue-register model.

We identify two cases when controlling the offset references by shared memory communication method generates programs with heavy memory traffic. The first case is inside basic blocks and the second case is across basic blocks.

5.1.1 Intra-block Communication

Depending on the compilation scope, the compiler reduces the basic block from trees into expression DAGs or it may generate a larger DAG covering all statements in the basic block. Figure 5.1(a) shows a basic block consisting of three statements. The program listed in Figure 5.1(b) is the result of compiling the basic block on statement basis. Redundancies are eliminated within each statement. For example, the "sum" variable is loaded once but used twice by the *rsh* and *sub* instruction. An offset reference of -2 is needed for the second consumer instruction in line 5. The Figure 5.1(c) lists the result of compiling the basic block as a whole. As a result of removing redundancies (lines 7, 8, 12) in a larger scope the offset in the "add" instruction of line 14 becomes -6 as it reuses the "x" variable loaded in line 3. The maximum offset reference value grows from -2 to -6 as highlighted in the programs.

Figure 5.2 shows the maximum offset reference value for a set of benchmark programs when compiled on a expression DAG scope and basic block DAG scope. Programs

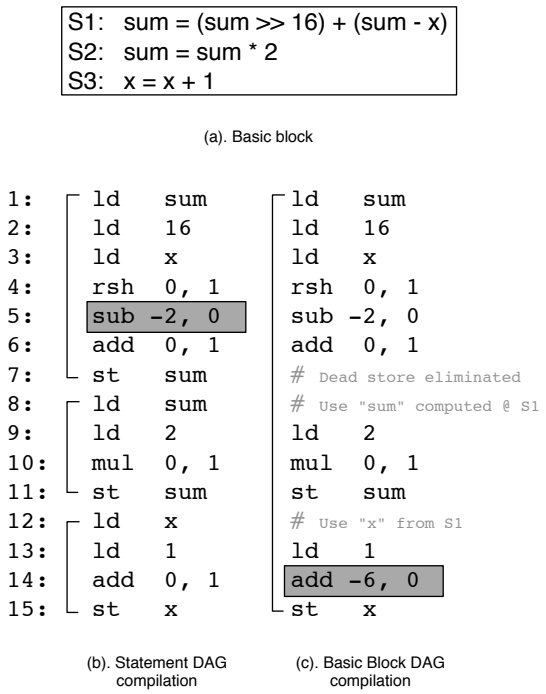


Figure 5.1: Effect of compilation scope on offset characteristics. (a) sample basic block, (b) resulting program of statement-based compilation scope with a maximum offset of -2, and (c) resulting program of basic block compilation scope with a maximum offset of -6.

compiled from a basic block DAG raise the maximum offset reference from 4 to 10 times. Therefore, limiting compilation to a statement scope is a good alternative to generate programs with modest offset reference values at the cost of increasing memory traffic.

Despite that offset reference values can be relaxed by the compilation scope, the queue register file utilization remains an unsolved problem. Employing the same shared memory communication model the queue utilization can be controlled effectively [15]. Whenever the compiler detects a program exceeding the available hardware queue, the data flow graph is partitioned into *clusters* fitting in the available queue registers. And spill/fill instructions are inserted to communicate the values defined and used in different clusters. Although this technique efficiently controls queue utilization, it produces extra memory communication. As it has been reviewed, shared memory communication is an effective way to control offset references and queue utilization. However, the load/store distribution of programs is large and may lead to poor execution times.

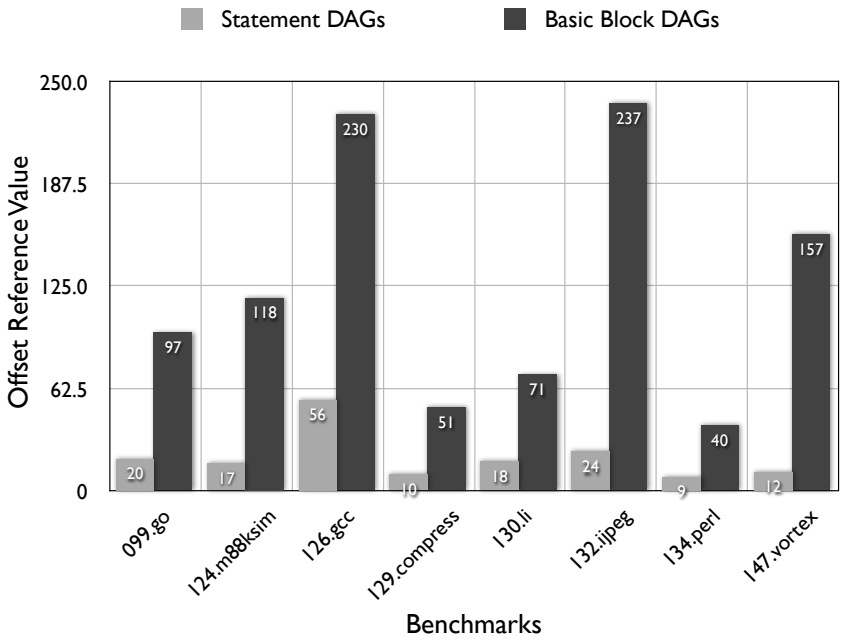


Figure 5.2: Maximum offset reference value for statement-based and basic block compilation scopes.

5.1.2 Inter-block Communication

To communicate live variables across basic blocks, offset references may be used at the cost of increasing its length and also more queue to hold dead element fragments of two blocks. Figure 5.3(a) shows an example of a live variable "i" in BB1 accessed by an operation in BB2. More queue is required to hold all the values between the definition of "i" in BB1 and its use in BB2. Excluding the previous implications, inter-block communication may arise a problem of offset inconsistency. Consider the merging block BB3 in Figure 5.3(b). Let "x" be alive in BB3 and defined in both BB1 and BB2, using offset reference to retrieve the correct "x" value depends on whether the BB1 or BB2 was executed. Such situation cannot be determined at compile time and therefore generating an offset reference would lead to inconsistent execution unless a kind of predicated execution is available in the hardware.

The shared memory model for inter-block communication has identical effect as presented in the previous section. Offset references and queue utilization can be reduced by introducing memory operations to communicate the live variables as shown in Figure 5.4(a). The offset inconsistency problem on merging blocks disappears as the two

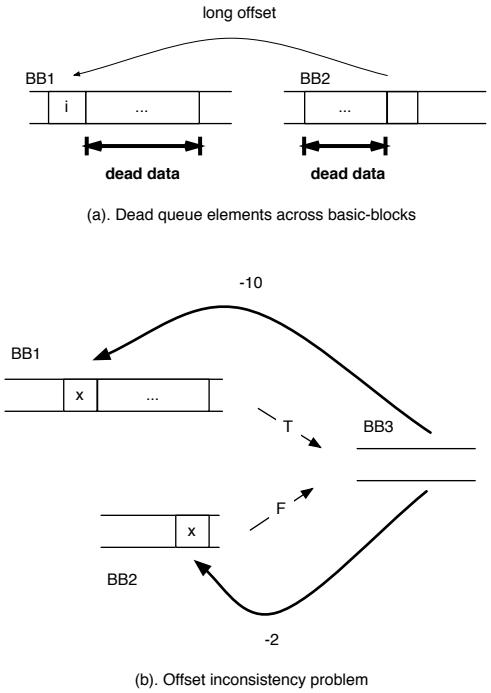


Figure 5.3: Problems of sharing data in the queue across basic blocks. (a) long offset references of live variables across basic blocks produce large amounts of dead data. (b) BB3 faces an offset inconsistency problem since the correct offset value depends on runtime behavior and cannot be determined at compile-time.

possible offset values are replaced by a consistent memory access as shown in Figure 5.4(b). For inter-block communication, using the shared memory model becomes a necessity as it provides consistency and guarantees correctness.

5.2 Operand Queue for Reducing Memory Traffic in Queue Programs

In this section we propose a new method to improve the memory traffic in queue programs by using the operand queue for data communication within and across basic blocks. The main premise is to use the main queue as the high-speed storage to hold and forward-propagate values consumed multiple times. Replacing the memory traffic generated to constrain offset references for data transfer instructions speeds up the execution of the

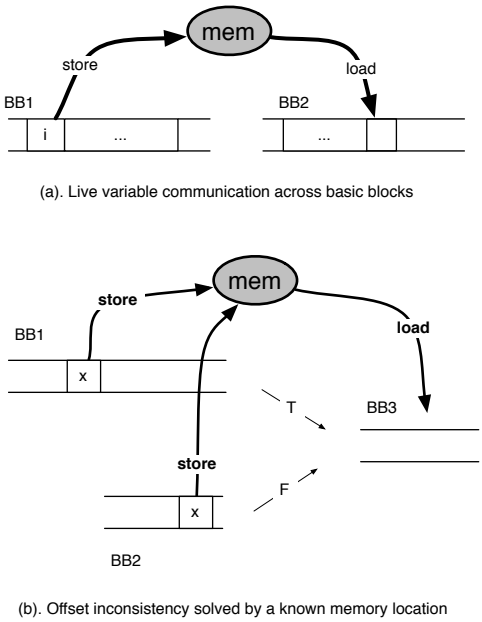


Figure 5.4: Shared main memory for basic block communication. (a) long offsets and dead data problems are solved with a store and load instructions. (b) offset inconsistency problem is solved by accessing a known memory location.

program.

The length of offset references can be controlled if temporaries holding copies of used variables are inserted in the program and the uses of the original definition are replaced by uses of the temporaries. Short life ranges of variables represent short offsets and small queue utilization in the queue computation model. For example, let the offset in Figure 5.5 be outside the range allowed by the hardware. The transformed program shown in Figure 5.5(b) includes a compiler inserted temporary i_{new} holding a copy of i. And the use of i in the last statement is replaced by a use of i_{new}. Assuming that i_{new} is inserted in right between the first and last statements, the original offset has been reduced by half. To implement such functionality the queue processor must include an instruction that produces a copy of a queue register, and the compiler must find the correct location in the program to insert such instructions. The strength of this technique is that without addition of extra hardware registers the frequently used values can be kept accessible within short offset values.

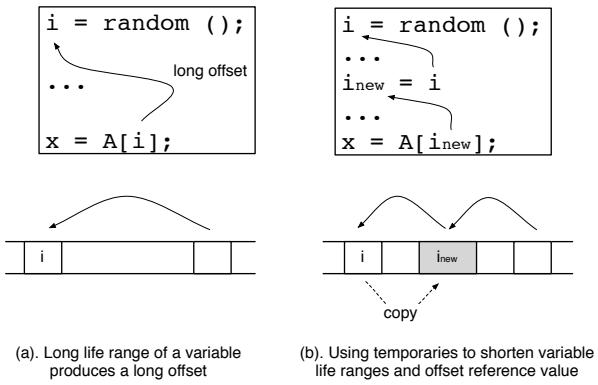


Figure 5.5: Offset reduction by in-queue copies: (a) original program with long offset, (b) compiler inserted copies to shorten offset references.

5.2.1 Semantics of `dup` instruction

To enable this feature we have added a new instruction called *duplicate* or `dup` instruction. The instruction syntax is a producer order-like instruction consisting of the opcode and one offset reference. Same as all producer order instructions in the Parallel Queue Processor the `dup` instruction is encoded in 16 bits.

dup	offset

The `dup` instruction copies the contents of a queue register specified by an offset reference relative to QH and produces the result in QT.

$$QT \leftarrow \text{contents\_of} (QH - \text{offset})$$

The `dup` instructions produce values in the queue that most probably will be consumed by an offset reference. Data having no explicit consumers (consumers that dequeue a value directly from QH) would induce a data ordering problem. Explicit repositioning of the QH and QT registers is available using instructions. However, we extend the functionality of the queue processor so that no explicit instructions to reposition QH are required, and that offset reference values are identical as in the shared memory model program. The `dup` instruction flags the queue register pointed by QT to be ignored whenever a read access is attempted by QH. For better understanding of such new feature consider the example in Figure 5.6. Let a long offset reference using variable “a” be shortened by

a `dup` instruction inserted between the addition and the load of "c". During execution, right after the addition is performed, the "`dup` −2" instruction produces a copy of 'a' into the tail of the queue. The queue register pointed by QT is flagged as shown by the oversized 'a' element in the queue. Execution continues until the division requests its both operands to be read from the QH. After the first operand 'a+b' is consumed, QH is updated and now it points to the flagged 'a' element. Once this situation is discovered by the processor, the QH position is updated to the next non-flagged element and the second operand 'c' is read from QH directly. Thus, the offset values of the division instruction appear as if there was no element in between the operands making the utilization of `dup` instructions transparent to the program.

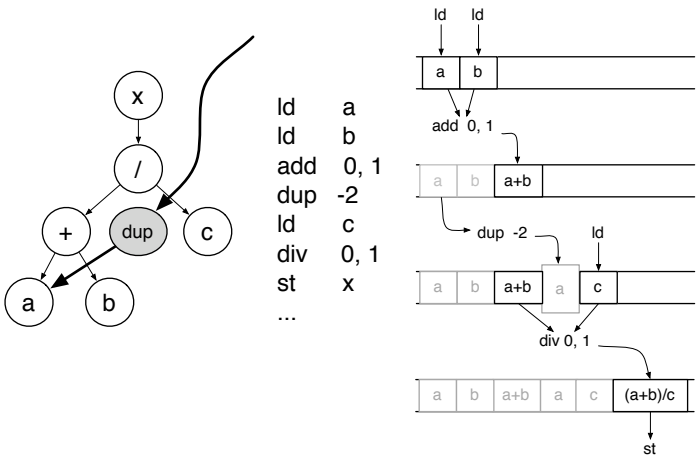


Figure 5.6: Semantics of `dup` instruction do not affect other instructions offset references.

5.2.2 Algorithm for inserting `dup` instructions

The *threshold* is the maximum offset reference value allowed to exist in the program. The algorithm inserts `dup` instructions in the program's data flow graph to reduce the length of edges violating the threshold value. It is a good idea to let the user decide the maximum offset value since this constraint may different for different implementations of queue processors.

If a single `dup` instruction is not enough to constrain the length of an edge then a chain of `dup` instructions is inserted as shown in Figure 5.7. For this example, two `dup`

instructions (definitions of i_1 and i_2) are needed to constrain all offsets to a threshold of 8. For the few cases when a very long offset [15] is constrained by a small threshold, the large number of chained `dup` instructions may incur in lower performance than using the shared memory model. Although we discard it from our implementation, we notice here that allowing the algorithm to reason about using the shared memory communication model whenever the chain of `dup` instructions is too long may lead to higher performance code. Such hybrid solution can be extended from our proposed algorithm by the addition of a MAX_DUP parameter and the logic to discern whether a long offset is reduced by `dup` instructions or the shared memory communication model.

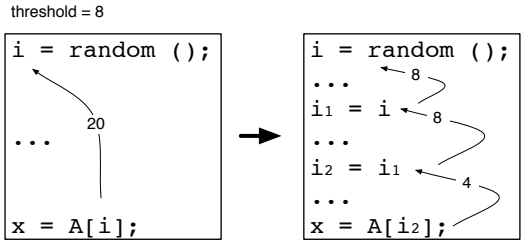


Figure 5.7: Chain of `dup` instructions fit any offset reference into the threshold value.

Algorithm 13 shows the implementation of the `dup` instruction placement. The outer loop iterates over the instructions of a basic block reducing all offset references exceeding the threshold value by inserting `dup` instructions. The number and location of the `dup` instructions, *ndup*, is decided by dividing the maximum incoming offset by the threshold and inserting the instructions in fixed intervals between the sink and source instructions. All other incoming edges are relinked to the most immediate generated copy. Since the `dup` instructions are copies of variables that occupy space in the queue they affect other offset references. The "keep_going" flag is set to indicate the outer loop to iterate once more over the basic block and reduce any new possible violating offsets. The main idea is letting the first iteration on the basic block to guess the initial number and location of `dup` instructions. Further iterations refine the placement taking into account the new offsets. The function in Line 11 recomputes the offset values of the basic block after placing the new instructions.

```
   Input: DAG representation of the basic block, BB
   Input: Maximum offset, threshold
   Output: BB with offset references <= threshold
 1 begin
 2   while keep_going do
 3     keep_going ← 0;
 4     forall nodes n in BB do
 5       if any incoming edge e ∈ InEdges (n) > threshold then
 6         ndup ← InsertDup (MAX(InEdges(n)), threshold);
 7         Relink_to_dups ( InEdges (n), ndup);
 8         keep_going ← 1;
 9       end
10     end
11     RecomputeOffsets (BB);
12   end
13 end
```

Algorithm 13: queue_communication (BB, threshold)

5.2.3 Using queue for inter-block passed variables

To communicate frequently used and live variables across basic blocks we propose to use the queue instead of memory to reduce memory access and increase performance. We limit the number of variables that can be passed in the queue to five as the common knowledge indicates [36]. All others are passed through shared memory communication model. In our implementation, we copy the variables with the most uses in the successor blocks. Subroutine calls are treated similarly, up to five arguments are passed in the queue and are available at the beginning of the first block of the callee function. The return value is always copied before returning.

The selected candidates are copied by `dup` instructions inserted at the end of the basic block. These instructions place copies from one block to the beginning of the successor block. In case of a merging block this method provides a consistent offset access as in the shared memory model. Copies are placed at the beginning of the merging block and thus the offset references are known. In Figure 5.8 the predecessor blocks BB1 and BB2 include code their corresponding code for copying the variable “x” into the beginning of their merging block BB3. All references to “x” in BB3 are local to the block and can be determined at compile-time.

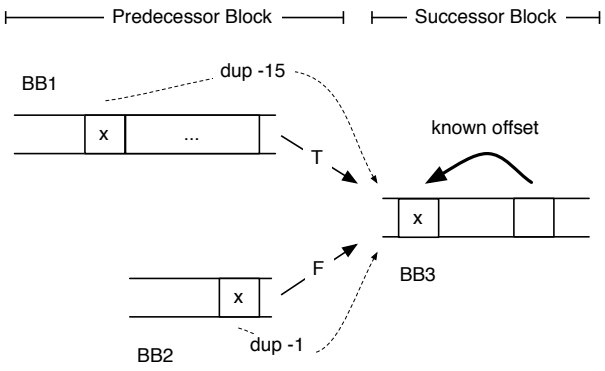


Figure 5.8: Insertion of `dup` at the end of blocks solves the problem of offset inconsistency for the successor blocks and allows the communication of frequently used variables in the queue.

5.2.4 Enabling Compiler Support

We implemented the new algorithm in the Queue Compiler. Some modifications of the original compiler were made to support the queue-based communication model. The most important change is the generation of a basic block DAG rather than statement-based DAGs. The original compiler was designed to process one statement at a time and all algorithms worked on this assumption. For example, the offset calculation phase receives as input single rooted DAGs. However, in a basic block DAG there may be multiple roots. Figure 5.9 shows the block diagram of the queue compiler. The gray blocks represent the modified phases.

The Queue Code Generation phase generates a DAG representation of the basic block. The new algorithm (Algorithm 13) represented by the third phase of the compiler optimizes the program by using the operand queue for sharing frequently used data. The Offset Calculation phase computes the offset reference values of the program. This algorithm was updated to handle the new basic block DAGs correctly. The Level-Order scheduler schedules the program and lowers the LDAGs representation into a linear low level intermediate representation (QIR). This QIR representation is consumed by the last phase of the compiler which generates assembly code for the PQP.

The new algorithm demands detailed data dependency information which is facilitated by a tree-based representation. But also requires the ability to traverse, modify, and

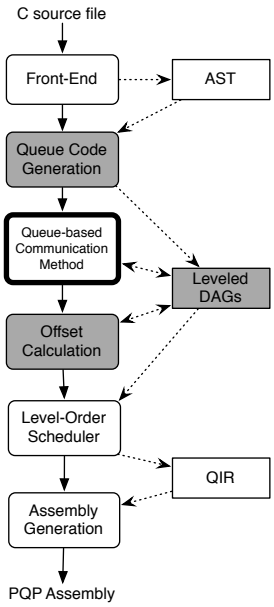


Figure 5.9: Block diagram of the queue compiler.

gather precise location information of instructions in the program for the insertion of `dup` instructions. The latter is facilitated by a linear representation. Given the location of the new phase (between queue code generation and offset calculation), we had to extend the original LDAGs [35] which is tree-based to be combined with a linear-based representation. The new LDAGs can be traversed as a tree or as a list of instructions.

5.3 Evaluation

Memory traffic in queue programs can be reduced by reusing data as much as possible within computations of a basic block, and communicating frequently used variables in the queue across blocks. If data reutilization is maximized then offset references grow in length and in number. Therefore, in this section we look at the comparison of memory traffic of programs generated by our queue-based communication method against shared-memory communication model. Then, we analyze the effectiveness of `dup` instructions on controlling the length of offsets. We do this by investigating the overhead of `dup` instructions when limiting the offsets references to different thresholds. We based our algorithm on the Queue Compiler [13] infrastructure. To compare the new method against

the baseline code we use the modified, and the original queue compilers respectively. The selected benchmarks are from SPEC CINT95 suite [21].

5.3.1 Memory Traffic Reduction

Figure 5.10 shows the instruction distribution of loads, stores, and the rest of the generated instructions (including ALU, control flow, etc). For every benchmark a pair of results is shown. The first result represents the baseline instruction distribution compiled with the shared-memory communication model. The second result represents the instruction distribution obtained with the our proposed queue-based communication model. The queue-based communication model was configured with a threshold set to infinity and therefore no `dup` instructions are inserted in the programs.

For all benchmarks the memory traffic is reduced from 16% to 25%, 20% in average. This memory traffic reduction is explained by the scope of compilation. The original compiler generates code for statements and uses memory for holding and reusing data. The new method generates code for basic blocks and uses the queue for communicating values rather than memory. Non-memory instructions percentage remains about the same for the two methods. As a result, the code density of programs compiled by our new method is improved by about 20%.

Another important qualitative characteristic to be noticed by these results is that the memory instructions in the original compiler account for about 50% of all instructions in the program, and in the new method are about 40% of the total.

5.3.2 Expense of `dup` instructions for offset constrain

Offset characteristics can be classified in two: number and length. Number refers to the amount of instructions using an offset reference to access an operand. Length is the distance, in queue words, between the head of the queue and the accessed value. Programs with high degree of data reusability require an extensive number of offset references. Long live ranges of reaching definitions demand large offset values. Therefore, the overhead of `dup` instructions is a function of the offset characteristics of the program and the selected threshold value.

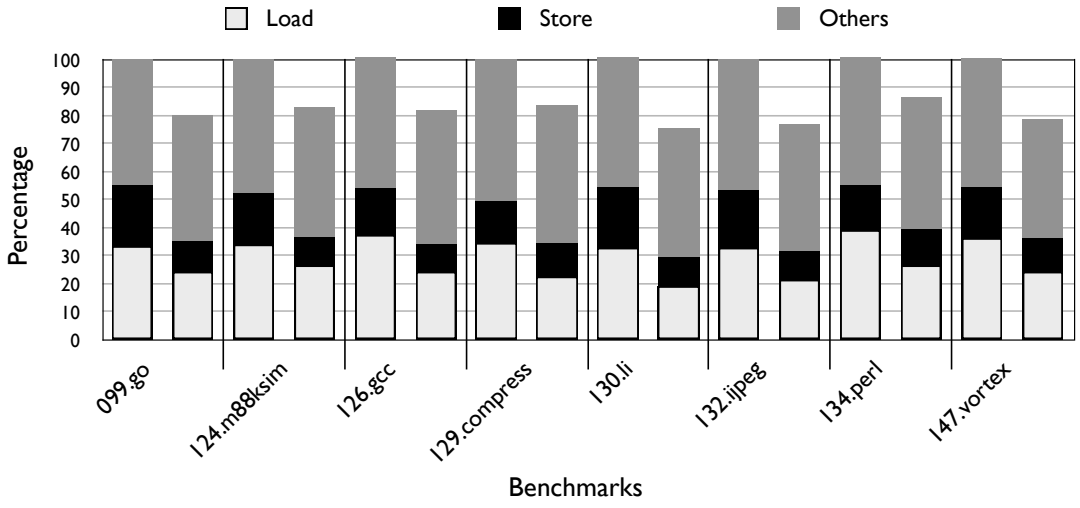


Figure 5.10: Memory traffic reduction. For every benchmark the first column represents the program compiled shared-memory communication model, and the second column represents the program compiled with the new queue-based communication model.

Figure 5.11 shows the results of constraining offsets to threshold values of 4, 8, 16, 32, 64. We use the unconstrained offset compilation result (INFTY in the graph) as the lower bound for our technique since memory traffic is reduced and no efforts are made to reduce the offsets. The lower bound provides the reference point to measure the overhead of our technique. And we use the number of instructions generated by the shared-memory model as the baseline.

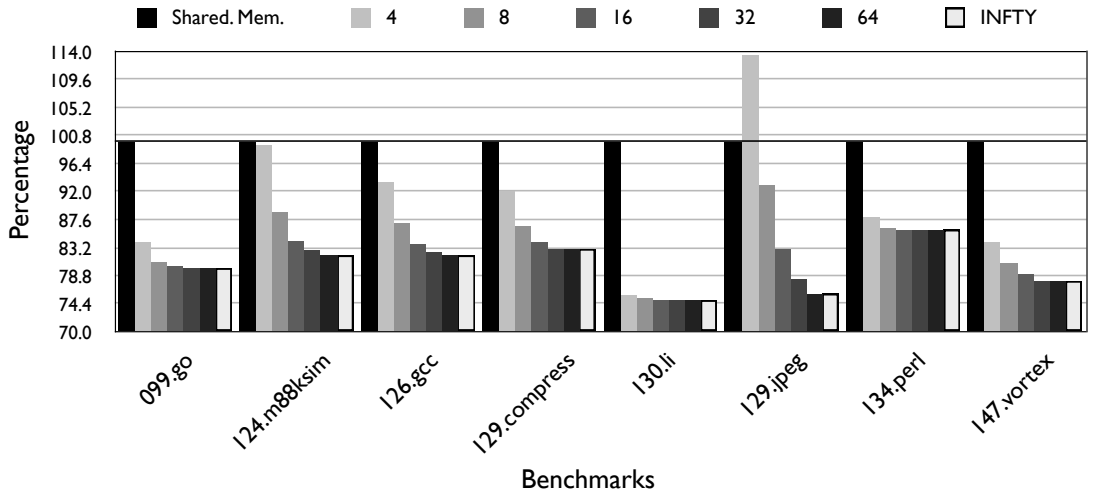


Figure 5.11: Overhead of `dup` instructions for different threshold values.

For the most restrictive threshold, 4, the overhead ranges from a 1% (130.li) to 37.47% (132.ijpeg) more `dup` instructions than the lower bound. The geometric mean is 6.47%. These percentages represent the density of `dup` instructions in programs. The 132.ijpeg and 124.m88ksim benchmarks benefits from the larger scope of compilation and produces a large number of long offset references. As a result, the programs are saturated with `dup` instructions. Compiling for a threshold of 8 produces an average overhead of 4.6%, for 16 an average of less than 1.69%. and for larger thresholds an average less than 1%. In terms of code size, for all programs and for all thresholds except 132.ijpeg, our proposed technique achieves smaller programs than the original shared-memory model compiler. These results show that sharing data in the queue reduces memory traffic, and effectively handles the problem of long offsets with a small overhead of `dup` instructions.

5.4 Conclusion

We presented an efficient compilation method for reducing memory traffic in queue processors with a single operand queue. The key idea is to keep data in the operand queue during their life times without seriously affecting the offset references limitations and queue register file utilization. Our proposed method avoids redundant memory accesses within a basic block by duplicating and propagating data near its consumers, and across blocks by duplicating the most frequently used variables to the beginning of the successor block. We successfully implemented the queue-based communication compilation method in the queue compiler and analyzed its effectiveness on a set of benchmark programs. The new method reduces memory traffic by about 20% when compared to the shared memory communication model. And the overhead of `dup` instructions to constrain the offset reference lengths is small even for very restrictive thresholds, about 6.47% more instructions for a threshold of 4. This means that our method effectively substitutes a significant number of long-latency memory accesses for a small number of fast queue-to-queue communication. In our future work we expect to measure the execution time improvement of our technique.

Chapter 6

Queue Computing Evaluation

This chapter has the objective to demonstrate, with the use of the Queue Compiler Framework, the benefits and good characteristics of queue computing for real world applications. We selected ten applications commonly used in embedded systems from MiBench and MediaBench suites [32, 52]. 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. The target architecture for this comparison is the QueueCore processor (See Section 4.2.2). As the instruction set of the QueueCore processor allows at most one encoded offset reference, the 1-offset P-Code generation algorithm presented in Section 4.1 was enabled.

6.1 Code Size Comparison

We compare the size of QueueCore binaries with the code size of two dual-instruction embedded RISC processors: MIPS16 [43], ARM/Thumb [31]. With two traditional RISC machines: MIPS I [41], ARM [69]. And with a traditional CISC architecture: Pentium processor [5]. 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 6.1 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 queue 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. The QueueCore binaries are able to achieve the smallest code sizes due to instructions being 16-bit.

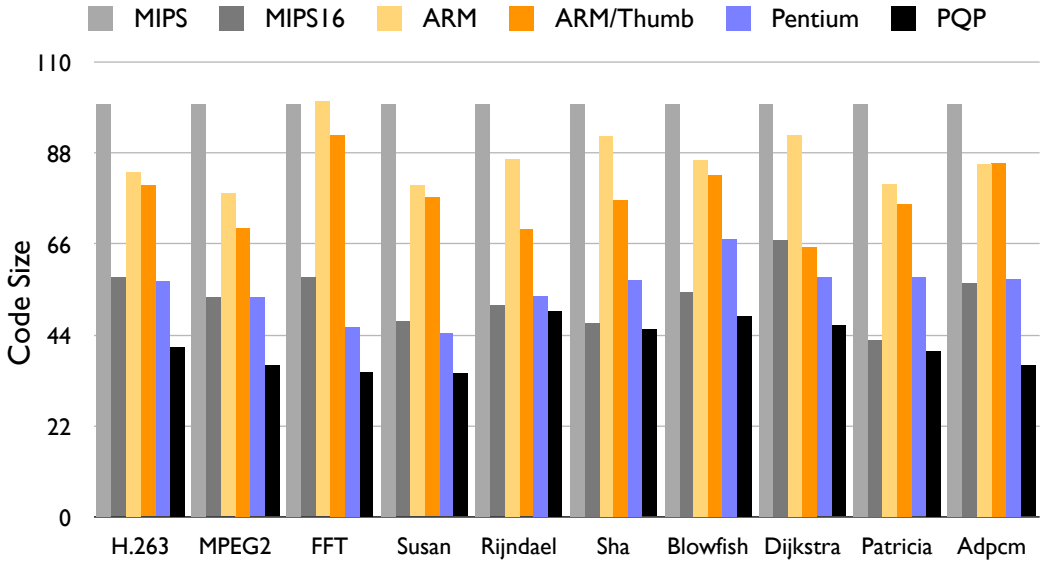


Figure 6.1: Code Size Comparison

6.2 Compile-time Extracted ILP Comparison

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 [19] 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.

To match the 32 architected registers of MIPS I, we constrained the compilation of queue code to a queue register file of 32 words using the algorithm presented in Section 4.2. Figure 6.2 shows the extracted parallelism by the two compilers. Our compiler extracts, in average, 1.16 times more parallelism than fully optimized RISC code. The -O3 optimization in the register compiler rearranges instructions according sophisticated list scheduling heuristics. The order of instructions in the register program resembles data flow scheduling which is very similar to the level-order scheduling. From these results we observe that both, register and queue programs, have similar ILP properties. However, the MIPS code is limited by the architected registers and register pressure is solved by the insertion of spill code. For some applications, the queue programs are able to extract all available parallelism as the limitation of physical registers does not exist in the queue computation model.

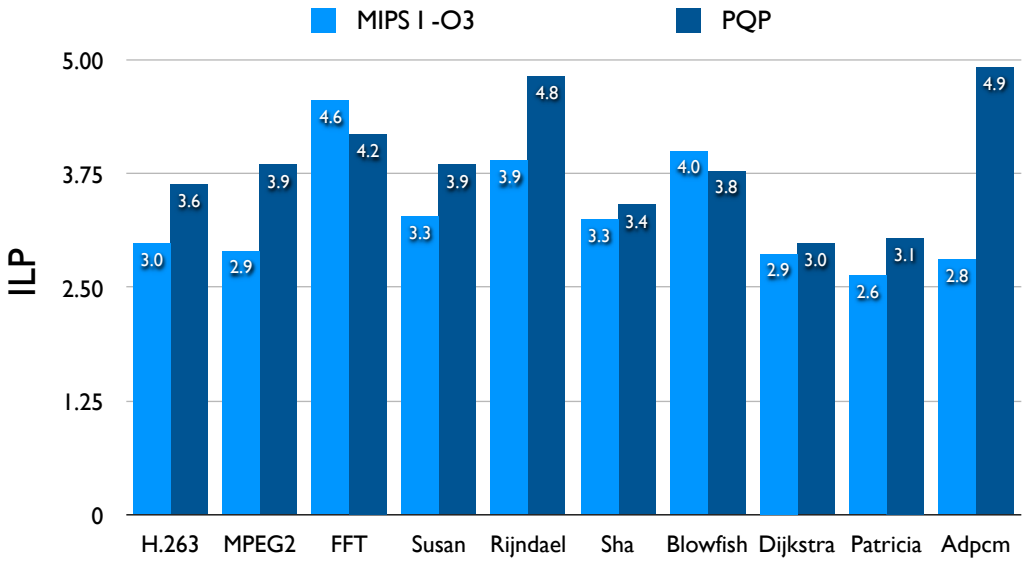


Figure 6.2: Compile-time extracted instruction level parallelism

Chapter 7

Conclusion

This doctoral dissertation presented the design and development of the first queue compiler framework. Many aspects of code generation for queue machines were investigated from a different yet natural perspective and, as a result, novel techniques have been discovered, implemented, analyzed, and evaluated. The presented methods integrate the queue computation principle into all stages of automatic compilation of high-level programming languages to executable code. The most important contribution of this work, I believe, is the establishment of principles that facilitate code generation for queue machines taking into consideration all the demands and peculiarities of such computation model.

Chapter 2 presented a summary on the queue computing variants and concluded that the producer order queue computation model offers the best characteristics and flexibility. Chapter 3 defined the concepts on which the queue compiler was developed. It presents novel and fundamental algorithms to build leveled directed acyclic graphs (LDAGs), to calculate the position of `QH` and to perform offset calculation. A queue compiler framework is developed and the compilation phases are described together with the internals of the compiler. This part of the dissertation establishes the methodology to build a queue compiler. We presented that the complexity of the queue compiler is similar to that of a research and production compilers in terms of lines of code and compile-time. These results show that the developed algorithms and internal data structures that drive the queue compiler are efficient.

Chapter 4 introduced the problems of compiling for actual queue processors. First,

we have presented a technique to make the compiler aware of the reduced instruction set of a queue processor to achieve high code densities. The idea is to constrain the instructions to encode at most one offset reference and let the compiler handle the cases when two offsets are required by inserting extra instructions to copy data and locate it in the correct place to eliminate one offset reference. The evaluation of the proposed technique shown that the compiler is able to generate programs about 16% and 36% smaller than two embedded processors while keeping the insertion of extra instructions less than 5% of the total generated instructions. Second, we developed a technique to control the queue register file utilization of peak-parallelism fragments of programs. The algorithm breaks the data flow graph into clusters that fit into the physical queue while retaining the semantics of queue computation model and inserting inter-cluster communication and spill code. For scalar applications, our technique is able to control queue utilization by around 2% of extra communication and spill code. Third, we explored the impact of classical, target-independent, data flow optimizations in queue machines. We introduced how common subexpression elimination raises the offset requirements of instructions and queue utilization. Our results shown that queue computing does not restrict this kind of transformations and execution time is reduced together with the total number of instructions. However, the raise in offsetted instruction increased from 1% to 3% for 2-offset instructions and 9% to 28% for 1-offset instructions. Fourth, we presented a code motion technique to improve the level-order scheduling by increasing the static instruction level parallelism. The statement merging technique shown that it is capable of extracting in average, 1.66 times extra parallelism.

Chapter 5 presented a global compilation technique, Queue Allocation, to reduce memory traffic in programs by allowing data reutilization. The key idea is to copy and propagate data inside and across basic blocks using the queue instead of long latency memory accesses. The argument is that fast queue to queue data transfer instructions replacing memory accesses may improve the execution time of programs. The presented algorithm is able to reduce memory traffic by about 20% for a set of scalar applications while keeping the queue-to-queue communication small with about 7% of total instructions.

In Chapter 6, by using the here presented queue compiler, we have been able to

demonstrate that our framework generates code with practical quality comparable to that of production compilers. Having small binaries and high amounts of compile-time instruction level parallelism suggest that an actual queue processor may make efficient use of cache memories due to its small instructions, may have a simple fetching and decoding buses and logic, a small instruction window, and no register renaming mechanisms. Reducing the complexity of the hardware may allow us to produce a very power-efficient parallel processor.

Engineering the queue compiler took big efforts and special attention. New data structures and suitable intermediate representations were designed, and the algorithms were designed and crafted to be part of a production compiler. The compiler has been completed and it is able to compile any program, including itself which makes it ready for deployment in a queue-based computer system. Making the compiler aware of the target hardware configuration allows the generation of customized and optimized code that helps the current and future research and development of queue machines. I expect this work will be the starting point for future works and discoveries in queue-based computer systems. I believe, the queue compiler is a sophisticated tool that will be used to develop new ideas.

Appendix A

P-Code Instruction Set Architecture

A.1 Notation

We use the following convention for the instructions: a field is specified by its definition in monospace characters enclosed by '<' and '>', e.g. "`<type>`". The following table describes the meaning of fields used to describe the syntax and semantics of P-Code instructions.

Field	**Description**	**Example**
`opcode`	Mnemonic of the instruction	`add`
`<type>`	Data type	`iws`, (See Table 3.2)
`<QHoffset>`	Offset reference relative to `QH`	`not` −3
`<QHoffset2>`	Second offset reference relative to `QH`	`add` −3, **-4**
`<register>`	Special purpose registers used for memory addressing	Frame pointer (**$fp**) Stack pointer (**$sp**) Return address (**$ra**)
`<displacement>`	Memory displacement addressing mode	`ld` **32**($fp)

`<label>`	Label describing a target of a jump
	for conditional jumps the label is PC relative
	for unconditional jumps is an absolute address
		
		`$L3`
		`$malloc`

A.2 Arithmetic & Logic Instructions

The format for binary instructions is "`opcode <type>, <QHoffset>, <QHoffset2>`" and for unary instructions is "`opcode <type>, <QHoffset>`".

Opcode	Description	Operator
`add`	Addition	+
`sub`	Subtraction	−
`mul`	Multiplication	∗
`div`	Division	/
`mod`	Modulo	%
`neg`	Negation (Unary)	−
`rsh`	Right shift	>>
`lsh`	Left shift	<<
`and`	Logic AND	&
`ior`	Logic inclusive OR	|
`xor`	Logic exclusive OR	
`not`	Logic NOT (Unary)	!
`rrot`	Bit right rotation	
`lrot`	Bit left rotation	
`abs`	Absolute value (Unary)	$\|abs\|$

A.3 Memory Instructions

Opcode	Description	Format
`ld`	Load from memory	"`ld <type>, <displacement>(<register>)`"
`st`	Store to memory	"`st <type>, <QHoffset>, <displacement>(<register>)`"
`ldi`	Load immediate value	"`ldi <type>, <constant>`"
`sld`	Unary load	"`sld <type>, <QHoffset>, <displacement>(<register>)`"
`sst`	Binary store	"`sst <type>, <QHoffset>, <QHoffset2>`"

The hardware implementation of PQP [87] exposes a small set of special purpose registers to be used as the stack pointer (`$sp`), frame pointer (`$fp`), return address

register, two address registers (`$a1, $a2, $a3, $a4`), and two data registers (`$d1, $d2`). Local variables, incoming and outgoing arguments are accessed through `$sp, $fp` using displacement addressing mode. Data registers serve to access global variables and addresses. The queue compiler employs a conventional register-indirect addressing mode to access memory. First the address is loaded into a data register and that register is used to compute the effective address. When a pointer access or computed address for array addressing is required, the compiler makes use of the address registers in similar fashion. First, the address is computed in the queue. Second, the computed address is moved from the queue to the address register. Third, the memory operation uses the register for accessing the effective address.

Although this addressing mechanism is convenient for conventional register computers, the PQP faces serious challenges that degrade the parallel capabilities of the queue computation model. Naturally the PQP code is free of false dependencies. Using the address registers to access memory introduces false dependencies in the queue code that limit the available parallelism. The quality of the code is seriously affected whenever more than two address registers are required to perform computation. Since the PQP has a reduced bit-width instruction set the addressable number of registers is limited. To solve this limitations we propose a new addressing method for the queue processor that relies entirely on the queue rather than registers.

A.3.1 Efficient Addressing Method for Queue Processors

We have developed a new addressing method for queue processors which: (1) follows the single assignment rule by using queue to compute addresses and accessing memory rather than registers; (2) increases the parallelism by fully computing the addresses every time that is required and eliminating unnecessary false dependencies; (3) eliminates all data transfer instructions from queue to registers; (4) presents two new instructions that read and write to memory taking the address and data directly from the queue.

Two new instructions are required to facilitate the new proposed addressing method. We have named these instructions *binary store* (`sst`) and *unary load* (`sld`). The new store instruction requires two operands that are implicitly read from the `QH` of the queue, thus, a binary operation. Similarly, the new load instruction requires a single operand

to be read from the QH of the queue making it a unary operation. Figure A.1 shows the semantics of the two instructions. The sst instruction requires its first operand to be the computed address, and its second operand to be the data to be stored in that address. The sld instruction requires its only operand to be the computed address from where to read data. After reading the data from the specified address the value is written into QT.

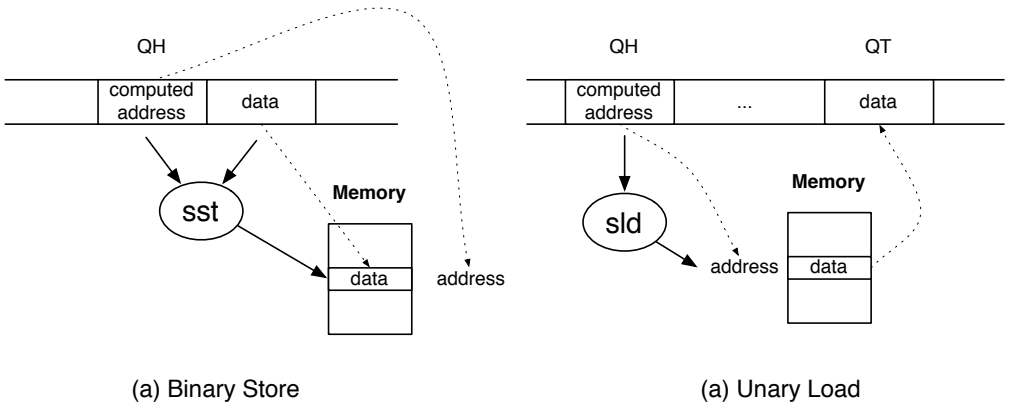


Figure A.1: Semantics of the new memory instructions.

Consider the following C fragment of a program that copies array y to x:

```
for(i=0; i<n; i++) {
    x[i] = y[i]; /* kernel */
}
```

For the simplicity of the explanation we concentrate on the kernel of the loop. The statement "x[i] = y[i];" requires the addresses of arrays x, y to be computed by the following equation:

$$addr = base + (index * size) \tag{A.1}$$

Figure A.2 shows the parse tree of the kernel of the loop including instructions to compute addresses of "x[i], y[i]" as dictated by Equation A.1. On the right side of the figure the queue program is shown. The queue program is obtained from making a level-order traversal of the parse tree. A level-order traversal visits all the nodes on every level from left to right, from the deepest level towards the root [71]. Notice that the instruction sequence includes the sst and sld instructions. The sld instruction is

connected to the (+) node in level L_2. This addition computes the address of `y[i]` (`T1`). The `sst` instruction is connected the two nodes in level L_1. Its first children holds the address of `x[i]` (`T2`), and the second children holds the value loaded by the `sld` instruction.

In Figure A.3 the contents of the queue are shown before the execution of the last two instructions of the program. For the `sld` instruction, notice that `QH` points to the result of the addition that holds the computed address `T1`, or `y[i]`. After `sld` is executed it retrieves the value stored at address `T1` and writes it into `QT`. Then, before the `sst` instruction is executed, `QH` points at the result the addition that contains the computed address `T2`. As `sst` reads two operands, the first is the address, and the second is the value written by `sld`. Thus, the evaluation of the kernel of the sample loop is performed solely by using the queue.

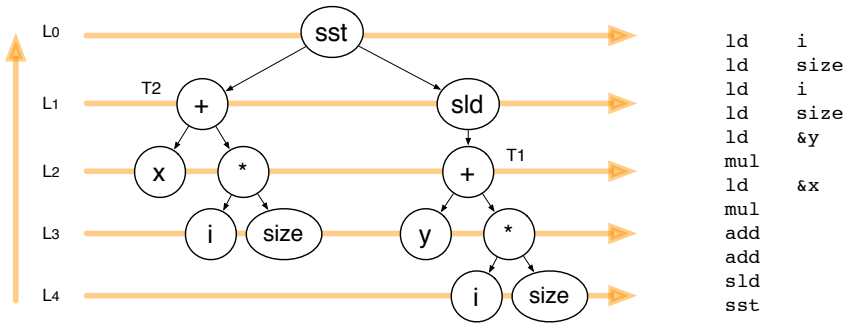


Figure A.2: Level order traversal of parse tree with array address calculation

Our proposed addressing method for queue processors is applicable to all aggregate types such as pointers, arrays, and structures that require their addresses to be computed at runtime. The new addressing method using unary load and binary store operations increases parallelism by eliminating false dependencies introduced by the native PQP's displacement addressing mode using data and address registers. Instead, se use the queue to compute and hold effective addresses rather than registers.

A.4 Comparison Instructions

Comparison instructions are binary instructions whose format is "`opcode <type>, <QHoffset>, <QHoffset2>`". After two operands are compared and the condition is

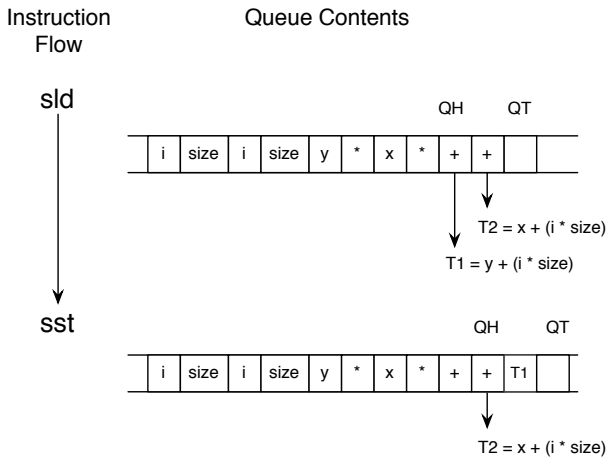


Figure A.3: Level order traversal of parse tree with array address calculation

as expected by the operation, the result is automatically placed in the condition code register `$cc`. This condition code register is checked by conditional branch instructions to enable control flow.

Opcode	**Description**
`ceq`	Equal
`cne`	Not equal
`clt`	Less than
`cle`	Less or equal
`cgt`	Greater than
`cge`	Greater or equal

A.5 Control Flow Instructions

Conditional branch instructions check the result stored in condition code register to branch on *true* or *false*. Compiler generated target labels for jumps are defined by '$L' followed by a number, e.g. '$L7'. All control flow instructions using labels with this format are PC-relative jumps.

Opcode	Description	Format
bt	Conditional branch to target if $cc is true	"bt <label>"
bf	Conditional branch to target if $cc is false	"bf <label>"
j	Unconditional jump to PC-relative target Unconditional jump to absolute address target	"j $L3" "j 0xffff0000"
jal	Subroutine call	"jal $subroutine"
ret	Return from subroutine call	"ret"

A.6 Data Type Conversion Instructions

To convert operands to other data types the conv instruction is employed. This instruction takes two types, destination and source types, and the operand specified by an offset reference. The format of the instruction is as follows:

Opcode	Description	Format
conv	Data type conversion	"conv <type>, <type>, <QHoffset>"

A.7 Special Instructions

Opcode	Description	Format
dup	Duplicates an operand referenced by an offset relative to QH placing the result in QT	"dup <type>, <QHoffset>"
rot	Rotates the operand in QH to QT	"rot <type>"

A.8 Queue Control Instructions

For compatibility reasons with the hardware implementation of the PQP [87] the compiler includes these two instructions. However, the queue compiler does not utilize the instructions. These instructions allow the QH and QT pointers to be explicitly moved to any location. As the compiler statically schedules the instructions, and knows at every point of execution the precise location of the queue pointers, the explicit control is unnecessary.

Opcode	Description	Format
moveqh	Moves QH to the specified location	"moveqh <QHoffset>"
moveqt	Moves QT to the specified location	"moveqt <QHoffset>"

Bibliography

[1] B. Abderazek, A. Canedo, T. Yoshinaga, and M. Sowa. The QC-2 Parallel Queue Processor Architecture. *Journal of Parallel and Distributed Computing*, 68(2):235–245, February 2008.

[2] 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.

[3] 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.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[5] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *Micro, IEEE*, 13(3):11–21, June 1993.

[6] S. Amarasinghe. Multicores from the compiler's perspective: A blessing or a curse? In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 137–137, Washington, DC, USA, 2005. IEEE Computer Society.

[7] J. Ayala, A. Veidenbaum, and M. Lopez-Vallejo. Power-Aware Compilation for Register File Reduction. *International Journal of Parallel Programming*, 31(6):451–467, December 2004.

[8] P. Boytchev. QRP-GCC a GCC-based C compiler for QRP. Technical Report Sowa Laboratory SLL050301, University of Electro-Communications, 2005.

[9] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Not.*, 24(7):275–284, 1989.

[10] A. Canedo. Code Generation Algorithms for Consumed and Produced Order Queue Machines. Master's thesis, University of Electro-Communications, Tokyo, Japan, September 2006.

[11] A. Canedo, B. Abderazek, and M. Sowa. A GCC-based Compiler for the Queue Register Processor. In *Proceedings of International Workshop on Modern Science and Technology*, pages 250–255, May 2006.

[12] 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*, 34(4):184–194, June 2007.

[13] A. Canedo, B. Abderazek, and M. Sowa. Compiler Support for Code Size Reduction using a Queue-based Processor. *Transactions on High-Performance Embedded Architectures and Compilers*, 2(4):153–169, 2007.

[14] A. Canedo, B. Abderazek, and M. Sowa. New Code Generation Algorithm for QueueCore - An Embedded Processor with High ILP. In *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2007)*, pages 185–192, 2007.

[15] A. Canedo, B. Abderazek, and M. Sowa. Queue Register File Optimization Algorithm for QueueCore Processor. In *to appear in Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[16] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[17] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 266–275, 1991.

[18] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[19] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, 1998.

[20] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[21] J. J. Dujmovic and I. Dujmovic. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):2–9, December 1998.

[22] M. A. Ertl and D. Gregg. Stack Caching in Forth. In *EuroForth 2005*, pages 6–15, 2005.

[23] K. Farkas, P. Chow, and N. Jouppi. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, page 40, 1996.

[24] M. Feller and M. D. Ercegovac. Queue machines: an organization for parallel computation. In *CONPAR '81: Proceedings of the Conference on Analysing Problem Classes and Programming for Parallel Computing*, pages 37–47, London, UK, 1981. Springer-Verlag.

[25] M. Fernandes. Using Queues for Register File Organization in VLIW Architectures. Technical Report ECS-CSG-29-97, University of Edinburgh, 1997.

[26] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995.

[27] J. Fujita. Implementation of c compiler for the parallel queue processor. Master's thesis, University of Electro-Communications, January 2004.

[28] GCC, the GNU Compiler Collection, http://gcc.gnu.org.

[29] T. Genda, B. Abderazek, and M. Sowa. Design of a Parallel 2-Dimensional Queue Processor. In *In Proceedings of SWoPP 2007, IPSJ*, 2007.

[30] A. Gordon-Ross, S. Cotterell, and F. Vahid. Tiny instruction caches for low power embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(4):449–481, November 2003.

[31] L. Goudge and S. Segars. Thumb: Reducing the Cost of 32-bit RISC Performance in Portable and Consumer Applications. In *Proceedings of COMPCON '96*, pages 176–181, 1996.

[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, 2001.

[33] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau. 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*, page 402, 2002.

[34] W. A. Havanki. Treegion Scheduling for VLIW Processors. Master's thesis, North Carolina State University, July 1997.

[35] 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.

[36] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 2007.

[37] Y. Ishizaki, P. Boytchev, T. Yoshinaga, and M. Sowa. Queue compiler development based on gcc. In *FIT2005*, 2005.

[38] S. Jang, S. Carr, P. Sweany, and D. Kuras. A Code Generation Framework for VLIW Architectures with Partitioned Register Banks. In *Proceedings of the 3rd International Conference on Massively Parallel Computing Systems*, 1998.

[39] J. Janssen and H. Corporaal. Partitioned Register File for TTAs. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 303–312, 1995.

[40] M. Junger, S. Leipert, and P. Mutzel. Level Planarity Testing in Linear Time. Technical report, Zentrum fur Angewandte Informatik Koln, 1999.

[41] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[42] S. Kawashima. Queue Java Compiler. Master's thesis, University of Electro-Communications, Tokyo, Japan, September 2004.

[43] K. Kissel. MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group, 1997.

[44] T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W.-M. Hwu. Register connection: a new approach to adding registers into instruction set architectures. *SIGARCH Comput. Archit. News*, 21(2):247–256, 1993.

[45] J. P. Koopman. A preliminary exploration of optimized stack code generation. *Journal of Forth Application and Research*, 6(3):241–251, 1994.

[46] A. Krishnaswamy. *Microarchitecture and Compiler Techniques for Dual Width ISA Processors*. PhD thesis, University of Arizona, September 2006.

[47] A. Krishnaswamy and R. Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, pages 56–64, 2002.

[48] A. Krishnaswamy and R. Gupta. 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*, pages 254–264, 2003.

[49] G. Kucuk, O. Ergin, D. Ponomarev, and K. Ghose. Energy efficient register renaming. *Lecture Notes in Computer Science*, 2799/2003:219–228, September 2003.

[50] Y. Kwon, X. Ma, and H. J. Lee. Pare: instruction set architecture for efficient code size reduction. *Electronics Letters*, pages 2098–2099, 1999.

[51] lcc, A Retargetable Compiler for ANSI C, http://www.cs.princeton.edu/software/lcc/.

[52] 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.

[53] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *SIGOPS Oper. Syst. Rev.*, 32(5):46–57, 1998.

[54] S. Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 272, 1995.

[55] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 1999.

[56] J. Llosa, E. Ayguade, and M. Valero. Quantitative Evaluation of Register Pressure on Software Pipelined Loops. *International Journal of Parallel Programming*, 26(2):121–142, April 1998.

[57] The LLVM Compiler Infrastructure, http://llvm.org/.

[58] A. Maeda and M. Nakanishi. A Queue-Machine-Based Implementation of Parallel Functionalk Programming Language. In *Proceedings of the Fifteenth IASTED International Conference on Applied Informatics*, pages 67–70, 1997.

[59] S. A. Mahlke, W. Y. Chen, P. P. Chang, and W. mei W. Hwu. Scalar Program Performance on Muliple-Instruction-Issue Processors with a Limited Number of Registers. In *Proceedings of the 25th Annual Hawaii Int'l Conference on System Sciences*, pages 34–44, 1992.

[60] S. A. Mahlke, N. J. Warter, W. Y. Chen, P. P. Chang, and W. mei W. Hwu. The effect of compiler optimizations on available parallelism in scalar programs. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 142–145, 1991.

[61] M. Maierhofer and M. A. Ertl. Local Stack Allocation. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 189–203, 1998.

[62] H. McGhan and M. O'Connor. Picojava: A direct execution engine for java bytecode. *Computer*, 31(10):22–30, 1998.

[63] W. mei Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 754–759, New York, NY, USA, 2007. ACM.

[64] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of GCC Developers Summit*, pages 171–180, 2003.

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

[66] D. Novillo. Design and Implementation of Tree SSA. In *Proceedings of GCC Developers Summit*, pages 119–130, 2004.

[67] S. Okamoto, H. Suzuki, A. Maeda, and M. Sowa. Design of a superscalar processor based on queue machine computation model. *Communications, Computers and Signal Processing, 1999 IEEE Pacific Rim Conference on*, pages 151–154, 1999.

[68] Y. Okumura, T. Yoshinaga, and M. Sowa. Parallel C Compiler for Queue Machine. In *Joho Shori Gakkai Kenkyu Hokoku*, pages 127–132, 2002.

[69] V. Patankar, A. Jain, and R. Bryant. Formal verification of an ARM processor. In *Twelfth International Conference On VLSI Design*, pages 282–287, 1999.

[70] M. Postiff, D. Greene, and T. Mudge. The Need for Large Register File in Integer Codes. Technical Report CSE-TR-434-00, University of Michigan, 2000.

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

[72] R. Ravindran, R. Senger, E. Marsman, G. Dasika, M. Guthaus, S. Mahlke, and R. Brown. Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor. *IEEE Transactions on Computers*, 54(8):998–1012, August 2005.

[73] K. Sankaralingam and K. Agaram. Evaluating the instruction folding mechanism of the picojava processor.

[74] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. *SIGPLAN Not.*, 25(6):28–39, 1990.

[75] V. Sarkar. Code optimization of parallel programs: evolutionary vs. revolutionary approaches. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 1–1, New York, NY, USA, 2008. ACM.

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

[77] M. Shannon and C. Bailey. Global Stack Allocation - Register Allocation for Stack Machines. In *EuroForth 2006*, pages 13–20, 2006.

[78] L. Sheayun, L. Jaejin, and S. Min. Code Generation for a Dual Instruction Processor Based on Selective Code Transformation. In *Lectures in Computer Science*, pages 33–48, 2003.

[79] H. Shi and C. Bailey. Investigating available instruction level parallelism for stack based machine architectures. *dsd*, 0:112–120, 2004.

[80] S. Shigeta, B. Abderazek, T. Yoshinaga, and M. Sowa. Proposal of High Parallelism QJava Bytecode. In *Proceedings of the Advanced Computing Systems and Infrastructures*, 1999.

[81] S. Shigeta, L. Wang, N. Yagishita, B. Abderazek, S. Shigeta, T. Yoshinaga, and M. Sowa. QJava: Integrate Queue Computation Model into Java. In *Proceedings*

of the Joint Japan-Tunisia Workshop on Computer Systems and Information Technology, pages 60–65, 2004.

[82] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70–83, October 2000.

[83] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinle, and J. Burrill. Compiling for edge architectures. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–195, Washington, DC, USA, 2006. IEEE Computer Society.

[84] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 central processor. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 22, pages 199–204, New York, NY, 1987. ACM Press.

[85] Sowa Laboratory, http://www.sowa.is.uec.ac.jp.

[86] M. Sowa. Summary of Parallel Queue Machine. Technical Report Sowa Laboratory SLL973002, University of Electro-Communications, 1999.

[87] 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.

[88] Trimaran, An Infrastructure for Research in Backend Compilation and Architecture Exploration, http://www.trimaran.org/.

[89] G. Tyson, M. Smelyanskiy, and E. Davidson. Evaluating the Use of Register Queues in Software Pipelined Loops. *IEEE Transactions on Computers*, 50(8):769–783, August 2001.

[90] V. živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994.

[91] N. Vijaykrishnan. *Issues in the Design of a Java Processor Architecture*. PhD thesis, University of South Florida, 1998.

[92] D. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *ACM SIGARCH Computer Architecture News*, 17(2):272–282, April 1989.

[93] D. Wall. Limits of instruction-level parallelism. *ACM SIGARCH Computer Architecture News*, 19(2):176–188, April 1991.

[94] D. W. Wall. Global register allocation at link time. *SIGPLAN Not.*, 21(7):264–275, 1986.

[95] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 81–91, 1992.

[96] W. Wulf. Evaluation of the WM Architecture. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 382–390, 1992.

[97] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[98] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Software and Hardware Techniques to Optimize Register File Utilization in VLIW Architectures. *International Journal of Parallel Programming*, 32(6):447–474, December 2004.

[99] H. Zhou and T. M. Conte. Code Size Efficiency in Global Scheduling for ILP Processors. In *The 6th Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 79–90, 2002.

[100] H. Zhou and T. M. Conte. Using Performance Bounds to Guide Pre-scheduling Code Optimizations. Technical report, N.C State University, 2002.

List of Publications

Journals (Refereed)

1. Arquimedes Canedo, Ben A. Abderazek, Masahiro Sowa, "A New Code Generation Algorithm for 2-offset Producer Order Queue Computation Model", *Journal of Computer Languages Systems & Structures*, Vol. 34, Number 4, pages 184-194, 2007, (Chapter 2, 3)

2. Arquimedes Canedo, Ben Abderazek, and Masahiro Sowa, "Natural Instruction Level Parallelism-aware Compiler for High-Performance Embedded QueueCore". *Journal of Embedded Computing, To appear in 2008*, (Chapter 3, 4.4, 6).

3. Ben A. Abderazek, Arquimedes Canedo, Tsutomu Yoshinaga, Masahiro Sowa, "The QC-2 Parallel Queue Processor Architecture", *Journal of Parallel and Distributed Computing*, Volume 68, Issue 3, pp. 235-245, 2008, (Chapter 2, 3, 6)

4. Arquimedes Canedo, Ben Abderazek, and Masahiro Sowa, "Compiler Support for Code Size Reduction using a Queue-based Processor". *Transactions on High-Performance Embedded Architectures and Compilers*, Volume 2, Issue 3, pp. 153-169, 2007, (Chapter 4.1).

Books (Chapters)

1. Ben Abderazek, Arquimedes Canedo, "Processor for Mobile Applications", Handbook of Research on Mobile Multimedia, 1st Edition, Information Science Reference, *To appear 2008*, (Chapter 3)

International Conferences (Refereed)

1. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Queue Register File Optimization Algorithm for QueueCore Processor", *In Proceedings of 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, Gramado, Brazil, pp. 169-176, 2007, (Chapter 4.2)

2. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "New Code Generation Algorithm for QueueCore - An Embedded Processor with High ILP" **Best Paper Award**, *In Proceedings of the Eight International Conference on Parallel*

and Distributed Computing Applications and Technologies (PDCAT'07), Adelaide, Australia, pp. 185-192, 2007, (Chapter 3, 4.4, 6)

3. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Quantitative Evaluation of Common Subexpression Elimination on Queue Machines", *In Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2008)*, Sydney, Australia, pp. 25-30, 2008, (Chapter 4.3)

4. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "An Efficient Code Generation Algorithm for Code Size Reduction using 1-offset P-Code Queue Computation Model", *In Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC07)*, Taipei, Taiwan, pp. 196-208, 2007, (Chapter 4.1)

5. Yuki Nakanishi, Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Optimizing Reaching Definitions Overhead in Queue Processors", *Journal of Convergence Information Technology*, Vol. 2, Number 4, December, pp. 36-39, 2007, (Chapter 4.3)

6. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Compiler Framework for an Embedded 32-bit Queue Processor", *In proceedings of the International Conference on Convergence Information Technology (ICCIT07)*, Gyeongju, South Korea, pp. 877-884, 2007, (Chapter 3)

7. Teruhisa Yuki, Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Novel Addressing Method for Aggregate Types in Queue Processors", *In proceedings of the International Conference on Convergence Information Technology (ICCIT07)*, Gyeongju, South Korea, pp. 1793-1796, 2007, (Chapter 3)

Others

1. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Queue Compiler Development", *In Proceedings of the Summer United Workshops on Parallel Distributed and Cooperative Processing (SWoPP'07)*, Asahikawa, Japan, 2007, (Chapter 2, 3)

2. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "A GCC-based Compiler for the Queue Register Processor (QRP-GCC)", *In proceedings of the International Workshop on Modern Science and Technology (IWMST06)*, Wuhan, China, 2006

3. Arquimedes Canedo, Ben A. Abderazek, Masahiro Sowa, "Queue Assembler", *Proceedings of the 67th IPSJ conference*, Tokyo, Japan, 2005

Currently Under Review

1. Arquimedes Canedo, Sayaka Akioka, Masahiro Sowa, "Compilation Techniques for Reducing Memory Traffic in Queue Processors", *IEEE/ACM International Symposium on Microarchitecture MICRO-41*, Submitted on May 2008, (Chapter 5)

2. Arquimedes Canedo, Ben A. Abderazek, Masahiro Sowa, "Compiling for Reduced Bit-Width Queue Processors", *Journal of Signal Processing Systems*, Springer, Submitted on April 2008, (Chapter 4.1, 6)

3. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Design and Implementation of a Queue Compiler", *Journal of Microprocessors and Microsystems*, Elsevier, Submitted on November 2007, (Chapter 3, 6)

4. Arquimedes Canedo, Ben Abderazek, Masahiro Sowa, "Efficient Compilation for Queue Size Constrained Queue Processors", *Journal of Parallel Computing*, Elsevier, Submitted on August 2007, (Chapter 4.2)

Author Biography

Arquimedes Canedo received his B.Eng. degree with honors of excellence in computer engineering from the National Polytechnic Institute (IPN) of Mexico in 2004. In 2006 he received his M.Eng. degree from the Graduate School of Information Systems of the University of Electro-Communications, Tokyo, where he is currently pursuing his doctor degree with the investigation of code generation, compilation, and optimization techniques for queue processors. His research interests include optimizing compilers, algorithms and data structures, computer languages, non-conventional computer architectures, embedded systems, and queue computing. He is member of IEEE and ACM.

Acknowledgments

First and foremost, I would like to thank my main advisor Prof. Masahiro Sowa for his guidance during the course of my doctoral research, for many hours invested in technical and philosophical discussions, for all the motivation and inspiration to become a better researcher and person, and for his faithful belief in queue computing. The pride of being his student and scholar will last forever. I would also like to extend my appreciation to my advisors, Professor Masanori Idesawa, Professor Toshihiko Kato, Professor Hiroki Honda, and Professor Tsutomu Yoshinaga.

I am grateful to all the members of the thesis supervisory committee for their valuable comments and suggestions during my doctoral defense: Professor Masahiro Sowa, Professor Masanori Idesawa, Professor Toshihiko Kato, Professor Hiroki Honda, Professor Tsutomu Yoshinaga, Professor Tsuneyasu Komiya, and Professor Masaaki Kondo.

My special thanks to Professor Ben Abderazek who, for the first half of my doctoral studies, supervised my research progress and gave me valuable comments and offered me his trust and friendship. And also to Professor Sayaka Akioka for helping me reviewing my thesis and papers.

To all the members of the Queue Compiler Group: Teruhisa Yuki, Akihiro Nagai, Yuki Nakanishi, and Keisuke Takaki, for pointing out technical issues and improvements for the queue compiler, for finding and fixing bugs, and for having the strong will to learn and discover new things.

I would like to thank, from the very core of me, to my parents, my brother, and my motherland México. Toinkins, gracias por todos los sacrificios hechos para darme educación.

Finally, all the cosmic propagation to my wife Petty Sukarsaatmadja. Thank you for all the encouragement, support, and very unique love.