1-1-2002

An analysis of reconfigurable memory queues/computing units architecture

Matthew John Patitz

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Patitz, Matthew John, "An analysis of reconfigurable memory queues/computing units architecture" (2002). Retrospective Theses and Dissertations. 20198.
https://lib.dr.iastate.edu/rtd/20198

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
An analysis of reconfigurable memory queues/computing units architecture

by

Matthew John Patitz

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee
Akhilesh Tyagi, Major Professor
Simanta Mitra
Arun Somani

Iowa State University
Ames, Iowa

2002

Copyright © Matthew John Patitz, 2002. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master's thesis of

Matthew John Patitz

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
TABLE OF CONTENTS

ABSTRACT............................................................................................................. vii

CHAPTER 1 OVERVIEW .......................................................................................... 1
  1.1 The balanced computing problem in modern processors ......................... 1
  1.2 A solution to the balanced computing problem ........................................ 2
  1.3 Thesis organization ..................................................................................... 3

CHAPTER 2 RECONFIGURABLE FUNCTIONAL UNITS ....................................... 4
  2.1 The DLX architecture ............................................................................... 4
  2.2 Pipelined functional unit structure ............................................................ 5
  2.3 Architectural changes for reconfigurable functional units ......................... 6
    2.3.1 Hardware modifications to functional units ....................................... 6
    2.3.2 Instruction set modifications ......................................................... 8
  2.4 Compiler support ...................................................................................... 9
  2.5 Architectural summary ............................................................................ 10

CHAPTER 3 COMPILER TOOLSETS ..................................................................... 11
  3.1 The SUIF2 Compiler Infrastructure ........................................................... 11
    3.1.1 The SUIF architecture ....................................................................... 11
      3.1.1.1 The kernel .................................................................................. 12
      3.1.1.2 Modules .................................................................................... 12
      3.1.1.3 Drivers ..................................................................................... 13
    3.1.2 SUIF program representation ............................................................ 13
  3.2 Machine SUIF ............................................................................................ 14
    3.2.1 Machine SUIF OPI ......................................................................... 15
    3.2.2 Usefull Machine SUIF libraries .................................................... 16
      3.2.2.1 The Machine SUIF Control Flow Graph Library .................... 16
      3.2.2.2 The Machine SUIF Control Flow Analysis Library ............... 17
    3.2.3 Machine SUIF summary ................................................................. 19

CHAPTER 4 OPERAND MAPPING TO REGISTER QUEUES .................................. 20
  4.1 General usage scenario .............................................................................. 21
    4.1.1 Analysis implementation ................................................................. 21
    4.1.2 Results .......................................................................................... 25
    4.1.3 Result analysis ............................................................................... 25
  4.2 Targeted circular queue usage .................................................................... 28
4.2.1 Potential targets ........................................................................................................ 29
  4.2.1.1 Blocked matrix multiplication ........................................................................ 29
  4.2.1.2 Loop unrolling .......................................................................................... 31
    4.2.1.2.1 Related work .................................................................................... 33
    4.2.1.2.2 Using circular register queues in unrolled, software pipelined loops .......... 36
    4.2.1.2.3 Loop analysis ................................................................................. 38
    4.2.1.2.4 Analysis implementation .................................................................. 38
    4.2.1.2.5 Results .......................................................................................... 39
  4.2.2 Targeting methods .......................................................................................... 40
    4.2.2.1 Programmer added directive ................................................................. 41
    4.2.2.2 Compiler pattern detection .................................................................... 41

CHAPTER 5 CONCLUSIONS AND FUTURE WORK ................................................................ 42
  5.1 Conclusion ........................................................................................................ 42
  5.2 Future work ...................................................................................................... 43

APPENDIX A: ANALYZED SOURCE CODE ................................................................ 44
APPENDIX B: OPERAND ACCESS PATTERN ANALYSIS RESULTS .................. 45
BIBLIOGRAPHY .............................................................................................................. 47
LIST OF TABLES

Table 4.1 First occurrence access patterns ................................................. 25
Table 4.2 Full depth access patterns .............................................................. 25
LIST OF FIGURES

Figure 2.1 Pipelined functional unit ................................................................. 6
Figure 2.2 Reconfigurable pipelined functional unit ........................................ 7
Figure 2.3 DLX instruction set modifications .................................................... 8
Figure 2.4 Example assembly code using register queues .......................... 9
Figure 3.1 Top levels of the SUIF IR ................................................................. 14
Figure 4.1 Operand access analysis, part 1 ....................................................... 22
Figure 4.2 Operand access analysis, part 2 ....................................................... 23
Figure 4.3 Operand access analysis, part 3 ....................................................... 24
Figure 4.4 First occurrence results graph ....................................................... 26
Figure 4.5 Full depth results graph ................................................................. 27
Figure 4.6 Algorithm and array access for standard matrix multiplication ...... 30
Figure 4.7 Algorithm and array access for blocked matrix multiplication ... 31
Figure 4.8 Loop unrolling example, part 1 ....................................................... 32
Figure 4.9 Loop unrolling example, part 2 ....................................................... 32
Figure 4.10 RQ architecture with 32 architected registers, queue depth 4, and 256 physical registers ................................................................. 35
Figure 4.11 Loop unrolling example, part 3 ....................................................... 37
Figure 4.12 Circular register contents during first loop iteration for Figure 4.11 . 37
Figure 4.13 Results of multiple live instance analysis ...................................... 40
Figure B.1 First occurrence results graph, separated by access type ............ 45
Figure B.2 Full depth results graph, separated by access type .................... 46
ABSTRACT

The clock speed of modern processors allows for the processing of huge amounts of data, but the limited amount of on chip data storage creates situations where the processing bandwidth of the processors overwhelm their memory bandwidth. A solution to this problem is to make portions of the processor dynamically configurable so that they can switch between functioning as processing units and storage units. Pipelined functional units can offer this functionality with only slight hardware and instruction set modifications. However, the additional register storage provided is in the form of a circular queue rather than a random access set of registers. The usefulness of such a set of registers was determined by studying operand access patterns and their potential mapping to such a circular queue. The results demonstrated that while few operand sets map well to circular queues in the general case, targeting specific operations such as matrix multiplication or loop unrolling provides an excellent opportunity to make use of these queues to reduce register pressure and therefore gain an overall speedup.
CHAPTER 1 OVERVIEW

In this chapter we look at the problem of unbalanced computing in modern processors. We introduce an architecture containing reconfigurable functional units that provides a solution for this problem. At the end of this chapter we describe the organization of this thesis.

1.1 The balanced computing problem in modern processors

For the past several decades, the processing speed of microprocessors has increased at a fantastic pace. This increase in computing speed has its roots in several different aspects of technological and architectural improvements. Among these are an amazing increase in the amount of transistors that can be etched into a microchip, improved instruction set designs, deeper pipelines, and many others. In order to feed the data demands of these ever faster processors, there has also been a similar explosion in cache, memory, and hard drive capacities. Although the amount of information that microprocessors can devour has been matched by storage capacity, another key element to the equation has been lagging: bandwidth. The speed at which the huge amounts of data can be shuttled to and from the processor’s engine hasn’t been able to keep pace. The result of this imbalance is a penalty in the net processing speed, causing actual instructions executed per cycle to fall far short of the processor’s potential.

A solution to this situation is of course an increase in memory and cache bandwidth and on chip register storage that is sufficient to feed the processor enough data to eliminate stalls. However, technical problems notwithstanding, the above solution still retains an intrinsic problem. To reach balance (Kung introduced this term [3]) in computing, the memory bandwidth and data requirements would be exactly equal, resulting in an ideal situation with no stalls and complete memory bandwidth utilization. By simply increasing bandwidth by a fixed amount, the processor is still provided with a fixed quantity of data. Real world computations,
though, vary greatly in their data and computation requirements. This means that while one particular computation may be balanced in its data and computation requirements, other computations will stall due to insufficient memory bandwidth, while still others will underutilize the memory bandwidth.

1.2 A solution to the balanced computing problem

To solve this fundamental problem, Tyagi [9] has proposed making two pieces of the equation for balanced computing dynamically configurable. His proposal is to make computation bandwidth, the quantity of data that can be simultaneously processed by the processor, and local memory size, the amount of data that can be stored on-chip in registers, dynamically tradeable. In so doing, the data demands of the processor for a given computation can be more closely met by both increasing available storage and decreasing computation bandwidth. As the nature of the computation shifts to require more computation bandwidth, the opposite can be done, therefore dynamically shifting to bring about a situation that is closer to balanced.

In order to realize the potential benefits of dynamically tradeable computation bandwidth and local memory size, functional units capable of performing both computation and data storage must be identified or created. Modern processors have a large number of separate functional units. These units are each designed for special purpose computations, such as integer division or floating point multiplication. While each functional unit is obviously necessary for a particular type of computation, several of them are completely unused while the processor performs other computations. Floating point arithmetic units are a good example of functional units that remain idle for large amounts of time. Besides their copious idle time, these units also have a deeply pipelined structure with registers between each pipeline stage. With such storage capacities already existing, these functional units are excellent candidates for use in increasing local memory size during periods where they are unused.

The proposed solution to the problem of unbalanced computing is to make the necessary hardware and instruction set modifications to enable a set of pipelined functional units to dynamically switch their functionality between computational units and additional register sets. This allows flexibility in the computation and memory
bandwidths of the processor so that it can dynamically change to match the nature of particular computational tasks, therefore increasing the likelihood of balanced computing.

1.3 Thesis organization

The remainder of this thesis is organized as follows. Chapter 2 describes the architecture to be used and the changes required to support the reconfigurable functional units. Chapter 3 gives a description of the SUIF and MachineSUIF toolsets used, and expanded on, for the operand analyses performed later in the thesis. Chapter 4 details the different operand analyses performed to determine the feasibility of circular register queue usage and the results of those analyses. Chapter 5 is a brief summary of the results and potential future work.
CHAPTER 2 RECONFIGURABLE FUNCTIONAL UNITS

In this chapter we present the architecture that we will be working with and the modifications to that architecture needed to create the reconfigurable functional units discussed in Chapter 1.

2.1 The DLX architecture

The processor architecture that we will be using as the basis for our implementation will be the DLX [2] architecture. DLX is a simple RISC-type architecture. Some of the important features of DLX are described below:

DLX is a 32-bit word oriented architecture with the following data types:
  * Integer data types:
    * 8-bit bytes
    * 16-bit half words
    * 32-bit words
  * Floating point data types
    * 32-bit single precision values
    * 64-bit double precision values

DLX has two register sets:
  * General purpose register set (GPR) consisting of 32 32-bit registers
    * Register R0 hardwired to the value 0
  * Floating point register set (FPR) consisting of 32 floating point registers that can be used as follows:
    * 32 32-bit single precision floating point registers
    * Pairs of even/odd registers to hold double precision floating point values (e.g. F0, F2, ...)

DLX is a simple load/store instruction set designed for pipeline efficiency:
  * Memory references are load/store between memory and GPR's or FPR's
  * Two addressing modes: immediate and displacement
- Register deferred and absolute addressing with 16-bit field can be accomplished using R0
- All accesses must be aligned and for GPR's can be to byte, half, or word

DLX instruction types:
- I-type instruction
  - Format: \textit{op rs1 rd immed}
  - Usage:
    - Loads and stores
    - All immediate value operations
    - Conditional branch
    - Jump register, jump and link register
- R-type instruction
  - Format: \textit{op rs1 rs2 rd func}
  - Usage:
    - Register-register ALU operations
- J-type instruction
  - Format: \textit{op offset}
  - Usage:
    - Jump, jump and link
    - Trap, return from exception (RFE)

(For each instruction type format \textit{op} is 6 bits, \textit{immed} is 16 bits, \textit{func} is 11 bits, \textit{offset} is 26 bits, and all \textit{r*} fields are 5 bits)

\section*{2.2 Pipelined functional unit structure}

The basic structure of pipelined functional units is shown in Figure 2.1. That figure displays a functional unit that consists of four pipeline stages and has two inputs operands to each stage. Each stage consists of the computational logic that is performed on the two inputs, \textit{A} and \textit{B}, and is performed in one clock cycle. The pipeline latches are sets of registers that store the intermediate values of the operands between stages.

At the beginning of each clock cycle, the operand values are transferred from the pipeline latches that serve as the inputs to each stage and then through the computational logic of the proceeding stage. After each stage there is a pipeline
Pipeline latches

Stage 1
Stage 2
Stage 3
Stage 4

Source A
Source B

Figure 2.1 Pipelined functional unit

latch that stores the results. At the end of four clock cycles the resultant values will be available in the final pipeline latch.

2.3 Architectural changes for reconfigurable functional units

To provide the functionality of reconfigurable, pipelined functional units, the DLX architecture requires only slight modifications. Both the functional units themselves and the instruction set must be enhanced.

2.3.1 Hardware modifications to functional units

Given the existing structure of pipelined functional units, significant architectural changes would be required to create a random access type register set that is similar to the existing DLX register sets. However, with just the simple addition of a multiplexor at each pipeline stage that would allow the latched operands to bypass the computational logic of each stage, these pipelined units could easily be transformed into a circular register queue. These modifications are shown in Figure 2.2.
Logic bypass signal

Source A

Source B

2:1 Multiplexors

Stage 1

Stage 2

Stage 3

Stage 4

Figure 2.2 Reconfigurable pipelined functional unit

When the logical bypass signal is set, the 2:1 multiplexors will allow the input operand values to bypass the computation logic at each stage. Values will be moved forward one pipeline stage per cycle, unmodified, and with another slight addition, from the end back to the beginning of the queue if necessary. The result is a circular register queue that is four entries deep.

To minimize the changes required for the instruction set as well as the architectural changes, the only input and output ports for these register queues would be the beginning and the end of the pipeline, respectively. However, this constraint could be relaxed if experimental data were to show typical register access patterns that don't conform well to the necessary latencies required by this design, at a slight penalty of additional hardware complexity. These parameters provide for flexibility in both the queue design and the instruction set design. For instance, read and write ports could be placed at locations in the queues that allow for them to leverage the latency displayed by a significant portion of operands.
### Additional Instructions for functional unit reconfiguration

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Q1</td>
<td>Configures functional unit Q1 to behave as a circular register queue. This will cause the logic bypass signal shown in Figure 2.2 to be set.</td>
</tr>
<tr>
<td>Function Q1</td>
<td>Configures functional unit Q1 to behave as a functional unit, executing the computational logic at each stage. This will cause the logic bypass signal shown in Figure 2.2 to be off.</td>
</tr>
<tr>
<td>Toggle Q1</td>
<td>(optional) Causes the behavior of functional unit Q1 to toggle to the opposite configuration.</td>
</tr>
<tr>
<td>Advance Q1</td>
<td>(optional) Causes the data in the pipeline latches of functional unit Q1 to rotate forward by one position.</td>
</tr>
<tr>
<td>Add Q2, Q1, R5</td>
<td>Add the contents of register R5 and the data at the output port of queue Q1, and put the result into the input stage of queue Q2.</td>
</tr>
</tbody>
</table>

---

#### 2.3.2 Instruction set modifications

The instruction set modifications required for the DLX instruction set were suggested by Tyagi [9] and are listed in Figure 2.3. The reconfigurable functional units are represented by the set \{Q0, Q1, \ldots, Qk\}.

Figure 2.4 shows an example snippet of code using such a queue. In that example, the queue depth is assumed to be 4, and both read and write operations to the queue cause the queue to rotate one position. The sample code loads four values from memory and sums them into R5. Furthermore, the instruction set could be extended to include additional instructions that, while reading and writing to the queues, can affect the queues in different ways, such as reading and rotating the queue and reading and not rotating the queue.
lw Q1, 0(R4)
lw Q1, 4(R4)
lw Q1, 8(R4)
lw Q1, 12(R4)
add R5, R0, Q1
add R5, R5, Q1
add R5, R5, Q1
add R5, R5, Q1

Figure 2.4 Example assembly code using register queues

In order to be able to address the circular register queues formed by the reconfigured functional units, the register fields of instructions must be expanded. DLX currently uses 5-bit register fields, so adding a single bit allows for 32 additional targets which is sufficient to accommodate the number of reconfigurable functional units available. To provide this extra bit, the size of the *immed* field in I-type instructions would have to shrink from 16-bits to 14-bits. Additionally, the *func* field in R-type instructions would have to shrink from 11-bits to 8-bits. Both field size changes are easily accommodated in the DLX architecture, with the only penalty being a slightly larger number of already existing instructions needed to perform operations that require immediate values of 16-bits.

### 2.4 Compiler support

The compiler is responsible for register allocation, which is the process of assigning operands to registers. The complexity of this task greatly increases with the addition of circular register queues due to the latency between the writing of an operand and the cycle at which it is next available for reading. The compiler must only schedule operands to a queue that match the queue’s latency. The determination of sets of operands that map well to queues could either be done by predefined notations in the source code or by enhanced pattern detection within the compiler.
2.5 Architectural summary

Given the above dynamically configurable functional units, it must be shown that they can help achieve a situation closer to balanced computing for a sufficient proportion of computational tasks. Although they can be used to increase local memory size by trading off computational bandwidth (by removing themselves from the available pool of computational resources), the circular queue nature of the register storage they provide must be useful to a general class of computations. The remainder of this paper will concentrate on different strategies for making use of register queues and the results of analyses based on these strategies, as well as the toolsets used to analyze those strategies.
CHAPTER 3 COMPILER TOOLSETS

In the following chapters we will discuss several methods of operand analysis that we performed to determine how best to make use of circular register queues for reducing register pressure and increasing balance in computational tasks. In order to perform these analyses, we made use of two major compiler toolsets: the SUIF2 Compiler Infrastructure and the Machine SUIF infrastructure for creating compiler back ends. These two toolsets will be briefly introduced in this chapter.

3.1 The SUIF2 Compiler Infrastructure

The Stanford Compiler Group, with funding from DARPA and NSF has created the SUIF2 compiler infrastructure [1], which is a modular and extensible compiler system that is based upon a program representation known as SUIF (Stanford University Intermediate Format). SUIF is a formatted program representation that encapsulates each level of program decomposition and allows for various types of analysis and transformation at all stages of compilation. The SUIF2 infrastructure is designed to provide useful abstractions and frameworks for the development of new compiler passes and an environment in which existing and new compiler passes can effectively and efficiently cooperate.

3.1.1 The SUIF architecture

The architecture of the SUIF system [1] is composed of three main divisions: the kernel, which implements the basic functionality used by all compilation passes, a set of dynamically loaded modules controlled by the user, and a driver that controls the operation of the system.
3.1.1.1 The kernel

The kernel is composed of two layers: the IO kernel and the SUIF kernel. The function of the IO kernel is to implement a persistent object system that is used to handle the reading and writing of data structures. This is done by writing out data structures known as meta objects that contain information about the compositions of the data structures along with the data structures. The IO kernel can then restore data in the form of data structures from files without having been compiled with the definitions of those objects. This is done in such a way as to be independent from the individual modules and thus insulates this functionality from an author of a compiler pass.

The function of the SUIF kernel is to define and implement an object known as the SUIF compiler environment (SuifEnv). The SuifEnv contains all of the data that a user needs to know when writing a SUIF program, including the entire state and loaded components of the compiler system, which accounts for all global data used by all compiler modules. The SuifEnv contents are the current intermediate program representation, an object factory for creating new persistent objects, and the ModuleSubSystem. The ModuleSubSystem is the object that keeps track of all of the currently registered modules in the system. Its two main functions are to register modules when libraries are loaded and to invoke modules with the proper arguments when necessary.

3.1.1.2 Modules

The SUIF compiler system consists mainly of dynamically loaded modules. These modules are each C++ classes that must be registered within the ModuleSubSystem in the SuifEnv, via the SUIF kernel. Then, when a module is to be invoked, the ModuleSubSystem initializes variables within the module, passes command line arguments to the module, then executes it.

A module can be either a set of nodes within the intermediate representation, or IR, (such as the basicnodes module which contains representations of several basic programming constructs, or the suifnodes module which contains representations of standard programming constructs in high level programming languages such as C and Fortran), or a program analysis pass.
Program analysis passes, or compiler passes, are code transformations that are performed on the current program representation at the time the pass is executed, and can be defined to be applied to any of the various components in a program representation, such as the global symbol table, the procedure definitions, or the variable definitions. They can be derived from either the \textit{Pass} class or the \textit{PipelineablePass} class (which is derived from the \textit{Pass} class). The standard method of application for a compiler pass is for the pass to be performed on all procedures in a SUIF program before the next pass is invoked, and this is the case for a pass derived from \textit{Pass}. However, to support passes that should be performed in a pipelined fashion, a pass can be derived from \textit{PipelineablePass}. This means that a series of different passes could be performed on a single procedure before moving on to the next procedure. This improves the locality of the compiler and can therefore be useful for compiling large programs.

3.1.1.3 Drivers

A driver is the "main" program that executes to perform a series of passes on a program. Drivers perform the following tasks, in order:
1. Create the SuifEnv
2. Import all required modules
3. Load a SUIF program
4. Apply a series of passes to the SUIF program
5. Output the resulting SUIF program

3.1.2 SUIF program representation

The SUIF program representation is provided by an extensible class hierarchy which attempts to capture all the concepts useful in compilation. This object hierarchy captures the different levels of abstraction from high-level programming language semantics to hardware description language primitives, and can easily be extended to accommodate new requirements.

The basic SUIF Object Hierarchy, made up of a set of IR nodes, is shown in Figure 3.1 [1]. Briefly, the \textit{Object} class is used to contain the meta data about a
Figure 3.1  Top levels of the SUIF IR

class for use by the IO Kernel, the SuijObject class provides all the basic user-level functions such as printing and cloning, and the AnnotableObject provides a framework on which to hang annotations which contain all derived data about a program. A FileSetBlock is the root representation of a program that contains some global information about the program, symbol tables, procedure definitions, etc. ScopedObjects contain the definitions of most of the objects in a program, including procedure and variable definitions, and can be further subclassed into ExecutionObjects into which fall all IR objects representing computations. Statements and Expressions represent the equivalent structures within high level programming languages.

3.2 Machine SUIF

Michael D. Smith and Glenn Holloway of the Division of Engineering and Applied Sciences at Harvard University created a similar compiler infrastructure that
ties into the SUIF2 infrastructure and provides a flexible and extensible platform for the construction of compiler back ends, called Machine SUIF [6]. Using Machine SUIF, machine-level intermediate forms can be constructed and manipulated, and the output can range from binary object code, to assembly language code, to C code. Back ends are provided for Alpha and x86, and it is simple to add additional machine targets.

Machine SUIF provides a framework for creating compiler optimization and analysis passes that are as nearly as possible independent from the compiler environment and compilation targets. Although the Machine SUIF distribution contains a working compiler based on the SUIF compiler, the analysis and optimization passes in the distribution do not directly reference any SUIF constructs or contain any constants based on the target machine. Instead, Machine SUIF makes use of an interface layer referred to as the Optimization Programming Interface (OPI), which encapsulates a standardized view of the underlying compiler environment.

3.2.1 Machine SUIF OPI

Machine SUIF can be thought of as a machine-level IR that represents a "lowered" version of SUIF, with class Inst roughly correlating to SUIF's Expression or Statement, and many additional libraries and passes that allow machine-specific optimizations for existing or future architectures. The "lowering" of SUIF code is the conversion from the SUIF IR to the Machine SUIF IR and can be accomplished using a tool provided with the Machine SUIF distribution.

The Machine SUIF IR has been designed so that it is extensive and expressive, with a 1:1 mapping of machine instructions to IR instructions in order to support fully functional optimization passes. However, to avoid the major difficulties that could be encountered when trying to change the IR and therefore every dependent pass, the OPI hides the implementation details of the IR. The functionality needed by the writer of an optimization pass is exposed by the OPI without revealing the structure of the underlying IR.

Additionally, the OPI provides a standard Machine SUIF IR known as suifvm (SUIF virtual machine). All Machine SUIF IR files share this binary representation and are merely interpreted differently for each target.
3.2.2 Useful Machine SUIF libraries

In order to perform the operand analyses described in the next chapters, we made use of the following additional libraries and functionality from the Machine SUIF distribution.

3.2.2.1 The Machine SUIF Control Flow Graph Library

Typically, procedures are represented as sequential lists of instructions that execute in order chronologically, with the exceptions of branch or jump statements that can cause execution to flow in a non-linear path. To better allow for many types of optimizations and analyses, different models of control flow can be very useful. One such model is a control flow graph (CFG). A control flow graph is a directed graph that represents a procedure as a series of interconnected nodes, each of which is composed of a basic block. A basic block is a sequential set of instructions with no control flow branches in the body, meaning that the order of execution of instructions in the body always proceeds linearly from the first instruction in the body to the last. Control flow can only branch at the beginning of the basic block and the end. Therefore, the CFG is formed with nodes representing basic blocks which are connected only to their ancestors and predecessors.

The Machine SUIF Control Flow Graph (CFG) Library [8] provides an abstraction of control flow graphs. This library provides the ability to translate the linear instruction flow representation to and from CFG form, transformation of the derived CFG by adding, removing, and reconnecting nodes, as well as low level control over the individual instructions within nodes. The layout of the nodes can be precisely controlled to ensure that when the CFG is re-linearized, the instructions are in the desired ordering.

The building blocks of a Machine SUIF CFG are the individual nodes in the graph, which are represented by instances of the class CfgNode. CfgNodes within a CFG can be accessed by a number or by following the graph flow through the list of successors or predecessors of a given node. The lists representing the successors and predecessors of a node can be modified by adding and removing entries,
thereby changing the control flow paths within the CFG. The contents of individual nodes, which are the instructions themselves, can also be accessed. Instructions can also be added, removed, or modified within a node. An example of a case for instruction modification is when the successor to a node, $X$, is switched to a newly created node, $Y$. In order for this to succeed, a branch or jump statement at the end of node $X$ must be changed so that its target becomes the initial instruction in node $Y$.

The Machine SUIF CFG library provides an excellent framework for the construction and modification of control flow graphs. However, it is lacking in tools for control-flow and data-flow analysis, which is why the next library is useful.

### 3.2.2.2 The Machine SUIF Control Flow Analysis Library

Based on Machine SUIF's CFG library, the Control Flow Analysis (CFA) library [7] extends it and adds a few important tools that are useful when analyzing the dynamic flow of execution paths through a procedure, or its control flow. The CFA library provides two different sets of control flow analysis tools: dominator analysis and natural loop analysis.

In order to understand the CFA library implementation, the following definitions are provided from [4]:

- **Dominance**: binary relation on CFG nodes. We say that node $d$ dominates node $i$, written $d \text{ dom } i$, if every possible execution path from 'entry' to $i$ includes $d$. Clearly, $\text{dom}$ is reflexive (every node dominates itself), transitive (if $a \text{ dom } b$ and $b \text{ dom } c$, then $a \text{ dom } c$), and antisymmetric (if $a \text{ dom } b$ and $b \text{ dom } a$, then $b = a$). We further define the subrelation called **immediate dominance** ($\text{idom}$) such that for $a \sim b$, $a \text{idom } b$ if and only if $a \text{ dom } b$ and there does not exist a node $c$ such that $c \sim a$ and $c \sim b$ for which $a \text{ dom } c$ and $c \text{ dom } b$, and we write $\text{idom}(b)$ to denote the immediate dominator of $b$. Clearly the immediate dominator of a node is unique. The immediate dominance relation forms a tree of the nodes of a CFG whose root is the 'entry' node, whose edges are the immediate dominances, and whose paths display all the dominance relationships.
• Dominance frontier: for a CFG node x, the dominance frontier of x, written \( DF(x) \), is the set of all nodes y in the CFG such that x dominates an immediate predecessor of y but does not strictly dominate y.

• Natural loop: given a back edge \( m \rightarrow n \), the natural loop of \( m \rightarrow n \) is the subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the CFG without passing through n and the edge set connecting all the nodes in its node set.

The CFA library’s dominator class, \( DominanceInfo \), provides the functionality to traverse the CFG in either the forward or reverse direction and to determine the following data about a CFG node, n, in the forward direction (and its opposite in the reverse direction):

- The immediate dominator of n.
- The set of dominators of n.
- The dominance frontier of n.

The natural loop analysis class, \( NaturalLoopInfo \), provides the functionality to compute information about the natural loops that exist in a CFG. This class uses an instance of the DominanceInfo class as a basis for constructing the sets of nodes that form natural loops in a procedure.

Given a CFG node n or that node’s number, a NaturalLoopInfo object can return the following information:

- A natural set of the CFG nodes that are in the same natural loop as n
- The loop depth of n, meaning how many nested levels of loops contain n
- Whether or not n is the beginning node of a natural loop, meaning the entry point for a loop
- Whether or not n is the ending node of a natural loop, meaning a node that jumps back to the entry node
- Whether or not n is an exit node, meaning a node from which the loop can be exited
3.2.3 Machine SUIF summary

Using the functionality provided by Machine SUIF and its included CFG and CFA libraries, it is possible to create control flow graphs for procedures and to extract data about the dynamic control flow, which can be analyzed at a very granular level. These analyses can provide the information necessary to perform many types of optimizations, and the interfaces to the data structures provided by these libraries allow for easy modification of the structure, sequencing, and logic of the underlying instructions.
CHAPTER 4 OPERAND MAPPING TO REGISTER QUEUES

Modern processors provide sets of registers that can be individually accessed in random order. These register sets are frequently a set of integer registers and a set of floating point registers, with each register in each set being uniquely addressable and randomly accessible. Given this paradigm, a compiler, which controls register allocation, can treat each register and register data access as independent from all other registers.

However, to make use of the circular register queues formed from the reconfigured functional units which were described in Chapter 2, the compiler must perform a much more intelligent scheduling task. To leverage speedup from the additional register space in the queues, the queues must only be assigned operand values that are accessed with frequencies that map well to the depth and configuration of the queues. This means that the queues must have well placed read and write ports, and the latency between operand accesses must be the same as that provided by the depth and rotation of the circular queue. For a data operand to be effectively mapped into a queue of depth $d$, the read of the operand must occur at least $d-1$ cycles after its write. This means that to fill the queue and benefit from its use, a set of operands of size $d$ must be identified that can be simultaneously mapped to the queue. This set of operands should be accessed multiple times and in the same order each time, with write access occurring only when the operand is at the write port and read access only when it is at the read port for maximum benefits. Of course, using the Advance instruction suggested above, queue data could be rotated as necessary, though with a single cycle penalty per use.

With the above constraints inherent in the scheduling of operands to a circular register queue, we decided to perform a series of operand access analyses to determine which code sections or scenarios may provide the best potential for efficient queue usage. Throughout the rest of this chapter, we will describe the analyses we performed, their implementation, and the results we received when we performed them on various benchmarks.
4.1 General usage scenario

The simplest and most comprehensive use of circular register queues would occur if the frequency and latency of accesses to sufficiently large sets of operands map well to the available queues throughout the entirety of an average section of code. The first operand access analysis that we performed involved testing this general usage scenario in order to determine if operand mapping to circular queues would be feasible in the general scenario, or if it would instead be necessary to target more specific code structures for conformity of operand access patterns to the circular register queue model.

The basic goal of this analysis was to generate statistics on the general trends of operand access. This includes the number of times that operands are accessed, the types of accesses (read or write), the ordering of the access types (RR, RW, WR, WW), and the latency, or number of clock cycles, between each access. The implementation and results of this analysis follow.

4.1.1 Analysis implementation

With the goal of analyzing every operand access and each possible ordering of those accesses, it was necessary to convert the targeted code into a representation that allowed for easy access to individual instruction components as well as allowing easy discovery and traversal of all possible code execution sequences. Since the Machine SUIF CFG representation provides this functionality, we first converted 'C' code into SUIF representation using the tools provided with the SUIF2 distribution, and then performed the 'lowering' to Machine SUIF’s suifvm representation using the Machine SUIF toolset. The next step was to run the resultant suifvm code through our own SUIF compiler pass which performed the following analysis:

1. Create a control flow graph of each procedure using the CFG library.
   Let \( n \) be the number of nodes in the CFG. An example of this can be seen in Figures 4.1 and 4.2.
2. Iterate over every node in the CFG, from 1 to \( n \). For each node, \( N \), record every path along the CFG originating from \( N \) for a predetermined number of nodes \( d \) (each node approximately
Operand access analysis example

Sample 'C' code:

```c
int a, b, c, d;
a = 0;
b = 2;
c = 0;
d = 10;
if (a > b) {
    c = a;
} else {
    c = b;
}
while (c < d) c++;
```

Equivalent assembly (split into basic blocks):

```assembly
$BLOCK1:
    move $2, $0 # a = 0
    li $3, 2 # b = 2
    move $4, $0 # c = 0
    li $5, 10 # d = 10
    slt $6, $3, $2 # if (a > b)
    beq $6, $0, $BLOCK3

$BLOCK2:
    if (a > b) {
        move $4, $2 # c = a
        j $BLOCK4
    }

$BLOCK3:
    move $4, $3 # c = b

$BLOCK4:
    slt $6, $4, $5 # while (c < d)
    beq $6, $0, $BLOCK6

$BLOCK5:
    addi $4, $4, 1 # c = c + 1
    j $BLOCK4

$BLOCK6:
```

Figure 4.1  Operand access analysis, part 1

representing one basic block, with \( d \) equal to 4 for the results included). This resulted in a complete set, \( P_i \), of the control flow paths originating at node \( N \) and of the specified depth (or shorter if the path happens to contain a node with no successor in the procedure, i.e. the procedure exit) that occur during program execution. The end result of this step was the set of \( \{P_1, P_2, \ldots, P_n\} \), which included all possible control flow paths from every node in the procedure, with a maximum path length of \( d \).

3. For each \( P_i \), traverse each of the recorded paths and create a list of the operands that are accessed in each instruction of each node, along with their position and type (source or destination). The result is the set \( \{L_1, L_2, \ldots, L_n\} \), where \( L_i \) is the set of lists of operand accesses for each control flow path of depth \( d \) beginning at CFG node \( i \).
4. Iterate through the set of operand sequences created in Step 3. For each sequence, $S_k$, in $L_i$ iterate through each operand access, $a$, of each instruction within the first CFG node of that sequence. For every $a$, generate the list of the additional accesses to that operand that occur within $S_k$, recording the distance between the first access, $a$, and subsequent accesses to that same operand. This record contains both the distance between the accesses and the types of the accesses (e.g. RR: read followed by read, RW: read followed by write, WR: write-read, and WW write-write). Figure 4.3 shows an example operand access analysis based on the code and CFG from Figures 4.1 and 4.2, respectively. Note that since this analysis is performed before register allocation, the registers being referenced are actually virtual registers.
Operand accesses corresponding to control flow path "BLOCK1 → BLOCK2 → BLOCK4 → BLOCK5":

(Read accesses of register $0$ are ignored because it is a special purpose register that always contains the value '0')

Write $2$
Write $3$
Write $4$
Write $5$
Read $3$, Read $2$, Write $6$
Read $6$
Read $2$, Write $4$
Read $4$, Read $5$, Write $6$
Read $6$
Read $4$, Write $4$

Example Full Depth Access Patterns:

$2$ (access type: instruction distance)

- RR: 2
- RW: <none>
- WR: 4, 6
- WW: <none>

$4$

- RR: 2
- RW: 0, 2
- WR: 1, 3, 5, 7
- WW: 3, 4, 7

Figure 4.3  Operand access analysis, part 3

Additionally, Step 4 was performed in two different ways. The first method was to trace from an operand access in the first block of a path all the way through the entire path, recording every subsequent access of that operand. This method was referred to as the 'full depth search.' The second method was to do the same trace but stopping after the first access of that operand. Therefore, the second method only records the distance between an access and the very next access of the same operand, while the first method records the distances between each access of an operand as long as it is within the node depth specified earlier. This method was referred as the 'first occurrence search.'

The results of this analysis are included in the following section. Note that the distance between accesses was only recorded as long as it was less than 30 instructions.
4.1.2 Results

Tables 4.1 and 4.2 display the results of the analysis described in Section 4.1.1 above when performed on the source code listed in Appendix A, but for convenience an instruction separation distance of up to 15 is shown. Figures 4.4 and 4.5 display the complete sets of results, up to an instruction separation of 30, in the form of graphs. Additionally, graphs of the results separated for each type of access (RR, RW, WR, and WW) are located in Appendix B.

4.1.3 Result analysis

As is quite obvious from both sets of results, the most common access pattern for operands occurs when an operand is written to a register and then read on the very next instruction. Other than that obvious trend, it can also be seen that read after read (RR) access patterns occur with a generally decreasing frequency as the instruction distance increases, though these types of accesses are the most

<table>
<thead>
<tr>
<th>Distance</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>106</td>
<td>127</td>
<td>147</td>
<td>162</td>
<td>175</td>
<td>137</td>
<td>104</td>
<td>126</td>
<td>97</td>
<td>98</td>
<td>74</td>
<td>72</td>
<td>75</td>
<td>69</td>
<td>83</td>
</tr>
<tr>
<td>RW</td>
<td>552</td>
<td>102</td>
<td>24</td>
<td>27</td>
<td>13</td>
<td>27</td>
<td>5</td>
<td>10</td>
<td>17</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>14</td>
<td>19</td>
<td>22</td>
<td>15</td>
</tr>
<tr>
<td>WR</td>
<td>0</td>
<td>10716</td>
<td>2153</td>
<td>749</td>
<td>988</td>
<td>576</td>
<td>451</td>
<td>197</td>
<td>212</td>
<td>183</td>
<td>114</td>
<td>170</td>
<td>117</td>
<td>66</td>
<td>73</td>
<td>42</td>
</tr>
<tr>
<td>WW</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.2 Full depth access patterns

<table>
<thead>
<tr>
<th>Distance</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>629</td>
<td>183</td>
<td>163</td>
<td>182</td>
<td>196</td>
<td>225</td>
<td>160</td>
<td>196</td>
<td>179</td>
<td>163</td>
<td>191</td>
<td>143</td>
<td>139</td>
<td>162</td>
<td>197</td>
</tr>
<tr>
<td>RW</td>
<td>552</td>
<td>102</td>
<td>24</td>
<td>30</td>
<td>18</td>
<td>32</td>
<td>10</td>
<td>18</td>
<td>21</td>
<td>21</td>
<td>33</td>
<td>23</td>
<td>21</td>
<td>27</td>
<td>27</td>
<td>24</td>
</tr>
<tr>
<td>WR</td>
<td>0</td>
<td>10716</td>
<td>2766</td>
<td>845</td>
<td>1021</td>
<td>611</td>
<td>477</td>
<td>235</td>
<td>243</td>
<td>220</td>
<td>141</td>
<td>209</td>
<td>138</td>
<td>79</td>
<td>101</td>
<td>71</td>
</tr>
<tr>
<td>WW</td>
<td>0</td>
<td>543</td>
<td>9</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>3</td>
<td>9</td>
<td>16</td>
<td>8</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>18</td>
<td>31</td>
</tr>
</tbody>
</table>
Operand Access Patterns for First Occurrence

Figure 4.4  First occurrence results graph
Figure 4.5  Full depth results graph
predominant after WR accesses with instruction distance equal to 1. The only exception to this is that the RW access pattern within the same instruction has a large frequency.

These patterns show that the life span of the vast majority of operand values is very short, usually on the order of only a single instruction. Therefore, the most basic model of the circular queues will most likely be insufficient as a model to which a great deal of the operands can be mapped, since it assumes a single write port at the beginning of the queue and a single read port at the end. It also assumes that accesses to the queue automatically rotate the contents of the queue.

In order to effectively support a larger amount of operand access patterns, it would be necessary to put a read port in the pipeline stage immediately following the stage containing the write port, or at the same stage as the write port and an instruction that doesn't rotate the queue on a write would then be necessary. The second option would be ideal for handling the RW access patterns within the same instruction if an instruction was available that performs a read with no queue rotation. To provide for the RR accesses of different instruction distances, it will be necessary to combine a series of rotating and non-rotating accesses on queues of appropriate depths.

With the above data on typical operand access patterns, which showed that general operand accesses don't map well to the circular queue model, it was necessary to look for more specific scenarios that could benefit by using this model.

### 4.2 Targeted circular queue usage

Given the lack of suitable access latencies and frequencies for the majority of operands to be mapped into circular register queues in a general code segment, we turned to the search for specific circumstances or code structures whose operand accesses may lend themselves to such a mapping more readily. If the general occurrence of a specific code structure, such as a particular function, operation, or algorithm, could be found to be a good fit for circular register queue usage, it would then be possible to target occurrences of that structure by the compiler for register queue use.
4.2.1 Potential targets

Given the need to discover specifically targetable code structures that can efficiently use circular register queues to decrease register pressure, we looked for circumstances that fit the following criteria:

1. They involve repetitive access to data elements.
2. The data sets requiring repetitive access are of sufficient size to fit into a set of available reconfigured register queues.
3. They typically experience high amounts of register pressure, and are therefore likely to experience a notable speedup from a decrease in register pressure that could be created by effective use of the register queues.

The remainder of this section describes two code structures that we found to fit the above criteria, and the results of analyses we performed on their operand access patterns.

4.2.1.1 Blocked matrix multiplication

Matrix multiplication is an excellent example of a procedure that is a good fit for criteria 1 and 3 above. The product of multiplying an mXn matrix A and an nXp matrix B is an mXp matrix, C, whose elements are calculated as follows:

\[ C_{ik} = A_{i1} \times B_{1k} + A_{i2} \times B_{2k} + \ldots + A_{in} \times B_{nk} \]

The algorithm for performing this multiplication in C code is shown in Figure 4.6, along with the pattern of accesses to the elements of the matrices. The repetitive read accesses to the data elements in X and Y can be clearly seen, and for an N of any significant size criteria 3 will be met, as there will be considerable register pressure that causes a huge amount of thrashing, or loading, replacing, and reloading of operand values due to limited available register space. However, since the value of N, and therefore the size of the sets of data that are regularly cycled
through, is determined solely by the dimensions of the arrays, there is no way to ensure that criteria 2 is met.

A standard optimization that has been applied to matrix multiplication in order to solve the problem of data set size and thrashing is **blocking**. Blocking is a slight modification to the matrix multiplication algorithm that causes the size of the repeating data set to be defined as a constant, which can be set to a value appropriate for mapping to a circular register queue. An example of blocked matrix multiplication, along with the pattern of accesses to the elements of the matrices can be seen in Figure 4.7.

When blocking is used, the operand access patterns change in such a way that operand accesses map perfectly to a circular queue. Using blocking, accesses to $X$ remain constant for $B$ iterations, where $B$ is the blocking factor, and the elements of $Y$ that are accessed are still not mappable to circular queues. The access pattern for array $Z$, though, is now perfectly suited for a circular queue whose depth is equal to $B$.

With such a good fit mapping to circular register queues, the high register pressure that exists during even blocked matrix multiplication can be substantially
Blocked Matrix Multiplication:

```c
int B = BLOCKSIZE;
int N = ARRAYSIZE;
int X[N][N], Y[N][N], Z[N][N];
...
for (kk=0; kk<=N; kk=kk+B) {
    for (jj=0; jj<=N; jj=jj+B) {
        for (i=0; i<=N; i=i+1) {
            for (k=kk; k<=MIN(kk+B-1, N); k=k+1) {
                r = X[i][k];
                for (j=jj; j<=MIN(jj+B-1, N); j=j+1) {
                    Z[i][j] = Z[i][j] + r*Y[k][j];
                }
            }
        }
    }
}
```

Array Accesses:

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0,0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0,1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0,2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0,3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>B-1</td>
<td>0</td>
<td>0,B-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1,0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1,1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1,2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B-1</td>
<td>1,B-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.7  Algorithm and array access for blocked matrix multiplication

alleviated by the additional register storage provided. This makes matrix multiplication a worthy target for the use of the reconfigurable functional units as circular register queues.

### 4.2.1.2 Loop unrolling

A very general code structure that obviously exhibits the properties mentioned in criteria 1 at the beginning of Section 4.2.1 is a loop. While criteria 2 and 3 don't necessarily follow for loops in general, there is another well-known optimization, loop unrolling, which is frequently applied to loops due to various gains in performance that it brings, and which does have the possibility of meeting these criteria.

In loop unrolling, the body of a loop is copied $n$ times and the copies are interleaved so that each iteration of the loop executes $n$ passes of the original loop. Figure 4.8 shows an example of a loop in C code that is transformed to MIPS assembly. In Figure 4.9, this loop is then unrolled, using an unrolling factor of 4.
Simple loop in C code:

```c
x = ArrayLen - 1;
while (x => 0) {
    Array[x] = Array[x] + n;
    x--;
}
```

Loop represented in MIPS assembly:

```
la    R2, ArrayAddr
addi  R1, R2, ArrayLen
Loop: lw R4, 0(R1)
      add  R4, R3, R4
      sw   R4, 0(R1)
      lw R4, -4(R1)
      add  R4, R3, R4
      sw R4, -4(R1)
      lw R4, -8(R1)
      add  R4, R3, R4
      sw R4, -8(R1)
      lw R4, -12(R1)
      add  R4, R3, R4
      sw R4, -12(R1)
      subi R1, R1, 16
      bge R1, R2, Loop
```

Figure 4.8 Loop unrolling example, part 1

MIPS assembly for unrolled loop:

```
la    R2, ArrayAddr
addi  R1, R2, ArrayLen
Loop: lw R4, 0(R1)
      add  R4, R3, R4
      sw R4, 0(R1)
      lw R4, -4(R1)
      add  R4, R3, R4
      sw R4, -4(R1)
      lw R4, -8(R1)
      add  R4, R3, R4
      sw R4, -8(R1)
      lw R4, -12(R1)
      add  R4, R3, R4
      sw R4, -12(R1)
      subi R1, R1, 16
      bge R1, R2, Loop
```

Unrolled and scheduled loop:

```
la    R2, ArrayAddr
addi  R1, R2, ArrayLen
Loop: lw R4, 0(R1)
      lw R5, -4(R1)
      lw R6, -8(R1)
      lw R7, -12(R1)
      add  R4, R3, R4
      add  R5, R3, R5
      add  R6, R3, R6
      add  R7, R3, R7
      sw R4, 0(R1)
      sw R5, -4(R1)
      sw R6, -8(R1)
      sw R7, -12(R1)
      subi R1, R1, 16
      bge R1, R2, Loop
```

Figure 4.9 Loop unrolling example, part 2

Note that for convenience, only the main unrolled loop body is shown, and any follow up code to account for remaining loop iterations where ArrayLen is not evenly divisible by 4 has been excluded.
The un-optimized assembly code for the unrolled loop in the left hand side of Figure 4.9 accomplishes the goal of reducing loop maintenance overhead caused by a decrease in the number of executions of the loop control logic to only 1/nth the original number (where \( n \) is the unrolling factor), and a decrease in the number of branch stalls incurred. However, more instruction-level parallelism (ILP) can be exposed within this loop, further increasing speedup by avoiding I/O stalls, with optimized scheduling. The right hand side of Figure 4.9 shows how the unrolled loop could be more effectively scheduled.

The optimization in the scheduling of the instructions is based on the fact that in most architectures, in this particular case the DLX architecture, there will always be a latency for integer loads that would cause stalls to occur if the resultant value was to be immediately used. Even enhancements such as data forwarding do not completely erase this latency in the best case scenario of a cache hit, and the latency can turn out to be much larger in the case of cache misses. Due to this penalty, it is optimal to schedule as many instructions between the loading of a value and its subsequent use. This allows the effects of the latency for each load to be erased, as the subsequent accesses to the values will be sufficiently delayed. However, this separation between definition and use increases both the lifetime of the variable and the number of simultaneously live instances of the variable that must exist from different iterations of the loop body. It will now be necessary to have a unique register to hold each instance, and it is for this reason that unrolled loops that are optimally scheduled to avoid I/O stalls can significantly increase the register pressure during the loop body execution.

### 4.2.1.2.1 Related work

The process of overlapping the execution of successive loop iterations in an attempt to optimize the utilization of the processor's scheduled resources and avoid unnecessary stalls is known as software pipelining. Optimal software pipelining will result in the number interleaved loop instances being equal to the number of cycles in the largest latency of operations within the loop body. As shown in [4], software pipelining can decrease the register pressure when compared to straight loop unrolling. However, with the potentially sizeable latency in the memory accesses of actual processors, such a large number of interleaved loop iterations and the
corresponding increase in live variable instances creates a demand for registers that is greater than non-optimized loops, and that cannot typically be met by standard register sets.

In [5], Smelyanskiy, Tyson, and Davidson [ST&D] proposed a hardware mechanism they called Register Queues (RQs). Using these RQs, which effectively behave similarly to the circular register queues introduced in Chapter 2, they gathered experimental data to support their claim that RQs can be used to significantly decrease the register pressure and code expansion that are caused by software pipelining.

The RQ implementation as shown in Figure 4.10 and described in [5] has three main parts:

• A set of register queues: each queue has a pointer to the tail called $Q_{tail}$, and is formed by a set of contiguous registers that share a common namespace with the conventional register set but that are logically, and likely physically, separate. Registers in the queue are not directly indexed by architected registers but must be explicitly mapped to an architected register.

• A physical register file: the remaining set of registers that are not allocated to a register queue. This is similar to a typical register file, such as the GRP set in the DLX architecture.

• An architected register map table: a table that maps each architected register to either a physical register in the register file, or a register queue. Each map entry contains a physical register index and a read offset. The physical register value identifies either a physical register from the register file or a register queue from the set of queues. The offset, which is only used when the index points to a register queue, specifies the offset into the queue. This offset determines which queue register is actually mapped to the architected register.

Only a single instruction is required to be added to the ISA: $rq$-$connect$. This instruction is used to map, unmap, or remap an architected register to one of the register queues. $rq$-$connect$ is used as follows:

• $rq$-$connect$ $srq,$ $sar,$ imm: architected register $sar$ is mapped to register queue $srq$ by writing the queue number into the pri field for the architected
Figure 4.10  RQ architecture with 32 architected registers, queue depth 4, and 256 physical registers

register in the map table. The ro field in the map table is set to imm, indicating that reads to $ar$ will map to $Q_{tail} + \text{imm}$.

- $rq\text{-}connect\ 0, \ $ar, \ 0$: architected register $ar$ is mapped to a free register in the physical register file.

Based on the RQ architecture described above, ST&D performed a series of analyses to determine the effects on register pressure that optimal software pipelining could impose. They gathered data related to the number of live instances of variables that would be needed given particular memory latencies as well as the necessary growth in queue sizes as latency increases. Assuming a memory latency of 13 and analyzing 983 loops extracted from the Perfect Club Suite, SPEC, and the Livermore Kernels, ST&D found that over half of the variables would only require 2 instances. The largest number of instances they found was 13. Additionally, by
averaging the required number of instances over all variables as the load latency increased, they found a slow linear growth, beginning at 2.5 for low latency, climbing to 6.5 for a latency of 45. While these numbers show a required set of architected registers that would overwhelm the capacities of most register sets, they could easily be accommodated for using RQs.

Their results indicate that with the ability of RQs to keep the demand for architected registers steady while increasing the number of live instances of loop variables, it is possible to schedule load operations optimally to avoid stalls even assuming large miss penalties. The use of RQs therefore enables more aggressive implementations of software pipelining and the greater speedup that entails.

4.2.1.2.2 Using circular register queues in unrolled, software pipelined loops

Although the RQ architecture in [5] is different in several aspects from the use of circular register queues, the basic model of using register queues to contain the multiple simultaneously live instances of loop variables is similar. In figure 4.11, there is an example of how circular register queues could be effectively employed to reduce register pressure caused by loop unrolling and software pipelining. In this example, it is assumed that writes and reads to the queue cause the queue to rotate one position. This would cause the queue to rotate two positions for the instruction "add Q1, R3, Q1", which would put the data in the wrong position for the follow up access. For this reason, an extra register, R4, is used and the add and sw operations are interleaved so that only a single additional register is required to hold the sums of each queue element and R3.

Based on the code from Figure 4.11, assuming that the data values loaded from 0(R1), -4(R1), -8(R1), and -12(R1) are a, b, c, and d, respectively, the contents of Q1 following the execution of selected instructions during the first loop iteration can be seen in figure 4.12.

Our work in addition to [5] has shown that unrolled and software pipelined loops have the potential to be good candidates for targeted circular register queue usage. There are sets of data requiring multiple accesses in fixed ordering, and there is significant register pressure that must be alleviated in order to gain the full benefits. The remaining criteria from Section 4.2.1 that must be met is that the data
Unrolled and scheduled loop using circular register queue Q1 (assuming queue depth of 5):

```
la R2, ArrayAddr
addi R1, R2, ArrayLen
Loop: lw, Q1, 0(R1)
      lw Q1, -4(R1)
      lw Q1, -8(R1)
      lw Q1, -12(R1)
      advance Q1
      add R4, R3, Q1
      sw R4, 0(R1)
      add R4, R3, Q1
      sw R4, -4(R1)
      add R4, R3, Q1
      sw R4, -8(R1)
      add R4, R3, Q1
      sw R4, -12(R1)
      subi R1, R1, 16
      bge R1, R2, Loop
```

Figure 4.11 Loop unrolling example, part 3

**Pipeline Contents by Stage**

<table>
<thead>
<tr>
<th>After Inst</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>d</td>
<td>c</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>

Figure 4.12 Circular register contents during first loop iteration for Figure 4.11
set must be of a reasonable size and fit into the set of available register queues.

4.2.1.2.3 Loop analysis

In order to determine the typical size of the set of live variable instances that occur in an unrolled loop, and therefore whether or not the set would fit well into a set of circular register queues, it was necessary to perform an analysis of simultaneously live, iteration dependent variables within typical loops.

The example in Figure 4.11 of an unrolled loop that uses a circular register queue shows how a single queue of depth $n$ could be used to contain $m$ live instances of a variable, where $m \leq n$, and $m$ is the unrolling factor. This will be sufficient to remove any additional register pressure caused by loop unrolling where:

- $x$ = the number of variables with $m$ simultaneously live instances
- $N$ = the number of available register queues with $m$ or more stages
- $x \leq N$

for the basic case where there is a direct one-to-one mapping of variable to queue. However, $x$ could dominate $N$ in situations where the queues are not deep enough, and if for $S =$ sum of the depths of all available queues, $x \cdot m \leq S$, the compiler could perform scheduling so that multiple queues contain the live instances of a single variable. In the most complex scheduling scenario, multiple variables could be assigned to a single queue.

In order to determine if the product $x$ is likely to be within the upper bound of $N$ for the majority of unrolled loops, it was necessary to determine the average value for $x$. This required an analysis of live operand values within loops. Only those operands that are read from unique memory locations at each loop iteration needed to be counted since only those operands would need to be loaded $m$ times within the unrolled loop body. All other loaded operands could be ignored when calculating $x$.

4.2.1.2.4 Analysis implementation

To perform this analysis, we again used the SUIF and MachineSUIF libraries to decompose source code into CFG's, then ran it through a custom compiler pass, which performed the following analysis:
1. A loop detection pass was performed on the CFG's to create the set of loops, which was represented as a set of basic blocks with predecessor and successor information.
2. Each node of each loop was traversed to detect load operations.
3. For each load, the operands that made up the memory address for the load were recorded.
4. Each reverse control flow path returning to the beginning of the loop was traversed in reverse order, from the load operation, to determine which operands were used to calculate each operand used in the address of the load operation. These were used to create a list of operands that determined which memory location would be referenced by the load operation.
5. Using this set of operands, a forward pass of the loop was performed to determine if any of the operands in the list changed during any control flow path of the loop during an iteration.
6. The total number of updated variables for each loop was recorded.

4.2.1.2.5 Results

The above analysis was performed on the source code of the SPEC95 Integer benchmark suite, covering 3332 loops. The mean number of variables per loop that require unique instances per iteration was 3.2 for all loops analyzed, and 4.1 for loops that contain one or more such variables. (A graph of these results is provided in Figure 4.13.) This closely conforms to similar data provided in [2] where a similar set of experimental data was presented. These two independent analyses confirm that greater than 90% of loops contain 10 or less such variables, meaning that criteria 2 from Section 4.2.1 can in fact generally be met for unrolled and software pipelined loops.

We have shown that unrolled and software pipelined loops exhibit the necessary criteria to be good candidates for targeting by the compiler for circular register queue use. Given the sufficiently small set of variables that need multiple simultaneous instances, and therefore need to be stored within circular register queues, a large load latency can be accommodated. The potential speedup to be gained by maximizing ILP through loop unrolling and minimizing I/O stalls by
aggressively implementing software pipelining, if the burden of additional register pressure is eliminated or minimized using circular register queues, can be sizeable.

### 4.2.2 Targeting methods

There are two different methods that could be employed to ensure that the necessary code segments that we have selected as good fits are targeted by the compiler to make use of the reconfigured circular register queues. The first method would be a high level approach that puts the burden of recognizing such a code segment on the programmer by requiring the programmer to add new compiler directives to high level code to notify the compiler that a given code section would be a good candidate for queue allocation. The second method would be to build into the compiler the ability to detect these predefined code patterns. Both approaches still require that the compiler have the ability to perform optimal register allocation using circular queues, but they reduce the significant overhead of constant operand access pattern analysis throughout all code sections.
4.2.2.1 Programmer added directive

By simply supporting a directive such as "#QUEUE_VAR x" which designates x as a variable that should be mapped to a queue, the compiler could easily perform register allocation using the reconfigurable functional units as circular register queues.

4.2.2.2 Compiler pattern detection

To make use of circular register queues for loop unrolling, it is obviously a simple task for the compiler to identify loops. Then, an analysis of the loop would be performed to determine which reconfigurable functional units are available during execution of the loop body to be configured as register queues. Next, the set of loop variables that would require unique live instances for each loop iteration would be determined. Using a predetermined, architecture dependent value for the potential latency that could be encountered during load operations and the size of the variable set from the last step, the unrolling factor would be determined and the loop would be unrolled and scheduled.
CHAPTER 5 CONCLUSIONS AND FUTURE WORK

In this chapter we present our overall conclusions and future work to be done on the topic.

5.1 Conclusion

In Chapter 2, we described a reconfigurable architecture that provides the potential to turn idle pipelined functional units into circular register queues that can be used to supplement the on-chip register storage capacity. We then performed a detailed analysis of operand access patterns throughout general code sections. The results of this analysis suggested that typical operand access patterns very seldom lend themselves to the repetitive patterns and latencies necessary to map well to circular register queues. Our next step was to determine a set of criteria that, if met, would display good potential for such a mapping so we could pre-select code segments that would be good fits. These criteria were: 1) the code segments involve repetitive access to data elements, 2) these data sets are typically of sufficient size to fit into a set of available reconfigured register queues, and 3) high amounts of register pressure are usually experienced. We were able to present and display evidence of two scenarios that meet these criteria: matrix multiplication and loop unrolling. Finally, we presented a method with which the compiler could target predetermined code constructs for circular register queue use.

In summary, it was shown that the slight changes required to support the reconfigurable architecture presented by Tyagi in [9] can be useful in supplementing the speedup acquired through the use of other compiler optimizations, especially loop unrolling and software pipelining, by significantly alleviating register pressure and allowing aggressive use of these techniques to fully expose the ILP within loops.
5.2 Future work

Future work on this topic could include the following:

- Modification of the Simple Scalar simulator to simulate the proposed architecture
- Completion of the Simple Scalar backend we began in order to target SUIF/MachineSUIF to the Simple Scalar simulator, as well as additional work to target the modified Simple Scalar which supports reconfigurable functional units
- Modifications of the MachineSUIF register allocation to utilize register queues for the scenarios mentioned in Chapter 4
- Generation of performance data for the modified Simple Scalar environment while executing code that makes use of the circular register queues.
APPENDIX A: ANALYZED SOURCE CODE

The following were the programs whose operand access patterns were analyzed to produce the results in Section 4.1.

From the SPEC95 Integer benchmark suite:

- 099.go/src/g2.c
- 099.go/src/g22.c
- 124.m88ksim/src/dc.c
- 124.m88ksim/src/pc.c
- 129.compress/src/compress.c
- 129.compress/src/harness.c
- 130.li/src/xleval.c
- 130.li/src/xlisp.c
- 130.li/src/xlmath.c
- 132.ijpeg/src/libppm2.c
- 132.ijpeg/src/jcmaster.c
- 132.ijpeg/src/jmemname.c
- 147.vortex/src/draw07.c
- 147.vortex/src/grp0.c
- 147.vortex/src/mem10.c
- 147.vortex/src/rect.c
- 147.vortex/src/vchunk.c

Others:

- matrix-mult.c (short program that performs matrix multiplication using the standard algorithm)
- blocking.c (short program that performs matrix multiplication using an optimized blocking algorithm)
APPENDIX B: OPERAND ACCESS PATTERN ANALYSIS RESULTS

This appendix contains additional graphs of the results from the operand access analysis performed in Section 4.1.

Figure B.1  First occurrence results graph, separated by access type
Figure B.2  Full depth results graph, separated by access type
BIBLIOGRAPHY


