# ABSTRACT

# Virtualizing Register Context

**by**

**David W. Oehmke**

Chair: Trevor N. Mudge

A processor designer may wish for an implementation to support multiple register contexts for several reasons: to support multithreading, to reduce context switch overhead, or to reduce procedure call/return overhead by using register windows. Conventional designs require that each active context be present in its entirety, increasing the size of the register file. Unfortunately, larger register files are inherently slower to access and may lead to a slower cycle time or additional cycles of register access latency, either of which reduces overall performance. We seek to bypass the trade-off between multiple context support and register file size by mapping registers to memory, thereby decoupling the logical register requirements of active contexts from the contents of the physical register file.

Just as caches and virtual memory allow a processor to give the illusion of numerous multi-gigabyte address spaces with an average access time approach-

ing that of several kilobytes of SRAM, we propose an architecture that gives the illusion of numerous active contexts with an average access time approaching that of a single active context using a conventionally sized register file. This dissertation introduces the virtual context architecture, a new architecture that virtualizes logical register contexts. Complete contexts, whether activation records or threads, are kept in memory and are no longer required to reside in their entirety in the physical register file. Instead, the physical register file is treated as a cache of the much larger memory-mapped logical register space. The implementation modifies the rename stage of the pipeline to trigger the movement of register values between the physical register file and the data cache. With the same size register file as a non register windowed machine, this architecture is within 1% of the execution time of an idealized register window machine. The virtual context architecture enables support for both register windows and simultaneous multithreading without increasing the size of the register file, increasing the performance by 50% over a single thread and 30% over a conventional multithreaded architecture.

# Virtualizing Register Context

by

**David W. Oehmke**

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2005

Doctoral Committee:

Professor Trevor N. Mudge, Chair
Associate Professor Todd M. Austin
Associate Professor Steven K. Reinhardt
Assistant Professor Dennis M. Sylvester

© David W. Oehmke

All rights reserved

2005

To Cathy

# Acknowledgements

Many people were helpful in my pursuit of a Ph.D. by providing support, encouragement and/or helping me navigate through graduate school. I would like to thank all of them and just say that without their help, I would never have finished.

I am grateful to my advisor, Trevor Mudge, for supporting me both financially and emotionally. He reassured me that I really did belong in graduate school. He was always patient, but he didn't let me get too far off course. He was an incredible source of ideas, and I always enjoyed the times we spent discussing research. I am also especially grateful to him for the opportunity I had to travel with him to India; it was one of the most interesting experiences I've had.

I would also like to thank my committee: Todd Austin, Dennis Sylvester, and especially Steve Reinhardt, who, along with Trevor and Nate, spent many an afternoon discussing my work and the direction it should take. Steve also provided timely support when I had problems with m5.

Several graduate student friends were also invaluable. They provided information, support, and an audience for complaints. Jeff Ringenberg was a good friend through the ups and downs of grad school. We worked on several projects together and I'll always remember our culinary tour of San Diego. David Greene helped me with MIRV and answered all my newbie questions. He also helped me move into the next phase of life by helping me get a job. Nate Binkert's original idea eventually became the virtual context architecture and he was instrumental in shaping my research.

Some people from my previous job are also deserving of thanks. Bob Stack made the effort to get to know me, and I appreciate his friendship. He would help me talk through things when I was depressed and provide some perspective. I

would also like to thank the management of K.J. Law Engineers (now Verimation Technology), especially Ken Law, Jr., for giving me the opportunity to go to graduate school, and for being flexible about scheduling.

I would like to thank my parents, who have been supportive of everything I've ever done or tried to do, despite the fact that my mom believed that computers would just be a fad. They encouraged me to go to graduate school, even though that meant enduring life with both of their children in graduate school at the same time. They didn't even say I told you so after I had insisted that I would stop with my masters over and over, yet eventually went on for my doctorate.

Thanks also to my brother, Bob, who has been one of my best friends ever since we were kids. Who knew that writing code for the Commodore 64 would lead to two doctorates in computer science? He helped get me started in graduate school and was always there when I needed him. Thanks to my sister-in-law, Shana, for being a good friend and for letting me hang around their house when I needed someone to talk to. I appreciate you listening while Bob and I discussed computers for hours. Double extra super thanks for introducing me to my wife, Cathy. Bob & Shana, thank you for letting me share in the joy of your daughter, my new niece, and in the process helping me to understand what is really important.

Finally, thank you with all my heart to Cathy Oehmke. She was there for me tirelessly editing and keeping me focused when that was what I needed. When I needed to get away from things she was there to help distract me. She blazed the way by getting her Ph.D. first and showing me that it was possible. She has become my best friend and I couldn't imagine my life without her. With the two simple words "I do" she made me the luckiest person to have ever lived. I love you.

## Table Of Contents

**Dedication** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ii

**Acknowledgements** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

**List of Figures** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . x

**List of Tables** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii

**List of Appendices** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiv

**Chapter 1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Mapping compiler virtual registers to memory . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Mapping ISA logical registers to memory . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.4 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**Chapter 2 Mem-Machine** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.1 Design of the ISA and ABI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1.1 Design Decisions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1.2 ISA and ABI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.1.2.1 ISA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.1.2.2 ABI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2.1 Compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2.2 C Library . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.3 Binary Utilities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.4 Simulator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.3.1 Code Profile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.3.2 Comparison vs. PISA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

**Chapter 3 VCA Design** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.1 Theory. . . . . . 31
3.1.1 Processor Model . . . . . 32
3.1.2 Memory Mapping. . . . . 32
3.1.3 Physical Register States . . . . . 35
3.1.4 Register Rename Stage . . . . . 36
3.2 Implementation . . . . . 41
3.2.1 Rename Stage . . . . . 41
3.2.1.1 Rename Table . . . . . 41
3.2.1.2 Tracking Physical Register States . . . . . 42
3.2.2 Implementing Register Spills and Fills. . . . . 43
3.2.3 Branch Recovery. . . . . 45
3.2.4 Operating System . . . . . 46
3.3 SMT and Multiprocessor Implications. . . . . 48
3.4 Rename Table Optimizations . . . . . 49
3.4.1 Rename Table Organization . . . . . 49
3.4.2 Rename Table Tags . . . . . 50
3.5 Summary of Pipeline Modifications. . . . . 54
**Chapter 4 Experimental Methodology**. . . . . 58
4.1 Benchmarks . . . . . 58
4.2 Workloads. . . . . 60
4.2.1 Register Windows . . . . . 62
4.2.2 Simultaneous Multithreading . . . . . 62
4.3 Simulation . . . . . 63
4.4 Statistics . . . . . 65
4.4.1 Register Windows . . . . . 65
4.4.2 Simultaneous Multithreading . . . . . 67
**Chapter 5 Parameter Studies** . . . . . 69
5.1 Spill/Fill Implementation . . . . . 69
5.1.1 Register Replacement Policy . . . . . 69
5.1.2 Method Used To Implement Spills and Fills . . . . . 74
5.1.2.1 Architectural State Transfer Queue Parameters . . . . . 78

5.1.2.2 Operation Parameters . . . 84
5.2 Rename Stage . . . 86
5.2.1 Associativity . . . 86
5.2.2 Ports . . . 93
5.2.3 Cost of Extra Logic . . . 95
5.3 Summary . . . 97
5.3.1 Spill/Fill Implementation . . . 97
5.3.2 Rename Stage . . . 98
**Chapter 6 Register Window Studies** . . . 100
6.1 Four Issue Pipeline . . . 101
6.1.1 Benchmark Details . . . 102
6.1.2 Two Data Cache Ports . . . 104
6.1.2.1 Execution Time Results . . . 105
6.1.2.2 Cache Results . . . 109
6.1.3 One Data Cache Port . . . 116
6.1.3.1 Execution Time Results . . . 116
6.1.3.2 Data Cache Results . . . 122
6.1.4 Summary . . . 123
6.2 Eight Issue Pipeline . . . 124
6.2.1 Three Data Cache Ports . . . 125
6.2.2 Two Data Cache Ports . . . 129
6.2.3 One Data Cache Port . . . 131
6.2.4 Summary . . . 137
6.3 Two Issue Pipeline . . . 138
6.3.1 Execution Time Results . . . 139
6.3.2 Data Cache Results . . . 142
6.4 Single Issue In-order Pipeline . . . 144
6.5 Summary . . . 147
**Chapter 7 Simultaneous Multithreading Studies** . . . 149
7.1 Two Thread SMT . . . 150
7.1.1 Execution Time . . . 151

7.1.2 Data Cache Accesses . . . 156
7.2 Four Thread SMT . . . 160
7.2.1 Execution Time . . . 161
7.2.2 Data Cache Accesses . . . 166
7.3 Threads Study . . . 169
7.3.1 Execution Time Study . . . 170
7.3.2 Data Cache Accesses Study . . . 173
7.3.3 Speedup Study . . . 174
7.4 Summary . . . 176
**Chapter 8 Spill Optimization** . . . 178
8.1 Delay Queue . . . 178
8.2 Register Window Deallocation . . . 181
8.3 Dead Value Information . . . 183
8.4 Evaluation . . . 185
8.4.1 Data Cache Accesses . . . 185
8.4.2 Execution Time . . . 189
8.5 Summary . . . 194
**Chapter 9 Related work** . . . 196
9.1 Memory As A Backing Store To The Register File . . . 196
9.2 Reducing Save/Restore . . . 199
9.3 Aggressive Physical Register Management . . . 202
9.4 Efficient Large Physical Register Files . . . 206
9.5 Simultaneous Multithreading . . . 207
9.6 Summary . . . 209
**Chapter 10 Conclusion** . . . 211
10.1 Summary . . . 211
10.2 Future Work . . . 214
10.2.1 Improving The VCA Implementation . . . 214
10.2.2 Exploring New Uses . . . 215
10.2.2.1 Decoupling The Physical Registers . . . 215
10.2.2.2 Fast Context Switching . . . 216

10.2.2.3 Directly Mapping Addresses To Physical Registers. . . . . . . . 216
**Appendices** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 218
**References** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 220

## List of Figures

**Figure 1.1:** Register Specifier Transformations . . . . . 3
**Figure 2.1:** Instruction Binary Encoding Format . . . . . 11
**Figure 2.2:** Mem-Machine Stack Layout . . . . . 13
**Figure 2.3:** Dynamic register usage. . . . . . 19
**Figure 2.4:** Dynamic indirection level. . . . . . 20
**Figure 2.5:** Dynamic instruction count comparison versus PISA . . . . . 21
**Figure 2.6:** Average data accesses per instruction . . . . . 23
**Figure 2.7:** Performance impact of extra data cache misses. . . . . . 24
**Figure 2.8:** Performance impact of extra instruction cache misses . . . . . 25
**Figure 2.9:** Code size comparison versus PISA . . . . . 26
**Figure 3.1:** Finding A Physical Register To Use . . . . . 37
**Figure 3.2:** Finding A Rename Table Entry . . . . . 38
**Figure 3.3:** Source Register Logic . . . . . 39
**Figure 3.4:** Destination Register Logic. . . . . . 40
**Figure 3.5:** Address Translation Scheme . . . . . 52
**Figure 3.6:** Optimized rename process.. . . . . . 53
**Figure 3.7:** Rename Stage Summary . . . . . 55
**Figure 3.8:** Issue Stage Summary . . . . . 56
**Figure 3.9:** Commit Stage Summary . . . . . 57
**Figure 5.1:** Operation Execution Time . . . . . 75
**Figure 5.2:** Operation Data Cache Accesses . . . . . 77
**Figure 5.3:** ASTQ Spill / Operation Fill Execution Time . . . . . 78
**Figure 5.4:** Architectural State Transfer Queue Ports . . . . . 79
**Figure 5.5:** Architectural State Transfer Queue Size . . . . . 81
**Figure 5.6:** ASTQ Size: Cache Accesses . . . . . 83
**Figure 5.7:** Operation With Optimized Load Store Queue . . . . . 85

**Figure 5.8:** Average Rename Table Entries . . . . . . . . . . . . . . . . . . . . . . . . 87
**Figure 5.9:** Single Thread Rename Table Associativity. . . . . . . . . . . . . . . . . . 88
**Figure 5.10:** Two Thread Rename Table Associativity . . . . . . . . . . . . . . . . . . 89
**Figure 5.11:** Four Thread Rename Table Associativity . . . . . . . . . . . . . . . . . . 90
**Figure 5.12:** Four Thread Optimized Rename Table. . . . . . . . . . . . . . . . . . . . 92
**Figure 5.13:** Rename Table Ports . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
**Figure 5.14:** Cost of Extra Logic In The Rename Stage . . . . . . . . . . . . . . . . . 96
**Figure 6.1:** Register Window Binaries Execution Time . . . . . . . . . . . . . . . . 102
**Figure 6.2:** Register Window Binaries Data Cache Accesses. . . . . . . . . . . . 104
**Figure 6.3:** Four Issue Execution Time Comparison . . . . . . . . . . . . . . . . . . 105
**Figure 6.4:** VCA Execution Time Relative To Ideal . . . . . . . . . . . . . . . . . . . 106
**Figure 6.5:** Four Issue Data Cache Accesses Comparison. . . . . . . . . . . . . . 110
**Figure 6.6:** Spill And Fill Rate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
**Figure 6.7:** Four Issue One Data Cache Port Execution Time . . . . . . . . . . . 117
**Figure 6.8:** Execution Time Relative To Ideal . . . . . . . . . . . . . . . . . . . . . . . 119
**Figure 6.9:** One Port VCA Versus Two Port Baseline . . . . . . . . . . . . . . . . . 121
**Figure 6.10:** Four Issue One Data Cache Port Data Cache Accesses . . . . . . 122
**Figure 6.11:** Eight Issue Three Data Cache Ports Execution Time . . . . . . . . . 125
**Figure 6.12:** Eight Issue Three Data Cache Ports Data Cache Accesses. . . . 128
**Figure 6.13:** Eight Issue One Data Cache Port Execution Time . . . . . . . . . . . 132
**Figure 6.14:** Eight Issue VCA Execution Time Improvement. . . . . . . . . . . . . 134
**Figure 6.15:** Eight Issue VCA Execution Time Versus Ideal . . . . . . . . . . . . . 135
**Figure 6.16:** Eight Issue VCA Data Cache Accesses . . . . . . . . . . . . . . . . . . 136
**Figure 6.17:** Two Issue Execution Time Comparison . . . . . . . . . . . . . . . . . . 139
**Figure 6.18:** Two Issue Data Cache Accesses Comparison . . . . . . . . . . . . . . 143
**Figure 6.19:** Single Issue In-order Execution Time Comparison . . . . . . . . . . . 145
**Figure 7.1:** Two Thread Execution Time Comparison . . . . . . . . . . . . . . . . . 152
**Figure 7.2:** Two Thread High Call Rate Execution Time Comparison. . . . . . 154
**Figure 7.3:** Two Threaded SMT Speedups . . . . . . . . . . . . . . . . . . . . . . . . 155
**Figure 7.4:** Two Thread Data Cache Accesses Comparison. . . . . . . . . . . . . 157
**Figure 7.5:** Four Thread Execution Time Comparison. . . . . . . . . . . . . . . . . . 161

**Figure 7.6:** Four Threaded SMT Speedups . . . . . . . . . . . . . . . . . . . . . . . . . 165
**Figure 7.7:** Four Thread Data Cache Accesses Comparison . . . . . . . . . . . . 167
**Figure 7.8:** Threads Execution Time Comparison . . . . . . . . . . . . . . . . . . . . . 170
**Figure 7.9:** Threads Data Cache Access Comparison . . . . . . . . . . . . . . . . . 173
**Figure 7.10:** Thread Speedup Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . 175
**Figure 8.1:** Delay Queue Time Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 180
**Figure 8.2:** Delay Queue Size Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
**Figure 8.3:** Spill Optimization With Two Data Cache Ports . . . . . . . . . . . . . . 186
**Figure 8.4:** Spill Optimization With One Data Cache Port . . . . . . . . . . . . . . . 189
**Figure 8.5:** Optimization Execution Time With Two Data Cache Ports . . . . . 190
**Figure 8.6:** Optimization Execution Time With One Data Cache Port . . . . . . 192

## List of Tables

**Table 2.1:** Address Indirection Level Description . . . . . 12
**Table 2.2:** Dynamic instruction value types . . . . . 18
**Table 2.3:** Code size statistic . . . . . 26
**Table 2.4:** Compression scheme.. . . . . 27
**Table 4.1:** Register window ABI . . . . . 59
**Table 4.2:** Four Issue Processor Description . . . . . 64
**Table 4.3:** Path Length Ratio . . . . . 66
**Table 5.1:** Replacement Policy Execution Time Comparison. . . . . 71
**Table 5.2:** Replacement Policy Memory Comparison . . . . . 72
**Table 6.1:** VCA Configurations For The Four Issue Pipeline . . . . . 101
**Table 6.2:** VCA Configurations For The Eight Issue Pipeline. . . . . 124
**Table 6.3:** VCA Configurations For The Two Issue Pipeline . . . . . 138
**Table 6.4:** VCA Configurations For The Single Issue In-order Pipeline . . . . 145
**Table 7.1:** VCA Configurations For Two Thread SMT . . . . . 150
**Table 7.2:** VCA Configurations For Four Thread SMT . . . . . 160

## List of Appendices

**Appendix A:** Mem-Machine Supported Operations . . . . . . . . . . . . . . . . . . . . . . 218

**Appendix B:** Mem-Machine Assembly Language . . . . . . . . . . . . . . . . . . . . . . . 219

# Chapter 1

# Introduction

## 1.1 Motivation

Registers are a central component of both instruction-set architectures (ISAs) and processor microarchitectures. From the ISA's perspective, a small register namespace allows the encoding of multiple operands in an instruction of reasonable size. Registers also provide a simple, unambiguous specification of data dependences, because—unlike memory locations—they are specified directly in the instruction and cannot be aliased. From the microarchitecture's point of view, registers comprise a set of high-speed, high-bandwidth storage locations that are integrated into the datapath more tightly than a data cache, and are thus far more capable of keeping up with a modern superscalar execution core.

As with many architectural features, the abstract concept of registers can conflict with real-world implementation requirements. For example, the dependence specification encoded in the ISA's register assignment is adequate given the ISA's sequential execution semantics. However, out-of-order instruction execution requires that the ISA's logical registers be renamed into an alternate, larger physical register space to eliminate false dependencies.

We address a different conflict between the abstraction of registers and its implementation: that of *context*. A logical register identifier is meaningful only in the context of a particular procedure instance (activation record) in a particular thread of execution. From the ISA's perspective, the processor supports exactly one context at any point in time. However, a processor designer may wish for an implementation to support multiple contexts for several reasons: to support multithreading, to reduce context switch overhead, or to reduce procedure call/return

overhead (e.g., using register windows) [8, 19, 24, 30, 35, 37]. Conventional designs require that each active context be present in its entirety; thus each additional context adds directly to the size of the register file. Unfortunately, larger register files are inherently slower to access. Thus, additional contexts generally lead to a slower cycle time or additional cycles of register access latency, either of which reduces overall performance. This problem is further compounded by the additional rename registers necessary to support out-of-order execution.

The context problem has already been solved for memory. Modern architectures are designed to support a virtual memory system. From the process's perspective, each process has its own independent memory, only limited by the size specified by the ISA. The architecture and operating system efficiently and dynamically manage the hardware memories (cache, RAM and disk) to maximize the memory performance of each process. Virtual memory never requires the process memory to exist in its entirety at any level. Instead, it is only required that those portions actively used exist at the last level (disk). All the other memory levels simply cache a fraction of the memory to provide good performance. This is in sharp contrast to how a traditional architecture manages the registers. Instead, all the logical registers for the active contexts are kept in the physical register file. Even in previous research that examined register caches, the full set of logical registers is kept in its entirety in the relatively limited register cache hierarchy.

A system with the advantages of a register, but with the context free nature of memory is desired. Such a system can be realized by mapping the registers to memory. The question becomes: when does this mapping occur? The registers go through several transformations between source code and execution in a processor. These steps are summarized in Figure 1.1. At initial compilation, the source code variables and temporary values are converted into virtual registers. At register allocation, the compiler maps the virtual registers into logical registers and the assembly code/binary is generated. Finally, in an out-of-order processor, at rename, the logical registers are mapped into physical registers to remove any false data dependencies imposed by the logical registers.

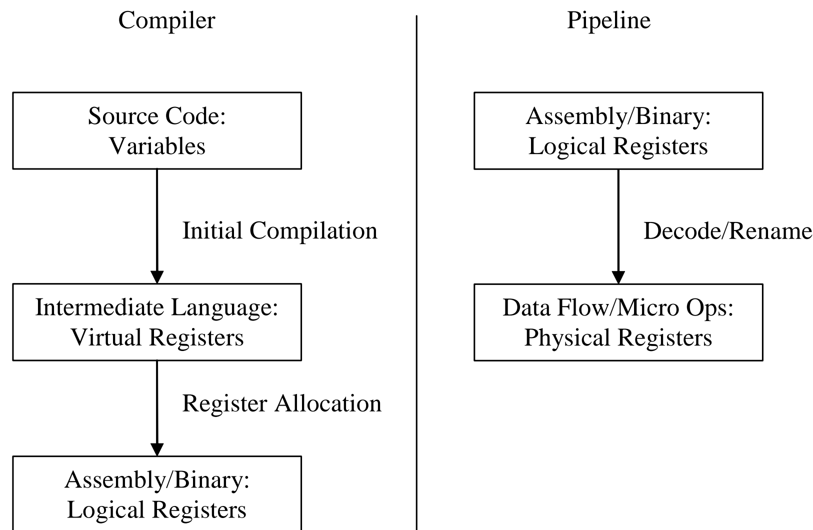



**Figure 1.1: Register Specifier Transformations**
The registers transformations from source code until the time the assembly instructions are executed.

The two obvious choices for memory mapping the registers are at register allocation or decode/rename. The first possibility is to have the compiler map virtual registers directly into memory at register allocation. In this case, the ISA will no longer have a notion of registers, and will instead directly work with memory addresses. The second possibility is to have the microarchitecture manage this mapping. In this case, the compiler generates a more traditional assembly with logical register specifiers. In the frontend of the pipeline, the microarchitecture maps the logical register into memory, thus alleviating the context problem.

## 1.2 Mapping compiler virtual registers to memory

By mapping the compiler virtual register directly to memory, we completely remove registers from the instruction set architecture. This creates a memory to memory instruction set architecture which we call the mem-machine. Such an ISA has the advantage of unlimited resources (all of virtual memory) to use as storage locations. The code is also free of explicit loads and stores, resulting in efficient execution. However, this type of instruction set architecture also suffers from

some serious disadvantages. As previously stated, registers allow for efficient encoding, unambiguous data dependencies and tight coupling into the microarchitecture's data path. One serious disadvantage would be the size of the instructions. As stated previously, a register specifier can be encoded in a small number of bits. Encoding a memory address would require a large number of bits per operand. Data dependencies are also no longer as unambiguous. Instead, potentially every source and destination could be accessed using a dynamic address calculated at run time. The tight coupling of the physical registers file is also lost. For this architecture the data cache that requires tags becomes the first level of storage. Reasonable performance requires a much larger and therefore slower structure. Chapter 2 contains a detailed description of the mem-machine and reports some results.

## 1.3 Mapping ISA logical registers to memory

Delaying the memory mapping to the frontend of the pipeline, gives all the advantages of registers with the automatic context management of virtual memory. We seek to bypass the trade-off between multiple context support and register file size by decoupling the logical register requirements of active contexts from the contents of the physical register file. Just as caches and virtual memory allow a processor to give the illusion of numerous multi-gigabyte address spaces with an average access time approaching that of several kilobytes of SRAM, we propose an architecture that gives the illusion of numerous active contexts with an average access time approaching that of a single active context while still using a conventionally sized register file. Our design treats the physical register file as a cache of a practically unlimited memory-backed logical register space. We call this scheme the virtual context architecture (VCA). An individual instruction needs only its source operands and its destination register to be present in the register file to execute. Inactive register values are automatically saved to memory as needed and restored to the register file on demand. Compared to prior proposals (see Chapter 9), the VCA:

- unifies support for both multiple independent threads and register windowing within each thread;
- completely decouples the physical register file size from the number of logical registers by using memory as a backing store;
- enables the physical register file to hold just the most active subset of logical register values, instead of the complete register contexts, by allowing the hardware to spill and fill registers on demand;
- is backward compatible with existing ISAs at the application level for multithreaded contexts, and requires only minimal ISA changes for register windowing;
- requires no changes to the physical register file design and the performance-critical schedule/execute/writeback loop;
- is orthogonal to the other common techniques for reducing the latency of the register file—register banking and register caching;
- does not involve speculation or prediction, avoiding the need for recovery mechanisms.

The virtual context architecture provides a near optimal implementation of register windows, improving performance and greatly reducing traffic to the data cache (by up to 10% and 20% respectively, in our simulations). The VCA naturally supports both simultaneous multithreading (SMT) and register windowing simultaneously with a conventionally sized register file, avoiding the multiplicative growth in register count that a straightforward implementation would require. Enable register windows allows the virtual context architecture to achieve more than a 10% increase in performance on top of the performance improvement associated with SMT. The VCA also enabled four simultaneous threads on a pipeline without an increase in the number of physical registers, providing over three times the best speedup achieved with the baseline architecture with the same number of physical registers.

Historically, register windows have been implemented on in-order machines exemplified by the Sun SPARC family and the Intel Itanium. In contrast, SMT has been implemented on out-of-order machines such as later versions of Intel's Pentium and the (cancelled) Alpha 21464. The VCA's unifying framework provides support for both techniques in an out-of-order pipeline. It provides a hardware scheme for efficiently managing register context. It is easily integrated into a conventional out-of-order processor. The VCA enables support for both register windows and simultaneous multithreading without increasing the size of the register file while still providing nearly ideal performance.

## 1.4 Organization

Chapter 2 contains a detailed description of the mem-machine and reports some results. Chapter 3 provides details on the theory and design of the virtual context architecture. Chapter 4 describes the methodology used to measure the performance of the virtual context architecture. Chapter 5 studies the effects that the various implementation parameters have on the performance of the virtual context architecture. Chapter 6 evaluates the performance of the virtual context architecture as an implementation of register windows. Chapter 7 evaluates the virtual context architecture in a simultaneous multithreading processor. Chapter 8 describes and evaluates three techniques that can be used to optimize the performance of the virtual context architecture. Chapter 9 briefly describes the research that is related to this dissertation. Chapter 10 concludes the dissertation and discusses future work.

# Chapter 2

# Mem-Machine

The compiler can map virtual registers directly into memory at register allocation. In this case, the ISA will no longer have a notion of registers, and will instead directly work with memory addresses. This creates a memory to memory instruction set architecture which we call the mem-machine. The ISA supports directly accessing memory for all operands, thus eliminating the need for general purpose registers. It has several advantages over a traditional register to register architecture. One performance advantage is the elimination of all explicit load and store instructions. The elimination of the general purpose registers also greatly reduces the size of the context necessary for each thread. This allows efficient context switches and large scale multithreading. The architecture is also very efficient at emulating other types of machines. To emulate a machine, the internal state, including the registers, is kept in memory. To execute an emulated instruction for a register to register machine requires a load of all the values from memory into registers, then the instruction is executed and finally the result copied back to memory. An architecture that can access memory for each operand can directly execute the operation without any extra loading or storing.

The main disadvantages of this architecture are its cache performance and code size. All values are stored in memory, and all operands access memory. This will place a much heavier burden on the caches than a register to register architecture. The code size is also likely to be much larger. To provide the information for an operand to access memory will require more bits than simply specifying one of a small set of logical registers. A memory address is usually 32 or 64 bits, while a register can be specified using only 5 bits for most architectures. This means

that instructions will be several times larger for this architecture than instructions for an equivalent register to register architecture.

This chapter contains three sections. The first section describes the design of the instruction set architecture (ISA) and application binary interface (ABI). The second section discusses the implementation of the build infrastructure and simulation environment. Finally, the third section evaluates the mem-machine.

## 2.1 Design of the ISA and ABI

To support a modern programming language like C, an ISA needs to have support for function calls and the separate compilation and linkage of execution units. This is usually accomplished by creating a stack in memory. In most ISAs, the top of the stack (stack pointer) is kept in a specific register. Any values that need to be communicated across a function call can be placed in memory at specific offsets from the stack pointer. Although many ABIs use additional specific registers to pass some of the information, for example the return address and several parameters, this is only done for efficiency. The notion of a stack is also convenient because it provides a simple and efficient mechanism for each function to allocate local storage in a dynamic fashion. This dynamic allocation is easily accomplished by growing the stack (adjusting the stack pointer). Separate compilation and linkage can be achieved because the stack pointer is a global resource available to all functions. The ABI specifies the location of all the arguments on the stack, and the current location of the top of the stack is communicated to the called function. The following two subsections describe the decisions made when designing the system and the final ISA and ABI chosen.

### 2.1.1 Design Decisions

To enable the mem-machine to support a stack based ABI, the stack pointer must somehow be communicated from the calling function to the callee. Another important issue is that in these stack based systems, all local values are addressed via offsets from the stack pointer. Thus, not only must the stack pointer be in some shared location, but for reasonable performance, there must be an

efficient way to access these local values. This becomes especially important if you eliminate all general purpose registers. In this case not only will local variables be kept on the stack, but so must all temporary values. This leads to an additional requirement: an operand or destination of an instruction must be able to address a stack location. In other words, a stack location becomes the basic location type, or at least the minimal that must be supported.

One solution is to assign the stack pointer a specific address. Since one fixed address is used, easy communication of the value can be accomplished. The callee can simply read the value from this location to determine the current position of the stack pointer. To support the minimal addressing mode for operands requires that each operand be able to access a location relative to the stack. In this case an address and offset would be specified for each operand. The address is read to provide a base pointer. The offset is added to the base pointer to generate a new address. The final value is read from this new address. There are two problems with this approach. One obvious problem is that both an address and offset must be allowed for each operand in an instruction. Since all local values and temporaries will be stored on the stack, large offsets should be supported. When combined with a 32 bit address, this would mean a large number of bits per operand. The second problem is that this would require two memory accesses to read/write a value to/from a location on the stack. Since all locals and temporaries are located on the stack, a typical two source and one destination instruction would require six memory accesses to execute.

The other possibility is to support a stack pointer register. The system will no longer be completely registerless, but it still won't have any general purpose registers, only a single special purpose one. As part of the opcode of the instruction, a bit would specify whether to use the stack pointer. This would allow the operand to be reduced to a single value. If the bit is set, this value is treated as an offset from the stack pointer. If the bit is not set, it is treated as an absolute address. This reduces the operand to a more manageable size. A typical instruction only requires three memory accesses to execute. Considering that all values are

stored in memory, this is as efficient as it can be. The advantages this solution has over using a fixed address make it the obvious choice.

This simple displacement addressing is not adequate. Languages like C support accessing a value through a pointer. To support this, the ISA must be able to read in a value from an address determined at runtime. At this point the operands can only do this based on the stack pointer. A more general solution is required. In particular the ISA must support reading a value from an address stored in a location on the stack. A few special instructions could be added to accomplish this. They would act like the standard load/store in a register ISA, but would only be needed when reading/writing to/from a dynamic address. However, one of the goals of the ISA is the elimination of explicit loads and stores. A more natural solution is expanding the addressing modes allowed for each operand. Specifically, one more level of indirection could be allowed for each operand. Instead of reading the value directly from the location on the stack, the value would be treated as an address and the final value would be read from that location in memory. A similar scheme would be used for destinations with the final access being a write instead of a read. This provides all the necessary functionality to support C.

The final design was created to support a stack based instruction set architecture and application binary interface. The ISA would support a small number of fixed purpose registers: none (zero), stack pointer, and frame pointer. The zero register is used to specify absolute addresses or constant values. The stack pointer specifies the end of the stack. It is used to communicate the stack location between function calls. The frame pointer is used to specify the top of the stack for the current function. It is needed to support a variable size function frame which is required to support alloca. Each operand would also support an indirection level specifier. This would allow constants, stack locations and dynamic addresses to be specified.

### 2.1.2 ISA and ABI

The instruction set architecture and application binary interface of the mem-machine are modeled on the Portable Instruction Set Architecture(PISA)[4]. PISA

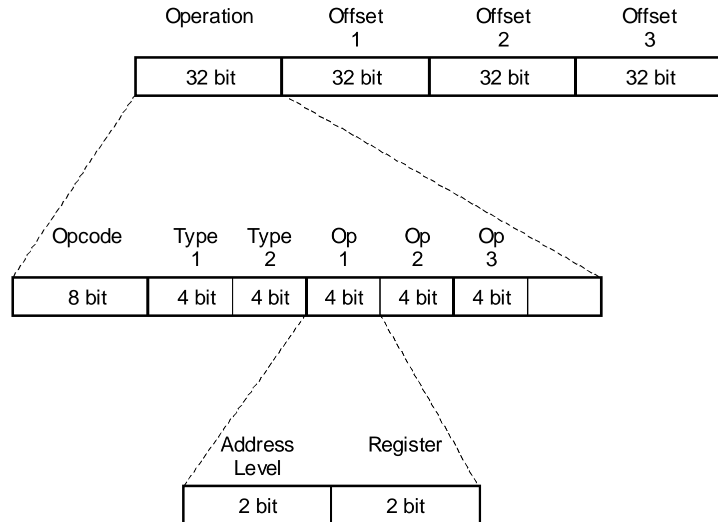



**Figure 2.1: Instruction Binary Encoding Format**

The mem-machine instructions are encoded in a 128 bit instruction. The instruction format is separated into set fields for easy decoding. The operation field contains the opcode of the instruction in the first 8 bits. The next 8 bits specify up to two value types for the instruction. Finally, the operation field contains one 4 bit field for each operand. The first 2 bits in this field specify the address indirection level of the operand. The other 2 bits specify one of three possible registers to use for the operand: none(zero), stack pointer, or frame pointer. The rest of the instruction is composed of three 32 bit offsets, one for each of the operands.

is supported by both the compiler and simulator the mem-machine was going to be implemented on and is typical of a modern day reduced instruction set architecture.

#### 2.1.2.1 ISA

The instruction set architecture is a specification of the specific operations supported, the operand specification (including a specification of the logical registers) and the instruction encoding. The mem-machine supports all the standard operations and is similar to most RISC like ISAs (see Appendix A for a list of supported operations). The ISA was designed to be extremely easy to target for a compiler and easy to decode. In particular, the operation has separate fields for the various options.

The basic instruction is 128 bits, see Figure 2.1. It includes a 32 bit operation and a full 32 bit offset for each operand. The first field is the opcode and signifies the type of instruction. It supports all the standard operations including a full set of

| Address Level | Read(Source) | Write (Destination) |
|---|---|---|
| 0 | Value = Register + Offset | Offset must be zero and Register must be the stack pointer or frame pointer. Register = Value |
| 1 | Value = Mem[Register + Offset] | Mem[Register+Offset]=Value |
| 2 | Value = Mem[Mem[Register + Offset]] | Mem[Mem[Register+Offset]]=Value |

**Table 2.1: Address Indirection Level Description**

The mem-machine supports three levels of address indirection for each operand. Level 0 is used for constant values, addresses and to read or write to registers. This level requires no memory accesses. Level 1 is used to read or write to stack locations or global variables. It requires one memory access. Finally, level 2 is used for read or writing through a pointer value. It requires two memory accesses.

conditional branches and conditional sets. This was done to minimize the amount of work needed to translate from the compiler intermediate format to the final instruction. The opcode is independent of value type. The next two 4 bit fields specify up to two of the ten supported value types. The architecture supports 32 and 64 bit floating point types, and 8, 16, 32, and 64 bit signed and unsigned integer types. Depending on the opcode, zero, one or two types are needed. For example, a jump requires no type, add requires a single type, and convert requires two types.

Finally, there are three 4 bit fields specifying how to treat the three operands. Each of these fields has two subfields. The first is the address level. This specifies the number of levels of indirection to use when reading or writing the value, see Table 2.1. The second field specifies the register to use - none (zero), stack pointer, or frame pointer. See Appendix B for an example of the assembly language for a simple function.

#### 2.1.2.2 ABI

The application binary interface is primarily a specification of the stack layout, the communication of arguments/results between functions and the function epilogue and prologue. PISA uses registers to pass some of the arguments and the

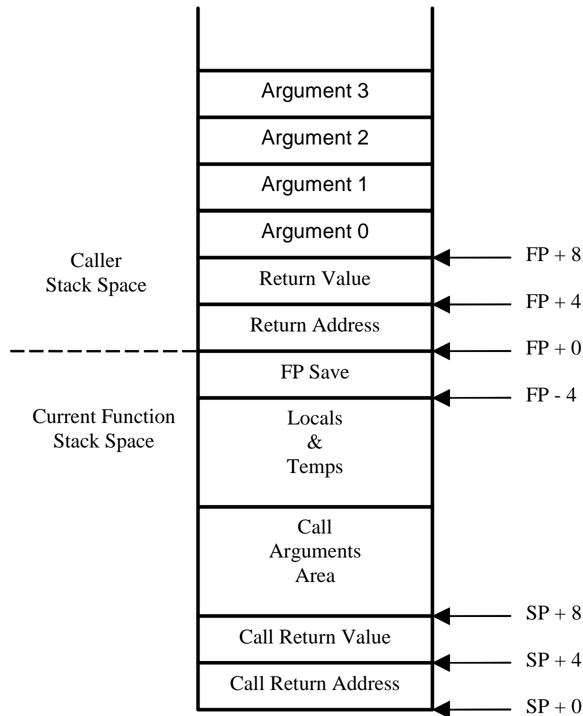



**Figure 2.2: Mem-Machine Stack Layout**

The stack layout of the mem-machine is similar to other RISC machines. The frame pointer (FP) is used to access the function arguments, function return, local variables and temporary values. The stack pointer (SP) is used to access the arguments and return value of any called functions. The return address is located at frame pointer + 0. The return value placeholder is at frame pointer + 4. Even if the function does not return a value, this space is reserved. The function arguments start at frame pointer + 8. The arguments are placed in order aligned to the appropriate address.

return address, and to return any values. The mem-machine uses the stack pointer register to communicate the location of the stack. All other values that need to be communicated are placed in fixed positions on the stack, see Figure 2.2.

Like PISA, the stack grows down. The return address is placed at the stack pointer. The return value placeholder is next on the stack at the stack pointer plus 4. If the return value is four bytes or less, it is placed directly at this location by the callee and no initialization needs to be done by the caller. If the return value is larger, the calling function allocates stack space for the value and initializes the placeholder with the address of this location. The callee uses this pointer to store

the result. This is similar to how PISA handles structure return values. Even if the function does not return a value, this space is reserved. This allows the arguments to always start at the same location and enables the compiler to setup a function call without knowing any information about the function being called. The arguments are placed starting at the stack pointer plus 8 and aligned accordingly.

The function prologue and epilogue are similar to other stack based systems. The prologue is composed of three things. First, the current value of the frame pointer is saved. Second, the stack pointer is copied into the frame pointer. Finally, a constant value is subtracted from the stack pointer to allocate all the local storage. The epilogue reverses the prologue. First, it restores the original stack pointer by setting it equal to the frame pointer. Second, it restores the original frame pointer by loading it back from its save location. Finally, the function returns using the return address saved at the stack pointer.

## 2.2 Implementation

The implementation of the mem-machine required creating a complete build tool chain and simulation environment. This required four major things. First, a compiler needed to be modified to target the new instruction set architecture and application binary interface of the mem-machine. Second, a C library needed to be ported. Third, a complete set of binary utilities needed to be created, including an assembler, linker and archiver. Finally, a simulator was ported to the new architecture to allow it to be evaluated.

### 2.2.1 Compiler

The compiler is a retargeted version of MIRV[13]. This required writing a checker module and printer module, and modifying the register allocation. The checker module is responsible for translating the internal assembly instructions into a form that could be executed on the target machine. For this ISA, this required two main things. First, the operands of each instruction had to be normalized into a form that the ISA could handle. In particular, operands that used displaced addressing (besides from the stack pointer) or indexed addressing had to

have an add instruction inserted before them to calculate the address and then place that address into a temporary variable. The problem operand was then switched to dereference this temporary value. Secondly, the ABI had to be implemented. This involved setting up the call stack for function calls and inserting the function prologue and epilogue. The printer module simply had to print the instructions in the format expected by the assembler. A very easy to parse format was used. Finally the register allocation needed to be modified. The standard allocator allocates architectural registers for each virtual register. If not enough are available or for certain other cases, the virtual register is instead assigned a spill location on the stack. This is appropriate for an ISA with general purpose registers, but is not appropriate for one using memory operands. A very simple allocator for this new architecture was implemented. It assigns a separate spill location to each virtual register. While not very efficient, it is very simple to implement. This has several repercussions though. MIRV relies on the register allocator to provide copy propagation. This simple allocator did not implement this. Therefore, the generated code will have extra moves. Secondly, this means that no reuse of stack locations will take place. This will have a negative effect on the cache performance and on the compressibility of the code.

### 2.2.2 C Library

A C library was provided by porting newlib[6]. Newlib was chosen because it was created to be as portable as possible. The task involved two main challenges. The first was to prepare the library for compilation by MIRV instead of a gnu compiler. This involved working around gcc specific options and defining the appropriate symbols. The second task was to provide the target specific code. This was primarily the startup code and the interface to the system calls. The system call interface is provided by a set of functions written in assembly language.

### 2.2.3 Binary Utilities

The binary utilities were created from scratch. The utilities included an assembler, archiver, ranlib and linker. In each case, the simplest but still functional ver-

sion was created. Specifically, they only needed to support the options and the functionality required by MIRV and the newlib makefiles.

The assembler is responsible for reading in the assembly file produced by the compiler and converting it into an object file. No attempt was made to perform any optimizations or to keep the file size down. The object file format is essentially a stored version of the structures used by the assembler.

The purpose of the archiver is to bundle several object files together to form libraries. In order to build the library, it needed to both insert and extract object files. A ranlib executable was also created, but it was just a stub and had no functionality. Its only purpose was to exist because it was called by the newlib makefiles.

The linker takes a set of object files and libraries and creates the final binary. Its two primary functions are to link all the labels and to layout the code and data. The label linkage needs to resolve all symbol labels to their final address. Some labels are local and defined in the same object file in which they are used. Other labels are global; these are defined in one object file, but may be used in other object files. To keep the code size of the executable small, the minimal set of object files should be linked. To accomplish this, the linker starts with the object file that defines the entry symbol and links it. As each object file is linked, it is checked for any undefined global labels. For each undefined label, the object files are searched until the one that defines it is found. If this object file is not already linked, it is linked now, and in turn checked. This recursive procedure continues until all the globals are resolved, in which case the compilation is successful. If a global label cannot be resolved, the linker exits with an unresolved symbol error. Once all the files have been linked, it inserts a special label to mark the end of the data. This is used by the code as the starting point for the heap. Next, the code and data layout is done. This determines the address of everything in the object files, obeying any alignment issues and size requirements. Finally, the executable is written to a file in a binary format.

### 2.2.4 Simulator

The simulator was provided by a new target for SimpleScalar. The port required creating several files. The header file describes the basic features of the architecture, including the instruction layout and register file description. The instruction definition file contains descriptions of the instructions, how they are decoded and their C implementation. The decode could be done directly, similar to PISA, because of the simplicity of the binary encoding. The loader file provides a function to load the executable from its file and copy the code and data into the simulated memory. It is also responsible for setting up the arguments and environment in the simulated memory in the positions expected by the target code. The final file primarily provides debugging features, in particular a function to print out a disassembled instruction in a human readable format. Most of the simulator files required no modification. However, no attempt was made to try to port the out of order simulator because of the complexity of the operands.

## 2.3 Evaluation

A few of the SPEC benchmarks[15] were compiled using the new system and the results verified against the reference output. The benchmarks were art00, compress95 and equak00. Each was compiled using three different levels of optimization: O0, O1, and O2. Two studies were performed. In the first, the dynamic instructions are characterized to generate a code profile. In the second, the mem-machine is compared against PISA to evaluate its effectiveness.

### 2.3.1 Code Profile

This section contains a profile of the dynamic instructions executed by the benchmarks. Three statistics are examined: instruction value type, operand register usage, and operand indirection level.

The results of the value type profile are shown in Table 2.2. In every case the most common type by far is the unsigned word. This is the type used for pointer

| | art00 | | | compress95 | | | equak00 | | |
|---|---|---|---|---|---|---|---|---|---|
| | O0 | O1 | O2 | O0 | O1 | O2 | O0 | O1 | O2 |
| Single Float | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Double Float | 11.01 | 20.54 | 20.54 | 0.43 | 0.46 | 0.47 | 11.27 | 18.19 | 18.11 |
| Signed Byte | 0.00 | 0.00 | 0.00 | 3.59 | 3.53 | 3.54 | 0.37 | 0.60 | 0.60 |
| Signed Half | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.07 | 0.12 | 0.11 |
| Signed Word | 11.04 | 9.55 | 9.55 | 28.98 | 33.12 | 33.23 | 11.31 | 16.01 | 16.22 |
| Signed Long | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Unsigned Byte | 0.01 | 0.01 | 0.01 | 7.45 | 7.99 | 8.02 | 0.40 | 0.65 | 0.64 |
| Unsigned Half | 0.00 | 0.00 | 0.00 | 1.97 | 2.11 | 2.12 | 0.20 | 0.32 | 0.32 |
| Unsigned Word | 74.57 | 57.94 | 57.94 | 57.52 | 52.50 | 52.35 | 73.70 | 58.53 | 58.35 |
| Unsigned Long | 3.37 | 11.95 | 11.95 | 0.06 | 0.28 | 0.28 | 2.70 | 5.60 | 5.64 |

**Table 2.2: Dynamic instruction value types**

The percentage that each value type appears in the complete dynamic execution of each of the benchmarks. The results are given for the three benchmarks at all three optimization levels.

arithmetic and for moving 32 bit values, regardless of the type. For 32 bit values, any extra moves in the system will be of this type. In general, as the optimization level is increased, the percentage of this type tends to decrease. This is most likely due to the reduction in extra moves and a general increase in the efficiency of the code. The two benchmarks that contain a significant amount of floating point code show an additional similar trend. As the optimization level increases, the percentage of code involving the double precision type increases. This is a good indication of the increasing efficiency of the code. Note that at the same time the percentage of unsigned long type also increases. Similar to the unsigned word, this type is probably used for moves involving 64 bit values, in this case double precision values. This is most likely the result of the reduction of address calculation and other extra code. In general, there is a relatively large difference between no optimization and the first level, but little difference between the next two.

The results of the register usage profile are shown in Figure 2.3. Unlike value type, in most cases there is very little difference between optimization levels. Most of the operands use the frame pointer register. This is the register used for tempo-

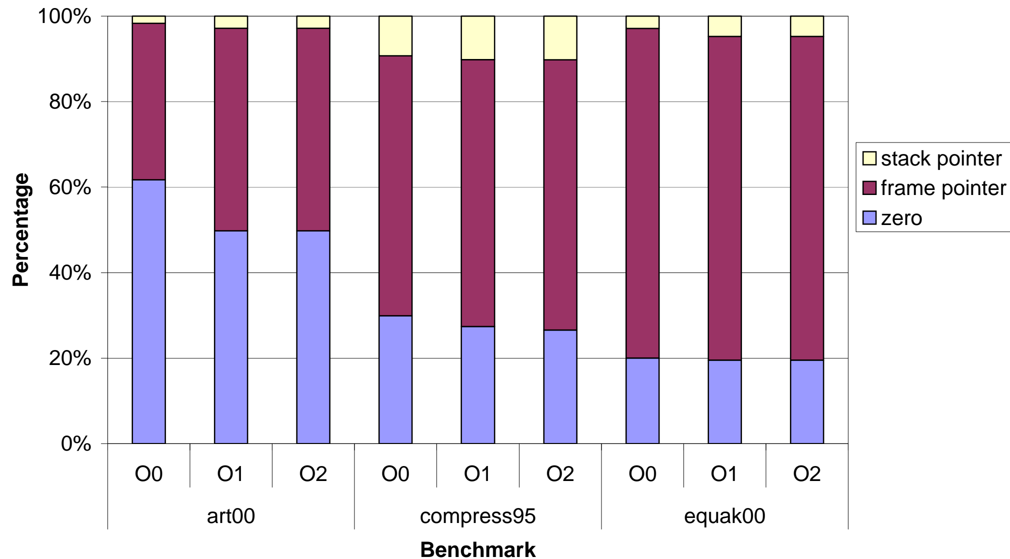



**Figure 2.3: Dynamic register usage**

Breakdown of the operand register usage. The results are given for all three benchmarks at all three optimization levels.

rary values and local variables. For compress, over 60% of the instructions use this type, while for equak it is over 75%. The next most often used register is the zero register. In most cases this is used when using a constant or specifying a label. The label could either be global data or a control flow instruction. It's interesting that art uses such a high percentage of zero register operands, and that the zero registers show a marked decrease in usage once optimization is applied. Art seems to make heavy use of global data, and in particular global arrays. The heavy zero register use is probably in address calculation instructions for these globals. The stack pointer isn't used very often. It is primarily specified for parameters passing. It can be used as an indirect indicator of the number of function calls occurring.

The results of the indirection level profile are shown in Figure 2.4. Like register use, optimization level seems to have very little effect on indirection. In general one level of indirection is the most common. This is the level used to access local values and parameters. The next most common is no indirection. Its percentage is almost exactly equal to the percentage of zero register usage. This would primarily be used for constants or labels, although it is also used to directly access the

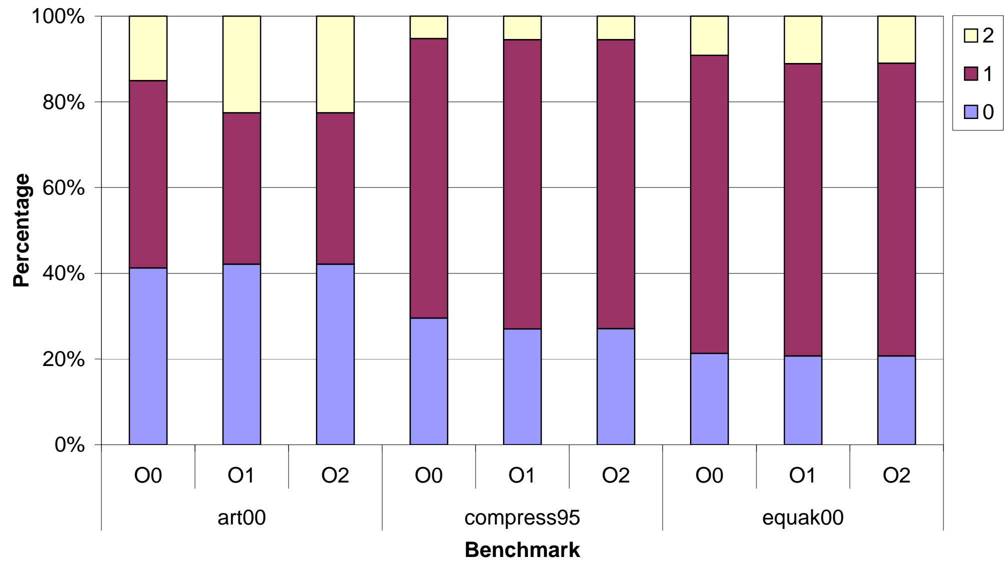



**Figure 2.4: Dynamic indirection level**

Breakdown of the operand indirection level. The results are given for all three benchmarks at all three optimization levels.

registers. If used for constants or labels, the zero register is used; this explains why the two percentages tend to be equal. In general, the direct register access would only be in the function prologue or epilogue. Two levels of indirection is not very common in the compress or equak benchmarks, but is used quite a bit more in the art benchmark. The art benchmark also has a relatively high percentage of no indirection operands. Once again this can be explained by its heavy use of global arrays. The no indirection operands are probably used in address generation. The two levels of indirection are necessary for reading or writing values using the generated addresses for array accesses.

## 2.3.2 Comparison vs. PISA

The same benchmarks were compiled using MIRV for the Portable Instruction Set Architecture(PISA) target and simulated using SimpleScalar. They also used a ported version of newlib. Using the exact same compiler and C library ensures a fair comparison.

The benchmarks were run to completion using the simulator, and a comparison of the number of executed instructions was made, see Figure 2.5. The num-

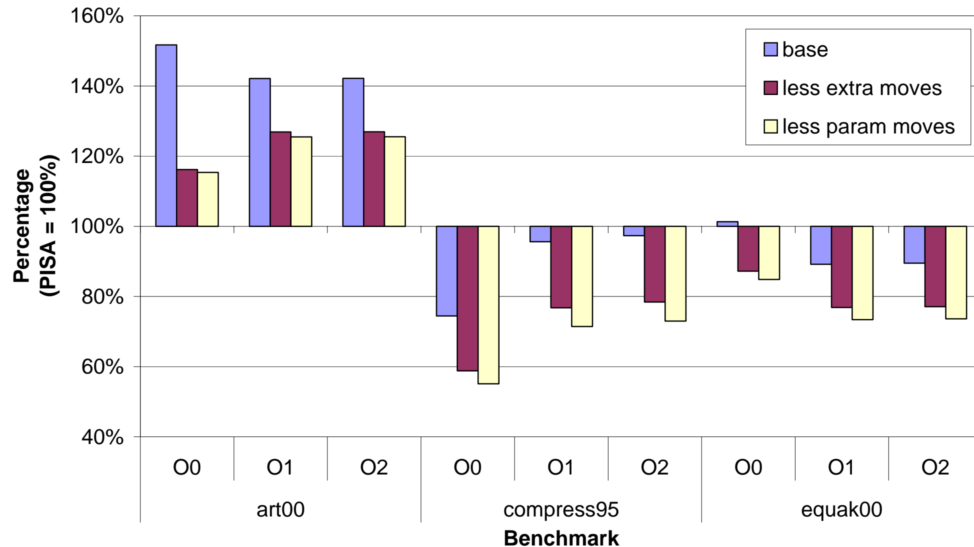



**Figure 2.5: Dynamic instruction count comparison versus PISA**
Less extra moves adjusts the number of dynamic instructions by removing those instructions that appear to be extraneous moves caused by the lack of copy propagation. Less param moves includes both the regular extraneous moves and the moves that use the stack pointer (param moves). The param moves are more likely to be actually required.

ber of instructions for PISA was normalized to 100 and the number executed by the mem-machine was compared to this. The base numbers correspond to the full number of instructions executed. However, the lack of copy propagation leads to extra moves. Any move with a source operand with one or zero levels of indirection and a destination with one level of indirection is a potential extra move. Any operand that uses the destination of the move as a source could theoretically use the original source of the move. Therefore, this is a rough estimation of the potential number of moves that could be eliminated. However, this would include moves to or from parameters and return values. These are much less likely candidates for elimination. Any moves that involved the stack pointers are probably in the second category. The less extra moves bar adjusts the number of instructions by removing all the potential extra moves. The less param moves bar also removes the parameter move instructions. This gives an approximate indication of the number of instructions once the compiler is more advanced. The number of extra moves is non trivial for all these benchmarks. In most cases it is around 15% of the instructions executed.

The performance of both compress and equak on the mem-machine compared favorably with that of PISA. Compress at no optimization only executed about 75% of the instructions compared to PISA. However, as the optimization level increases, this gap is reduced until at full optimization they are almost the same. It seems that the optimization of the PISA target is more effective on this benchmark then on the mem-machine. If the extra moves are taken into consideration, even at the highest optimization the mem-machine executed about 20% fewer instructions. Equak has the opposite trend. At low optimization, both executed about the same number of instructions. However, as soon as any optimization is done, the mem-machine required about 10% fewer instructions. Obviously some optimizations work better for RISC like targets, while others have more effect on the mem-machine. The performance of art on the mem-machine was very inefficient in comparison to PISA. Even when the potential extra moves are removed, it still required around 25% more instructions. Once again this comes down to the characteristics of the benchmark. In this case, art does quite a bit of access to global arrays. PISA is able to handle these using either displacement or indexed addressing. However, the mem-machine cannot support these directly for an operand. Therefore, it needs to use add instructions to do explicit address calculations. MIRV assumes that these addressing modes are allowed, so the internal assembly is generated to reflect this. The assembly code for any type of array access is optimized to make use of these address modes for efficient code on its usual RISC targets. The mem-machine backend is forced to insert the necessary adds, and it does it on a case by case basis and doesn't do any intra instruction optimization. If the assembly generation could be adjusted to take the lack of these instructions into account from the start, more efficient code could be generated.

The benchmarks were run over a range of cache sizes. All the caches are direct mapped with a 32 byte line size, and only the most optimized code was run. As expected, the mem-machine accesses the data cache much more often than the PISA target, see Figure 2.6. In general, it averages between 2 and 2.5

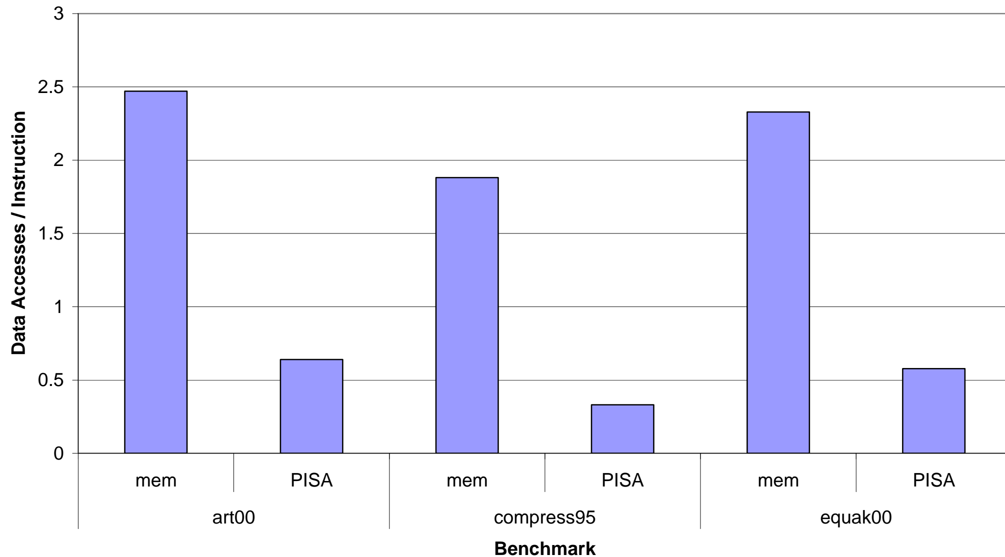



**Figure 2.6: Average data accesses per instruction**
The data cache accesses per instruction for the mem-machine and PISA. The results are given for all three benchmarks at the highest compiler optimization level.

accesses per instruction depending on the benchmark, while PISA averages between 0.3 and 0.7. Interestingly, the benchmarks display the same relative trend on both machines. This is somewhat unexpected. The mem-machine needs to access memory for all its temporary and local values besides just global data. In particular, about the same number of accesses should be required regardless of whether the data is global or local. I would have expected the average number of accesses to stay relatively consistent.

The most important cache consideration for the mem-machine is how all the extra accesses will affect performance. A simple comparison of miss ratios would not be of any use. The mem-machine has so many more accesses that the miss ratio should be very low. Therefore, a comparison of the absolute number of misses is needed. In particular, the number of extra misses that the mem-machine suffers will directly translate into decreased performance. To provide some context, this difference is given as a percentage of executed instructions. Therefore, it becomes the percentage of additional instructions (compared to PISA) that suffer a data cache miss, see Figure 2.7. For compress, this is around 1% for most of the cache sizes and remains relatively stable. Once the cache reaches 128 kilo-

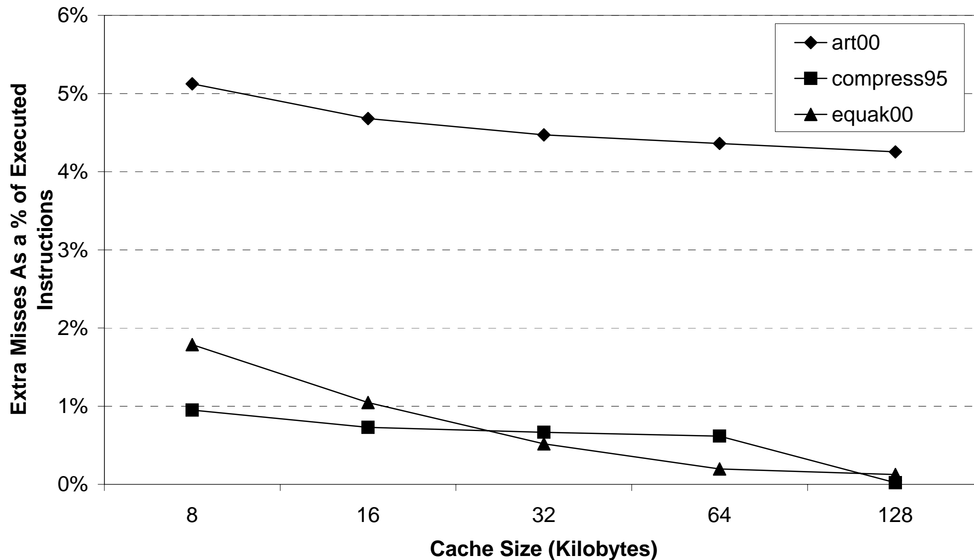



**Figure 2.7: Performance impact of extra data cache misses**
The difference in the number of data cache misses between the mem-machine and PISA, normalized to the number of instructions. This gives an indication of the relative performance cost of any extra data cache misses. The results are given for the most optimized compilation. All caches are direct mapped.

bytes, this percentage drops to almost zero. The sudden drop off seems to indicate this isn't a capacity problem but something else, possibly conflict problems. Equak shows a different trend; the percentage steadily decreases as the cache size increases. This seems to indicate that it's primarily capacity issues for this benchmark. The percentage starts at almost 2% for an 8 K cache, but quickly drops to 1% at 16 K, then 0.5% at 32, and eventually settles around 0.1%. The performance penalty these benchmarks suffer due to the additional data cache misses should be relatively low for reasonable size caches. With a 32 K cache, for both benchmarks, only about a half a percent more instructions suffer a data cache miss. Depending on the cycle penalty for a cache, this is probably not enough of a loss in performance for the PISA target to out perform the mem-machine. The art benchmark has a much larger penalty than the other two benchmarks; even with a 128 K cache, over 4% of the instructions are suffering a cache miss. Although it decreases as the cache sizes increase, it only falls from about 5.1% with an 8 K cache to 4.25% with a 128 K cache. Both PISA and the mem-machine suffer a large number of misses for this benchmark regardless of the size

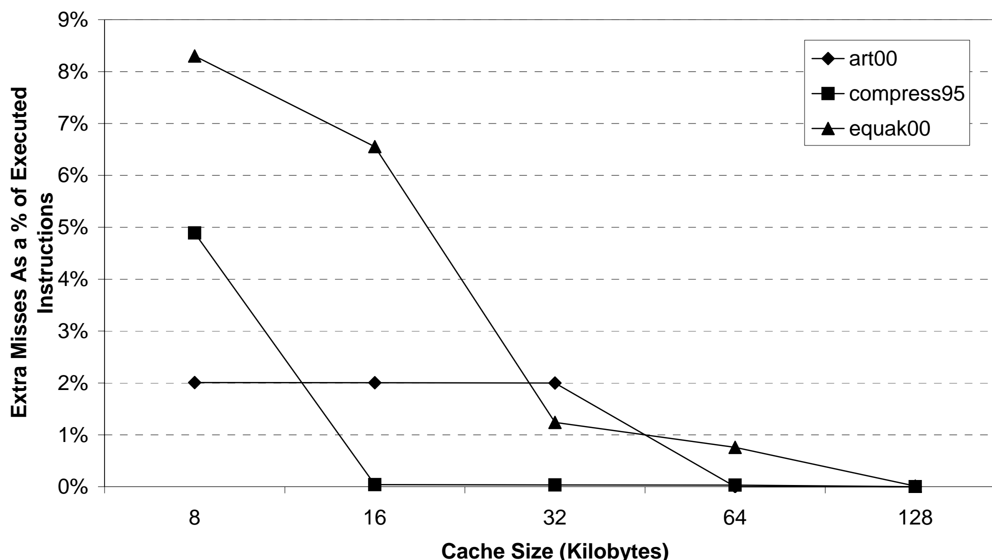



**Figure 2.8: Performance impact of extra instruction cache misses**

The difference in the number of instruction cache misses between the mem-machine and PISA, normalized to the number of instructions. This gives an indication of the relative performance cost of any extra instruction cache misses. The results are given for the most optimized compilation. All caches are direct mapped.

of the cache. However, because the mem-machine makes so many more accesses, the number of misses is greater by an almost fixed ratio.

The instruction cache performance is quite a bit different than the data cache, see Figure 2.8. In this case, the mem-machine is not making more accesses; however, each access is for 16 bytes instead of 4. The cache performance is primarily going to depend on how well loop bodies fit into the cache. If the entire body fits in, the cache performance will be good; if it doesn't it will be very bad. For a small instruction cache, the mem-machine suffers quite a larger performance penalty. Eventually when the threshold is reached, and the loops fit in the cache, the penalty drops to almost zero. As expected, it takes a cache about four times larger to show the same performance for the mem-machine as PISA. For these benchmarks, an 8 K cache is definitely too small to give good performance. At 32 K, the performance penalty is less than 2%, and it reaches about zero for a 128 K cache.

Code size is another important consideration for the mem-machine. The instructions for the mem-machine are four times the size of the PISA instructions;

| | art00 | compress95 | equak00 |
|---|---|---|---|
| PISA instructions | 13,644 | 11,465 | 15,433 |
| Base instructions | 11,753 | 9,953 | 13,478 |
| Less extra moves | 9,717 | 8,188 | 11,070 |
| PISA instruction size | 32.00 | 32.00 | 32.00 |
| Mem-machine instruction size | 128.00 | 128.00 | 128.00 |
| Base compressed instruction size | 33.33 | 32.62 | 37.37 |
| Less moves compressed instruction size | 34.13 | 33.39 | 38.15 |

**Table 2.3: Code size statistic**

The first three rows show the static number of instructions contained in each benchmark. The last rows show the average instruction size. The mem-machine statistics are shown both for the base and with the potential extra moves removed. The instruction size is shown for both uncompressed and with a simple compression scheme employed.

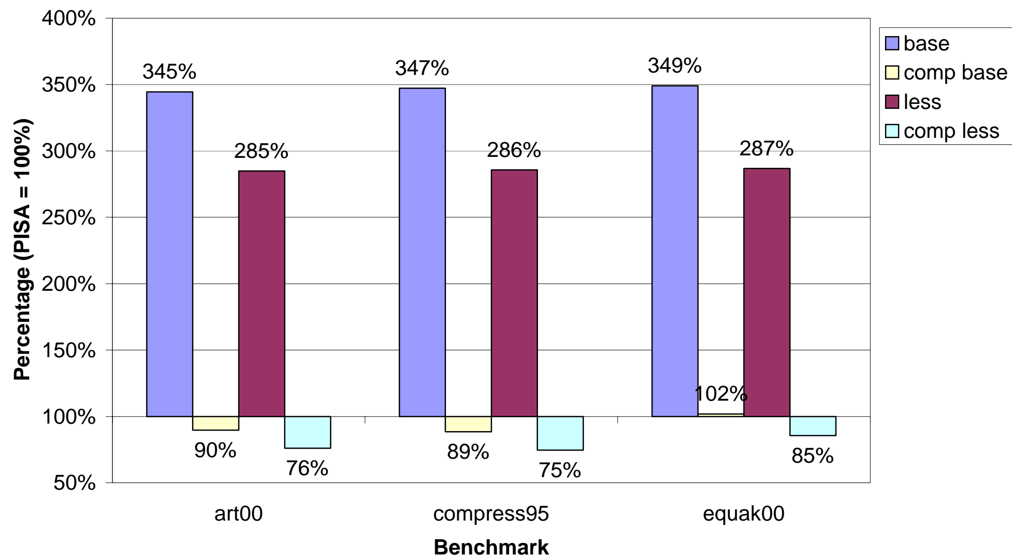



**Figure 2.9: Code size comparison versus PISA**

The static code size for all three benchmarks normalized to the size of the PISA code. The results are shown both base and with the extra moves removed (less) and uncompressed and compressed (comp).

however, the mem-machine executables tend to contain fewer instructions, see Table 2.3. For these benchmarks, the code size for the mem-machine is about 3.5x the size of the code for the PISA executable, see Figure 2.9. If the possible extra moves are removed, it becomes only about 2.8x the size. Code compression should work well on the mem-machine, because although it has 32 bit offsets

| Operation | | Offset | |
|---|---|---|---|
| Size (bits) | Number of Values | Size (bits) | Number of values |
| 4 | 8 | 1 | 1 |
| 7 | 32 | 7 | 32 |
| 11 | 256 | 12 | 512 |
| 36 | - | 36 | - |

**Table 2.4: Compression scheme.**
The simple compression scheme used to compress the mem-machine code. The operation and offset fields are compressed separately. The values are sorted by order of frequency and a simple replacement encoding scheme is used. Four different encodings are used. The first three provide compression. They encode the value in the number of bits specified by the size, but only for the number of unique values given by the second column. The final encoding is used for all the rest of the values.

and a 32 bit operation, the number of unique values these have in any executable should be low. In particular, some operations, for example unsigned word move, will be much more common while others are either not possible or not likely. The offset behavior is a bit more varied. Some operations require fewer than 3 operands, so zero will be a very common value. Standard offsets for locals and parameters will also be common. However, labels (both code and data) will tend to be unique, at least to the executable. A compression scheme could be used to reduce the size. A simple one is to sort the values by order of use and use a bit encoding to encode the most common values. For the mem-machine, it makes sense to treat the operation and offset fields separately, see Table 2.4 for the encodings used. If this simple compression scheme is applied, the base size of the mem-machine code shrinks from about the same size to about 10% smaller than the PISA. If the extra moves are removed, the code size becomes 15% to 25% smaller than the PISA. The average size of a compressed mem-machine instruction is still a couple of bits larger than a PISA instruction (see Table 2.3), but because there are fewer instructions, the total size is smaller. Notice that the average size of the compressed instruction increases when the extra moves are removed. This is because these moves probably all have the same or at least a

very common opcode and they only need two parameters. Therefore, they are probably some of the more compressible instructions.

## 2.4 Conclusion

The mem-machine exhibited mixed results. Some of the benchmarks were able to show significant performance improvements compared to a traditional RISC ISA. Because the mem-machine is a memory-to-memory architecture, it can eliminate explicit load and store instructions. The mem-machine removes all general purpose registers and therefore is also able to remove all the save and restore instructions normally necessary on a function call/return for a RISC instruction set. However, not all benchmarks show a performance advantage. In particular, equak00 requires more than 25% more instructions to execute on the mem-machine than on PISA. The mem-machine suffers from a limit on the addressing modes available to its operands. Specifically, it is unable to support displacement or indexed addressing. The mem-machine is therefore forced to insert extra instructions to explicitly calculate these addresses.

Although the mem-machine generates many more accesses to the caches, the performance impact is relatively minor. Only a small percentage of additional instructions will suffer a data cache miss, with the percentage decreasing as the size of the data cache is increased. The instruction cache shows similar results. Although the mem-machine instructions are four times the size of a typical RISC instruction, the extra misses in the instruction cache approach zero with a reasonable cache size. This is due to a combination of factors. First, the static code size was reduced. Therefore, the instruction cache can be used more efficiently. Second, the critical factor in instruction cache performance is that entire loop bodies fit into the cache. Once this threshold is achieved performance is relatively good.

The results of the mem-machine are affected by compiler limitations. The compiler used to generate code is a modified version of MIRV. The compiler was created to target reduced instruction set architectures. This has ramifications with regard to the quality of the code generated for the mem-machine. First, the com-

piler is optimized to generate intermediate code that uses indexed and displacement addressing. This forces the mem-machine to undo this addressing on a case by case basis. Although the final ISA had no general registers, compilers usually generate intermediate code that uses an unlimited pool of virtual registers to describe the data dependencies. Second, only a very simple allocator for this new architecture was implemented. It assigns a separate spill location to each virtual register. While not very efficient, it is very simple to implement. This has several repercussions though. MIRV relies on the register allocator to provide copy propagation. This simple allocator did not implement this. Therefore, the generated code will have extra moves. In addition, this means that no reuse of stack locations will take place. This will have a negative effect on the cache performance and on the compressibility of the code.

Perhaps the greatest complexity in the mem-machine would be designing an implementation. A simple single issue in-order core would be possible. However, it would still require complex hazard detection and a large number of data cache ports. The only realistic out-of-order superscalar implementation would require renaming the memory locations into physical registers (see Chapter 3 for an example of such a rename). This would be necessary both for dependency tracking and providing a practical data path. In such a system, all zero and one level indirection operands could be directly mapped into physical registers. However, the dynamic nature of a two level indirection operand makes this impossible. Instead, the operand would require the insertion of an additional micro op. Such a micro op would be responsible for generating the address (the first level of indirection). This is reminiscent of how loads and stores work in conventional RISC microprocessors. This relationship is further strengthened by the fact that two level indirection operands occur at a rate similar to loads and stores.

In general, the disadvantages of the mem-machine outweigh its advantages. The performance improvements are relatively mixed on the small set of benchmarks we examined. The instructions size is much larger. As the compression results show, these offsets can be compressed into approximately the same size as a register specifier in a RISC ISA, but this would require the addition of com-

plex decompression hardware. Finally, the mem-machine would be difficult to realize in hardware, especially in an out-of-order pipeline. The greatest advantage the mem-machine has is a complete lack of register context. This would allow a large number of simultaneous contexts to be supported on a processor. In almost all cases, register allocation is not the place to perform the memory mapping of the registers.

# Chapter 3

# VCA Design

This chapter provides details on the theory and design of the virtual context architecture. It is composed of four sections: 1) theory 2) implementation 3) SMT and multiprocessor implications and 4) rename table optimizations. The theory section describes the theory that underlies the operation of the virtual context architecture. The implementation section describes the specifics on how the VCA can be implemented. The third section examines some of the implications a multi-threaded or multiprocessor environment has on the virtual context architecture. The final section discusses some potential optimizations that can be applied to the implementation of the rename table. Finally, the chapter ends with a summary of the pipeline modifications.

## 3.1 Theory

The virtual context architecture is a modification to the processor pipeline. By memory mapping the logical registers, we seek to keep all the advantages of registers while adding the automatic context management of virtual memory. This section explores the theory behind these modifications. It is composed of four subsections. The first subsection describes the processor model that is used as a base for the virtual context architecture. The second subsection describes how the memory mapping is accomplished. The third subsection explains the new states required for the physical registers. The fourth subsection discusses the new operational requirements of the register rename stage.

### 3.1.1 Processor Model

The design of the virtual context architecture builds on that of a typical superscalar out-of-order processor [32]. Instructions are dynamically issued from an instruction queue out of program order, but commit in program order using a reorder buffer. To eliminate false dependencies and buffer speculative values, register renaming is used to dynamically assign logical registers to physical registers. We assume a merged register file implementation, like the ones used by the MIPS R10000 and Alpha 21264 [14, 38]. This implementation maintains both the speculative and architectural state of the registers in the physical register file with the reorder buffer containing just the physical register tags.

The processor model was chosen to reflect the current state of the art out-of-order pipeline. The VCA is in no way limited to this particular processor model. In general, the techniques we employ to virtualize the register context can be applied to most processor models. This includes processor models that use reservation stations instead of a reorder buffer and store the actual value of source registers in the buffer instead of just the tags.

Although it is not as natural a fit, the virtual context architecture could also be applied to an in-order processor. In this case, the pipeline would need to be modified to implement register renaming. This would require the addition of a new stage to perform register renaming. It would also require that the backend of the pipeline use the physical register tags for purposes of hazard detection and value forwarding.

### 3.1.2 Memory Mapping

The virtual context architecture maps logical registers to memory locations. The physical register file caches register values based on these memory addresses. As a result, a physical register can hold a value from any context at any point in time. The exact technique used to accomplish this mapping is in some ways independent of the virtual context architecture. There are only two requirements: independence and consistence. The two requirements when taken

together ensure a unique one to one mapping between a given logical register in a specific context and a memory address.

The memory mapping must result in addresses that are independent of each other. In particular, the only time two logical registers can map to the same memory address is if they are the same logical register within the same context. If either the logical register id or context (be it thread or function depending on the model) is different, the mapping address must be different. This is required to guarantee that there are no false positives when a physical register is looked up. In other words, when the memory address is used by rename to translate to a physical register, if the translation is successful (a physical register is caching that memory address), the value represented by that physical register must correspond to the value of the logical register.

The memory mapping must also result in addresses that are consistent. A particular logical register within a particular context must always map to the exact same address. This is a similar but subtly different requirement than independence. This is required to guarantee that there are no false negatives when a physical register is looked up. In other words, when the memory address is used by rename to translate to a physical register, if the translation is not successful (a physical register is not caching that memory address), the current value of that logical register must not be contained in the physical register file and must be present in memory at that address.

A simple way to accomplish this mapping is using a base address within the processes address space for each context. Depending on the usage model, a register context can be as specific as a particular activation record within a particular thread. Adding the scaled logical register index to the base address produces a memory address that is unique for that logical register within the context and also unique across all contexts.

In the case of simultaneous multithreading, the address space associated with each process provides the independence. In the case of register windows this base address is updated on each function call and return. The exact operation depends on whether the register windows are overlapping or not. In the case of

non overlapping windows, a function call would cause the base register to increment by a fixed amount equal to the total memory space of a single context's worth of registers. Thus the new function has a completely new set of memory locations for its logical registers. On a function return, the base address is decremented by a fixed amount. This restores the mapping of the previous function. In the case of overlapping register windows, the increment size is smaller. The memory locations of the two function contexts are allowed to overlap. In some sense this breaks the independence rule for the mapping. In particular, some registers in the calling function map to the same memory address as some registers in the called function. However, in the overlapping register window case this is the desired behavior. Although the logical registers are in different contexts, they are conceptually the same register. The virtual context architecture naturally and easily handles this.

To simplify the implementation, only the pipeline itself is allowed to read/write to this address range. In particular, loads and stores should be prevented from accessing this memory. If they were allowed to, complex dependency checking would be needed in the pipeline to detect loads/stores to addresses that are mapped into physical registers. Basically, once the address for a load/store was determined, the address would need to be sent to the rename table to determine if that location has been renamed to a physical register. This is further complicated by the fact that the in order semantics of the instructions would somehow need to be maintained. The address for a load/store is calculated some number of cycles after going through rename. It is possible that an instruction after the memory instruction defines the logical register corresponding to that memory location. The instruction is allocated a new physical register and the rename table is updated to map that address to this newly allocated physical register. The rename table would identify that new physical register as the current mapping, even though it wasn't the mapping when the memory instruction was renamed. To fix this, the pipeline would have to squash all the instructions following the problem memory access and restore the rename table to the state it was in when the memory instruction was renamed. This process is similar to recovering on a mispredicted

branch. The logic would be further complicated by things like byte stores to registers.

### 3.1.3 Physical Register States

In a conventional out-of-order machine the physical registers are in one of two states: free and not free. A register is not free when it contains architectural state, either speculative or not. All of the registers in the system that are not free are the committed state for all the logical registers, plus any physical registers renamed to destination registers of yet to be committed instructions (all those instructions between rename and commit). All the other physical registers are considered free and are available to be used by rename. A physical register becomes free when a later instruction which writes to a logical register commits. The physical register that previously held the architectural state of that logical register becomes free, because no instruction will ever need the old value.

Unlike the two states of a physical register in a typical machine, each physical register in the virtual context architecture is considered to be in one of four states: unavailable, available dirty, available, and free. The *free* physical registers are the same as those in a conventional out-of-order processor. These correspond to registers without valid state. In other words, the semantics guarantee that no later instruction will ever need the value held in this physical register. This registers are free to be used by rename. The non free physical registers in a conventional machine are split into the three other states in the virtual context architecture. The distinction is based on the physical register's availability or lack thereof to be used by the rename stage.

A physical register is *available* when two conditions are met. First, the physical register must contain architectural state, i.e. a committed value. A physical register meets this condition once the instruction defining it is committed, until a later instruction overwrites this value and the physical register becomes free. Second, there can be no instructions in the pipeline after the rename stage that have not read the value. If either of these conditions is not met, the physical register is considered *unavailable*. The second condition is critical. It is the guarantee that no

instruction after rename requires the value any more. If this is the case, the physical register can be moved to memory and the physical register becomes free. Available registers are additionally considered *available dirty* if they have been modified since the last time they were moved to memory. An available register that is not dirty can be used freely by the rename stage. However, if the available register is dirty then its value must first be moved to memory before it can be reused.

The virtual context architecture operates by treating the physical register file as a cache of memory mapped logical registers. This requires that physical registers be removed from the physical register file and moved into the cache hierarchy. The unavailable state of a physical register can be viewed as a lock on the physical register. Only an available physical register can be moved from the physical register file into memory and the corresponding physical register freed.

### 3.1.4 Register Rename Stage

To avoid modifying the scheduling and execution stages of the pipeline, we guarantee that the physical register indices used by instructions in the instruction queue are valid. Instructions issuing from the queue simply read their operands from (and write their results to) the physical register file without any tag checking or potential miss condition. Instruction scheduling is also performed based on physical register indices as in a conventional processor.

To guarantee valid physical registers for each instruction in the queue, the caching behavior of the physical register file is managed as part of the register renaming process. If a logical register does not have a corresponding physical register, the rename stage allocates a physical register and initiates a *fill* operation to bring the register value in from memory. If necessary, the rename logic will first *spill* an existing physical register's value back to memory so that the register can be reallocated. The new logic of the rename stage is shown in Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4.

Figure 3.1 details the logic used to find a physical register. First, the rename table must check to see if there are any free physical registers. If yes, it uses the

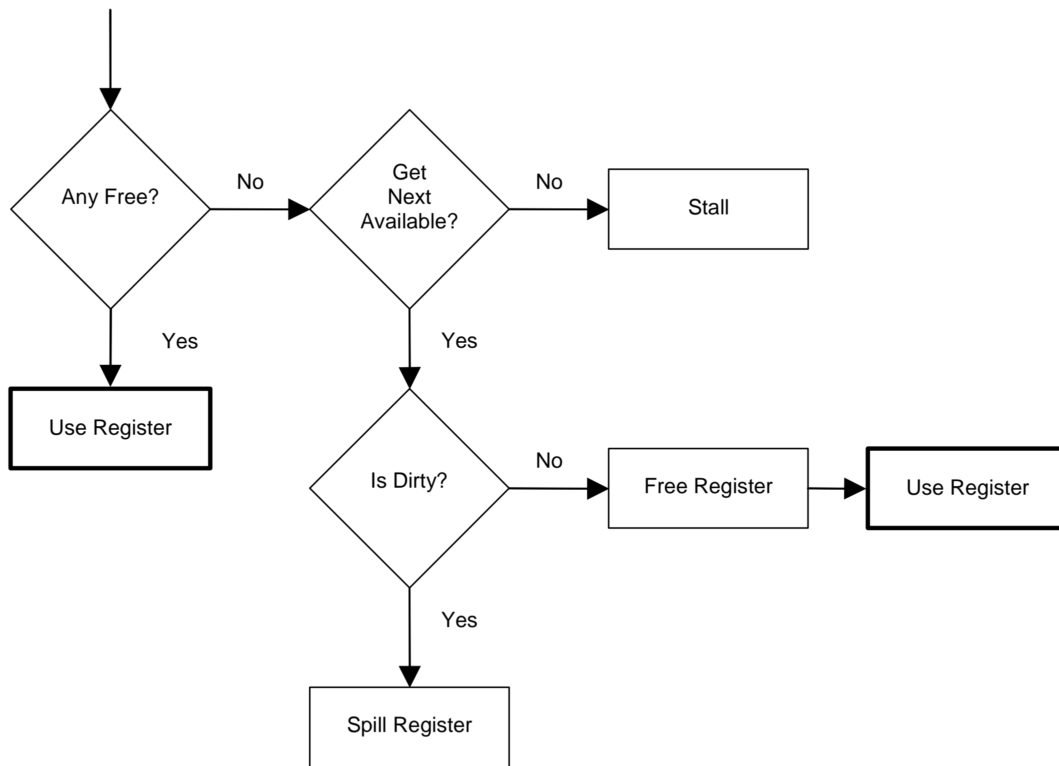



**Figure 3.1: Finding A Physical Register To Use**
The logic used by the rename stage to find the next physical register to use.

free physical register. If no, it uses some replacement policy to get the next available physical register to replace. If there are no available registers, the rename stage must stall this cycle. If there is an available register, it checks if it is dirty. If the register is not dirty, the rename stage frees the physical register then uses it. If it is dirty, the register must be spilled. Once the spill is complete, the dirty status of the physical register will be cleared, and the register can be freed.

Figure 3.2 details the logic used to find a rename table entry. First, the rename table checks if there is an unused entry. If yes, it uses that entry. If all the entries are used, the table checks if any of the entries are physical registers that are available. If all the physical registers are unavailable, the rename stage must stall this cycle. Eventually, one or more of the physical registers will become available. If one of the physical registers is available, the table checks if it is dirty. If the register is not dirty, the rename table frees the physical register then use its entry. If it

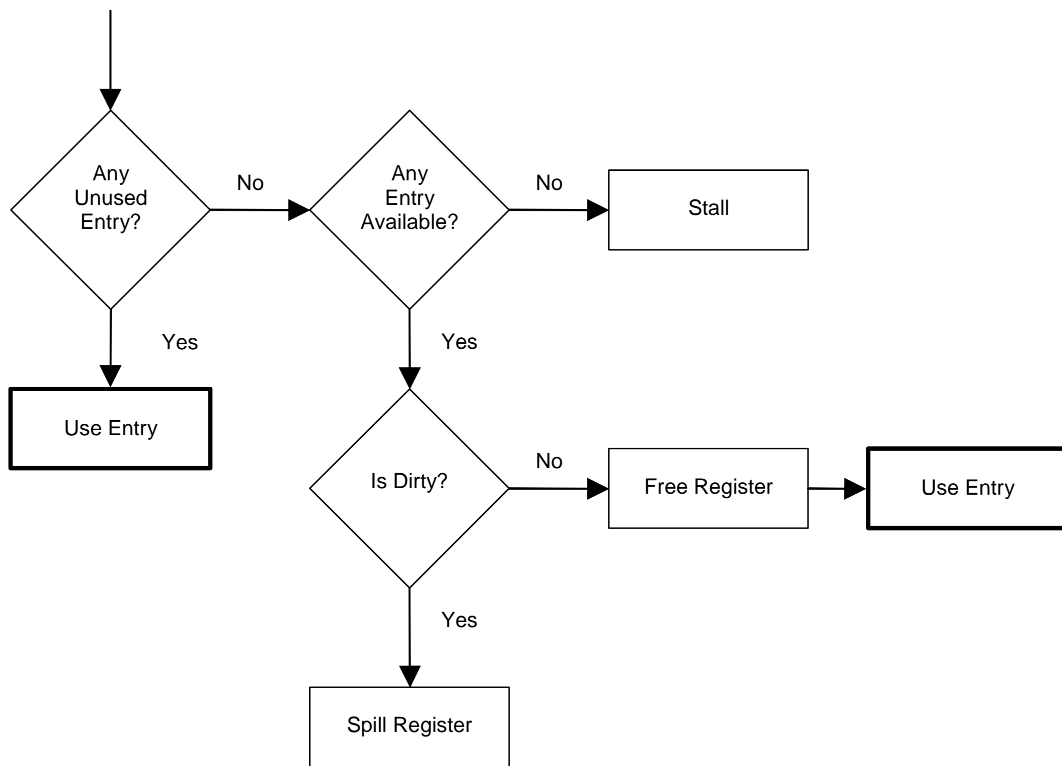



**Figure 3.2: Finding A Rename Table Entry**

The logic used by the rename table to find a rename table entry.

is dirty, the register must be spilled. Once the spill is complete, the dirty status of the physical register will be cleared, and the register can be freed.

Figure 3.3 details the logic used by the rename stage for a logical register acting as the source of an instruction. First, the memory address of the logical register source is looked up in the rename table. On a hit, the renamed physical register is used. On a miss, a physical register and rename table entry must be found. These operations can occur in parallel and are detailed in Figure 3.1 and Figure 3.2, respectively. If both a register and entry are found, a fill is done. The found physical register is used and the rename table entry is updated. If either a register or entry cannot be found, the stage is stalled this cycle.

Figure 3.4 details the logic used by the rename stage for a logical register acting as the destination of an instruction. A destination register requires two things: a physical register and a rename table entry. A physical register is found using the logic detailed in Figure 3.1. For the rename table entry, first the memory address

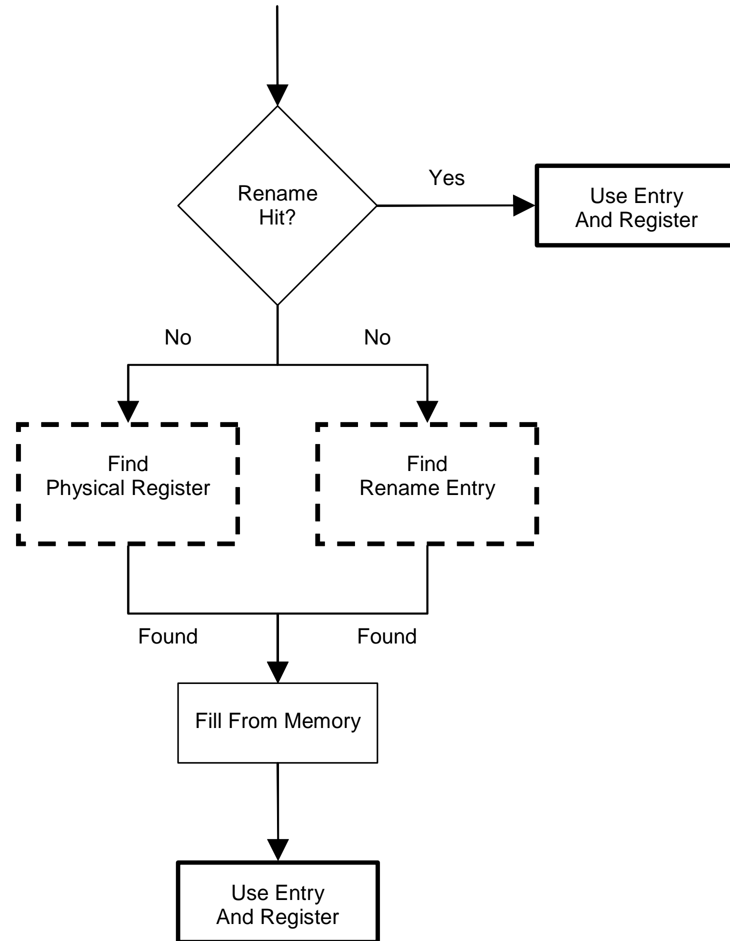



**Figure 3.3: Source Register Logic**
The logic used by the rename stage when a source register is renamed.

of the logical register is looked up in the rename table. On a hit, the matching entry is used. On a miss, the logic detailed in Figure 3.2 is used to find an entry. If either a register or entry cannot be found, the stage is stalled this cycle.

The virtual context architecture enables the physical register file to hold just the most active subset of logical register values, instead of the complete register contexts, by allowing the hardware to move registers to and from memory (the cache hierarchy) on demand. By using memory to maintain the architectural state of the processor, the number of physical registers is decoupled from the number of logical registers. This decoupling is what allows the processor to give the illusion of a nearly infinite set of logical registers. In particular, this enables the pro-

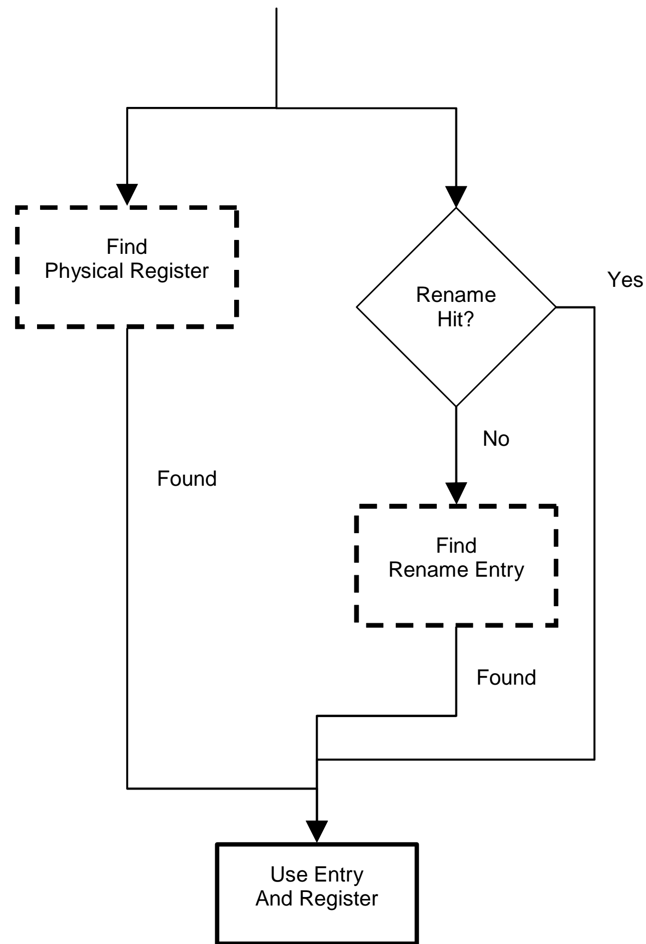



**Figure 3.4: Destination Register Logic**

The logic used by the rename stage when a destination register is renamed.

cessor to have fewer physical registers than logical registers. Normally, in a merged register file, the number of logical registers must be less than the number of physical registers to avoid a register-release deadlock [25]. In the VCA no deadlock is possible. Note that the physical register file contains all the operands and destination registers needed by instructions past the rename stage, so instructions will continue to execute, eventually making registers available and preventing deadlock. In the worst case, since instructions can continue to execute, the pipeline can eventually completely drain. At this point all the physical registers become available. Therefore it is never possible for the pipeline to deadlock because of a lack of physical registers.

## 3.2 Implementation

Building a machine based on the virtual context architecture requires some modifications to the traditional pipeline. These modifications can be grouped into four categories: rename, spill/fill implementation, branch recovery and operating system changes. Most changes focus on the rename stage, including extensive changes to the rename table itself. The pipeline requires minor modifications to allow register spilling and filling to/from memory. The increased size of the rename table influences the choice of branch recovery scheme. Lastly, the memory mapping of the registers requires some changes to the operating system.

### 3.2.1 Rename Stage

There are two main modifications that are needed to the rename stage. First, the rename table is indexed by memory address, requiring modifications to the table. Second, the pipeline must track the additional states of the physical registers.

#### 3.2.1.1 Rename Table

The rename table is no longer indexed by logical register; instead, it is indexed by a memory address. Therefore, the rename table can no longer be sized to handle all possibilities. The rename table must therefore become more like a cache. The rename table must have tags and valid bits. The tags are used to identify the memory addresses that the rename table entry represents (since only part of the address is used to index the table). The valid bit is used to signify a rename table entry that does not contain a mapping.

A simple direct mapped table may not be adequate. The rename table must be designed to prevent potential deadlocks. A deadlock is possible if two or more different logical registers in a single instruction map to a single rename table entry. Preventing deadlock in a single instruction is sufficient because the entire pipeline can be allowed to drain, which would make all the entries in the rename table available. In the case where a thread contains only a single context (no register windows), a simple direct mapped table with more entries than the number of log-

ical registers would work. In the case where a single instruction can contain logical registers from multiple contexts, such as global and windowed registers, the rename table must have some type of associativity. This associativity could be accomplished in many ways: fully associative table, set associative table or direct mapped table with a victim table. The only condition is that a single address must be able to be mapped to at least as many entries as there are potential contexts in a single instruction.

#### 3.2.1.2 Tracking Physical Register States

The second modification to the rename stage is that the pipeline must be able to track the additional states of each of the physical registers. A traditional pipeline only tracks if a physical register is free or not free. The physical registers in the VCA can be in four states (free, available, dirty available, not available). Therefore, the pipeline needs to track availability. A physical register is available when two conditions are met. First, the physical register must contain architectural state, i.e. a committed value. Second, there can be no instructions in the pipeline after the rename stage that are waiting to read the value. This can be simplified to the requirement that no instructions in the reorder buffer (ROB) can be using the physical register (depending on replay policy this may be overly cautious).

The first condition can simply be tracked with a commit vector, with one bit for each physical register. When an instruction is committed, the bit corresponding to the destination physical register is set. For the second condition, the pipeline must keep track of which registers are sourced by instructions in the reorder buffer. As in previous work[1, 22], this can be accomplished by a set of counters for each physical register that tracks the number of instructions after rename in the pipeline that use the physical register. The counters are incremented in the rename stage, and decremented at commit. Another way to maintain this is to keep a table indexed by physical register that contains the last ROB entry that uses the physical register. When an instruction is committed, it can index into the table with the source specifiers and if this ROB is the last entry, the physical register becomes available.

The second condition is much easier to track in an out-of-order pipeline that uses reservation stations or in an in order pipeline. In both these cases, the physical register file is accessed in an in order fashion. This is obvious in the in order case, but also holds in the reservation station case. In a pipeline that uses reservation stations the physical register file is only read from at rename, and the values are stored in the reservation station. Physical registers that are still pending are read from the broadcast network, however the first condition already precludes these from being available (they are not committed yet). Therefore, in both cases the physical register file is read in program order at rename. Therefore, the pipeline can guarantee that for any committed physical register, any instruction after rename will have already read the value from the physical register. This means that condition two is not relevant in these cases and therefore need not be tracked at all.

The dirty status of an available register also needs to be tracked. A simple bit vector can be used for this. When an instruction commits, it marks all its destination registers as dirty. A fill does not mark its physical register as dirty; the value is already in memory. Once a spill has completed, the dirty status of the physical register is cleared.

### 3.2.2 Implementing Register Spills and Fills

The implementation of register spills and fills requires the insertion of the equivalent of loads (fills) and stores (spills) into the pipeline. A simple way to accomplish this is to insert these as operations directly into the pipeline. However, these operations are simpler in several ways from the other loads and stores.

First, there are no input dependencies for these operations. The only input a fill has is the already determined mapping address of the register. A spill has two inputs: the register mapping address and the physical register being spilled. One of the conditions to spilling a physical register is that it contain committed architectural state. Thus, the instruction defining the physical register must already have been committed, and the physical register is always ready. Therefore, these operations are ready to be executed as soon as they are inserted. Second, as previ-

ously stated, the memory address for these operations is known when they are inserted and has certain characteristics. A traditional memory operation is placed in the load store queue (LSQ) where complex memory disambiguation logic determines when it is safe for operations to execute. The logic also checks for bypassing between stores and loads. This logic is not necessary for these spill/fill operations. Memory disambiguation is not needed because all the operations are ready to be executed as soon as they are inserted and they never need to be executed out of order. Bypass logic is not needed because of the nature of these insertions - a spill and fill to the same memory address will never exist in the pipeline at the same time. If a fill is already in the pipeline, a spill cannot be inserted because the physical register is not committed state until the fill has been committed. If a spill is already in the pipeline, a physical register will already exist that is mapped to the register and therefore a fill will never need to be generated. Third, these operations are never speculative. They are only transferring committed architectural state between memory and the physical register file. Although a speculative instruction can trigger them, the triggered operation itself is not speculative because it is only working with non speculative values. Therefore, these operations do not in fact need to be part of the speculative/squash system at all.

A more efficient way to implement spills and fills is to add a small queue—the architectural state transfer queue (ASTQ)—to the pipeline. On one end, the rename stage adds an entry to the queue when a spill or fill is necessary. On the other end, the ASTQ feeds a mux that merges cache accesses coming from the ASTQ and the instruction queue. If a memory function unit does not issue an instruction during a cycle, the next entry from the ASTQ is issued. A conventional tag broadcast is used to signal dependent instructions for register fills. In the case of spills, the rename stage is stalled until the spill has issued. The addition of the ASTQ means that spill and fill operations will not take up valuable resources by residing in the LSQ, instruction queue (IQ) and ROB. In an in order pipeline, these structures don't exist, and therefore directly inserting the spill and fill as operations in the pipeline would be the easiest solution.

In order for the rename stage to add a spill operation, it requires the address of a physical register to spill it. A table is needed to store these mappings. The table requires one entry for each physical register and contains the memory address tag. At commit, the memory address of the destination registers are stored in the table. The table is only read when a register is spilled. It requires as many write ports as the commit bandwidth of the pipeline, but only a few read ports since the rename stage only dispatches a small number of spills at a time.

The purpose of a spill is to move the register value out to memory (specifically the first level data cache) so that the physical register can be replaced by the rename stage. The rename stage must stall until it is safe for the register to be overwritten. The amount of time it must stall is dependent on the pipeline. In the tag only out-of-order pipelines we are concerned with, the stage must stall until the physical register has been read from the physical register file and put into the store buffer. The rename stage must wait until this is done to guarantee that the physical register would not be overwritten before it was read. If the rename stage did not stall, the renamed instruction could theoretically be issued and writeback before the spill operation was issued, causing the value to be spilled to be overwritten and lost. In an in order pipeline or an out-of-order pipeline that uses reservation stations, the rename stage does not need to stall at all. As discussed in Section 3.2.1.2, the order that instructions read from the physical register file is fixed. In particular, all the reads and writes occur in the order that instructions are dispatched from rename. Therefore, the rename stage can simply dispatch the spill before the instruction that uses the replaced physical register. As the spill operation is dispatched it will read the value from the physical register file. Thus, the overwriting instruction can be dispatched immediately after it without any stalls.

### 3.2.3 Branch Recovery

Branch recovery is a problem that all speculative pipelines face. Commonly, architectures checkpoint the rename table at each branch. Our rename table is larger than a conventional one, so this solution would be expensive in area.

Another recently proposed solution, which is used in the Pentium 4 processor, is a retirement map table [1]. This solution is more practical in the VCA because it requires only a single duplicate of the rename table. This duplicate rename table is kept in the commit stage. As each instruction is committed, it updates this retirement map table. When a mispredicted branch is committed, the retirement map table is now the correct rename table. To recover, the retirement rename table is copied to the rename table, and the pipeline is flushed. A simple optimization is to detect the misprediction at writeback time. The retirement map table is copied immediately to the rename table. The ROB is then walked from the oldest instruction backwards until the branch is reached, and each entry updates the rename table as if it was just renamed.

This branch recovery scheme integrates well with the counter scheme used to track the use of a physical register mentioned in Section 3.2.1.2. When the retirement map is copied to the rename table, all the use counters are reset to zero. If the optimized recovery is used, the counters are updated as the ROB is walked. The other scheme also fits in naturally with this type of recovery. Instead of zeroing the entries, all the entries are simply marked as invalid. In the optimized scheme, as the ROB entries are walked, the last use entries are updated.

### 3.2.4 Operating System

The virtual context architecture requires some minor changes to the operating system. Most of these changes are actually simplifications to the operating system. These changes include: reserving a location in virtual memory for the register mapping, implementing context switches/interrupts and changes to virtual memory management.

The location in virtual memory used for memory mapping is processor/operating system dependent. An unused range of addresses in the virtual address space is reserved for the registers. The operating system can then set up the base address to point to this location. When the operating system is asked to create new threads, it assigns each thread a new space in these reserved addresses. This is similar to what it is forced to do for the stack.

On a context switch the operating system saves the current base address to part of the process control block (like it used to do for all the registers) and loads the new thread's base address from its process control block. The saving and restoring maintains the consistency necessary in the mapping. Unlike a conventional machine, the logical registers themselves do not need to be explicitly saved or restored. The pipeline itself will move the new thread's registers from memory into the physical register file as they are needed, while it moves the old thread's registers into memory as it needs to free up physical registers. The physical register file is now simply a cache. Therefore, it no longer requires explicit management by the operating system in the case of a context switch. This is similar to how the data and instruction caches operate. On a context switch, nothing is explicitly done. Instead, the memory values of the new thread are efficiently loaded into the cache when they are needed while the old thread's memory values are move back into higher levels of the cache as they are replaced by the new thread's memory.

The same automatic context management that the VCA brings to context switching can also be used for interrupts. On an interrupt, the hardware can swap in a new base register. This effectively gives the interrupt handler an entire context of registers. There is no need for explicit saving or restoring. The same technique could also be applied to kernel code. In systems like the Alpha [31], a separate set of 8 shadow registers is used to optimize the exit and entry from PAL code. The VCA enables a very clean implementation of this. A separate base register for kernel/PAL code could be used. This would provide a completely independent set of registers for use in kernel code. A simple move instruction could then be added to move values between the different register sets.

Treating the physical register file as another level in the data cache hierarchy does add some complications to the management of virtual memory. In a conventional machine, when the operating system needs to move a page to the disk, it first issues a special instruction that forces all the caches to writeback any lines on that page to memory. For the VCA, this operation needs to be extended to the physical register file. In particular, if the operating system needs to reuse a page

of memory that is currently mapped to the part of the virtual address space that registers are mapped to, it must ensure that all registers mapped to that page are moved out of the physical register file. Therefore, the VCA requires a special instruction that spills all the registers that correspond to a particular page. This could be simplified to ease the complexity of implementation to either just the registers for a particular address space, or all the physical registers. These would trade efficiency for ease of implementation. The operating system would also have to pin any page that is used for register mapping of registers currently in use by the pipeline.

## 3.3 SMT and Multiprocessor Implications

To support multiple threads in the pipeline typically means that many structures in the pipeline must be duplicated. In the case of the virtual context architecture, most of the additional structures are completely independent of the number of threads, including the mapping table and use counters. Unlike a traditional pipeline, the rename table itself is not replicated. In fact, the only structures replicated are the pipeline registers holding the current base pointer. Thus, the VCA makes it practical to support large numbers of threads in a pipeline.

In a multiprocessor environment, treating the physical registers as another level of cache does raise the issue of coherence. Trying to make the physical register file coherent would be expensive because the physical register file has single word block sizes and logic would be needed to prevent an in use physical register from being removed from the pipeline. The physical register file is also accessed much more frequently than the data cache, and the latency of these accesses is critical to the performance of the processor. Our solution to this problem is two fold. First, we limit how these mapped addresses can be accessed. In our system these addresses should only be accessed through a register access; no code should ever need to explicitly address these mapped registers. This is desirable for several reasons, see Section 3.1.2. In addition, a protection scheme in the page table could potentially be used to prevent this from ever occurring. We

assume the operating system would keep the virtual memory segments separate for each software thread; then, only when a thread is migrated from one processor to another is coherence necessary. Second, we make use of the special instructions needed for virtual memory management in the VCA, see Section 3.2.4. Specifically, the VCA requires an instruction to force some or all of the physical registers to be spilled to memory when the operating system needs to reclaim a memory page used to hold register values. The operating system can detect when a thread is being migrated from one processor to another and use this instruction to force all the relevant architectural state (mapped physical registers) for the thread to be spilled into the data cache. Then, the standard coherence mechanism of the cache hierarchy would handle the coherence.

## 3.4 Rename Table Optimizations

The access pattern of the rename table is not random. The table will generally be accessed using memory addresses from a small number of contexts representing the active register windows of each thread. The maximum number of simultaneous entries is also fixed to the maximum number of different registers that can ever be renamed simultaneously. This is equal to the number of physical registers. These restrictions open up the possibility of optimizing both the organization of the rename table and the rename table tags.

### 3.4.1 Rename Table Organization

The proposed implementation for the rename table is a set associative organization with a tag and valid bit for each entry. Although this organization provides the greatest flexibility, it comes at the cost of a large rename table with complex logic. The fact that the maximum number of rename table entries is fixed to the number of physical registers means that a fully associative table sized to the number of physical registers is able to provide no capacity or conflict misses. However, a large fully associative table would be unlikely to meet cycle time limits. On the other end of the spectrum is a simple direct mapped table with valid bits for each entry, but only a single tag. The tag would represent the context that the

entire table supports. When the context is switched, all of the registers belonging to this context would need to be spilled to memory. Therefore, this design would be impractical in a register window implementation. However, if a table was provided for each thread, this would be a reasonable implementation in a simultaneous multithreading processor. The valid bits would allow single register granularity for caching in the physical register file. However, a more traditional implementation for context switches and interrupts, one that explicitly saves all the registers, would be required. Another theoretical possibility would be to provide a tag for some number of registers and a valid bit for each. While this would reduce the size of the table, it would still require tag checking for each access. It would also require more complex logic for determining when a tag can be replaced (all of the registers covered by this tag would need to be freed).

An intriguing possibility would be a combination of the two extremes. A mapping table is already required to hold the addresses for each physical register. Some additional information could be stored in the table, allowing us to regenerate the rename table based on the entries in the mapping table. One or more simple direct mapped tables would cache the rename information for current contexts. Thus, most accesses would simply use the direct mapped table and only need to check the valid bit. On a context change (for example a register window allocation or deallocation), the tables would be checked to find one holding this context. If one is not found, the mapping table would be used to rebuild the table. By providing more than one caching table, the fixup frequency could be lowered. The complexity of the logic needed to perform this fixup would need to be studied to determine if this idea is practical to implement.

### 3.4.2 Rename Table Tags

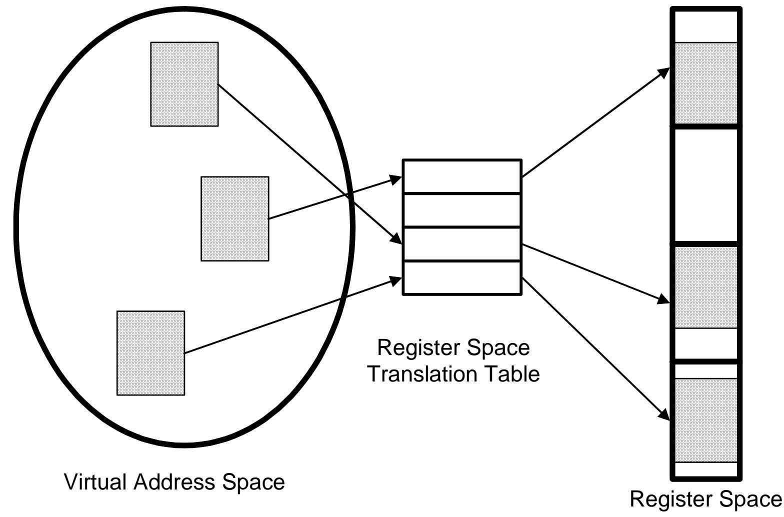
VCA requires a cache-like rename table with tags to map arbitrary addresses to physical registers. Adding a full address tag to each entry in the table significantly increases its size. Because the rename table will likely be replicated to implement the required number of ports, this size increase may be costly. We can take advantage of register address locality to drastically reduce the tag size with

only a modest increase in complexity. Although any memory address could be used as a base pointer, in practice only a relatively small number of addresses (i.e. a few function contexts from a small number of threads) will be used as base pointers within a given period of time. Register memory addresses also exhibit spatial locality around each base pointer address.

To exploit these characteristics, we introduce an additional translation table that maps the upper bits of each register memory address to a much smaller *register space identifier (RSID),* see Figure 3.5. The concatenation of the RSID and the remaining low-order memory address bits (the *register space offset*) are then used for the rename table lookup. Figure 3.6 illustrates this process. In the example shown there, the upper 48 bits of the 64-bit register memory address are mapped to a 4-bit RSID. As a result, the tag on each rename table entry is only 11 bits rather than the 55 bits required without RSIDs. In this case, the translation table could be implemented with a small 16-entry fully associative buffer where the RSID is simply the index of the matching entry.

If a register memory address does not find a match in the RSID table, it must allocate a new table entry, potentially replacing an existing valid table entry. In this (rare) situation, any physical registers using the current RSID must be flushed to memory before the RSID can be reused. In this unlikely event, a special instruction can be used to flush the physical registers to the data cache. The same instruction is used if a physical page that contains register values needs to be paged out by the virtual memory system, see Section 3.2.4. It may be desirable to associate reference counters with each RSID so that unused RSIDs can be identified and reused without requiring a flush operation. RSIDs can be managed in hardware (or low-level software, such as PAL code), and thus the very existence of RSIDs is hidden from the operating system and user-level code.

We can reduce the overhead of the translation process by caching the RSID associated with each base pointer, accessing the RSID table only when a base pointer is updated (due to a context switch or a register-window call or return). This optimization requires a modest alignment restriction on the base pointer so



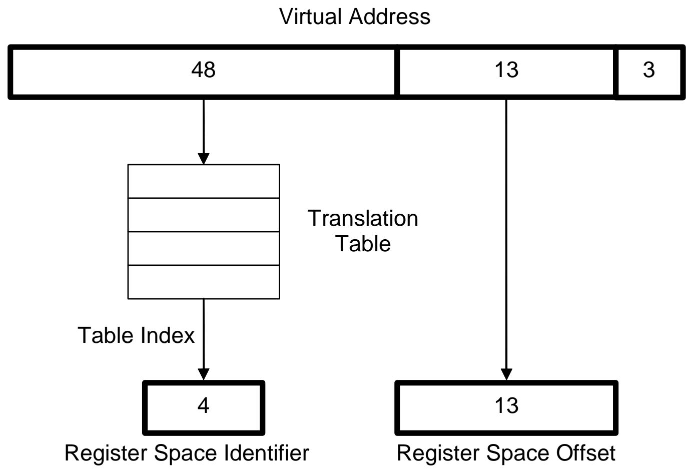




**Figure 3.5: Address Translation Scheme**

(a) The rename table will contain addresses from a small number of locations corresponding to the few active thread contexts for the small number of threads. A simple translation table can be used to map the large virtual addresses into the more compact register address space. (b) A 64 bit virtual address can be translated into a 17 bit register space address. The upper 48 bits of the virtual address are translated into a register space identifier using a small 16 entry table. The next higher 13 bits become the register space offset. This is a large enough offset for 182 register windows. The lower 3 bits are discarded because all the registers are 64 bit.

that a logical register offset cannot generate a memory address outside the range of the current RSID. A similar scheme could be used to cache the physical

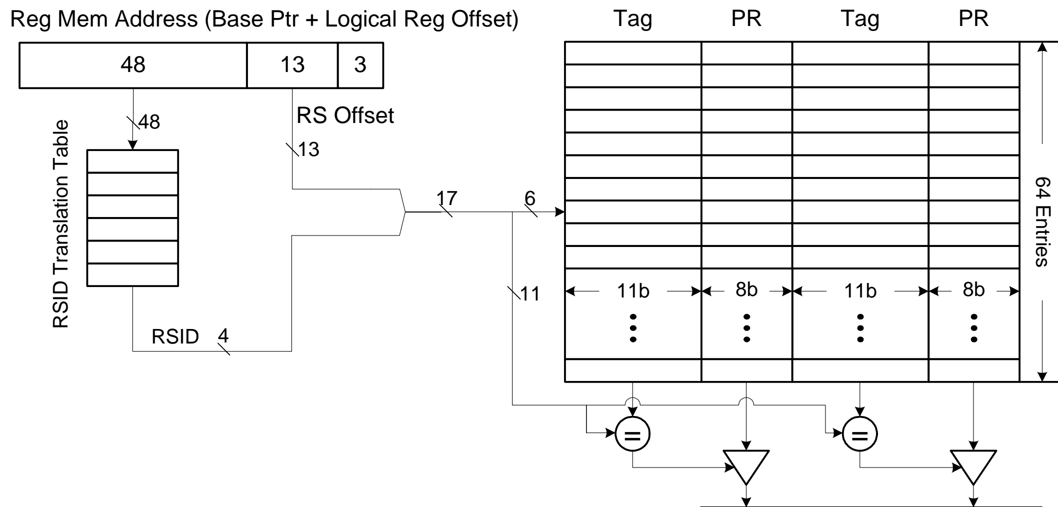



**Figure 3.6: Optimized rename process.**

The logical register index is added to the base pointer to produce a register memory address. The upper 48 bits of this address are translated to a 4 bit RSID and concatenated to the 13 remaining useful bits of the address. The resulting address is then looked up in the set-associative rename table. For a 256-entry register file and a 4-way associative table, the total size of the table is 4864 bits. This lookup logic must be replicated for each register being renamed.

addresses associated with each RSID, eliminating the need for TLB lookups (and the possibility of page faults) on spill and fill operations.

Note that the optimal configuration of the RSID scheme (both the number of RSIDs and the division of the register memory address bits between the RSID table lookup and the register space offset) will depend on the expected usage model. The number of RSIDs must be at least as large as the number of base pointers, and perhaps larger to enable caching of registers across context switches. For a system that does not support register windows, the register space offset need only be large enough to map a single logical register file, while a register-windowed machine would perform best with larger register spaces that can map the working set of an active register stack. The number of entries in the translation table will determine the number of simultaneous locations that can be translated. However, the larger the table, the greater the number of bits that need to be used in the RSID, thus requiring a corresponding increase in the size of the rename table tags. Similarly, the number of bits used to index the translation table

determines the size of a location mapped - the more bits used to index, the fewer bits left to form an offset. The determination of these sizes is dependent on the system and would need to be optimized for the ISA, ABI and expected workloads.

## 3.5 Summary of Pipeline Modifications

The virtual context architecture requires changes to three different stages in the pipeline: rename, issue and commit.

Most of the changes that the virtual context architecture requires are located in the rename stage of the pipeline. Figure 3.7 shows a summary of the new operation of the rename stage. The rename stage begins with a logical register identifier. This identifier is combined with the base register and index register to form an optimized address. This optimized address is used to index into the rename table. The rename table may require physical registers from the free list and/or available list. A physical register is needed for each destination register and for any source registers that require a fill. The rename table generates a physical register. In every case, this physical register is sent to the reorder buffer for use in instruction scheduling. The virtual context architecture also requires that the use of physical registers be tracked. Therefore, the physical register is used to index into the use counters, and the counter is incremented. If a spill or fill is generated, the rename stage requires additional processing. On a fill, the physical register and optimized address are sent directly to the architectural state transfer queue. On a spill, the physical register for the register being spilled is used to index into the mapping table. This lookup produces the optimized address for the physical registers. The physical register and optimized address are then sent to the ASTQ.

The issue stage of the pipeline also requires some modifications. Figure 3.8 shows a summary of the new operation of the issue stage. These changes are only necessary when spills and fills are implemented with an architectural state transfer queue. To implement the ASTQ, a priority mux is placed between the load/store queue (LSQ) and load/store function unit(s). The priority mux gives priority to instructions being issued from the load/store queue. If no instruction is

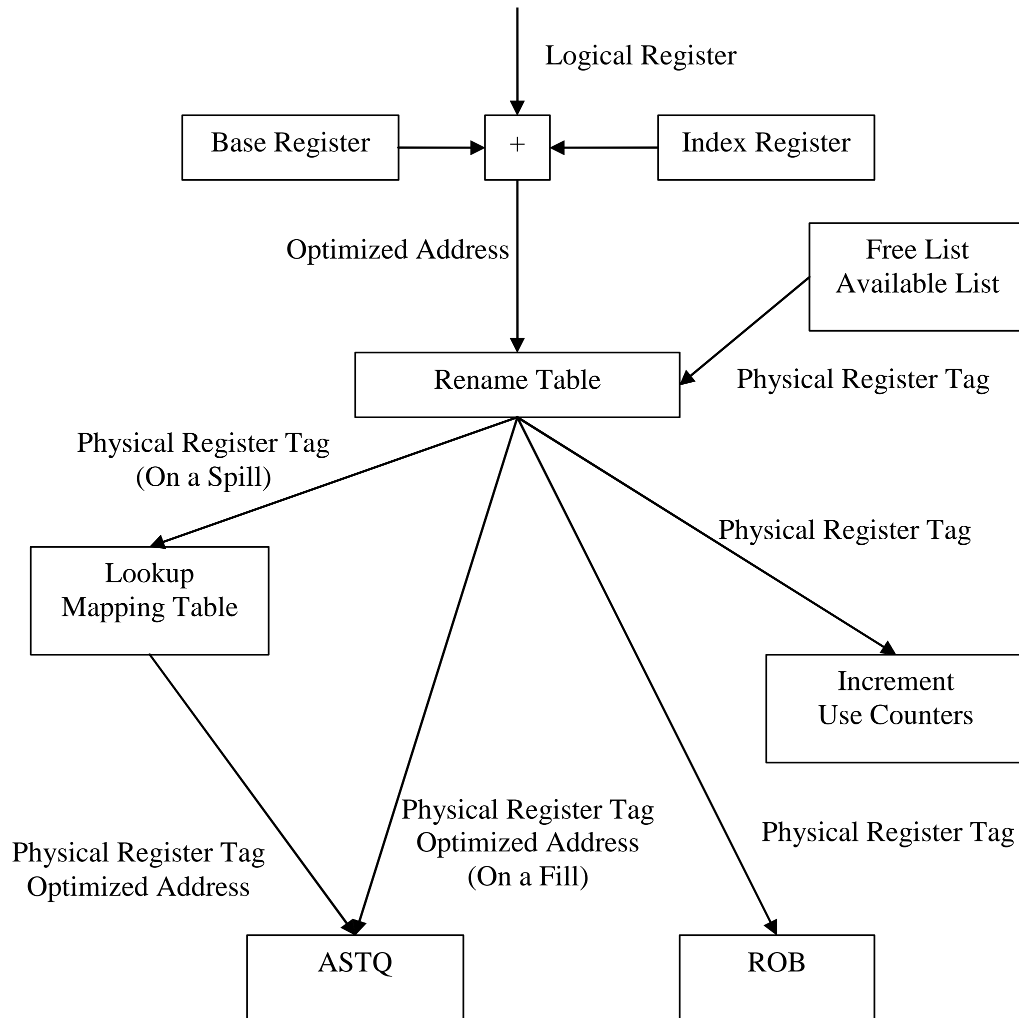



**Figure 3.7: Rename Stage Summary**
The virtual context architecture requires modifications to the rename stage.

being issued from the LSQ, the next operation is issued from the ASTQ. The operation is executed by the load/store function unit(s) and in almost the exact same way as a traditional load or store. Fills, in fact, work exactly like a traditional load. The address is sent to the first level of data cache. When the memory access is complete the result is broadcast on the bypass network to wake up any dependent instructions. A spill is handled almost exactly like a traditional store. Like a store, the memory address and value are placed into the store buffer. At some later time the store buffer will send the value to memory. Unlike a store, the spill

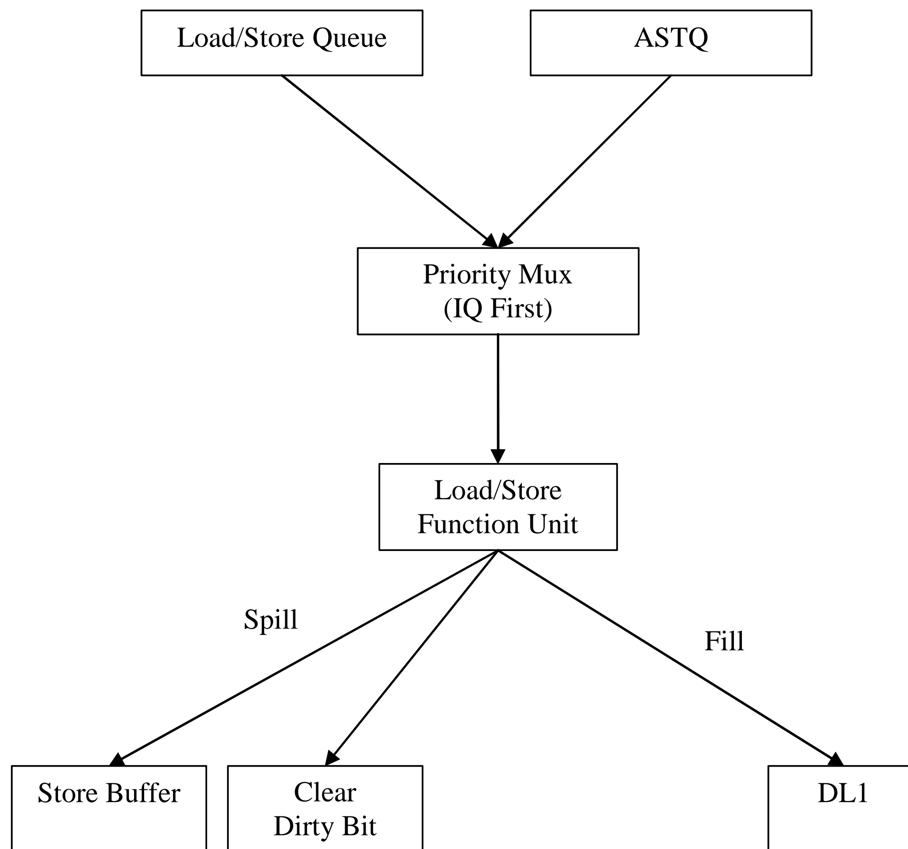



**Figure 3.8: Issue Stage Summary**
The virtual context architecture requires some minor modifications to the issue stage of the pipeline.

must also clear the dirty bit associated with the physical register being spilled. This signals the frontend that it may not replace this physical register freely.

The final stage that requires modification is the commit stage of the pipeline. Figure 3.9 shows a summary of the new operation of the commit stage. When an instruction is committed in the virtual context architecture it goes through a similar procedure as it does in rename. The logical registers from the ROB are converted into an optimized address by using a version of the base register and index register kept at commit. The optimized address is put into a copy of the rename table called the retirement map table. This table should be the exact configuration as the rename table. However, the logic associated with this table can be simplified by passing the way in the rename table where each logical register was renamed.

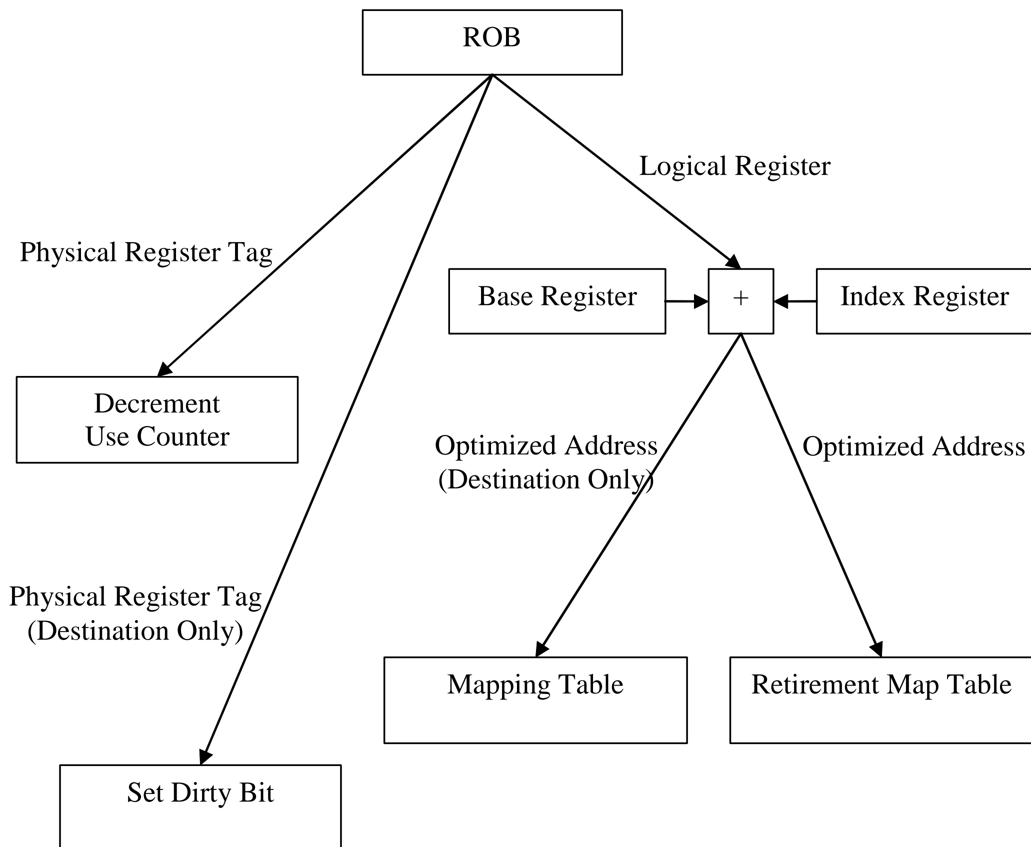



**Figure 3.9: Commit Stage Summary**
The virtual context architecture requires modifications to the commit stage of the pipeline.

Thus, the retirement map table can actually always be treated as a direct mapped table. The optimized addresses of destination registers are also put into the mapping table. This table keeps track of the memory address each physical register should be spilled to. The physical register tag associated with destination registers is also used to index into the dirty bit table, and set the physical register as dirty. There is one exception to this. In the case of fill instructions implemented using operations, the destination registers are not marked as dirty. Finally, all the physical registers are used to index into the use counter table and the uses for these registers is decremented.

# Chapter 4

# Experimental Methodology

This chapter describes the methodology used to measure the performance of the virtual context architecture. This chapter is composed of four sections: benchmarks, workloads, simulation and statistics. The benchmarks section describes which benchmarks were used and how the binaries were created. The workloads section discusses the techniques used to measure the performance of the benchmarks. The simulation section describes the simulator used and the machine models simulated. The statistics section describes the metrics used to measure the performance.

## 4.1 Benchmarks

The instruction set architecture (ISA) chosen for our experiments was the Alpha [31]. This represents a modern reduced instruction set computer that was a production machine and not just a research machine. The instruction set and application binary interface (ABI) were designed to be as efficient as possible while still being implementable. The ABI does not use register windows. Instead, the logical registers are divided into callee saved and caller saved registers. See Table 4.1 for a description of the Alpha registers. The first six function arguments are passed in registers, while any remaining are passed on the stack. The return value of the function is passed back in a register. The register allocation done by the compiler is optimized in terms of callee saved and caller saved assignments to minimize the save and restore instructions needed.

To test the capabilities of the virtual context architecture the application binary interface needed to be modified to support register windows. By carefully design-

| Register | Description | Windowed |
|---|---|---|
| $0 | Function Return | no |
| $1 - $8 | Temp, caller saved | yes |
| $9 - $14 | Temp, callee saved | yes |
| $15 | Frame pointer | yes |
| $16 - $21 | Function Arguments | no |
| $22 - $25 | Temp, caller saved | yes |
| $26 | Return Address | yes |
| $27 | Procedure Value | no |
| $28 | AT | yes |

| Register | Description | Windowed |
|---|---|---|
| $29 | Global Pointer | no |
| $30 | Stack Pointer | no |
| $31 | Zero | - |
| $f0 | Function Return | no |
| $f1 - $f9 | Temp, callee saved | yes |
| $f10 - $f15 | Temp, caller saved | yes |
| $f16 - $f21 | Function Arguments | no |
| $f22 - $f30 | Temp, caller saved | yes |
| $f31 | Zero | - |

**Table 4.1: Register window ABI**

The base Alpha application binary interface was modified to include the concept of register windows. Any register used to communicate values across a function (with the exception of the return address register) is non windowed. All other registers are treated as windowed. Each function invocation has access to a complete set of windowed registers.

ing the ABI, binary compatibility could be maintained with the conventional ABI. This greatly simplified the simulation and allowed the baseline binaries to run on the VCA. A non-overlapped register window implementation was chosen, although the VCA would also work with an overlapped interface. Our register window interface divided the registers into two types: windowed and non-windowed. The windowed/non-windowed assignments are summarized in Table 4.1.

Any register used to communicate values across a function call (in either direction) is treated as non-windowed. The mapping address for non-windowed registers does not change on function calls or returns. Therefore, these registers represent a global set of registers that are shared by all the function contexts for a given thread. However, the shared nature of these registers does require that the compiler manage any necessary saving and restoring.

All other registers are treated as windowed. Unlike the non-windowed registers, the mapping address of these registers changes on function calls or returns. Specifically, the mapping address is incremented by a fixed amount on function

calls and decremented by the same amount on function returns. In this case, the fixed amount is calculated as 45 windowed registers scaled by 8 bytes per register or 360 bytes. This ensures that each new function context has a completely independent set of windowed registers.

For simplicity, the call and return instructions were chosen to allocate and deallocate register windows. A special instruction could be used instead, to give the compiler the option of not allocating a new register window on a call, but this was beyond the scope of our research. The call instruction allocates a new register window for the next instruction. This saves the return address in the calling functions register window set. The return instruction deallocates the register window before it executes. Therefore, all of its registers are referenced from the original calling functions register window set (which is where the return address is stored). This was done so that no instruction uses registers from more than one register window set, thereby simplifying the decoding of an instruction.

The GNU compiler suite (gcc 3.3.3) [12] was chosen to compile the benchmarks because it is available in source code and supports cross compiling. The source code was necessary because the compiler needed to be modified to support the new ABI. This compiler modification was implemented by providing a compiler switch which would suppress the output of any save and restore instructions for the windowed registers. The GNU standard C library (glibc 2.3.2) was recompiled to support the new register window ABI.

To evaluate the architecture, the SPEC CPU2000 [15] benchmarks were used (except for four Fortran 90 benchmarks the GNU compiler suite could not compile). The benchmarks were compiled at -O3 optimization, which includes function inlining. Each benchmark was compiled and linked twice: once with the standard compiler and library, and once with the modified compiler and recompiled library.

## 4.2 Workloads

It is not practical to do full runs of the entire benchmark with all inputs. A detailed simulation of a full run of a single benchmark could take up to several

monthes to finish. Therefore, a more efficient way was needed to get good estimates of the overall performance while only running a small fraction of the instructions. Towards this end, a single representative input from the reference input set was chosen for each benchmark. In the case of benchmarks with more than one input, the input closest to the average IPC of all the inputs was selected.

To achieve practical simulation times, the common technique of SimPoints [29] was used. A SimPoint is one or more fixed length intervals of the dynamic execution of a program that are representative of the entire program run. The SimPoints of a program are first simulated in parallel. The statistics for the simulations are then later combined using weighting factors to estimate the statistics of the entire program. For our simulations we chose an interval size of 100 millions instructions.

The SimPoints for the baseline binaries were generated normally, using the basic block traces found by performing a fast functional simulation of the reference input sets. The SimPoints for the register window binaries were generated by finding the equivalent location to the SimPoints in the baseline binary. This was done by counting the conditional control instructions executed to reach the start of the SimPoint interval. Conditional control instructions were used for two reasons. First, the frequency of conditional control instructions in the benchmarks is relatively high; one occurs on average every 13 instructions. Second, the number of conditional control instructions is nearly equal between the windowed and baseline binaries. On average the difference is within 0.004% with the largest difference 0.03%.

Two sets of SimPoints were generated. For the first set, up to 10 SimPoints were generated for each of the benchmarks. This set allowed for a very accurate estimation of the performance at the expense of large numbers of simulations. For the second set, a single SimPoint was generated for each benchmark. This set was used when running multiple SimPoints for each benchmark was not practical.

The final step was selecting the workloads to use for the register window experiments and for the simultaneous multithreading experiments.

### 4.2.1 Register Windows

Register windows are used to reduce the save and restore overhead necessary for a function call. The benefits of register windows are only realized when there is a high frequency of function calls, therefore, we only gathered results for those benchmarks that made a function call at least once every 500 instructions, see Table 4.3. Both the 10 SimPoint and single SimPoint sets were used in these experiments. Depending on the number of configurations being tested, a tradeoff was made between the accuracy achieved using 10 SimPoints and the speed of using a single SimPoint. The more accurate 10 SimPoint runs were always used when comparing the virtual context architecture to a baseline.

### 4.2.2 Simultaneous Multithreading

Simultaneous multithreading requires that workloads be comprised of multiple benchmarks. In our experiments, we wanted to use two-thread and four-thread workloads. For our purposes, the workloads are composed of benchmarks from the SPEC CPU2000 benchmarks. Unlike the register window experiments, we did not limit the benchmarks to those with a high frequency of function calls. Instead, we wanted to evaluate the VCA in as diverse a multithreaded environment as possible. It was not practical to run the full cross product of benchmarks. This would result in too many simulations. Therefore, a scheme was needed to choose a representative set of two-thread workloads and a prepresentative set of four-thread workloads.

We used a scheme similar to that of Raasch [26] to generate representative two thread workloads. First, each of the 253 possible two-thread workloads was run using the baseline architecture. Next, we chose a set of statistics to characterize the benchmarks. The statistics fall into two groups. The first group characterizes the benchmark's absolute single-thread behavior, and includes: 1) data cache accesses; 2) floating point register usage; 3) total number of unique addresses to which registers are mapped; 4) ratio of dynamic instruction count between register window binary and baseline binary; and 5) the ratio of data cache accesses between the register window binary and the baseline binary. The

second group represents the relative performance of a thread in an SMT workload, and is composed of: 1) committed IPC; 2) fetch rate; 3) issue rate; 4) branch predictor accuracy; 5) instruction queue occupancy; 6) instruction queue ready rate; 7) reorder buffer occupancy; 8) data cache miss rate; and 9) second level cache miss rate. These statistics are normalized to the same benchmark's single thread values.

The statistics are all scaled to a mean of zero and variance of one. Principle components analysis [11] is then used to eliminate correlation. Enough components are used to cover more than 99% of the variance (in our case, 11 components were needed). The workloads are then mapped into 22-dimensional space and a linkage clustering algorithm is used to cluster the workloads. The Bayesian Information Criterion (BIC) score is calculated for each cluster assignment. The smallest number of clusters that has a BIC score within 2% of the maximum BIC score is used. This resulted in 43 two thread workloads.

To generate the four-thread workloads the scheme was repeated. However, instead of using the full set of possible four-thread workloads, the cross product of the two-thread workloads was used to provide a starting set of 919 possible four-thread workloads. After applying the scheme the result was a set of 127 four-thread workloads.

## 4.3 Simulation

The architecture was simulated using the M5 simulator [3]. M5 is a detailed execution driven simulator. It supports a realistic out-of-order pipeline with a separate instruction queue and reorder buffer. The memory model is an event driven model which accurately times memory accesses. It simulates not only multiple levels in the cache hierarchy, but the busses that interconnect them. Memory instructions are split into two separate micro operations. The first operation is placed into the instruction queue and calculates the memory address. The second operation is placed in the load store queue and is responsible for the actual memory accesses (either a read for loads or a write for stores). The load store queue is

| Pipeline | Issue and Operation Latencies |
|---|---|
| 4 wide | Int ALU: 1 issue, 1 operation |
| Fetch queue: 32 entries | Int Multiply: 1 issue, 3 operation |
| Instruction queue: 128 entries | Int Division: 19 issue, 20 operation |
| Reorder buffer: 192 entries | FP Add: 1 issue, 2 operation |
| Load/Store queue: 64 entries | FP Compare: 1 issue, 2 operation |
| Store buffer: 32 entries | FP Convert: 1 issue, 2 operation |
| ASTQ: 4 entry, 2 ports | FP Multiply: 1 issue, 4 operation |
| Rename: 64x4 | FP Divide: 12 issue, 12 operation |
| Hybrid branch predictor | FP Sqrt: 24 issue, 24 operation |
| Retirement table branch recovery | |

| Caches | Function Units |
|---|---|
| 2 read/write ports | Int ALU: 4 |
| DL1: 64K 4-way 3 cycle hit | Int Mul/Div: 2 |
| IL1: 64K 4-way 1 cycle hit | FP Add/Comp/Conv: 2 |
| L2: 1M 4-way 15 cycle hit | FP Mul/Div/Sqrt: 2 |
| Memory: 250 cycle | |

**Table 4.2: Four Issue Processor Description**

The processor description is composed of four parts: pipeline, cache, latencies (issue and operation) and function units. The pipeline description describes the size and type of the various structures in the processor pipeline. The cache section describes the size, associativity and latencies of the caches and memory. The latencies section provides the issue and operation latencies of the various types of integer (Int) and floating point (FP) operations. The issue latency is the number of cycles that the function unit is busy when performing this operation. An issue latency of one means that operations can be fully pipelined. The operation latency is the number of cycles until the result is ready. The number of function units of each type is specified in the function units section.

responsible for doing memory disambiguation. Our experiments assumed a normal memory disambiguation policy: a load is not allowed to issue until all earlier stores have a known address, or until an earlier store with the same address as the load occurs later than any stores with unknown addresses (in which case the value of the store can be bypassed to the load). The simulator was modified to support the virtual context architecture. Each simulation was allowed to warm-up for 5 million instructions. The simulation was then run until a thread reached 100 million executed instructions.

Several processor models were used. The main processor model was a modern four issue processor with a separate instruction queue and reorder buffer (see Table 4.2). To test the performance of the virtual context architecture in a wider pipeline, an eight issue version of the pipeline is also used. This pipeline is identi-

cal to the four issue version except the width and number of function units has been doubled. We also explored the performance of the virutal context architecture on a narrower two issue pipeline. The two issue pipeline has half the number of function units as the four issue. We also decreased by half the sizes of the instruction queue, reorder buffer, load store queue, and store buffer. Finally, we decided to evaluate the virtual context architecture in a simpler pipeline more suited to embedded applications. To this end, the final pipeline examined is a one issue in order pipeline. For this pipeline, we assume a single cycle hit latency for the first level data cache.

## 4.4 Statistics

A direct comparison of instructions per clock (a usual measure of performance) is not possible between windowed binaries and non windowed binaries. The binaries have different dynamic path lengths and therefore this metric is meaningless. Instead of comparing rates, the total must be compared. Therefore, the basic performance comparison used is total execution time. The totals are calculated by multiplying a rate statistic (such as cycle per instruction) by the total number of instructions needed to execute the benchmark. The rate statistic is gathered by doing a detailed simulation of the SimPoints. The total instructions needed to execute the benchmark were found by using a fast functional simulator to simulate the complete benchmark. Table 4.3 contains the path length ratio between the register window binaries and the baseline binaries.

### 4.4.1 Register Windows

For each experiment, we calculated the execution time and the number of 1st level data cache accesses. Execution time was calculated by multiplying the average committed CPI (cycles per instruction) of the benchmark (from detailed simulation of SimPoints) by the number of dynamic instructions needed to execute the complete benchmark (from a fast functional simulation of the entire benchmark). The cache accesses are calculated similarly, by multiplying the rate at which the

| SpecInt Benchmark | Ratio | SpecFloat Benchmark | Ratio |
|---|---|---|---|
| bzip2_graphic | 0.92 | ammp | 0.98 |
| crafty | 0.93 | equake | 0.94 |
| eon_rushmeier | 0.94 | mesa | 0.92 |
| gap | 0.91 | wupwise | 0.93 |
| gcc_expr | 0.92 | | |
| gzip_graphic | 0.92 | | |
| parser | 0.92 | | |
| perlbmk_535 | 0.85 | | |
| twolf | 0.99 | | |
| vortex_2 | 0.82 | | |
| vpr_route | 0.90 | | |

**Table 4.3: Path Length Ratio**

The ratio of the number of dynamic instructions required to execute the full benchmark for the register window binaries to the number of dynamci instructions required to execute the baseline binaries.

cache is accessed (measured as average accesses per committed instruction) by the total number of dynamic instructions.

Most of the results are normalized against a baseline configuration before averaging. This is done to ensure an equal contribution to the average by each benchmark. If the averaging was done before normalization, the value of the stat in question (for example CPI) would influence the results. For example, suppose we are measuring two benchmarks (one and two) under two different configurations (A and B). Benchmark one has a CPI of 100 for configuration A and a CPI of 110 for configuration B. Benchmark two has a CPI of 10 for configuration A and a CPI of 20 for configuration B. If we average first, then normalize to configuration A, we get an average increase in CPI of 18%. In this case, because the CPI of benchmark one is so much greater than benchmark two, benchmark one's CPI difference becomes much more important than benchmark two's. However, if we

normalize first to configuration A, then average, we get an average increase in CPI of 55%. Now, both benchmarks are equally weighted when determining the average CPI change.

### 4.4.2 Simultaneous Multithreading

Measuring the performance of a multithreaded workload is more complicated than measuring the performance of a single threaded workload. The most important performance characteristic of a simultaneous multithreaded architecture is the performance improvement achieved by allowing more than one thread to execute at the same time. If there is no speedup associated with running more then one thread, than the extra hardware and design complexity needed to support more than one thread is wasted. Previous work [28, 33] has used weighted speedup to measure the performance of a simultaneous multithreaded core. However, this metric is based on using instruction per clock (IPC) statistics. As discussed in Section 4.4, IPC is meaningless when comparing binaries with different dynamic path lengths. Therefore, we have adapted the idea behind weighted speedup, but applied it to different starting statistics.

The first measurement is calculated based on total execution time. We call this weighted execution time. This metric is calculated by summing the relative execution time of all threads—the execution time of each thread in the SMT workload, divided by the execution time of the same benchmark running as a single thread. The execution time was calculated by multiplying the cycles per instruction (CPI) by the dynamic path length of the benchmark. This measurement can be used to calculate the speedup associated with running multiple threads. However, since it is based on total execution time it can be used to compare workloads with different dynamic path lengths.

The second measurement is calculated based on total data cache accesses. We can this weighted cache accesses. It is calculated similar to weighted speedup, but using data cache accesses per instruction instead of execution time. This measurement reveals the overhead, in terms of data cache accesses, that running multiple threads simultaneously costs.

In the following four chapters, the methodologies described here will be used to evaluate the virtual context architecture.

# Chapter 5

# Parameter Studies

This chapter studies the effects that the various implementation parameters have on the performance of the virtual context architecture. The virtual context architecture has a large number of parameters that can be used to optimize performance. These parameters fall into two main groups. The first group is the implementation of spills and fills. The second group is the rename stage configuration. The chapter ends with a summary of the findings.

## 5.1 Spill/Fill Implementation

The implementation of spills and fills is governed by two different parameters. The first parameter is the register replacement policy used to choose which physical register to spill. The second parameter is the method used to implement spills and fills.

### 5.1.1 Register Replacement Policy

One of the critical parameters affecting the performance of the virtual context architecture is the physical register replacement policy. If the VCA does not have any free registers to use for rename, it must use a register replacement policy to choose an available register (dirty or not) to replace, see Figure 3.1. There are two costs associated with replacing an available physical register. First, if the physical register is dirty, the value must be spilled to memory before it can be replaced. This requires not only a memory address, but the pipeline must stall until it is safe for the physical register to be overwritten, see Section 3.2.2. The second cost is that the replaced register may be sourced by a later instruction.

This would result in a fill operation to bring the replace value back into the physical register file. The fill not only requires a memory access, but the instruction sourcing the value becomes dependent on the fill operation, possibly delaying its execution. Four different physical register replacement policies were investigated in order to determine which would be most effective for use in the virtual context architecture: least recently used, not dirty first, overwrite first, overwrite last.

The first policy is the least recently used policy (lru). This policy ignores the dirty status of a physical register and simply bases the selection on the time of last use. It attempts to always keep the more recently used logical registers in the physical register file. This policy is very common in computer architecture and is used in many places, including cache replacements. The effectiveness of the policy is dependent on temporal locality of the logical registers. A number of earlier studies have shown that logical registers exhibit high temporal locality [22, 25,34, 39].

The second policy studied is the not dirty first policy. The policy is to always favor spilling a non dirty available physical register over a dirty one. Thus, the policy treats the physical registers as three separate sets in terms of use by the rename stage: free, available, dirty available. The rename stage always uses a free register over an available register, and always uses an available register over a dirty available one. Within the set of non dirty and dirty physical registers the least recently used policy is used. This policy attempts to minimize the number of spills generated by only spilling as a last resort. However, this comes at the cost of potentially increasing the number of fill operations.

The overwrite last and overwrite first policies are slight modifications to the lru policy. They behave like the least recently used policy except in the case that the logical register a physical register is mapped to is possibly going to be overwritten. The possibility exists when an instruction enters the pipeline (after rename for our purposes) that writes to the same logical register. In this special case, these policies either move the physical register to the end of the list (overwrite last) or to the front of the list (overwrite first). By moving the physical register to the end of the list, the overwrite last policy attempts to minimize the number of unnecessary

| | 64 | 128 | 192 |
|---|---|---|---|
| lru | 1.000 | 1.000 | 1.000 |
| not dirty first | 0.998 | 1.003 | 1.004 |
| overwrite first | 1.212 | 1.093 | 1.029 |
| overwrite last | 0.996 | 1.000 | 1.002 |

**Table 5.1: Replacement Policy Execution Time Comparison**
The execution time of the various physical register replacement policies. The results are normalized to the execution time of the least recently used policy. The columns show the results for three different physical register file sizes.

spills (the physical register will become free as soon as the overwriting instruction commits). The overwrite first policy favors generating the unnecessary spill over spilling a register that may need to be reloaded later. In this case, as long as the overwriting instruction commits, the value will never be needed again.

In order to determine which policy yields the best performance, the full set of benchmarks were run using each of the policies at a variety of physical register file sizes. The execution time of the various policies are shown in Table 5.1. Overwrite last, lru and non dirty yield about the same execution time with overwrite last being slightly better (0.25% better than non dirty and 0.05% better than lru) on average. Overwrite first is slower than the others in all cases. Compared to the other policies the execution time of the overwrite first policy increases dramatically as the number of physical registers is decreased. At 192 physical registers it is within 3% of the other policies, at 128 physical registers it slows to 9% of the others and at 64 physical registers the benchmarks take over 21% longer to execute with this policy.

The memory characteristics of the various policies are shown in Table 5.2. The performance of the various policies can be explained by examining the data cache accesses. In particular, the execution time tracks the number of data cache accesses. The least recently used, non dirty and overwrite last policies all generate approximately the same memory traffic, while overwrite first generates more. This can be explained by examining the total number of transfers generated. On

Data Cache Accesses

| | 64 | 128 | 192 |
|---|---|---|---|
| lru | 1.000 | 1.000 | 1.000 |
| not dirty first | 1.003 | 1.008 | 1.005 |
| overwrite first | 1.217 | 1.105 | 1.061 |
| overwrite last | 0.998 | 1.001 | 1.000 |

Total Transfers

| | 64 | 128 | 192 |
|---|---|---|---|
| lru | 1.000 | 1.000 | 1.000 |
| not dirty first | 1.005 | 1.039 | 1.069 |
| overwrite first | 1.853 | 1.993 | 2.432 |
| overwrite last | 0.994 | 0.996 | 0.996 |

Fills

| | 64 | 128 | 192 |
|---|---|---|---|
| lru | 1.000 | 1.000 | 1.000 |
| not dirty first | 1.168 | 1.310 | 1.372 |
| overwrite first | 0.463 | 0.558 | 0.587 |
| overwrite last | 1.188 | 1.050 | 1.046 |

Spills

| | 64 | 128 | 192 |
|---|---|---|---|
| lru | 1.000 | 1.000 | 1.000 |
| not dirty first | 0.967 | 0.985 | 0.948 |
| overwrite first | 2.187 | 2.327 | 3.177 |
| overwrite last | 0.965 | 0.985 | 0.980 |

**Table 5.2: Replacement Policy Memory Comparison**

The memory characteristics of the various physical register replacement policies. The results are normalized to the least recently used policy. The columns show the results for three different physical register file sizes. The data cache accesses table shows the total number of read and write accesses on the first level data cache. The total transfers table shows the total number of spills and fills generated. The fills table shows the total number of fills generated by each policy while the spills table shows the total number of spills generated by each policy.

average, overwrite last generates the fewest number of transfers, while lru generates 0.5% more and not dirty results in nearly 4% more. Overwrite first generates more than twice as many transfers as the other policies. In fact, a strict order

exists at each physical register file size in terms of number generated. Overwrite last always generates the fewest transfers. Least recently used generates slightly more. Not dirty generates slightly more than lru. Finally, in all cases overwrite first generates around two times as many transfers as the other policies.

Overwrite last remains relatively constant with respect to least recently used. In contrast, the relative increase in transfers for both not dirty and overwrite first decrease as the number of physical registers is decreased. This can probably be explained by noting that the pressure on the physical register file increases dramatically as the number of physical registers is decreased. Therefore, when the physical register file is relatively small, the pool of available registers (both dirty and not) is going to be small and the rename stage will constantly be replacing them to free up more registers for renaming. Therefore, the not dirty first policy will start to behave more like the other policies. Although the relative number of transfers generated by the overwrite first policy decreases, it is still very large compared to the other policies. The overwrite first policy preferentially replaces those registers that are more than likely going to become free in a few cycles. When the spill has completed, the rename stage has a free register it can use. However, if it spilled a different register, once the spill is completed, the rename stage is likely to have two registers free, one that was just spilled and one that was just overwritten.

If you look at the spills and fills separately, the policies did act as expected in comparison to the least recently used policy. Not dirty generated fewer numbers of spills, about 3% fewer than least recently used. However, it generated between 17% and 37% more fills than lru. This shows that it is likely some registers are being used for a long time without being changed. The not dirty policy would tend to keep replacing these registers because they are not dirty, but would have to keep generating fills to bring them back in memory. This type of access could be seen for example in the frame pointer. It would tend to change only at the start of a function and when a function returns. However, it would tend to be accessed throughout the whole function.

By favoring the spilling of registers that were not likely to be used again, overwrite first did generate the fewest number of fills, between 41% and 53% fewer than least recently used. However, the decease in fills came at the cost of a huge increase in the number of spills, between two and three times the number generated by lru. As explained previously in this section, this policy suffers because it is very likely to be spilling registers that would be overwritten shortly. Once the register was overwritten, this would have freed that physical register. The other policies in contrast would tend not to be spilling these registers, and thus they would be freed without a spill.

The overwrite last policy performs the best overall, generating the fewest number of total transfers in all cases. By reducing the number of unnecessary spills it was able to decrease the total number of spills by between 2% to 4% in comparison to least recently used. However, this policy did generate more fills than least recently used, from 5% at 192 physical registers to nearly 19% at 64 physical registers. The increased fill traffic was more than compensated for by the decreased spill traffic.

With our particular implementation, the overwrite last policy provides the best performance, both in terms of execution time and transfers generated. Therefore this policy will be used in all of the following studies. However, the least recently used and not dirty first policies provided almost identical performance. If in a given implementation the costs of fills versus spills is drastically different one of the other policies might perform better. If the cost of fills was drastically greater, this would tend to favor the least recently used policy since it generated fewer fills. In contrast, if the cost of spills was drastically greater, this would tend to favor the not dirty first policy, which generated the fewest number of spills.

### 5.1.2 Method Used To Implement Spills and Fills

The implementation of register spills and fills requires the insertion of the equivalent of loads (fills) and stores (spills) into the pipeline. As discussed in Section 3.2.2, two possible methods could be used to accomplish the insertion. The first method is to add a small queue to the pipeline that we call the architec-

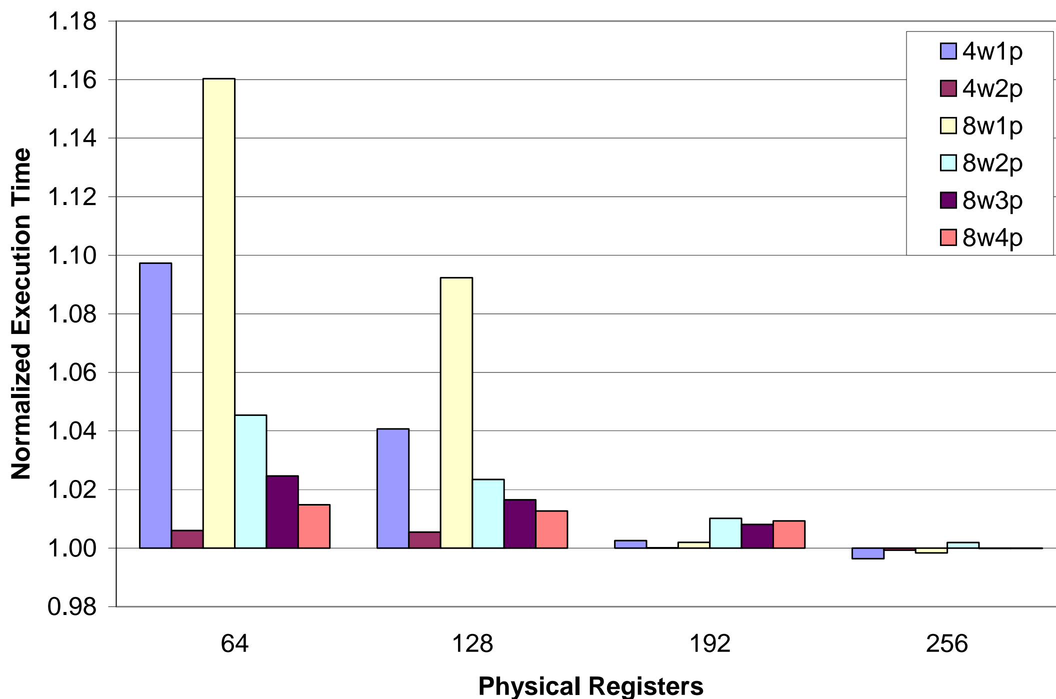



**Figure 5.1: Operation Execution Time**

The execution time of the benchmarks where both spill and fill are implemented by inserting operations into the pipeline. The execution time is normalized to the execution time when an ASTQ is used to implement both. The results are shown for a variety of pipeline configurations. The pipeline configurations are expressed in the form XwYp, where X is the issue width of the pipeline and Y is the number of read/write ports on the data cache.

tural state transfer queue (ASTQ). The second method is to insert the operations directly into the pipeline. It would also be possible to combine the methods. For example, the fills could be implemented by inserting operations directly into the pipeline while spills could be handled by an ASTQ.

Figure 5.1 shows the execution time of the workloads when both spills and fills are implemented by inserting operations into the pipeline. The results are normalized to the execution time when an ASTQ is used to implement both. In the vast majority of cases, the ASTQ is able to provide better performance than using operations. The relative performance is very dependent on two factors: the number of physical registers and the number of data cache ports.

The results clearly show a strong correlation between the size of the physical register file and the relative performance of using operations versus a queue. With

a large physical register file, the two techniques perform almost identically. On average, across all the pipeline configurations, with 256 physical registers the operations perform almost identically to the queue. With 192 physical registers, using operations results in a 0.5% increase in execution time on average, and a worse case of only 1%. However, the performance gap increases dramatically as the number of physical registers is reduced. With 128, using operations is 3% slower on average with a worst case of 9% and with only 64 physical registers, the average climbs to almost 6% with a worst case of 16%. With 192 or more physical registers, the rename stage does not to generate very many spills or fills. Therefore, the effect of the method used to implement will be relatively small. With fewer physical registers, the generation of spills and fills starts to climb drastically. At this point the method used to implement them becomes critical.

The connection between the number of physical registers and the relative performance of the ASTQ versus operations is clear. However, the results also showed a very strong connection to the number of data cache ports. In particular, the performance difference increases dramatically as the number of data cache ports is decreased. This trend can be explained by looking at the relative number of data cache accesses, see Figure 5.2. With 128 or fewer physical registers using operations not only results in an increase in execution time, but an even larger increase in data cache accesses. The results are nearly identical to the execution time. Obviously, an increase in data cache traffic is going to have a much greater effect on a pipeline with fewer data cache ports. It is possible for the virtual context architecture to enter a vicious circle with a small number of physical registers. As the register pressure increases, the rename stage will generate more spills and fills. This will in turn slow down the pipeline, which can lead to more instructions in the reorder buffer, and therefore an increase in register pressure. The net result is that the pipeline is flooded with spills and fills. The architectural state transfer queue is able to break this cycle because of its scheduling. In the ASTQ implementation priority is always given to instructions in the pipeline.

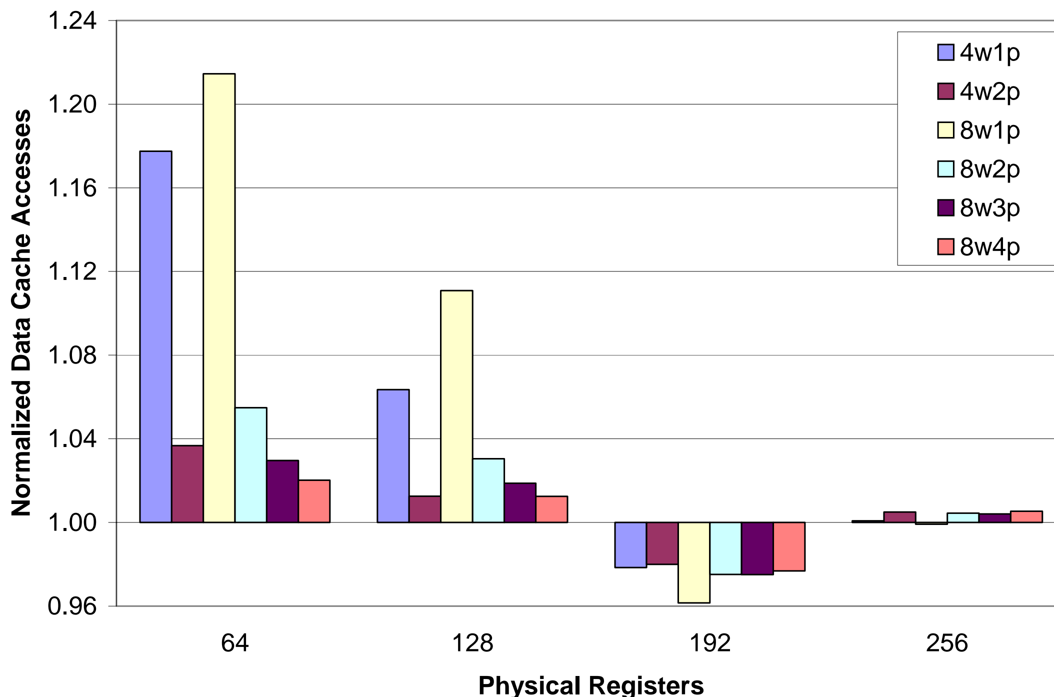



**Figure 5.2: Operation Data Cache Accesses**

The data cache accesses of the benchmarks where both spill and fill are implemented by inserting operations into the pipeline. The data cache accesses are normalized to the data cache accesses when an ASTQ is used to implement both. The results are shown for a variety of pipeline configurations. The pipeline configurations are expressed in the form XwYp, where X is the issue width of the pipeline and Y is the number of read/write ports on the data cache.

Therefore, spills and fills must wait until there are no other instructions to issue before they are allowed to execute. This helps to minimize the problem cycle.

It is possible to combine the two implementations. Figure 5.3 shows the execution time of the workloads with a mixed implementation. In this case, spills are implemented using the architectural state transfer queue while fills are implemented by inserting operations directly into the pipeline. This is the more natural implementation because the more numerous spills are still moved out of the load/store queue while fill which requires access to the bypass network is handled by an operation. The results are normalized to the execution time when an ASTQ is used to implement both. The mixed implementation performs between the two pure implementations. The trend of the relative performance is similar to the operation only implementation, but the magnitude of the performance differences is

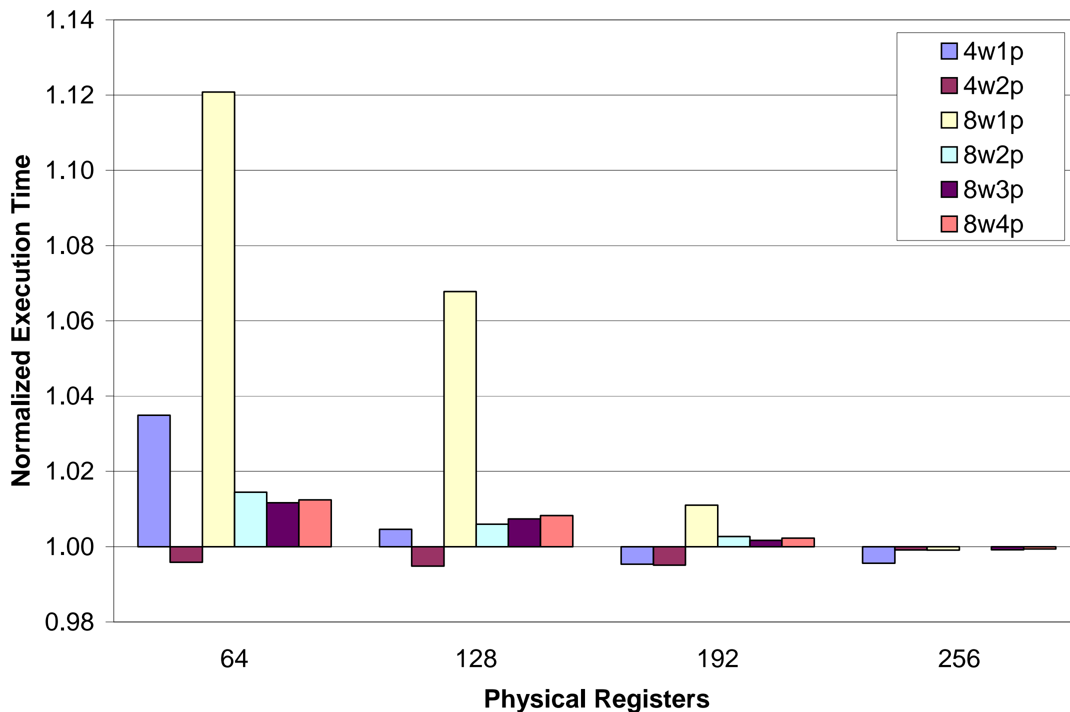



**Figure 5.3: ASTQ Spill / Operation Fill Execution Time**
The execution time of the benchmarks where spills are implemented by the architectural state transfer queue and fills are implemented by inserting operations into the pipeline. The execution time is normalized to the execution time when an ASTQ is used to implement both. The results are shown for a variety of pipeline configurations. The pipeline configurations are expressed in the form XwYp, where X is the issue width of the pipeline and Y is the number of read/write ports on the data cache.

less. Although it is possible to do a mixed implementation it is probably not worthwhile. It suffers the performance penalty of the operation implementation, and also the cost associated with implementing an architectural state transfer queue.

Both methods for implementing spills and fills have parameters associated with them. In the following two subsections we discuss and evaluate these parameters.

#### 5.1.2.1 Architectural State Transfer Queue Parameters

There are two parameters associated with the architectural state transfer queue: ports and size. The ports on the queue specify the maximum number of operations that can be added each cycle. The more ports the greater the complexity of the rename logic. The size of the queue limits the number of operations

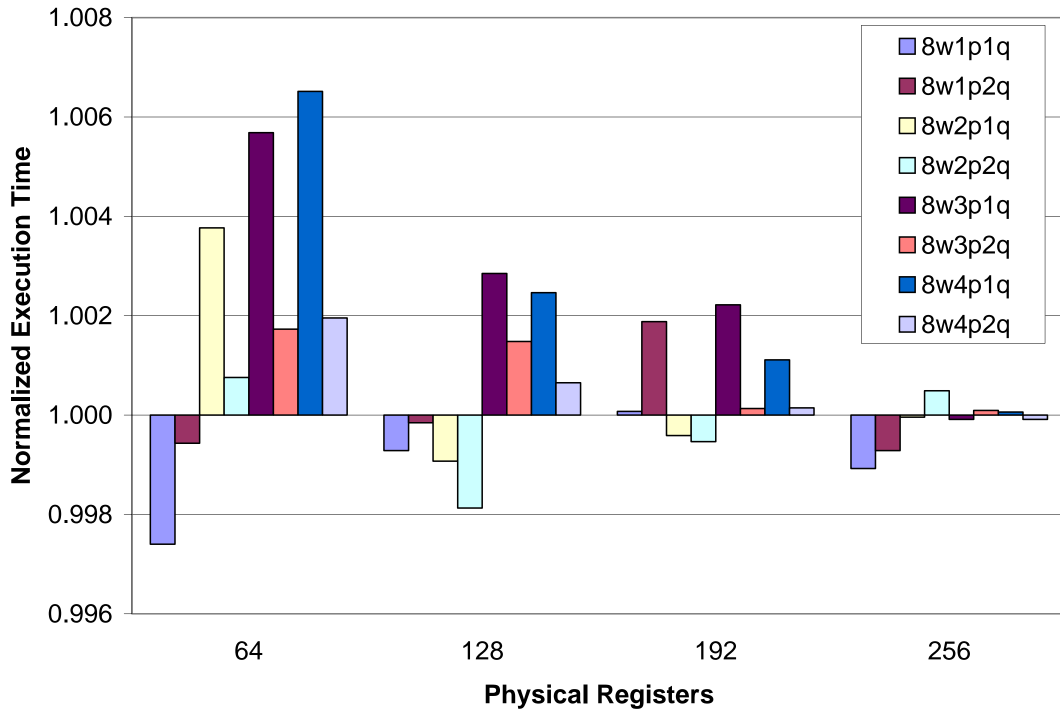



**Figure 5.4: Architectural State Transfer Queue Ports**

The execution time of the benchmarks with an ASTQ with varying ports. The execution time is normalized to the execution time with a four ported ASTQ. The results are shown for a variety of pipeline configurations. The pipeline configurations are expressed in the form XwYpZq, where X is the issue width of the pipeline, Y is the number of read/write ports on the data cache, and Z is the number of ASTQ ports.

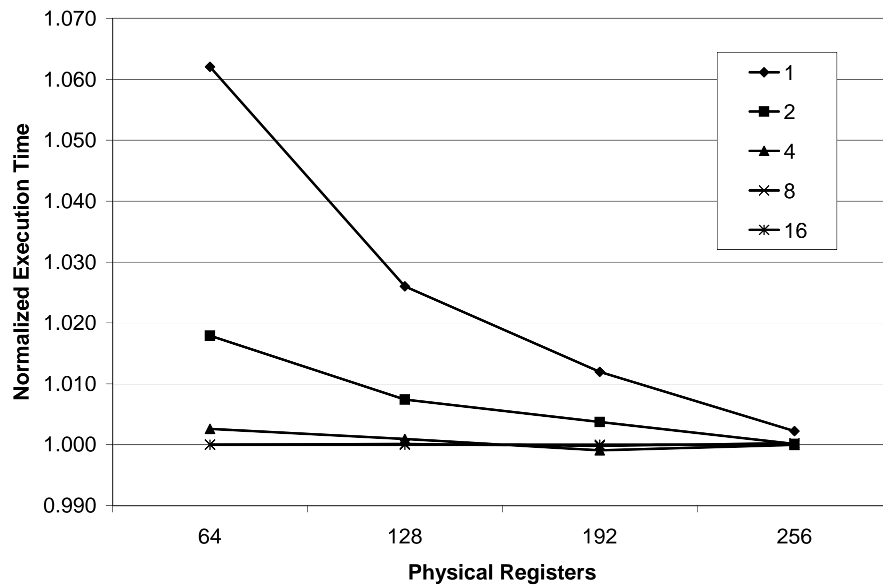
that can be in the queue at once. If the queue is full and the rename stage needs to add an operation, the rename stage is forced to stall.

Figure 5.4 shows the effect that reducing the number of ports on the ASTQ has on the execution time. The results are normalized to the execution time with a four ported queue. The chart shows the results for an eight issue pipeline. The results for a four issue were similar with even smaller differences in performance. The results show several things. First, reducing the number of ports (even to one) does not have a huge impact on the performance. This is true across all the pipeline configurations studied. The worst case performance decrease is 0.6% with respect to a four ported queue. The average execution time difference across all pipeline configurations and physical register file sizes is a 0.1% increase in execution time for one port and a 0.03% increase in execution time for two ports. Sec-

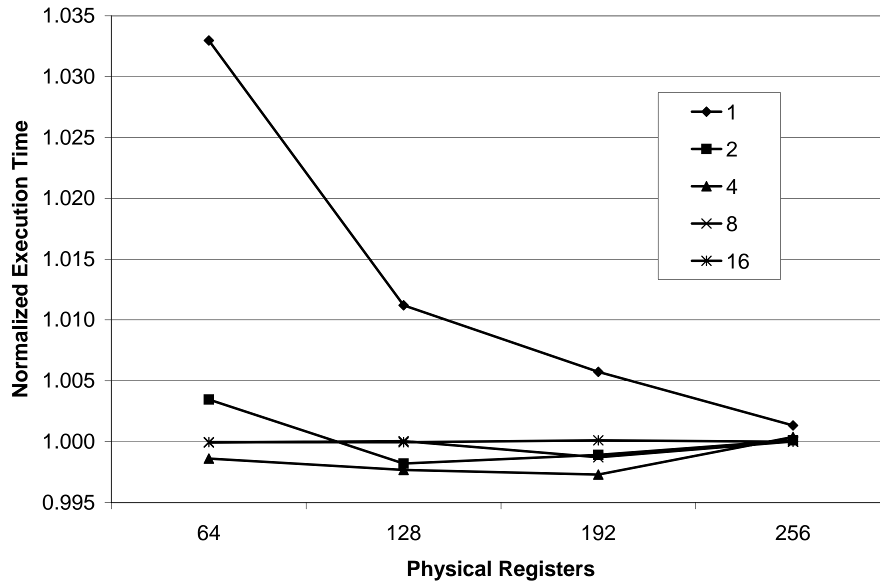
ond, the effect of reducing the ports is dependent on both the number of physical registers and the number of data cache ports. As the number of physical registers is decreased, the magnitude of the performance differences tends to increase. This makes sense because the ports are only going to have an effect when spills and fills are generated, and as the number of physical registers is decreased the number of generated spills and fills increases.

The performance is also effected by the number of data cache ports. In general, as the number of ports is decreased, the performance of an ASTQ with fewer ports tends to increase relative to an ASTQ with more ports. This is especially obvious in the results with only 64 physical registers. The one ported ASTQ is over 0.6% slower than a four ported ASTQ when the pipeline has four data cache ports. At three ports this is reduced to 0.5% slower, at two ports it's 0.35% slower, and with only one data cache port, the one ported ASTQ actually outperforms the four ported ASTQ by over 0.2%. Finally, as was just mentioned, it is possible that reducing the ports can improve performance. As stated in the previous section it is possible with a small number of physical registers to enter a vicious circle and flood the pipeline with spills and fills. Reducing the number of ports and the ASTQ is one way of slowing down this cycle. Thus, this leads to improved performance and a decrease in the number of spills and fills generated.

The second important parameter for the architectural state transfer queue is the size of the queue. Figure 5.5 shows the results of reducing the size of the architectural state transfer queue. The results are shown for an eight issue pipeline with one data cache port and with four data cache ports. Similar results are seen for four issue pipelines. The results are similar to the ASTQ port results in that they are dependent on the number of physical registers and number of data cache ports. Like the previous results, the effect of reducing the size of the ASTQ becomes more pronounced when the number of physical registers is reduced. Once again reducing the number of physical registers results in many more spills and fills which in turn puts more pressure on the ASTQ. With 256 physical registers, all sizes of queue provided nearly identical performance. In this configuration




Four Data Cache Ports




One Data Cache Port

**Figure 5.5: Architectural State Transfer Queue Size**

The execution time of the benchmarks with an ASTQ of varying size. The execution time is normalized to the execution time with a 32 entry ASTQ. The results are shown for an eight issue pipeline. One set of results is for a pipeline with four data cache ports, the other set is for a pipeline with a single data cache port. ASTQs with 1, 2, 4, 8 and 16 entries were studied.

the pipeline is generating so few spills and fills that a single entry, single ported ASTQ is adequate. As the number of physical registers is reduced this is no longer the case. This is especially true of a single entry ASTQ. The performance of this queue quickly decreases when the number of physical registers is reduced. In a four data cache port pipeline, with 192 physical registers the single entry queue is over 1% slower, this increases to over 2.5% with 128 and finally over 6% with 64 physical registers.

The performance of the queue is also dependent on the number of data cache ports in the pipeline. With four data cache ports, the relationship between execution time and the size of the architectural state transfer queue is straightforward. As the size of the queue is decreased, the execution time increases. The loss of performance is especially pronounced for one and two entry queues. When the ASTQ has at least four entries, the performance is almost identical in all cases. The worst case performance for a four entry queue occurs with 64 physical registers, and the execution time difference is less than 0.3%. In all other cases, the performance difference is less than 0.1%. The results for a single data cache pipeline are much more interesting. A single entry queue still results in worse performance in all cases, with the performance loss increasing as the number of physical registers is decreased. However, the other queue sizes in many cases actually result in increased performance. The two entry queue is able to provide better performance with both 192 and 128 physical registers, although it provides worse performance with 64 physical registers. The four entry queue actually provides the best performance at every size of physical register (although once again all perform nearly identically with 256 physical registers). The smaller queue improves performance by between 0.1% and 0.3% over a queue with thirty two entries. As discussed previously in this chapter, the virtual context architecture sometimes benefits from restricting the generation of spills and fills. This once again seems to be a case where this yields performance improvements. However, as the results for the one entry queue demonstrates, more restriction does not necessarily translate into improved performance. Instead, a careful balance is required to achieve this improvement. The balance occurs between the cost of

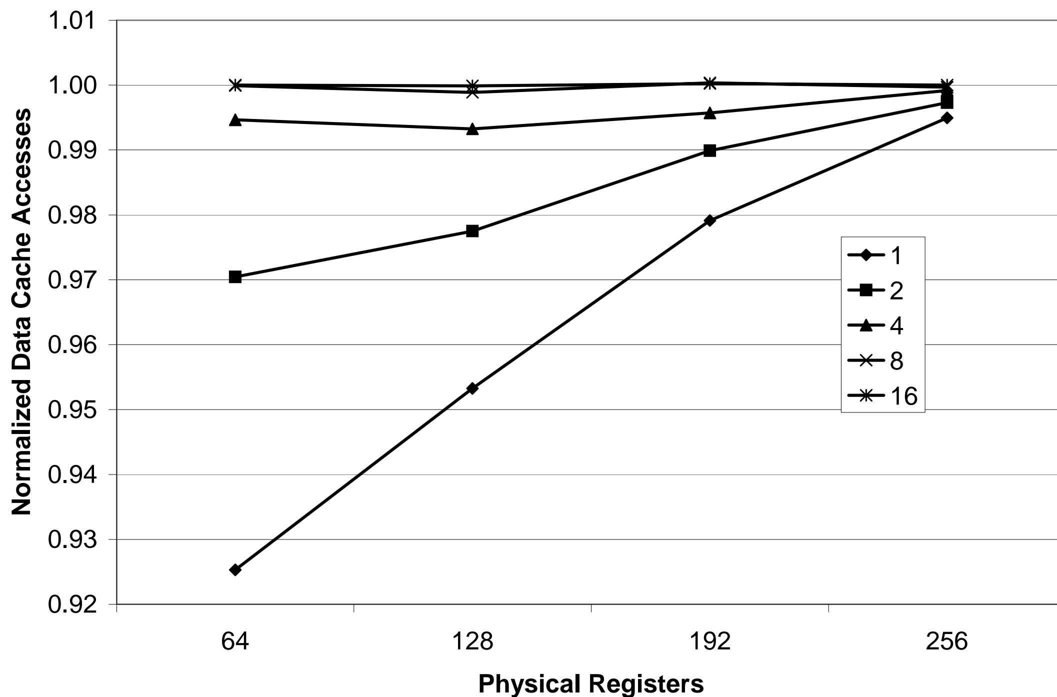



**Figure 5.6: ASTQ Size: Cache Accesses**
The data cache accesses of the benchmarks with an ASTQ of varying size. The cache accesses are normalized to the accesses with a 32 entry ASTQ. The results are shown for an eight issue pipeline with a single data cache port. ASTQs with 1, 2, 4, 8 and 16 entries were studied.

generating spills and fills and the cost associated with stalling the frontend while waiting for physical registers to be freed for reuse. This effect is more pronounced when the number of data cache ports is reduced. In such a pipeline, the cost of spills and fills is increased because of the fierce contention over the data cache ports.

Figure 5.6 shows the data cache accesses with various ASTQ sizes. The results are normalized to the data cache accesses with a thirty-two entry queue. The results plainly show that reducing the size of the queue results in fewer data cache accesses. This decrease becomes more pronounced with fewer physical registers. With 64 registers, a single entry queue has almost 8% fewer accesses to the data cache, a two entry has a savings of 3% while for a four entry the savings is less than 1%. The difference in data cache accesses occurs because of the change in spill and fill traffic. With a smaller queue and especially with fewer

physical registers, the spill and fill traffic can be reduced. In the case of two and four entry queues, this reduction is large enough to provide some performance benefit. In the case of a single entry queue, the reduction is so great that the performance suffers as the frontend is forced to wait for needed spills and fills to be generated.

These studies show that for the range of pipelines studied, a four entry queue with two ports provides the best performance overall. Furthermore, it shows that high performance is able to be achieved with a very modestly sized structure. In an actual virtual context architecture design, the size and ports of the ASTQ would need to be carefully balanced for the given pipeline configuration.

#### 5.1.2.2 Operation Parameters

The insertion of spills and fills into the pipeline by the method of inserting operations does not have the flexibility of design that the architectural state transfer queue has. As was shown earlier in this section, this method of implementing spills and fills suffers in comparison to the ASTQ in pipelines with a small number of data cache ports, in particular with a single data cache port. Unfortunately inserting spills and fills directly into the pipeline means that almost all the control of this implementation is lost to the operation of the pipeline itself. Therefore, this section only examines a single optimization. As noted in Section 3.2.2, spills and fills are never dependent on any other instructions and in fact never require bypass or scheduling hardware. It was noted that this is the functionality that the load store queue (LSQ) provides. Therefore, spill and fill operations are going to suffer from delays caused by memory disambiguation logic that is never needed by these particular operations. A possible optimization is to mark these operations in the load store queue and allow them to bypass the normal memory disambiguation logic. The result of this is that these operations are marked as ready as soon as they are inserted into the LSQ. Figure 5.7 shows the execution time results with an optimized load store queue. In all cases, the optimized load store queue is able to provide a performance improvement. The performance increase is between 0.4% for a mixed implementation on a four issue pipeline with 128 physi-

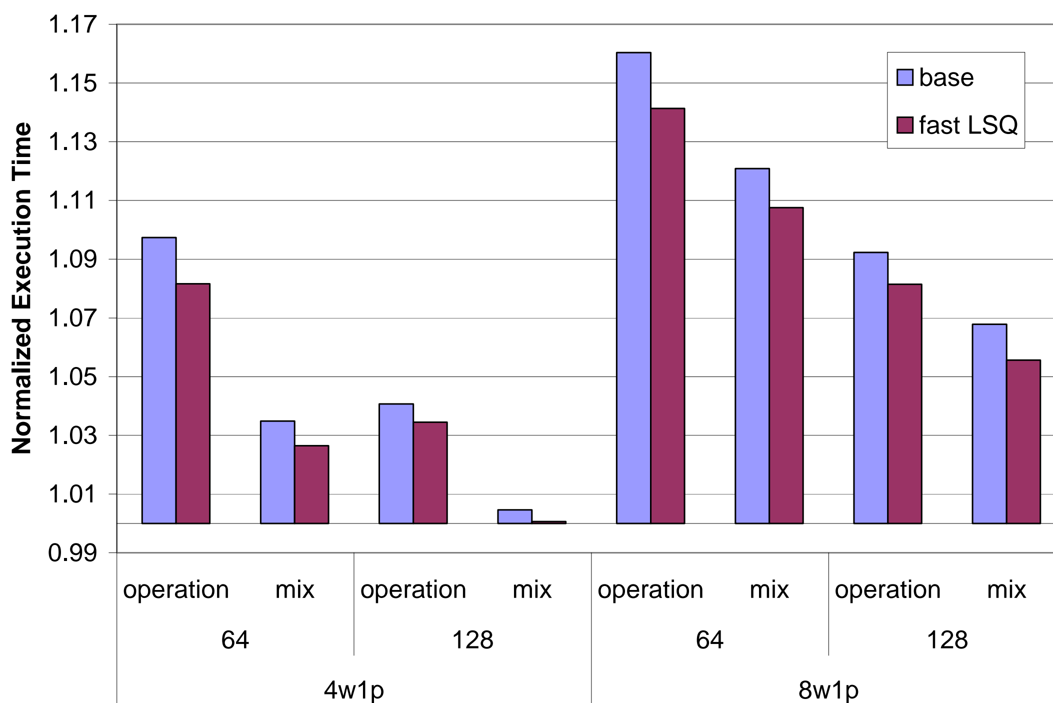



**Figure 5.7: Operation With Optimized Load Store Queue**
The execution time of operation only and mixed operation/ASTQ with an optimized load store queue. The execution times are normalized to an ASTQ only implementation. The base results are without any load store queue optimization. The fast LSQ results are when spills and fills are allowed to bypass the normal memory disambiguation logic. The results are shown for both four and eight issue pipelines, both with one data cache port and with either 64 or 128 physical registers.

cal registers to 1.9% for an operation only implementation on an eight issue pipeline with 64 physical registers. This results in a decrease in the performance difference between the ASTQ only implementation of between 10% and 25%. Although all the configurations show a performance increase, they are still all slower than an architectural state transfer queue only implementation. This seems to suggest that although memory disambiguation is a factor, it is not the major factor in the decreased performance of the operations implementation relative to the architectural state transfer queue implementation.

## 5.2 Rename Stage

We looked at several parameters that affect the operation of the rename stage, including the associativity of the rename table, limiting the number of ports on the table, and the cost of adding extra logic to the rename stage.

### 5.2.1 Associativity

The first parameter was the associativity of the rename table. As stated in Section 3.3 the size of the rename table is theoretically independent of the number of threads. This is in direct contrast to a traditional pipeline which requires a duplicated rename table for every supported thread. However, the rename table in the virtual context architecture must be sized to provide adequate room to rename all the registers from the currently active contexts, otherwise performance will suffer. A quick study of the average number of rename entries shows that supporting more threads leads to additional rename table entries. Figure 5.8 shows the average number of rename table entries with one, two and four threads. These results were measured using a fully associative table with no capacity or conflict misses. The results clearly show that two factors influence the number of rename table entries: the number of physical registers and the number of threads.

The number of rename table entries represents the number of unique logical registers active at any one time in the pipeline. The physical register file contains both these registers and additional registers used for renaming purposes. As the number of physical registers is increased, the pipeline can store registers from more contexts simultaneously. For example, the pipeline may not need to spill registers from the calling context, or even the context that called the calling context. The results show that a single thread uses on average 57 rename table entries with 192 physical registers and 96 with 256 physical registers. The multi thread workloads show a similar trend. Two threads has 79 entries with 192 physical registers, 114 with 256 and 162 with 320 physical registers. Four threads has

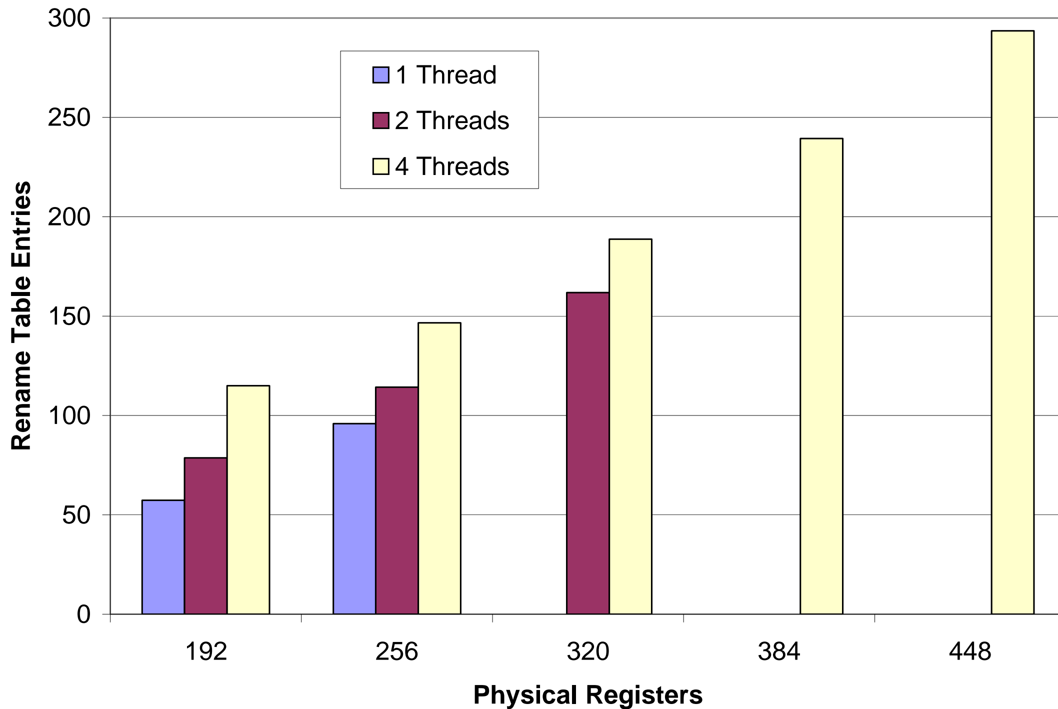



**Figure 5.8: Average Rename Table Entries**

The average number of rename table entries for the one, two and four thread workloads. The results are given at a variety of physical register file sizes.

115 entries with 192 physical registers up to 293 entries with 448 physical registers.

The rename table for a single threaded pipeline will mostly contain registers from the currently active context plus additional registers from some number of calling contexts. In a pipeline that supports more than one thread, the rename table will contain registers from the active context for each thread, plus some from their calling contexts. The effect of the number of threads is limited by the physical register file. With more threads in the pipeline, more registers are needed to hold the registers from the active contexts of the threads, leaving fewer to hold registers from calling contexts. Similarly, with more active contexts fewer physical registers will be available for rename. In comparison to a single thread, two threads requires 37% more rename entries with 192 physical registers and 19% more with 256 physical registers. For four threads, the percentages are 100% and 53% respectively. A similar comparison can be made between two threads and four

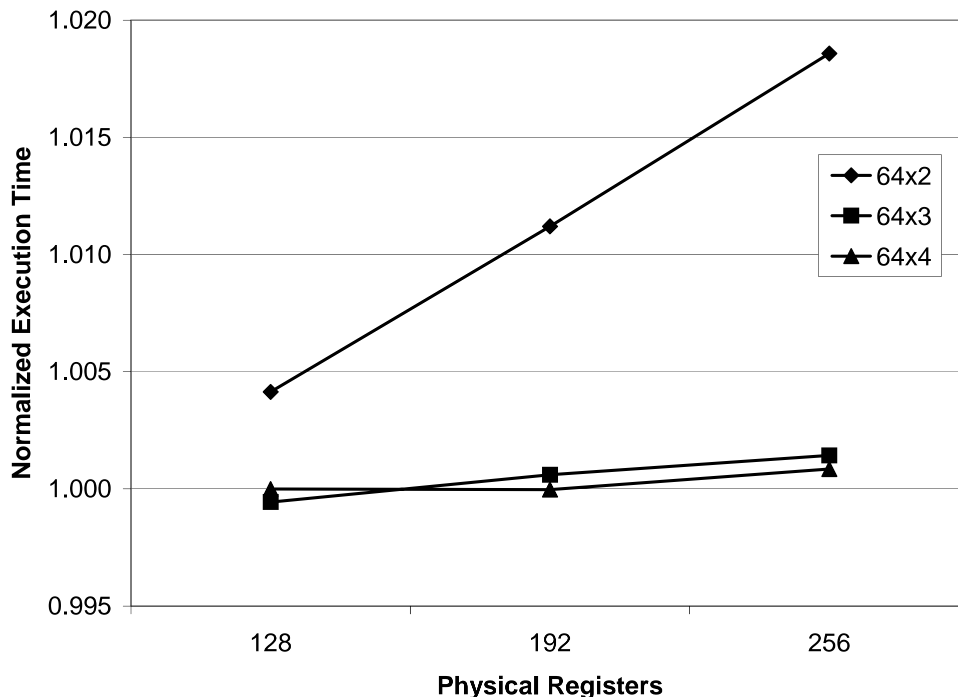



**Figure 5.9: Single Thread Rename Table Associativity**
The average execution time of the benchmarks when the associativity of the rename table is varied. The execution times are normalized to a fully associative rename table with one entry for each physical register. This ensures that the rename table would never suffer a capacity miss. The results are for 2, 3 and 4 way associative tables with 64 entries per way. The results are given for a variety of physical register file sizes.

threads. Four threads requires 46% more entries with 192 physical registers, 28% more with 256 physical registers and 16% with 320. The ratios all decrease as the number of physical registers increases. Therefore, when sizing the rename table, the number of threads in the pipeline must be taken into account.

We studied a fully associative table versus 64x4, 64x3 and 64x2 set associative rename tables for single thread workloads, see Figure 5.9 for the execution time results. The performance of the processor with a 64x4 table is within 0.1% of the fully associative table with 256 physical registers, and a negligible difference with fewer physical registers. Similarly, the 64x3 table is within 0.2% at 256 registers and a negligible difference with fewer. The 64x2 table has a larger impact on

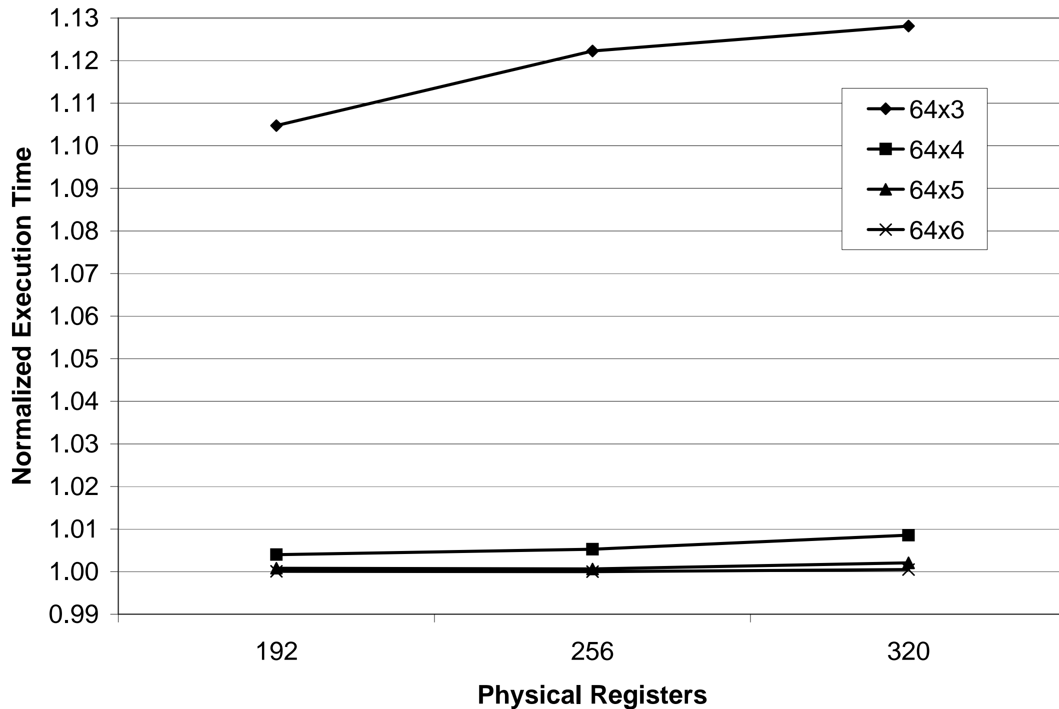



**Figure 5.10: Two Thread Rename Table Associativity**

The average execution time of the two thread workloads when the associativity of the rename table is varied. The execution times are normalized to a fully associative rename table with one entry for each physical register. This ensures that the rename table would never suffer a capacity miss. The results are for 3, 4, 5 and 6 way associative tables with 64 entries per way. The results are given for a variety of physical register file sizes.

the performance. It is 2% slower with 256 physical registers, 1% with 192 and 0.5% with 128.

Figure 5.10 presents the results for two threads. The results show that a higher associativity is needed than when only a single thread is used. While a single thread can achieve close to ideal performance with a 64x3 or a 64x4 table, two threads requires 64x5 or 64x6 to achieve a similar level. A 64x6 rename table provides performance identical to a fully associative table, with a difference of less than 0.05%. There is a slight performance loss with a 64x5 rename table of 0.2% with 320 physical registers. However, the difference decreases as the number of physical registers is decreased, with a difference of less than 0.08% with 256 or 192 registers. When the associativity is further decreased to four, the performance difference widens even further. With 320 physical registers, the 64x4 table is

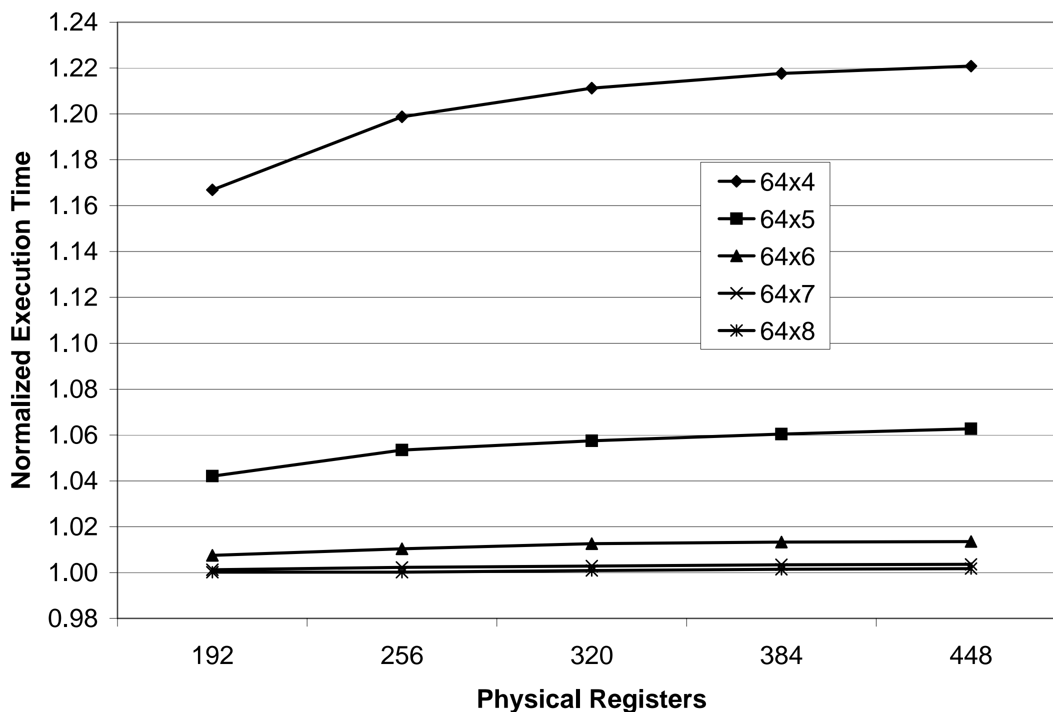



**Figure 5.11: Four Thread Rename Table Associativity**
The average execution time of the four thread workloads when the associativity of the rename table is varied. The execution times are normalized to a fully associative rename table with one entry for each physical register. This ensures that the rename table would never suffer a capacity miss. The results are for 4, 5, 6, 7 and 8 way associative tables with 64 entries per way. The results are given for a variety of physical register file sizes.

nearly 0.9% slower than a fully associative table, with 256 registers the difference is 0.5% and with 192 it is less than 0.4%. This performance difference is still relatively small. However, with an associativity of three, the performance difference becomes much larger. With 320 physical registers, the smaller table is 12.8% slower. The difference decreases to 12.2% with 256 registers and 10.5% with 192.

Figure 5.11 presents the results with four threads. These results show trends similar to two threaded results, although an increase in associativity is necessary to achieve good performance. While two threads shows good performance with 64x5 and 64x6, to achieve comparable performance with four threads, the rename table needs to be 64x7 or 64x8. With a 64x8 table, the execution time is within 0.2% of the performance with a fully associative table. With a 64x7 table, the execution time is within 0.4%. The 64x6 table yields execution times similar to

a 64x4 table for two threads. The performance loss is over 1.3% with 448 physical registers. The loss drops steadily as the number of physical registers decreases until the loss is 0.7% with 192 physical registers. The performance loss with a 64x5 table becomes quite large, ranging between 4% with 192 physical registers to over 6% at 448. A 64x4 table is 22% slower with 448 physical registers. This drops to 16.7% with 192 registers. The performance of a 64x3 table is not shown on the chart because the loss is so great, nearly 64% with 448 physical registers.

The previous results revealed that larger tables are needed for multithreaded pipelines in comparison to single thread pipelines. One potential reason for the poor performance of the smaller rename tables in a simultaneous multithreaded pipeline is conflicts between the non windowed registers. The non windowed registers for each thread will always have the same mapping address and therefore occupy the same sets in the table. These registers also will have definite patterns in their usage frequency. For example, the stack pointer will tend to be used very frequently, while the floating point return will only be used for a short time and only in certain functions. The results presented so far map the non windowed registers to the exact some memory location regardless of thread (the address space identifier in the tag will distinguish between them). With four simultaneous threads, this will cause some sets to have major conflict problems. For example one set will be associated with the stack pointer and may need to have the stack pointer for each thread held in the various ways. If a windowed register now needs to be renamed that maps to this set, this will lead to a large potential performance loss as the pipeline needs to be drained. Therefore, one potential optimization in the case of multiple threads is to ensure that the non windowed registers tend to be mapped to different sets for the different threads. This was achieved by simply offsetting the mapping address of the non windowed registers by a fixed amount multiplied by the thread number. The fixed amount is simply the number of sets (64 in our case) divided by the number of threads. The optimized results for four threads is show in Figure 5.12. The chart shows the execution time for the optimized rename normalized to the execution time for the non optimized rename table of the same

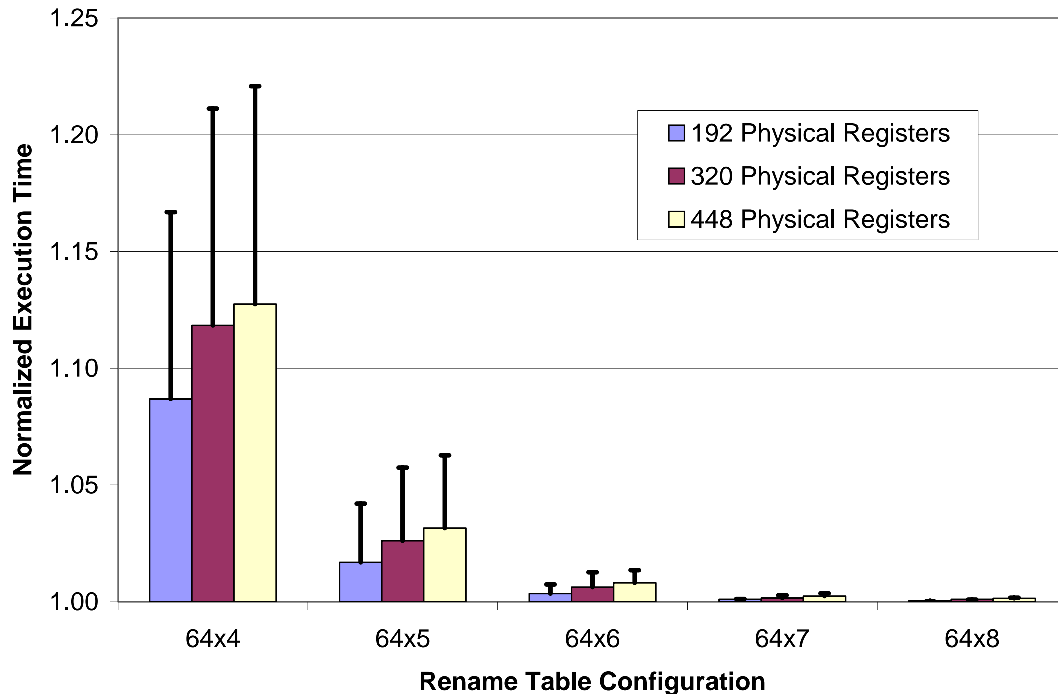



**Figure 5.12: Four Thread Optimized Rename Table**

The execution time difference achieved when the rename table is optimized for multiple threads. The execution times are normalized to a fully associative rename table with one entry for each physical register. This ensures that the rename table would never suffer a capacity miss. The error bars indicate the normalized execution time of the non optimized rename table. The chart presents the results for the four thread workloads with 192, 320 and 448 physical registers.

size. The results show that with the smaller table sizes this optimization is able to improve performance over a non optimized rename table. This optimization decreases the performance loss of the rename configurations by approximately half. While this is a sizable improvement, it is not large enough to necessarily change which configurations one would want to use. Specifically, at least a 64x6 rename table is probably necessary to achieve good performance with four threads.

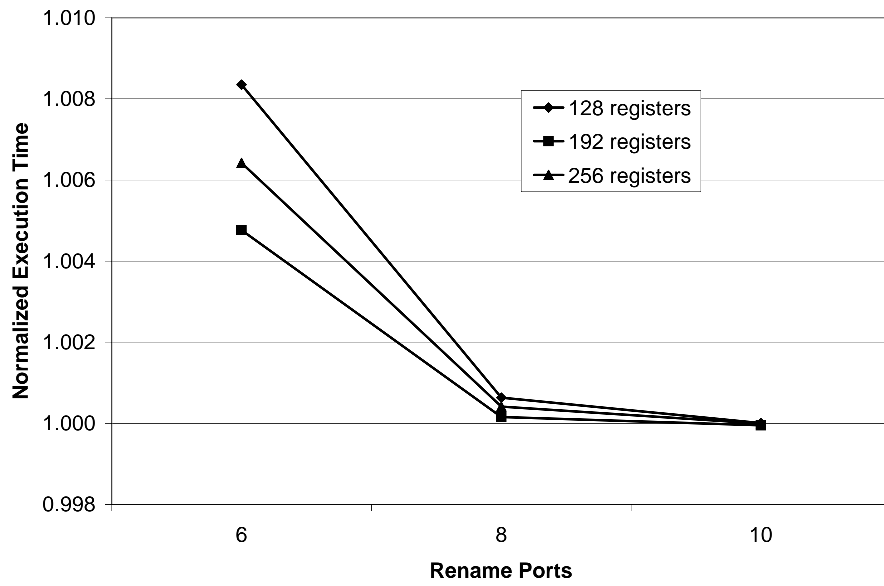
The results show that a moderately sized table of 64x3 is able to achieve performance approaching that of a fully associative table. Although larger than a conventional rename table, it would still be practical to implement in a pipeline. The studies in this section also indicated that larger tables are needed in simultaneous multithreading pipelines. In the two thread case, a 64x4 table is adequate,

although a 64x5 table would be preferred (especially with a larger physical register file). In a conventional two thread SMT pipeline, the rename table would double in size. In the virtual context architecture, the rename table increases, but the increase is less than double. This trend is even more pronounced in the four thread case. A conventional four thread SMT pipeline would require four complete rename tables. In contrast, the VCA is able to achieve nearly ideal performance with a 64x6 rename table. This is only twice the size of the single threaded virtual context architecture rename table.

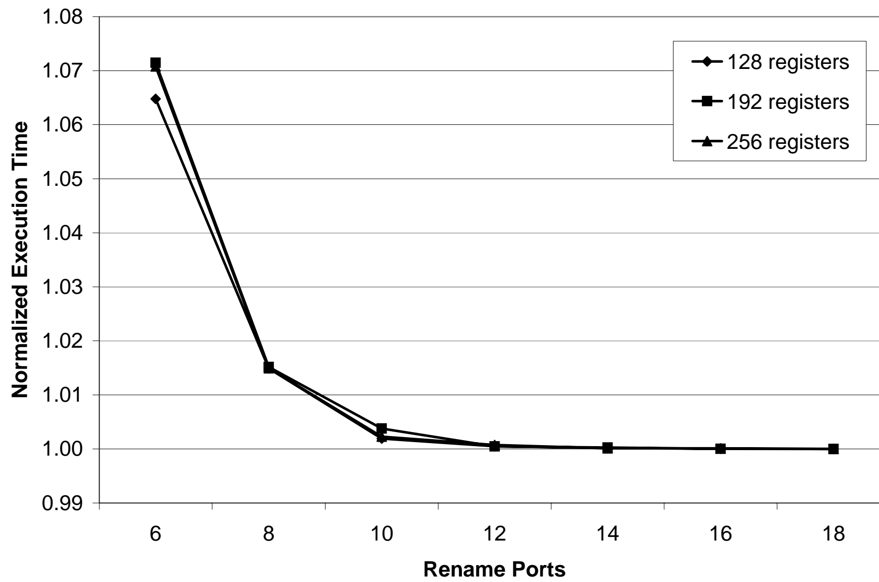
### 5.2.2 Ports

One of the factors that determines the size of a structure in a processor is the number of ports. Reducing the number of ports will drastically reduce the size of the structure. A fully ported rename table for a four issue processor requires 12 ports, while an eight issue processor requires 24 ports. It is unlikely that all these ports are needed. This is particularly true if one assumes that lookups of the same address can share a port. There are three situations where the full set of ports are not needed. First, a full set of instruction may not be ready for rename every cycle. Second, every instruction does not require two source register operands and one destination register operand. Third, if port sharing is allowed, it is very likely that instructions that are renamed at the same time would tend to share some registers between them. A quick study revealed that on a four issue pipeline an average of only about 3.3 ports are used per cycle, and 5.4 ports are used each cycle that an instruction is actually renamed. An eight issue processor uses on average only about 3.5 ports per cycle and 7.6 ports each cycle that an instruction is actually renamed. These results indicate that a fully ported rename table is not necessary.

We studied the performance cost of reducing the number of ports, see Figure 5.13 for the results. As expected, the results show a decline in performance as the number of rename table ports is reduced. The effect of the rename table ports seems to be mostly independent of the number of physical registers. For a four issue pipeline, we studied reducing the ports to 10, 8 and 6: 10 ports




Four Issue Pipeline




Eight Issue Pipeline

**Figure 5.13: Rename Table Ports**

The execution time of the benchmarks with varying number of rename table ports. The results are normalized to the execution time of a fully ported rename table. The top chart shows the results for a four issue pipeline. The bottom chart shows the results for an eight issue pipeline. The benchmarks were run with 128, 192 and 256 physical registers.

results in a less than 0.01% change in execution time; 8 ports results in a reduction of 0.05%; and 6 ports results in a reduction of 0.7% on average. For an eight issue pipeline, we studied reducing the ports between 18 and 6. Reducing the ports to 14 or more shows almost no difference in performance, with a maximum difference of less than 0.02%. Starting at 12 ports, the difference becomes to grow substantially. With 12 ports the difference is 0.06%, with 10 the performance loss is 0.3%. The performance now begins to degrade rapidly, with a difference of 1.5% with 8 ports and 6.9% with 6 ports.

This study demonstrates that good performance can still be achieved even when substantially reducing the ports of the rename table. In a four issue pipeline, the ports can be reduced from 12 to 8 while only incurring a 0.04% drop in performance. In an eight issue pipeline an even more drastic reduction is possible. In this case the ports can be reduced by half from 24 to 12 with only a 0.06% increase in execution time.

### 5.2.3 Cost of Extra Logic

The rename stage in the virtual context architecture is more complicated than a conventional rename stage. Not only is the rename table set associative instead of direct mapped, but additional logic is also needed because of the spill and fill generation. Because of the more complex logic necessary in the rename stage, it may not be possible to meet the cycle time requirements of the pipeline. A solution is needed to allow the rename stage to meet the cycle time. We studied the cost of two potential solutions: adding an extra stage and delaying rename on certain events. Figure 5.14 presents the costs of these solutions.

One possible solution is to add an extra stage to the frontend of the pipeline. The logic of the rename stage would be pipelined across the two stages. It results in a relatively modest decrease in performance that remains constant with different physical register file sizes.

A second solution is to implement some of the extra logic on an as needed basis. In other words, if an instruction requires more logic to process it than can be done in a cycle, the rename stage can stall for a cycle while it processes just

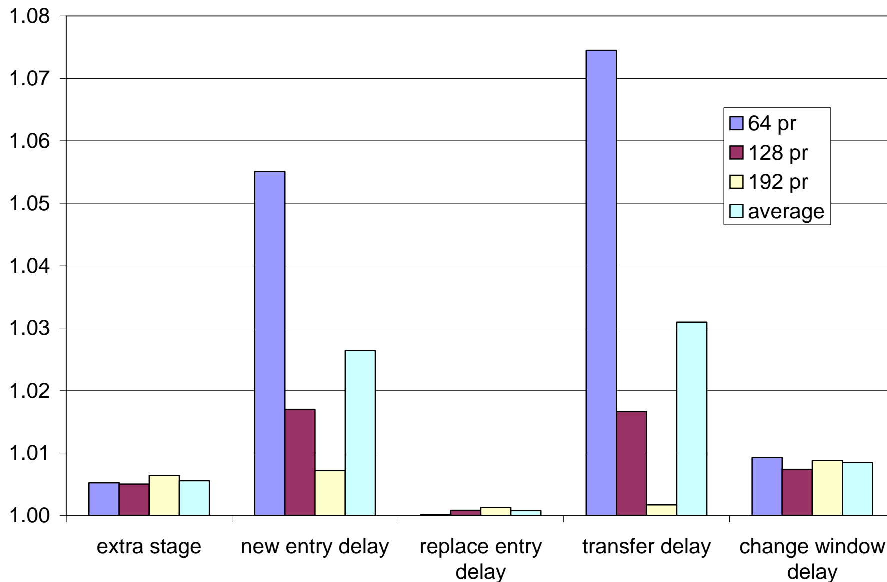



**Figure 5.14: Cost of Extra Logic In The Rename Stage**

The average relative execution time of processor when the given delay is added to the rename stage of the pipeline. All delays are one cycle. The rename table is 64x3. The bars represent runs with different number of physical registers(pr) and the average across all the runs.

that instruction. The next cycle it can dispatch the problem instruction, and resume normal processing on the cycle after that. The rename stage must already be able to stall when the instruction queue or reorder buffer are full. Therefore the capability to stall already exists in the rename stage. The events that may require these occasional stalls are as follows: 1) put a new entry in the rename table; 2) replace an entry; 3) generate a spill or fill (transfer); and 4) change the register window.

A new entry occurs when there is a miss in the rename table and a new table entry must be allocated. This results in a relatively large penalty. This event occurs whenever a logical register is defined that doesn't exist in the rename table (very likely to occur multiple times at the start of every function). The penalty grows larger with fewer physical registers.

Replace entry occurs when there is a miss in the table, and a free entry does not exist. In this case, the pipeline must check if any of the entries are available and if at least one is, it must be freed. The performance loss due to this event is

very small. The penalty actually decreases with fewer physical registers, because fewer valid rename table entries are possible.

A transfer delay occurs whenever the rename stage must generate a spill or fill. This has the largest average penalty, but is very dependent on the number of physical registers. With fewer physical registers, the VCA is forced to generate more spills and fills, thus increasing the penalty. This delay does have the benefit of reducing spill and fill traffic When the pipeline is forced to stall when generating a spill, there is an increased chance that an instruction will commit freeing up a physical register. At 64 physical registers, this delay reduces the data cache traffic by almost 13%.

A change window delay occurs whenever a function call/return instruction is renamed. This is also a relatively modest penalty and is independent of the number of physical registers.

The best solution is to add an extra stage to the pipeline—it only affects performance when branches are miss predicted. If the rename logic is still too complicated to meet cycle time constraints, a delay on entry replacement would make sense. Not only is this one of the smallest delays, but the event itself involves the most complex logic. The actual solution for a given implementation would be dependent on how aggressively the processor is pipelined.

## 5.3 Summary

This chapter studied the effects that the various implementation parameters have on the performance of the virtual context architecture. These parameters fall into two main groups. The first group is the implementation of fills and spills. The second group is the rename stage configuration.

### 5.3.1 Spill/Fill Implementation

The implementation of spills and fills was dependent on several factors. The first factor explored was the replacement policy for physical registers. The studies revealed that an overwrite last policy provided the best performance. The performance of the least recently used and not dirty first are very close. The overwrite

first policy yielded very poor performance. The second major factor was the implementation technique used for spills or fills, either an architectural state transfer queue or inserting operations directly into the pipeline, or a mix. The results show that the ASTQ provides better performance in almost every case. The operations are able to provide similar performance only with relatively large physical register files and more than one data cache port.

The configuration of the architectural state transfer queue was also examined. The two parameters studied were the number of ports and the size. The results showed that a one or two port queue was able to provide about the same performance as a four ported queue, with the two port providing slightly better performance than the one port. The studies of queue size revealed that a four entry queue is able to provide good performance in all pipeline configurations, and in cases with only one data cache port better performance than a larger queue.

### 5.3.2 Rename Stage

We looked at several parameters that effect the operation of the rename stage. The first set was the configuration of the rename table: associativity and ports. Our studies showed that for a single thread a 64x3 or 64x4 set associative rename was able to provide nearly ideal performance, within 0.2% and 0.1% respectively to a fully associative rename table. A pipeline that supports two threads requires a slightly larger rename table to achieve similar performance, 64x5 or 64x6. Similarly a four thread pipeline requires a larger table than those for two threads, in this case 64x6 or 64x7. In the case of multithreaded pipelines, better rename performance can be achieved by ensuring that the non windowed registers for the various threads map to different sets. We also found that it was possible to reduce the number of rename table ports without significantly effecting performance. The virtual context architecture was able to achieve good performance on a four issue pipeline with as few as 8 ports. Similarly, on an eight issue pipeline 12 ports was enough to provide good performance.

The second set of parameters we looked at was the cost of the extra logic in the rename stage of the pipeline. The best solution is to add an extra stage to the

pipeline—it only affects performance when branches are miss predicted. If the rename logic is still too complicated to meet cycle time constraints, a delay on entry replacement would make sense. Not only is this one of the smallest delays, but the event itself involves the most complex logic.

These results were used to choose configurations to evaluate in detail. The next several chapters provide a detailed evaluation of the virtual context architecture against several baseline architectures.

# Chapter 6

# Register Window Studies

This chapter evaluates the performance of the virtual context architecture as an implementation of register windows. This chapter is composed of four sections, each of which presents the results for a different pipeline configuration. The chapter begins with a detailed analysis of the virtual context architecture implemented in a four issue out-of-order pipeline. The second section examines the virtual context architecture in a wider issue pipeline, in this case an eight issue out-of-order pipeline. The third section looks at the performance in a narrower two issue out-of-order pipeline. The final pipeline section examines the VCA in a simpler single issue in-order pipeline. The chapter ends with a summary of the results. The primary metrics used to evaluate the performance of the VCA are the execution time of the benchmarks and the number of accesses made to the first level data cache.

The virtual context architecture is evaluated against three other architectures: a baseline architecture that does not support register windows, an ideal implementation of registers windows, and a conventional register window implementation. The first architecture is a baseline architecture that represents a typical non register window pipeline. The baseline pipeline executes the non register window binaries. The second architecture is an ideal architecture that represents a theoretical pipeline in which spills and fills are handled instantaneously, and never generate accesses to the data cache. It provides a theoretical limit on the register window performance. The third architecture is a conventional register window configuration. This represents a typical implementation of register windows and uses the same ABI as the virtual context architecture. The physical register file is split into several windows with the number of windows depending on the size of

|  | Rename Table Size | Rename Table Ports | Extra Logic Cost |
|---|---|---|---|
| vca1 | 64x4 | 10 | none |
| vca2 | 64x3 | 8 | extra stage |
| vca3 | 64x3 | 8 | vca2 + replace entry delay, change window delay |
| vca4 | 64x2 | 6 | vca3 + transfer delay |

**Table 6.1: VCA Configurations For The Four Issue Pipeline**
The virtual context architecture configurations studied for the four issue pipeline. The configurations represent a range of implementations. The configurations are composed of two things. The first is the size and ports of the rename table. The second is the cost assumed for the extra logic in the rename table.

the physical register file. The remaining registers are used for renaming. When a register window overflow or underflow occurs, the pipeline delays for 10 cycles (a generous simulation time for the hardware trapping to the operating system). After the delay, load/store instructions are inserted into the pipeline to either load all the windowed registers of the new window on an underflow, or to store all the dirty registers in the departing window on an overflow.

All the virtual context architecture configurations in these studies use the overwrite last policy and a four entry architectural state transfer queue with two ports. In Chapter 5 we determined that these provide the best performance. The rename table is configured according to the pipeline being studied. Specifically, the number of rename table ports needed is dependent on the issue width of the pipeline.

## 6.1 Four Issue Pipeline

This section presents the performance of the virtual context architecture in a four issue out-of-order pipeline. Several different VCA configurations were studied, see Table 6.1. These configurations represent a range of implementations, from vca1 with the least restrictions and most hardware to vca4 with the most restrictions and least hardware. The actual implementation used would depend on the cycle time restrictions of the rename stage. The results are presented in three subsections: benchmark details, results of two data cache ports and results for one data cache port. The benchmark details section shows the effect register win-

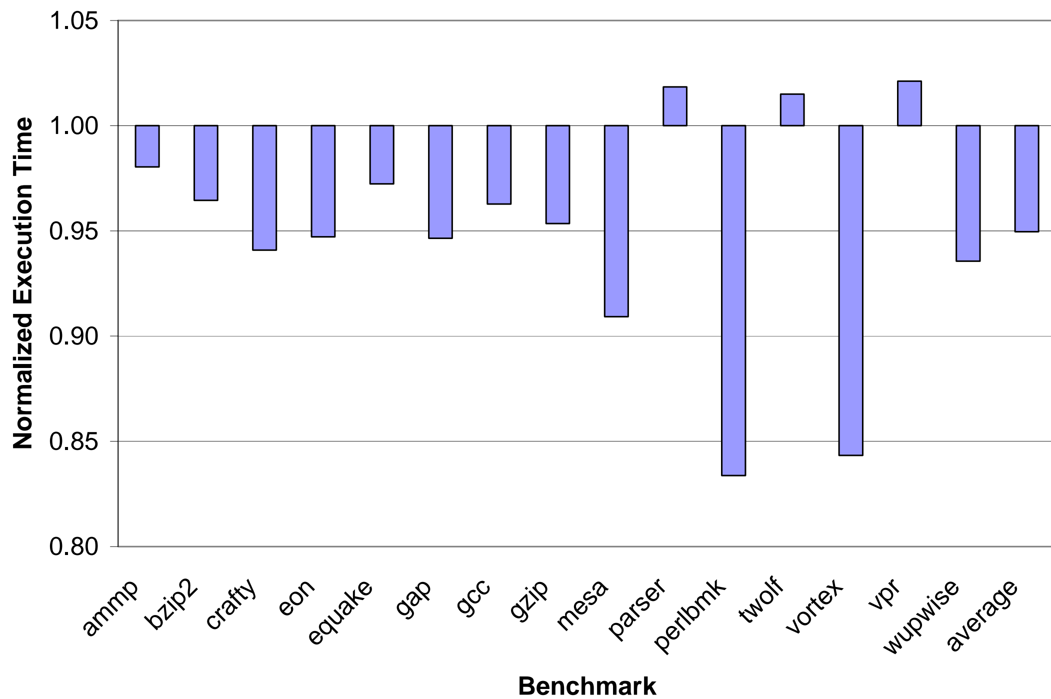



**Figure 6.1: Register Window Binaries Execution Time**
The execution time using the ideal configuration of the individual benchmarks used to measure the register window performance. The results are normalized to the execution time of the baseline binaries. These represent the theoretical limits of the improvement associated with register windows for this particular pipeline.

dows has on the individual benchmarks. The other sections present comparisons of the different virtual context architectures to the baseline architectures described in the introduction of this chapter.

### 6.1.1 Benchmark Details

This section presents the execution time and data cache access savings achievable by using registers windows. Figure 6.1 shows the ideal execution time of the register windows binaries for this pipeline normalized to the execution time of the baseline binaries. See Section 4.1 for a description of the register window and baseline binaries. The results show that register windows is able to decrease the execution time of the binaries by an average of 5%. The benchmarks tend to cluster into three distinct groups based on their relative performance. The majority of the benchmarks experience a decrease in execution time of 3-8%. There are

outliers on both sides, however. Three of the benchmarks actually experience a performance decrease relative to the baseline binaries. The decrease in performance is relatively small at approximately 2%. If one looks at the relative dynamic path lengths in Table 4.3, the performance of twolf is easily explained. Although it makes a lot of function calls, the code does not benefit very much from the register window optimizations. However, both parser and vpr show nearly a 10% decrease in dynamic path length for the register window binaries. A study of the statistics reveals that the register window binary of vpr suffers from more branch predictor mispredictions, approximately 10% more mispredictions per executed instruction. If a perfect branch predictor is used, the results for vpr match the dynamic path length ratio. On the other end of the spectrum, two benchmarks (perlbmk and vortex) experience very large decreases in execution time of 17% and 16% respectively. As the results show, the effect of registers windows is very dependent on the benchmark.

The second metric considered is the number of data cache accesses. Figure 6.2 shows the ideal data cache accesses of the register window binaries for this pipeline normalized to the data cache accesses of the baseline binaries. Register windows are used to reduce the save and restore overhead necessary for a function call and the results show that in every case register windows is able to decrease the number of data cache accesses. The average decrease is nearly 20%. The smallest decrease occurs for twolf which shows only a 3% reduction in data cache traffic. The largest decreases occur for vortex and perlbmk. These benchmarks show a decrease of almost 40%.

In general, the results show that the relative execution time of the benchmarks is tightly coupled to the reduction in data cache accesses. However, the two problem benchmarks (parser and vpr) both show significant reductions in data cache accesses of nearly 25% for parser and 15% for vpr, even though their execution times are not improved.

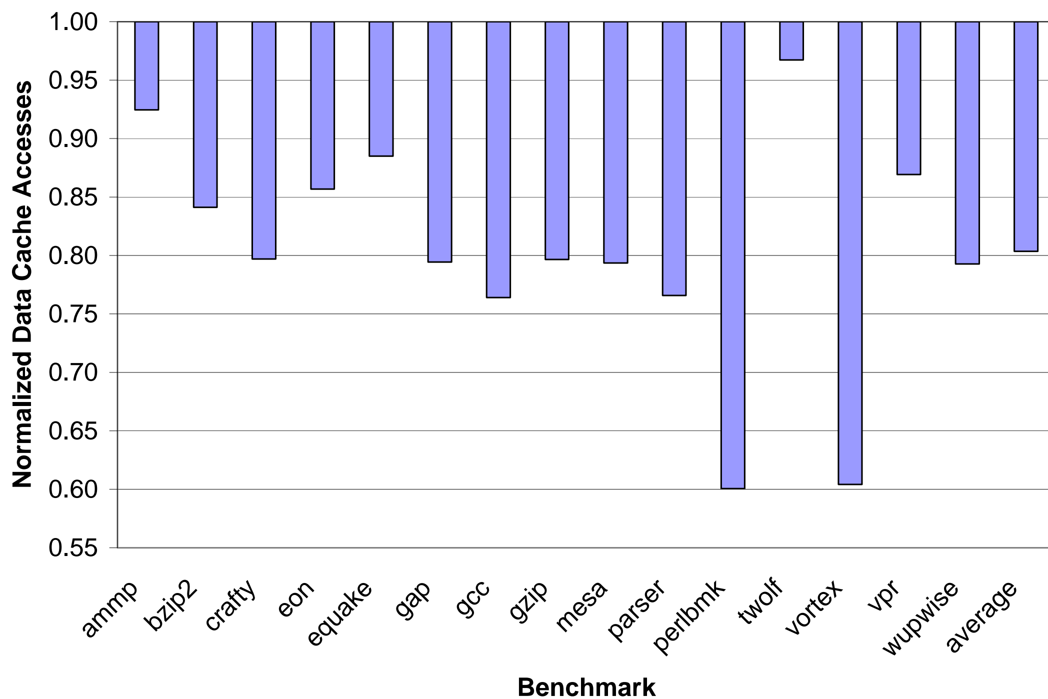



**Figure 6.2: Register Window Binaries Data Cache Accesses**

The data cache accesses using the ideal configuration of the individual benchmarks used to measure the register window performance. The results are normalized to the data cache accesses of the baseline binaries. These represent the theoretical limits of the improvement associated with register windows for this particular pipeline.

### 6.1.2 Two Data Cache Ports

This section provides a detailed comparison of the virtual context architecture to the baseline architectures for a four issue pipeline with two data cache ports. The results are presented for a variety of physical register file sizes. The smallest number of physical registers examined is 64 physical registers, which is just enough to hold the architectural state without rename registers. Only the virtual context architecture and ideal register window architecture are able to run with this few physical registers. The other architectures require their full architectural state be kept in the physical register file. With the addition of rename registers, this constrains them to a physical register file larger than 64. The largest number examined is 256 physical registers. This is enough registers to guarantee that the baseline architecture will never need to stall rename because of a lack of physical

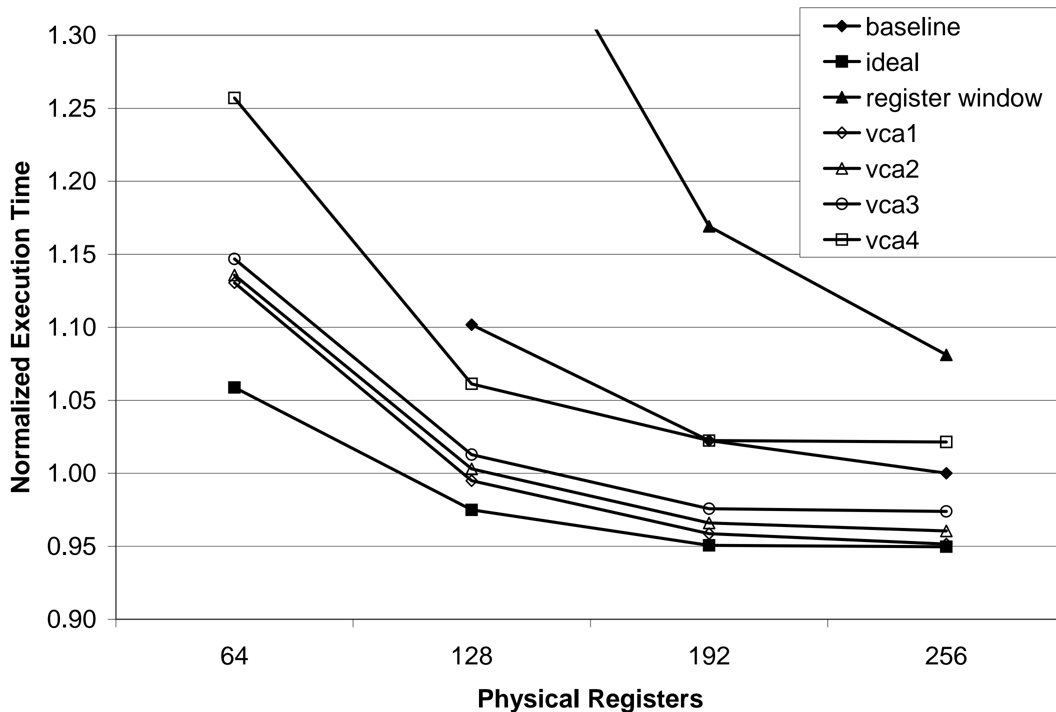



**Figure 6.3: Four Issue Execution Time Comparison**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and four different virtual context architecture configurations. The execution times are normalized to the execution time of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

registers. To ensure this, the processor must have one physical register for each architectural register plus one physical register for each reorder buffer entry (assuming instructions only have a single destination register). The results are presented in two sections. The first section presents the execution time results. The second section presents the cache results.

#### 6.1.2.1 Execution Time Results

This section examines the execution time of the benchmarks in the four architectures being compared. Figure 6.3 presents the execution time results. The results are all normalized to the baseline architecture with 256 physical registers. The first three VCA configurations provide better performance at every physical register file size than the baseline architecture. With 256 physical registers, vca1 is 5% faster, vca2 is 4% faster and vca3 is just under 3% faster. With 192 physical

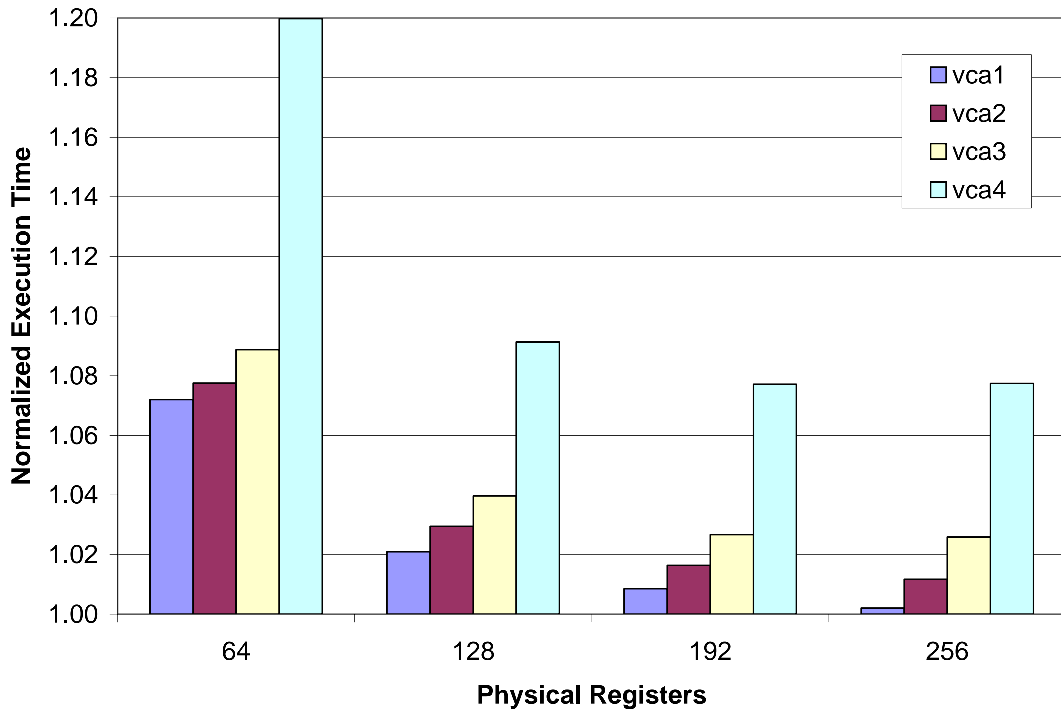



**Figure 6.4: VCA Execution Time Relative To Ideal**

The execution time of the four different virtual context architecture configurations. The execution times are normalized to the execution time of the ideal implementation with the same number of physical registers. The results are given for a range of physical register file sizes.

registers, the performance improvement jumps to 6.5% for vca1, 5.5% for vca2 and 4.5% for vca3. With 128 physical registers, the improvement is even greater. The first three configurations are between 10.5% and 9% faster. The fourth VCA configuration is slightly slower than the baseline at 256 physical registers, provides the same performance at 192 and is 4% faster at 128. The performance savings are great enough that the first three VCA configurations provide better performance with 192 physical registers than the baseline with 256. With only 128 physical registers, the first two VCA configurations provide the same performance as the 256 register baseline, while VCA configuration three is only 1% slower.

The results show that the virtual context architecture, at least the first three configurations, is able to provide a nearly ideal implementation of register windows. Figure 6.4 shows the relative execution times of the four virtual context architecture configurations compared to the ideal implementation. The execution

times are normalized to the ideal execution time with the same number of physical registers. With 256 physical registers, vca1 is within 0.2% of idea, vca2 is within 1.1% and vca3 is within 2.6%. The fourth virtual context architecture configuration does not fare as well. Even with 256 physical registers, it is nearly 8% slower than ideal. With 192 physical registers, the vca1 configuration begins to slow down relative to ideal, becoming nearly 0.9% slower. The second virtual context configuration also slows down, although the change is slightly less, with it slowing to just 1.6% slower. Interestingly, both the third and fourth configurations maintain the same relative execution time at 192 physical registers as they do at 256 physical registers. Both of these configurations include event delays to offset the cost of the extra rename logic. The delays limit these configurations from taking advantage of the additional physical registers available with 256 versus 192. Therefore, these configurations actually have approximately the same performance with both register file sizes. As the number of physical registers is decreased further, all of the configurations began to slow relative to the ideal configuration. With so few physical registers, the virtual context architecture is forced to generate many more spills and fills. The ideal architecture can handle these instantly, but the actual VCA configurations are forced to generate more data cache accesses and the pipeline slows while it waits for the accesses to complete.

Unlike the virtual context architecture, the conventional register window architecture is not able to provide good performance in this range of physical register file sizes. The conventional register window architecture is 8% slower than the baseline with 256 physical registers and almost 15% slower than ideal. The performance decreases rapidly from there. The reason for the poor performance is that even with 256 physical registers, the conventional register window implementation only has room for four register windows after reserving physical registers for the non windowed logical registers and some rename registers. The small number of windows forces many overflow and underflow conditions. When these conditions occur, a large number of loads or stores is generated, forcing the pipeline to slow down drastically while it processes them. Therefore, unlike the VCA, the con-

ventional register window performance dramatically decreases when the number of physical registers is decreased.

All of the architectures slow down as the number of physical registers is decreased, although their rates of slowdown are different. For the baseline, the number of physical registers needed for architectural state never changes. Therefore, any decrease in the number of physical registers directly decreases the number of registers available for rename. For any out-of-order pipeline to achieve good performance, it must have enough instructions in the instruction queue to find enough instruction level parallelism to issue close to the width of the pipeline. However, as you decrease the number of physical registers available for rename, the pipeline loses the ability to keep large numbers of instructions in the instruction queue. This in turn decreases the size of the instruction window that the pipeline can find independent instructions in and causes the pipeline to slowdown.

The ideal register window architecture also slows down with fewer physical registers. Although it can instantly move the unused architectural state into memory, it is still constrained to keep the used registers in the physical register file. Therefore, like the baseline architecture, as the number of physical registers is decreased it is able to keep fewer instructions in the instruction queue and reorder buffer. However, the ideal architecture is able to rename more instructions than the baseline with the same number of physical registers because it is only constrained to keep the used registers in the physical register file. This results in the ideal maintaining a higher level of performance as the number of physical registers is decreased.

Like the baseline architecture, the conventional register window architecture must also keep its architectural state in memory. For this architecture, the problem is further exacerbated by the large quantity of registers set aside for use as register windows. As the number of physical registers decreases, both the number of simultaneous register windows supported and the number of registers available for rename will decrease. This leads to the very drastic loss of performance as the number of physical registers is decreased.

Although the virtual context architecture also slows down as you decrease the number of physical registers, its rate of slowdown is less than that of the baseline. Therefore, the performance advantage that the virtual context architecture has over the baseline increases as the number of physical registers decreases. The VCA is able to take advantage of its ability to move architectural state out to memory to free more physical registers for rename. There are two situations in which this occurs. The first would be moving most of the integer or floating point state out of the register file because it is not currently being used. The second is the pipeline can dynamically balance the ratio of physical registers used for renaming and those used for architectural state. If, for example, the current section of code is a loop that uses a limited number of different logical registers, as the VCA runs out of registers for rename it will began to spill the unused logical register to memory, freeing them up for rename. This will continue until the pipeline naturally balances the cost of spilling to the number rename registers. The balance occurs when the rate at which instructions are being committed is enough to supply free registers for the rename stage.

The results in this section showed that the virtual context architecture is able to provide an execution time advantage over the baseline architecture with most of the configurations tested. The VCA has very close to ideal performance when the physical register file is relatively large. In contrast, the conventional register window architecture has very high execution times in this range of physical register file sizes. Finally, the virtual context architecture is able to take advantage of its ability to move architectural state to memory to increase its performance advantage over the baseline architecture as the number of physical registers is decreased.

**6.1.2.2 Cache Results**

The advantage of using register windows is a removal of the explicit save and restore instructions from the binary. Instead, the architectures themselves become responsible for managing this. Figure 6.5 presents the data cache results of the four architectures in a four issue pipeline with a range of physical register

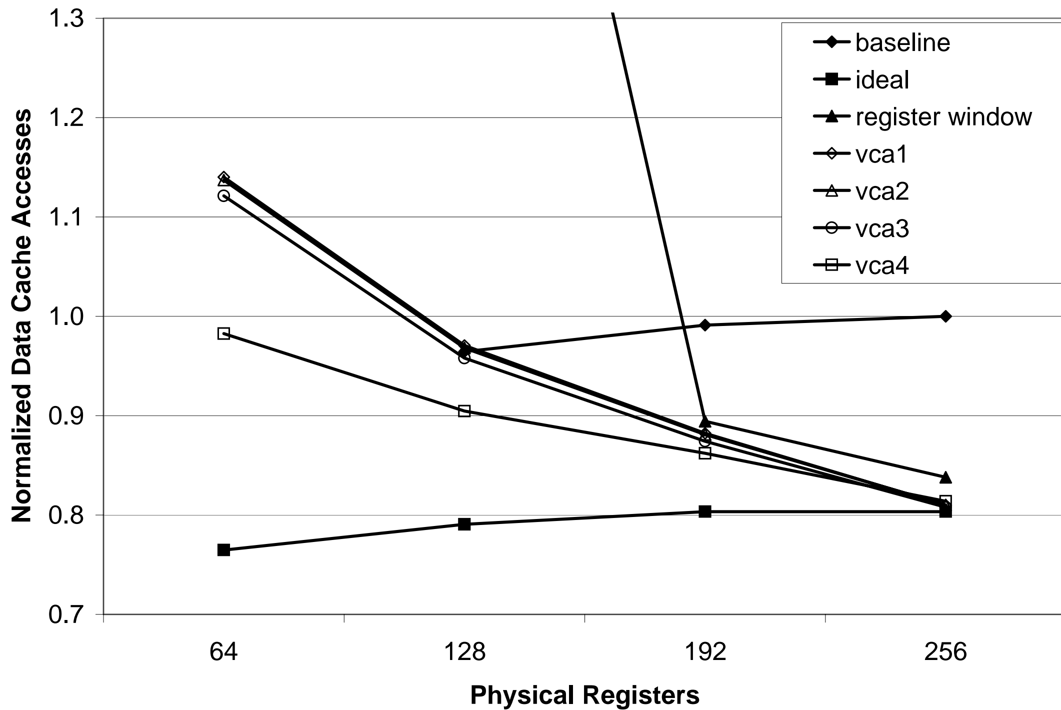



**Figure 6.5: Four Issue Data Cache Accesses Comparison**

The first level data cache accesses of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and four different virtual context architecture configurations. The accesses are normalized to the accesses of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

file sizes. The results have been normalized to the data cache accesses of the baseline binary with 256 physical registers. The data cache results are more complicated than the execution time results seen earlier. In particular, some of the architectures experience more data cache accesses as the number of physical registers is decreased. The other architectures experience the opposite effect; their data cache accesses decrease as the number of physical registers is decreased.

The two architectures that have decreasing accesses with decreasing numbers of physical registers are the baseline architecture and the ideal register window architecture. For both of these architectures, only instructions in the binary will ever generate a data cache access. Therefore, the minimum level of data cache accesses is fixed by the binary. Any additional accesses are caused by

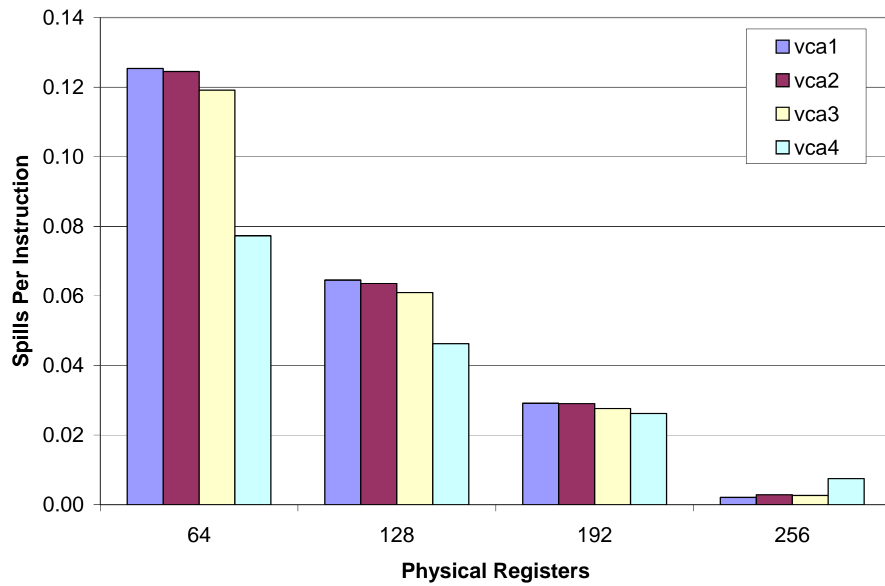
instructions executed as part of speculative execution after a mispredicted branch. The more speculative execution occurs, the more the number of data cache accesses will increase. As the number of physical registers is decreased, the execution rate is decreasing. When the execution rate slows down, fewer instructions will tend to be executed after a mispredicted branch. This leads to the phenomenon we see, the data cache results decreasing as the number of physical registers decreases.

For the virtual context architecture and conventional register window architecture, the program instructions are not the only source of data cache accesses. For these, architectural state is moved to and from memory as needed. In the case of the conventional register window processor, entire register windows need to be saved or restored on underflow or overflow conditions. As stated earlier in this section, as the number of physical registers decreases, the number of register windows that the physical register file can hold decreases. This leads directly to more underflow and overflow conditions, which in turn lead to many more data cache accesses. This effect completely overwhelms any decrease caused the execution of fewer speculative instructions. With 256 physical registers, the conventional register window processor is able to achieve a nearly 17% savings in data cache accesses. With this many physical registers, a relatively small number of overflows and underflows occur, allowing the configuration to obtain most of the cache savings that a register window binary can produce. A comparison against the ideal register window shows this to be the case. When the physical register file decreases to 192 registers, this savings drops by half to only 10%. The decrease in the number of register windows in the physical register file results in more overflows and underflows which lead directly to more data cache accesses. With only 128 physical registers, only two register windows are held in the register file. At this point, a conventional register window implementation suffers. The number of overflow and underflow events rises dramatically causing not only a large loss in performance, but a huge increase in cache traffic. With this few physical registers, the data cache accesses are nearly double those of the baseline architecture.

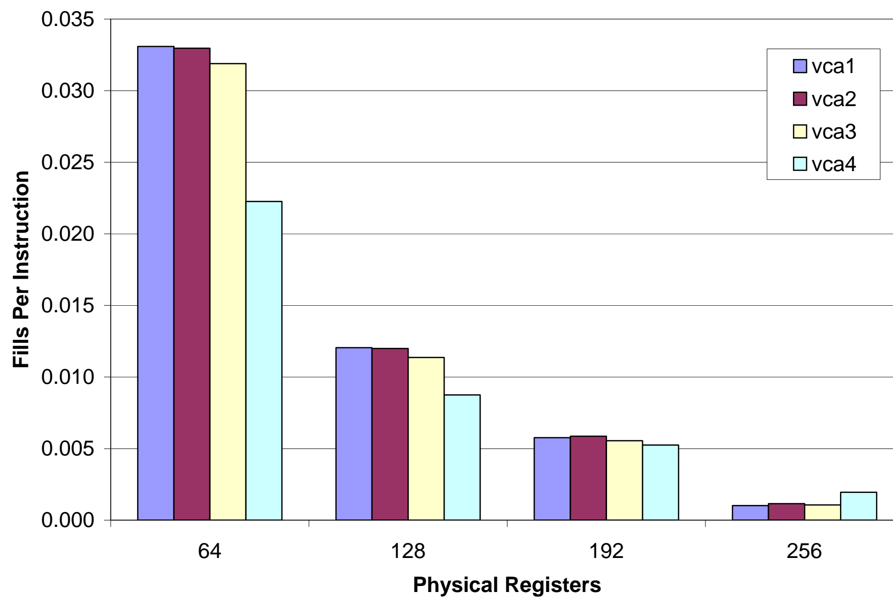
The virtual context architecture has in some ways the same cache behavior as the conventional register window architecture. The VCA also moves architectural state to and from memory. However, the virtual context architecture is able to do this on an as needed basis with single register granularity. Although it also generates more cache traffic as the number of physical registers is decreased, it does so at a much slower rate and in almost linear fashion. With 256 physical registers, all of the VCA configurations show a large reduction in data cache accesses relative to the baseline architecture with a nearly 20% reduction. This large a register file enables all of the virtual context architecture configurations to achieve within 1.5% of the data cache savings of an ideal implementation. With 192 physical registers, the VCA still provides a large savings in data cache relative to the baseline, with a 12% to 14% savings depending on configuration. The savings are slightly larger than those of the conventional register window design. With 128 physical registers, things become quite different. The conventional register window processor generated a huge number of data cache accesses at this point. The VCA also generates more cache accesses, but the level is greatly reduced. With this size physical register, the first three virtual context architecture configurations generate the same number of data cache accesses as the baseline does with this number of registers. The small physical register file causes the VCA configurations to generate enough spills and fills to remove the advantage of the register window binaries. However, they still generate somewhat fewer data cache accesses than the baseline does with 256 physical registers, while providing equivalent performance. The fourth VCA configuration still maintains a 5% improvement over the baseline with this size register file. As mentioned in Section 5.2.3, the transfer delay tends to trade performance for a decrease in transfers as the number of physical registers is decreased. The same trends continue when the physical register file is decreased to 64 registers. The increase in data cache accesses of the first three VCA configurations continues and they generate between 12% to 14% more accesses than the baseline did with 256 registers. The fourth configuration generates nearly 16% fewer accesses, and is still 2% less than the baseline with 256.

The data cache access behavior is strongly dependent on the generation of spills and fills. Figure 6.6 presents the spill and fill rate of the virtual context architecture. The rates are given as the number of spills or fills per committed instruction. As expected, the rates are very low with large physical register files, but rapidly increase as the number of physical registers is decreased. The first three VCA configurations generate between 2 and 3 spills per one thousand committed instructions with 256 physical registers. The fourth configuration generates over twice as many with approximately 7.5 spills per one thousand instructions. With 192 physical registers, all four configurations generate almost exactly the same number of spills of around 3 spills per one hundred instructions. The decrease in physical registers has caused the spill generation to increase by an entire order of magnitude. Unlike 256 physical registers, the fourth configuration is actually generating slightly fewer spills than the other three. As mentioned, the transfer delay implemented in the fourth configuration tends to limit the numbers of spills and fills at smaller physical register file sizes. This trend becomes even more pronounced with smaller physical register file sizes. With 128 physical registers, the rate of the first three configurations doubles to just over 6 spills per one hundred instructions, while the fourth configuration only generates 4.5 spills. With 64 physical registers the difference becomes even greater. The first three configurations generate around 12.5 spills per one hundred instructions (with the third configuration generating slightly fewer than the first two). The fourth configuration only generates 8 spills per one hundred instructions.

The fill rates show similar trends as the spill rates, but the rate of fills is much lower. A fill is only needed when architectural state has been moved out of the physical register file and into memory by a spill. However, a spill can move state out to memory and if the program never needs that state or overwrites the value of that logical register, a fill will not result from that spill. Therefore, the number of fills will always be less than the number of spills. With 256 physical registers, the first three VCA configurations generate 1 fill per one thousand instructions, while the fourth once again generates more, with 2 fills per thousand instructions. With 192




Spill Rate




Fill Rate

**Figure 6.6: Spill And Fill Rate**

The spill and fill rate of the four virtual context architecture configurations. The rates are given as spills or fills per committed instruction. The results are presented for a range of physical register file sizes.

physical registers, once again the rate is the same for all four configurations, with 5 fills per one thousand instructions. This is only one fifth as many fills as spills. This ratio continues at 128 physical registers, where the first three configurations generate approximately 12 fills per thousand instructions with the fourth generating only 8. With 64 physical registers, the rates become 3 fills per hundred instructions for the first three and 2 fills per hundred instructions for the fourth configuration. As the results show, the rates of spills and fills are very tightly coupled. Both rates show the exact same trends with changes in the number of physical registers and between the different configurations tested. This is expected because of the nature of spills and fills, with a fill in general only being needed when a value was previously spilled.

The virtual context architecture also shows a slight decrease in instruction cache accesses with respect to the baseline architecture with 256 physical registers, with an average decrease of 2% and a maximum decrease of 5%. This is easily explainable because of the decreased dynamic path length of the register window binaries. However, with fewer physical registers this is no longer the case. With 192 physical registers, the baseline and virtual context architecture have nearly identical numbers of instruction caches. Similar to the data cache accesses, this is explainable by the decrease in speculative execution attributable to a slow down in the execution rate. As the data presented in Section 6.1.2.1 showed, the baseline architecture tends to slow down faster than the VCA as you decrease the number of physical registers. With 128 physical registers, the baseline architecture accesses the L1 data cache almost 5% less than the virtual context architecture configurations.

The virtual context architecture does show a dramatic increase in the number of second level cache accesses compared to the baseline architecture (25% to 45%). However, the ideal register window processor shows an identical increase (43%). Therefore, the increase is related to the register window binaries, and not any overhead of the VCA. A quick look at the statistics for the individual benchmarks shows that most of the difference is attributable to the eon benchmark. The L2 cache accesses for this benchmark are over seven times higher with any of the

register window architectures compared to the baseline. Eon generates the fewest L2 cache accesses by over an order of magnitude. Therefore, a slight change in the absolute number has a huge impact on the relative numbers. For all four virtual context architecture configurations at all four sizes of physical register file, the VCA at most generates less than 2% more accesses than the ideal implementation, and often generates fewer. This shows that the extra cache accesses generated by spills and fills does not adversely affect the cache hit rate.

The advantage of using register windows is a removal of the explicit save and restore instructions from the binary. The results in this section showed that the virtual context architecture is able to provide most of this savings with larger physical register file sizes, providing a nearly 20% data cache access savings over the baseline with 256 physical registers. With a smaller number of physical registers, the VCA generates more spills and fills and the cache savings is decreased.

### 6.1.3 One Data Cache Port

This section provides a detailed comparison of the virtual context architecture to the baseline, ideal, and conventional register window architectures for a four issue pipeline with only one data cache port. The results are presented in two sections. The first section presents the execution time results. The second section presents the data cache results.

#### 6.1.3.1 Execution Time Results

This sections presents the execution time results of a four issue our-of-order pipeline with a single data cache port. Figure 6.7 shows the execution time results. The results in general are similar to the results for two data cache ports. The register window architecture performs worse than the baseline architecture, and the performance becomes drastically worse as the number of physical registers is decreased. The virtual context architecture still shows a performance advantage over the baseline at every physical register file size. However, the virtual context architecture is able to take advantage of the reduction in data cache accesses versus the baseline to provide an even larger performance advantage,

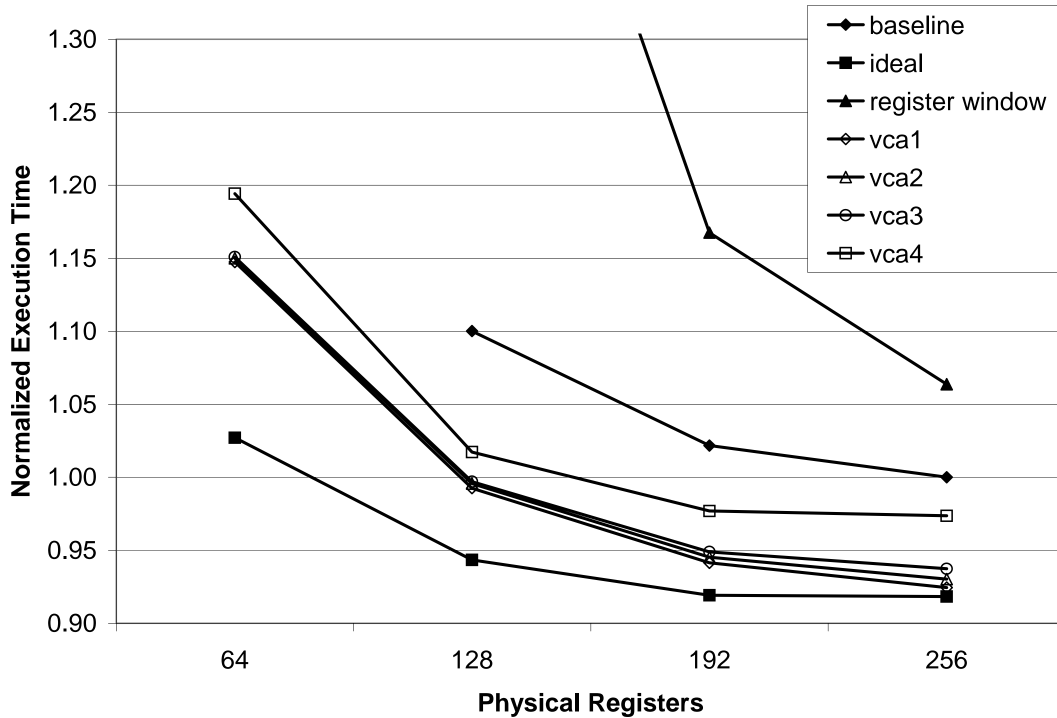



**Figure 6.7: Four Issue One Data Cache Port Execution Time**

The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and four different virtual context architecture configurations with a four issue pipeline with one data cache port. The execution times are normalized to the execution time of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

especially at 256 physical registers. The first VCA configuration is almost 8% faster than the baseline in the one data cache port pipeline versus only 5% faster in the two port pipeline. Both vca2 and vca3 see a similar decrease in execution time. The fourth configuration has an even greater difference. The difference is great enough to realize a 2% performance improvement over the baseline architecture. In contrast, vca4 was slower than the baseline in the two data cache port pipeline. With 192 physical registers this trend continues, with the first three configurations still showing a 2% to 2.5% improvement in the relative performance compared to the same configurations with two data cache ports. The fourth configuration retains the same advantage. With a smaller physical register file, things began to change for the first three configurations. With 128 physical registers, the performance difference drops to between no difference and a 1.5% improvement.

With 64 physical registers, the trend reverses. The relative performance is between 0.5% and 1.7% worse than the relative performance with two data cache ports. The behavior of vca4 is very different. This configuration continues to see an improvement relative to the results with two data cache ports. With 128 physical registers, the improvement is still 4.5% better, and with 64 physical registers the improvement actually increases to over 6%.

In a pipeline with a single data cache port, the number of data cache accesses is going to play a critical role in the overall performance. With a large physical register file, the VCA will decrease the overall number of data cache accesses relative to the baseline. The cache accesses removed by register windows are register save and restore instructions. These instructions are saving/loading their values to/from locations on the stack. These accesses will tend to hit in the data cache. Therefore, the instructions do not take a significant amount of time to execute (versus if they tended to miss in the data cache). However, with only a single data cache port, the bandwidth to the cache is limited. Therefore, any memory instructions will tend to be more expensive to execute. This means that removing memory instructions in this situation will tend to have a larger impact on the execution time. This explains the results for one data cache port. If one looks at the rate at which instructions were executed (measured in instructions per clock or IPC), with two data cache ports, the rate of instruction execution is 3% slower in the ideal architecture than the baseline. The removal of the relatively quick to execute stack accesses increases the average execution time of instructions. The decrease in dynamic path length can more than make up for it, leading to the 5% decrease in execution time for the idea case with 256 physical registers. However, with only one data cache port, the IPC of the ideal architecture is the same as the baseline. Because the execution of the save and restore instructions is more expensive given the limited cache bandwidth, their removal doesn't change the average execution time. The advantage of the decreased dynamic path length is fully expressed. This accounts for the increase in relative execution time that the virtual context architecture enjoys with a single data cache port.

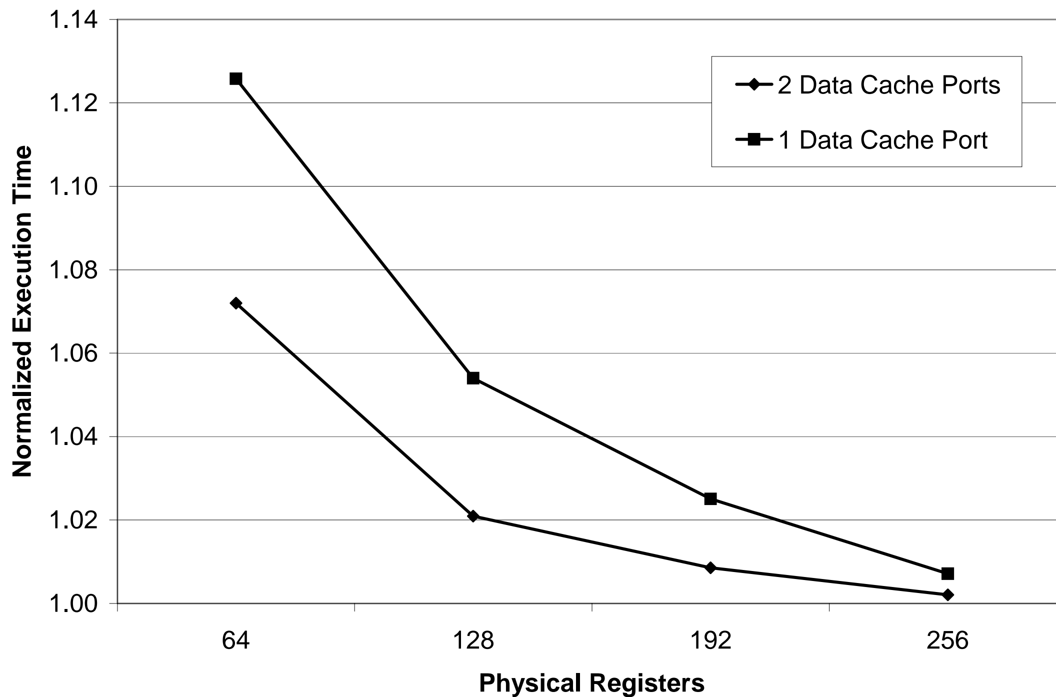



**Figure 6.8: Execution Time Relative To Ideal**
The relative execution time of vca1 to ideal in a pipeline with two data cache ports and in a pipeline with one data cache port. The execution times are normalized to the execution time of the ideal register window architecture with the same number of physical registers.

The removal of one data cache port also affects the performance of the virtual context architecture relative to the ideal. Figure 6.8 presents the execution time of vca1 relative to the execution time of the ideal register window configuration. The results are given for a pipeline with two data cache ports and for a pipeline with one data cache port. As previously noted, with a single data cache port, any type of memory access is going to cost more performance. The ideal architecture never generates additional data cache accesses for its spills and fills. In contrast, the virtual context architecture generates additional cache traffic for spills and fills. With 256 physical registers, although only a small number of spills and fills are generated, the difference in expense is enough for vca1 to be 0.7% slower than ideal with one data cache port versus only 0.2% slower with two data cache ports. The relative difference increases as the number of physical registers is reduced, forcing the VCA to generate more and more spills and fills. With 192 physical reg-

isters, the relative difference between one port and two ports grows to 1.6%. With 128, it increases to 3.3% and with only 64 physical registers it becomes over a 5% difference.

The relative performance difference of the various virtual context architecture configurations is also affected by the removal of one data cache port. In every case, the performance differences between vca1 and the other three configurations is smaller with one data cache port than with two data cache ports. The two port results show that all the configurations maintain an almost constant difference between themselves. This is not the case with only one data cache port, especially for vca2 and vca3. These two configurations with 256 physical registers show less of a performance decrease with respect to vca1 when the pipeline only has a single data cache port. As the number of physical registers is decreased, this difference shrinks even further, until the first three configurations display almost the exact same performance. The behavior of the fourth configuration is somewhat different. As mentioned, at every size of physical register file, the performance difference between it and vca1 is smaller with one data cache port than with two. The fourth configuration follows a somewhat similar trend, with the difference decreasing as the number of physical register is decreased. However, with only 64 physical registers, the performance difference increases back to the same level it was with 192 physical registers. The increase is probably related to the capability of the transfer delay in limiting the number of spills and fills generated. As the physical register file size is decreased, the performance influence of this capability increases relative to the cost associated with delaying on each transfer. However, when the physical register file size reaches 64, the balance probably begins to shift the other way, with the cost becoming more of a factor.

The virtual context architecture is able to provide a performance advantage over the baseline architecture. The advantage is greater when the pipeline has a single data cache port. The performance advantage is great enough for the virtual context architecture to achieve the same performance with a single data cache port that the baseline needs two data cache ports to achieve. Figure 6.9 presents

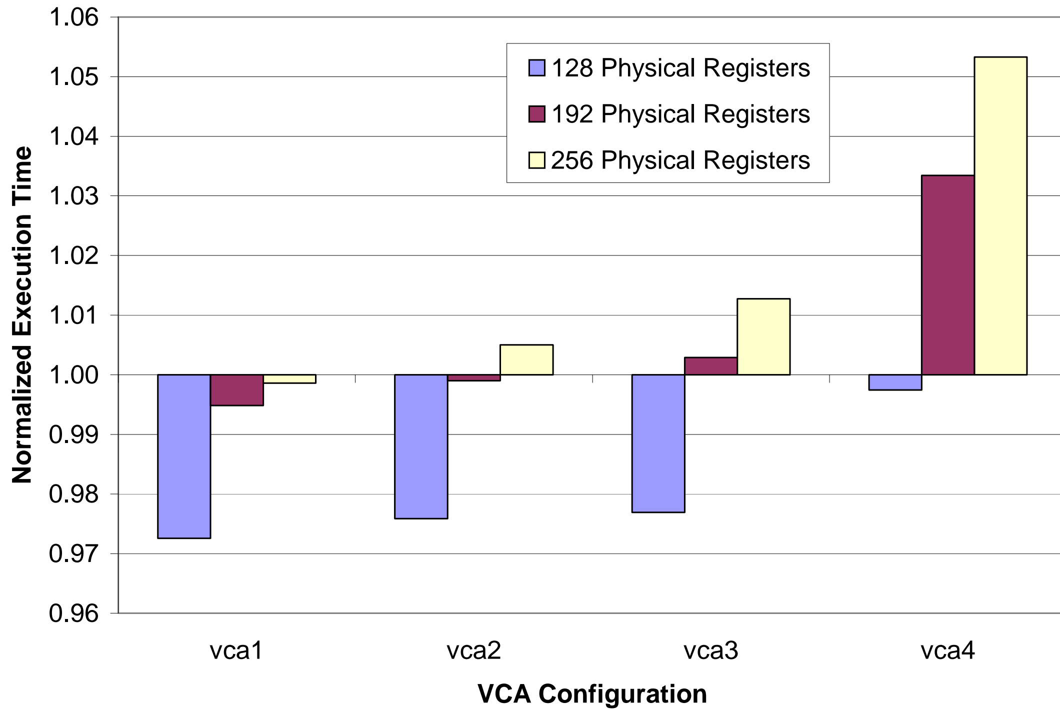



**Figure 6.9: One Port VCA Versus Two Port Baseline**

The execution time of the four virtual context architectures on a pipeline with one data cache port. The results are normalized to the execution time of the baseline architecture on a pipeline with two data cache ports with the same number of physical registers. The results are given for a range of physical registers.

the execution time of the virtual context architecture on a pipeline with one data cache port normalized to the execution time of the baseline architecture with two data cache ports. All four VCA configurations outperform the baseline when the physical register file is 128 physical registers. With this size register file, the first three configurations are almost 2.5% faster even though the pipeline has only one data cache port. With 192 physical registers, the first three configurations show nearly the same performance as the baseline. The fourth configuration is over 3% slower. This trend continues with 256 physical registers. The first configuration provides the same performance, vca2 is now 0.5% slower and vca3 is over 1% slower. The fourth configuration has also slowed further; relative to the baseline, it is now over 5% slower.

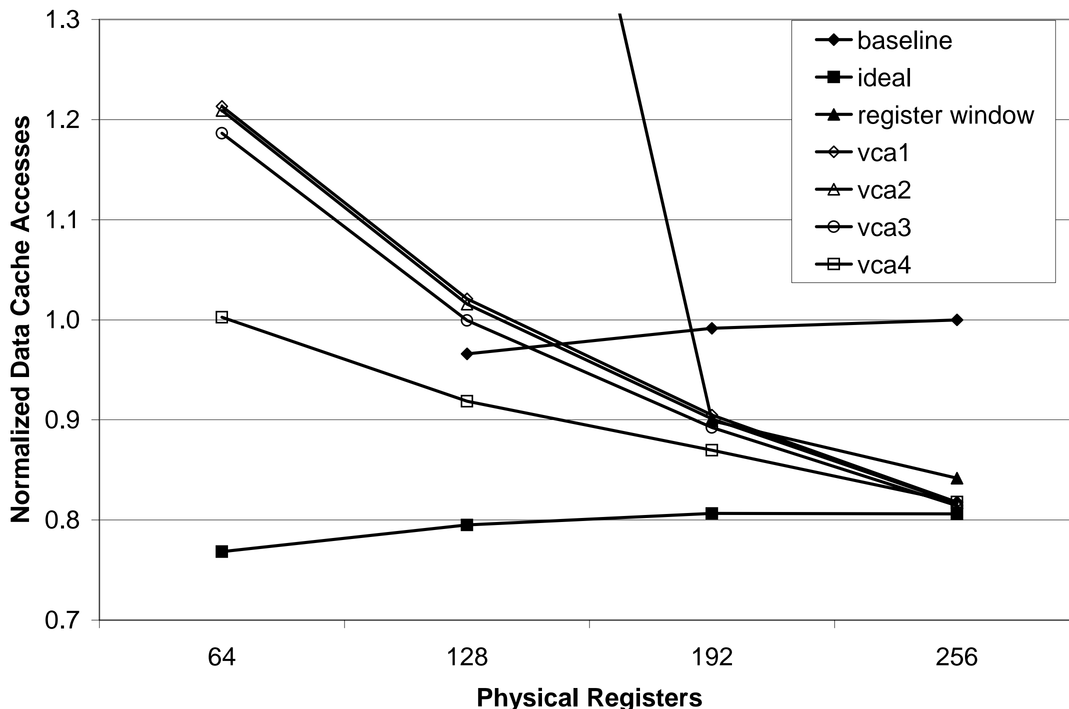



**Figure 6.10: Four Issue One Data Cache Port Data Cache Accesses**
The data cache accesses of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and four different virtual context architecture configurations with a four issue pipeline with one data cache port. The data cache accesses are normalized to the data cache accesses of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

### 6.1.3.2 Data Cache Results

The data cache accesses are also affected by the number of data cache ports. Figure 6.10 presents the data cache access results for a pipeline with one data cache port. As already seen in the execution time results, the data cache results for one data cache port are very similar to the results seen for a pipeline with two data cache ports. In fact, the results for all the architectures except for the virtual context architecture are almost exactly the same. In all of these architectures, the non speculative data cache accesses are a constant, or in the case of a conventional register window architecture, a constant for a given number of physical registers. Therefore, the slight differences in cache accesses for these architectures

are directly attributable to the variance of the number of speculative instructions executed with the execution rate of the pipeline.

The data cache accesses for the virtual context architecture are not constant. Instead, they are dependent on a complex interplay between the size of the physical register file, the execution rate of the pipeline and the configuration of the VCA architecture. In particular, the rate at which spills and fills can be completed affects the number of spills and fills generated. The one data cache port pipeline has much less bandwidth to the first level data cache. Therefore, in general, spills and fills will tend to take longer to complete. This will tend to slow down the execution, resulting in more pressure on the physical register file as more instructions become resident in the reorder buffer. This increased pressure leads to more spills, which results in more fills.

This trend is clear for the first three virtual context architecture configurations. With 256 physical registers, only a small number of spills and fills are generated, and we see the one port pipeline generating less than 1% more data cache accesses relative to the baseline architecture than the pipeline with two ports. Although the generation of spills and fills is small, the one data cache port pipeline will execute more slowly than the two, increasing the pressure on the physical register file. The difference grows as we decrease the size of the physical register file. With 192 physical registers, the difference becomes 2%. This grows to 5% with 128 physical registers and over 7% with 64 physical registers. The fourth virtual context architecture also shows this trend, but at a much reduced level. The ability of this configuration to restrict the generation of spills and fills results in a difference of less than 0.5% with 256 physical registers, 0.8% with 192, 1.4% with 128 and less than 2% with 64.

### 6.1.4 Summary

The results in this section show that the virtual context architecture is able to provide a nearly ideal implementation of register windows. In contrast to a conventional register window design, the VCA provides a performance advantage over the baseline architecture even with small physical register file sizes. Even

|  | Rename Table Size | Rename Table Ports | Extra Logic Cost |
|---|---|---|---|
| vca1 | 64x4 | 12 | none |
| vca2 | 64x3 | 10 | extra stage |

**Table 6.2: VCA Configurations For The Eight Issue Pipeline**
The virtual context architecture configurations studied for the eight issue pipeline. The configurations represent a range of implementations. The configurations are composed of two things. The first is the size and ports of the rename table. The second is the cost assumed for the extra logic in the rename table.

with an extra stage added, the virtual context architecture reduces execution time versus the baseline architecture with the same number of physical registers between 4% and 10% for a pipeline with two data cache ports and 7% to 10.5% for a pipeline with one data cache port. The improvement in performance is great enough that vca2 is 3%-5% faster with 192 physical registers than the baseline architecture with 256. The VCA has the same execution time with 128 physical registers as the baseline does with 256. These reductions are enough that the VCA can provide nearly identical performance over the range of physical registers studied on a pipeline with only one data cache port to the baseline architecture with two data cache ports. The virtual context architecture improves relative to the baseline as hardware becomes more restricted; that is, when either the number of physical registers or the number of data cache ports is reduced.

## 6.2 Eight Issue Pipeline

This section presents the performance of the virtual context architecture in an eight issue out-of-order pipeline. The purpose of this study is to examine the differences that a wider pipeline make in the performance of the virtual context architecture. Two different VCA configurations were studied, see Table 6.2. The configurations are equivalent to the first two configurations for the four issue pipeline. The only difference is the increase in the number of rename table ports because of the increased width of the pipeline. Everything else is the same. The other configurations are not repeated in this study because the relative perfor-

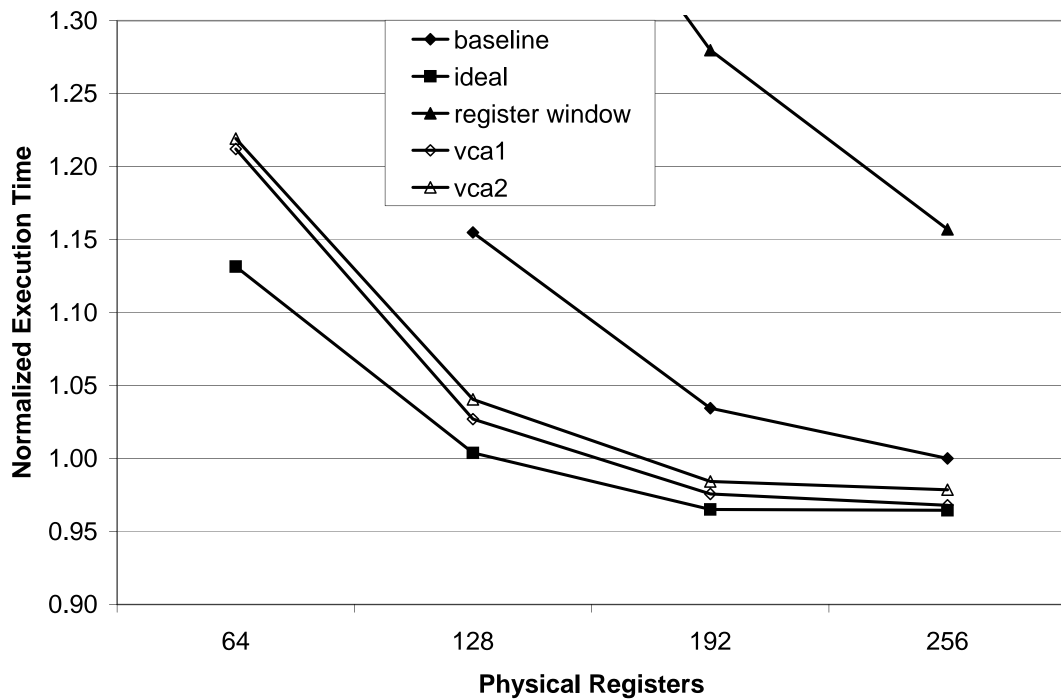



**Figure 6.11: Eight Issue Three Data Cache Ports Execution Time**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and two different virtual context architecture configurations with an eight issue pipeline with three data cache ports. The execution times are normalized to the execution time of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

mance characteristics were shown in the four issue pipeline. The results are presented for the same range of physical registers as was studied in the four issue results. The size of the reorder buffer is the same in the two pipelines, and therefore the number of physical registers needed by the baseline to provide no stalls is the same. Results are presented for pipelines with three, two and one data cache port.

### 6.2.1 Three Data Cache Ports

The execution time of the architectures with three data cache ports is presented in Figure 6.11. The results show trends similar to the four issue pipeline with two data cache ports. However, the execution times relative to the baseline architecture with 256 physical registers are all slower in the eight issue pipeline

than in the four issue. This is true for every architecture and with every physical register file size. In almost every case, the difference increases as the number of physical registers is decreased. As the number of physical registers is decreased, the number of registers available for renaming decreases.

For any out-of-order pipeline to achieve good performance, it must have enough instructions in the instruction queue to find enough instruction level parallelism to issue close to the width of the pipeline. It is always the case that with a fixed number of instructions available to execute, the pipeline is much more likely to find four independent instructions than to find eight. For example, assume a theoretical stream of instructions. For simplicity assume that for every eight instructions in the instruction window, one independent instruction can be found. Therefore, with a 64 instruction window we have 8 independent instructions, but with a 32 instruction window we have only 4 independent instructions. With a four issue pipeline, both a 32 and 64 instruction window can provide the four instructions it needs. However, the eight issue pipeline will only be able to issue four instructions if it has a 32 instruction window, costing it half its performance.

This effect is visible in this chart for all the architectures. Consider the baseline architecture. With 192 physical registers, the execution time of the four issue increases by 2% relative to 256 physical registers, while for an eight issue this increase is 3.5%. With 128 physical registers, the four issue is 10% slower while the eight issue is 15.5% slower. The only instance that does not see this increasing decline in performance relative to the four issue is the conventional register window. With 192 physical registers compared to 256 physical registers, the eight issue is 10% slower relative to the four issue. However, with 128 physical registers, this percentage drops to less than 5%. With so few register windows in the physical register file, this architecture benefits more from the extra data cache port than it loses from the lack of rename.

The results for the virtual context architecture on this pipeline are mixed. With 256 physical registers, vca1 is only a little over 3% faster than the baseline architecture while vca2 is only a little over 2% faster. This compares to 5% and 4% respectively for the four issue pipeline with two data cache ports. The larger num-

ber of data cache ports decreases the relative cost of issuing loads and stores, decreasing the impact register windows has on the performance. This is confirmed by the results of the ideal register window architecture. Similar to the four issue with two data cache ports, both VCA configurations are very close to ideal with 256 physical registers, with 0.2% and 1.5% respectively. The virtual context architecture also loses performance more quickly in the eight issue pipeline when the number of physical registers is decreased. With 128 physical registers, vca1 is over 2.5% slower than the baseline with 256 physical registers. In the four issue pipeline the two had the same performance. However, the decline is less rapid than occurs for the baseline architecture. The ability of the virtual context architecture to dynamically balance the number of physical registers used for rename allows it to not lose as much performance. With 128 physical registers, the difference between vca1 and the baseline architecture is 13% while in the four issue it was only 10%.

The data cache accesses for the virtual context architecture show trends similar to the execution time. Figure 6.12 presents the data cache access results for an eight issue pipeline with three data cache ports. The data cache access behavior of the various architectures is similar to that seen for a four issue pipeline. Unlike the execution time, the architectures can be divided into two distinct groups based on relative cache performance compared to the four issue pipeline. All of the architectures except for the virtual context architecture see a decrease in the data cache accesses relative to the change seen in a four issue pipeline. As mentioned previously in this chapter, these architectures produce a constant number of non speculative cache accesses (the conventional register window architecture is constant, but dependent on the size of the physical register file). The results show that all of the architectures lose performance faster with decreases in the physical register file size in the eight issue pipeline as compared to the results for a four issue pipeline. Therefore, the decrease in relative data cache accesses can be attributed to the decrease in speculative execution caused by the greater slowdown of the execution.

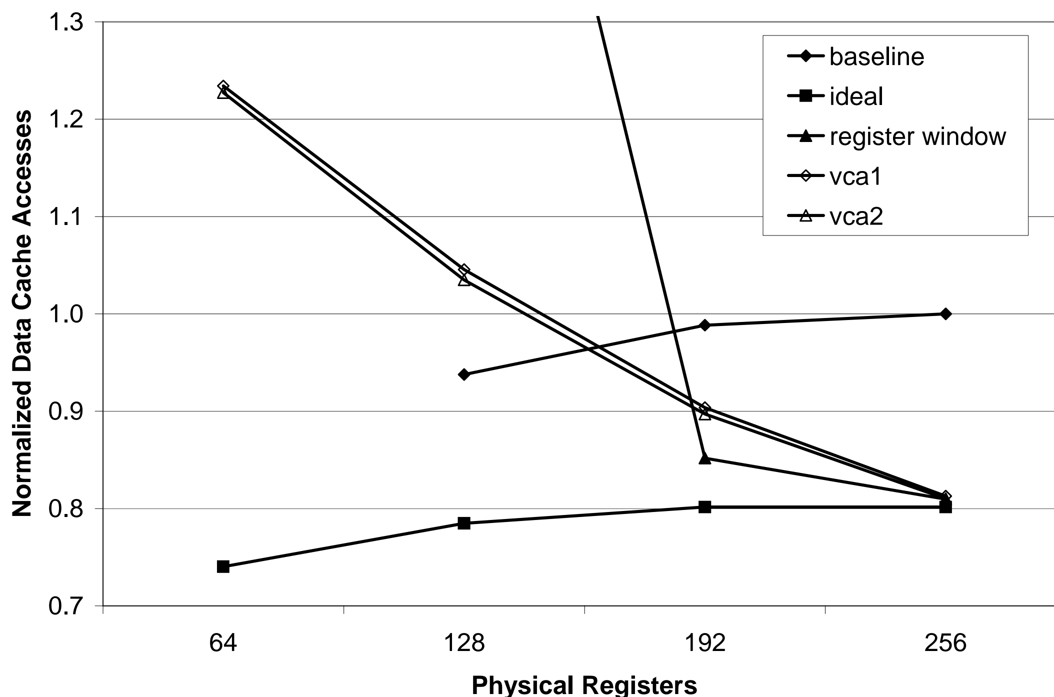



**Figure 6.12: Eight Issue Three Data Cache Ports Data Cache Accesses**
The data cache accesses of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and two different virtual context architecture configurations with an eight issue pipeline with three data cache ports. The data cache accesses are normalized to the accesses of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

The virtual context architecture does not generate a constant number of non speculative data cache accesses. The virtual context architecture dynamically balances the number of physical registers used for rename and the number of registers used for architectural state. As mentioned previously in this section, the VCA is able to use this to help with the decline in performance that all of the architectures experience with decreasing physical register file sizes. The cost of this balancing is the generation of spill and fill traffic to the data cache. In an eight issue pipeline, it is desirable to try to keep as many registers as possible used for renaming. This will keep as many instructions as possible in the instruction queue, increasing the number of independent instructions that can be found. The VCA is able to adjust to this situation, generating more cache accesses but decreasing the performance loss. With 192 physical registers, the virtual context architecture

generates 2% more relative data cache accesses than in a four issue pipeline. The difference climbs to 7.5% with 128 physical registers and 9% with 64.

The results of this section show that the performance of all of the architectures declines faster when reducing the number of physical registers than was seen with a four issue pipeline. The large number of data cache ports reduces the potential performance savings. Thus, the virtual context architecture shows only a small performance advantage over the baseline with 256 physical registers. However, the virtual context architecture is able to effectively trade additional cache accesses for reduced execution time as the number of physical registers is decreased. Therefore, with a small physical register file size, the virtual context architecture provides a 13% decrease in execution time relative to the baseline architecture.

### 6.2.2 Two Data Cache Ports

This section presents the results of an eight issue pipeline with two data cache ports. The results for this pipeline are very similar to those with three data cache ports. With 256 physical registers, the relative execution time for the virtual context architecture configurations are almost identical to those of the four issue pipeline with two data cache ports with a 5% decrease in execution time for vca1 compared to the baseline. The slowdown in relative execution time as compared to the four issue pipeline results are still present, but somewhat diminished. With only two data cache ports, the overall performance of the pipeline is decreased. The decreased performance lowers the pressure on the pipeline to supply a large number of independent instructions each cycle. This reduces the performance loss as the number of physical registers is reduced. With 128 physical registers, the virtual context architecture with two data cache ports is just over 1.5% slower than the baseline with 256 physical registers while with three data cache ports it is nearly 3% slower.

As seen in the four issue results, the relative execution times of the virtual context architecture are improved when the number of data cache ports is reduced. The difference decreases as the number of physical registers is decreased. With

256 physical registers, vca1 with two ports is 5% faster than the baseline, while with three ports it's only 3% faster. With 192 physical registers, vca1 with two ports is 7% while three ports is 6%. With 128 physical registers, the relative execution times are identical.

The execution time of the virtual context architecture relative to the ideal register window architecture remains nearly the same when moving from three data cache ports to two data cache ports. With 256 physical registers, vca1 with three data cache ports is 0.3% slower than ideal while vca1 with two data cache ports is 0.4% slower. The other three physical register file sizes show an almost constant difference between three ports and two ports. The vca1 with two ports is an additional 0.5% slower than ideal than in a pipeline with three ports. The removal of one data cache port makes the relative cost of a data cache access slightly higher. This not only accounts for the improved relative performance mentioned in the previous paragraph, but also for the increase in execution time relative to the ideal register window, which never accesses the cache for spills or fills. With 256 physical registers, the virtual context architecture generates so few spills and fills that the additional cost barely affects performance. When the number of spills and fills increases, the difference becomes greater.

The data cache accesses also show similar behavior with two data cache ports as was seen with three data cache ports. All the architectures except for the virtual context architecture generate nearly identical relative data cache accesses with two ports compared to three ports because the non speculative data cache accesses are constant for these architectures. The small differences in relative execution time of these architectures results in a negligible difference in the number of speculatively executed instructions. The virtual context architecture shows a slight increase in relative data cache accesses as the number of physical registers is decreased. With 256 physical registers, the difference is 0.5% more relative cache accesses with two data cache ports. The number grows until with 64 physical registers it becomes 2.5%. The decrease in ports causes the pipeline to slow, which in turn puts more pressure on the physical register file. This increased pressure results in the slight increase in spill and fill traffic. As the number of spills and

fills increases, more pressure is put onto the data cache ports. With one less port, the added pressure causes the pipeline to slowdown. The slowdown puts even more pressure on the physical register file. This cycle eventually reaches an equilibrium because of the limited resources in the ASTQ and physical register file.

### 6.2.3 One Data Cache Port

This section presents the results of an eight issue pipeline with a single data cache port. With only a single data cache port, the relative cost of a cache access becomes very high. The pipeline slows down drastically compared to pipelines with two and three data cache ports, especially with a large physical register file. With 256 physical registers, the average execution time of the benchmarks increases by nearly 25% for a single cache port compared to a pipeline with three ports. The increase is nearly 20% for a single port compared to a two port pipeline. These factors have a significant effect on the results.

The execution time results for this pipeline are presented in Figure 6.13. The basic trends seen so far are still present. All of the architectures slow down as the size of the physical register file is decreased. However, the rate of slowdown for the architectures has changed. Both the baseline architecture and ideal architectures are able to maintain a higher relative level of performance in this pipeline than in the previous eight issue pipelines studied. For example, with 128 physical registers the baseline architecture is 11% slower than with 256 physical registers. With two and three data cache ports, the slowdown is 14% and 15.5% respectively. Because the performance of this pipeline is so much slower to began with, restricting the number of rename registers has much less of an effect. The same behavior is seen in the ideal register window architecture. With 64 physical registers and one data cache port, the ideal architecture is 13% slower than with 256 physical registers. This compares with 16% and 17% for the two and three data cache port pipelines.

This is not the only change in behavior for the ideal register window architecture in this pipeline. As noted previously, the relative execution time of architectures that support register windows is decreased (improved) as the number of

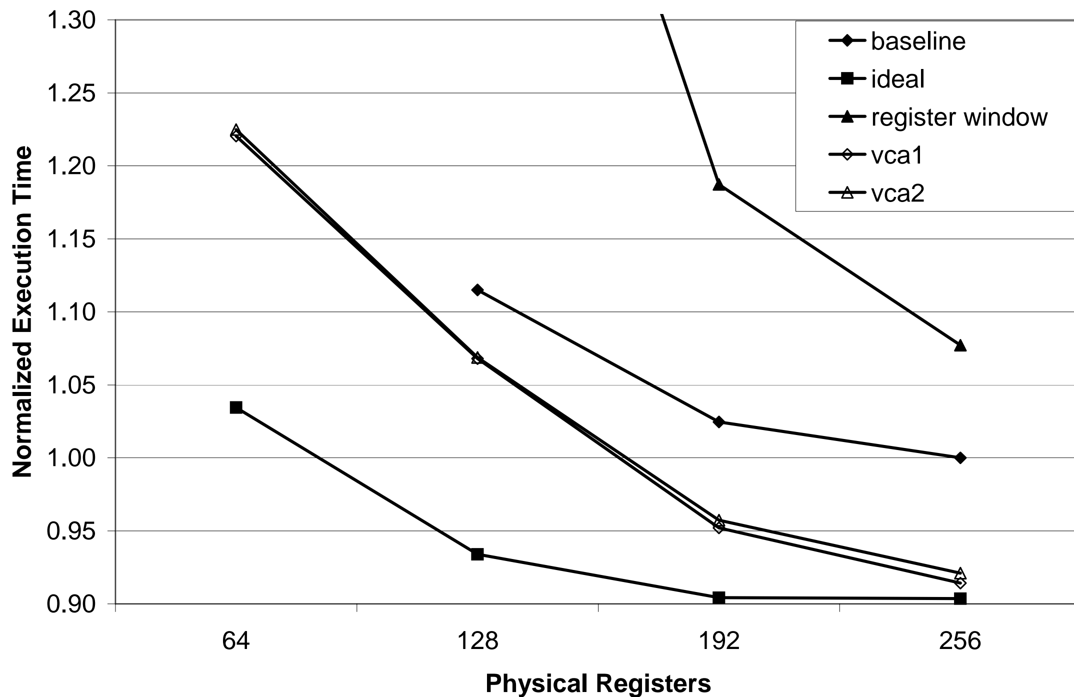



**Figure 6.13: Eight Issue One Data Cache Port Execution Time**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and two different virtual context architecture configurations with an eight issue pipeline with one data cache port. The execution times are normalized to the execution time of the baseline architecture with 256 physical registers. The results are given for a range of physical register file sizes.

data cache ports is decreased. With fewer data cache ports, the performance impact of data cache accesses is significantly increased. Therefore, the performance impact of removing some of these accesses becomes greater too. The ideal architecture experiences a nearly 5% drop in relative execution time to the baseline with respect to a two cache port pipeline. With 256 physical registers, the ideal register window is nearly 10% faster than the baseline, while with two data cache ports the ideal is only 5% faster than the baseline.

The conventional register window architecture also experiences this. With 256 physical registers it is 8% slower than the baseline in a pipeline with one data cache port versus nearly 14% slower than the baseline in a pipeline with two data cache ports and 16% slower with three data cache ports. The improvement is still present with 192 physical registers, but disappears with 128 physical registers.

With so few register windows in the physical register file, the architecture is forced to generate nearly double the cache accesses of the baseline architecture due to overflow and underflow conditions. In contrast, the conventional register window still provides a cache access savings with 192 physical registers of nearly 10%. Since the architecture generates even more accesses than the baseline, it suffers due to the increased cost of cache accesses instead of benefitting from it like it did with more physical registers.

The relative performance of the virtual context architecture is also greatly influenced by the increased cost of data cache accesses. Its behavior is reminiscent of the behavior of the conventional register window architecture. With a relatively large physical register file, the virtual context architecture generates very few spills and fills and therefore benefits from the increased cost of data cache accesses. However, like the conventional register window architecture, when the physical register file reaches a certain size, the VCA actually begins to generate more cache accesses than the baseline architecture. At this critical point, the high cost of data cache accesses that used to be an advantage becomes a disadvantage. The combination of all the factors causes the virtual context architecture in the eight issue pipeline with a single data cache port to behave in a way unlike all the other pipeline configurations, both four and eight issue. Specifically, as the number of physical registers is decreased, the performance advantage the virtual context architecture provides over the baseline architecture decreases. Figure 6.14 presents the execution time of the first virtual context architecture normalized to the execution time of the baseline with the same number of physical registers. The results are presented for all three eight issue pipeline studied. With 256 physical registers, vca1 is 8.5% faster than the baseline architecture. This compares to only 5% with two data cache ports and 3% with three data cache ports. With 192, the advantage vca1 has over the baseline decreases to 7.2% while with two ports it has increased to 6.8% and three ports has increased to almost 6%. With 128 physical registers, with one port the advantage is now only slightly over 4% while with two and three ports the advantage has grown to 11%.

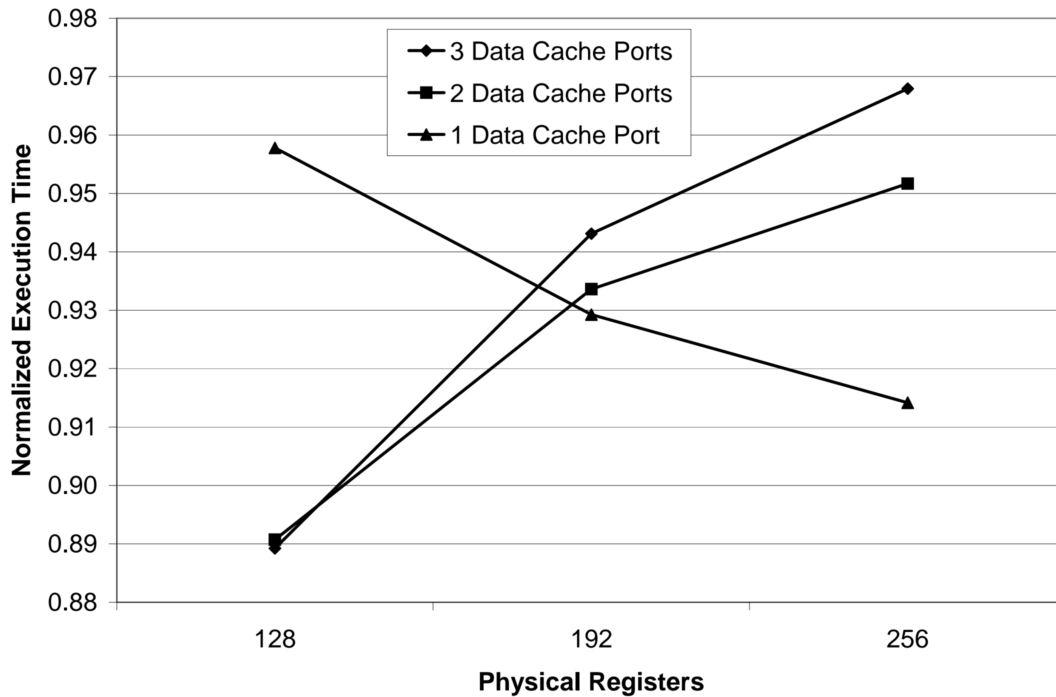



**Figure 6.14: Eight Issue VCA Execution Time Improvement**
The execution time of the first virtual context architecture configuration normalized to the execution time of the baseline architecture with the same number of physical registers. The results are given for an eight issue pipeline with three data cache ports, two data cache ports and one data cache port. A range of physical registers is shown.

The increased cost of cache accesses also effects the relative performance of the virtual context architecture to the ideal register window architecture. The virtual context architecture is forced to generate data cache accesses to implement the spills and fills necessary. However, the ideal never generates any additional cache accesses. Therefore, the greater the cost of cache accesses, the greater the difference between the VCA and the ideal. Figure 6.15 presents the execution time of the first virtual context architecture configuration normalized to the execution time of the ideal register window architecture with the same number of physical registers. The results are given for all three eight issue pipelines studied. With 256 physical registers, vca1 is 1% slower than ideal compared to only 0.4% slower with two and three ports. With 192 physical registers, the differences start to increase, with the one port becoming over 5% slower, while two ports is only 1.5% slower and three ports is within 1%. The difference increases dramatically

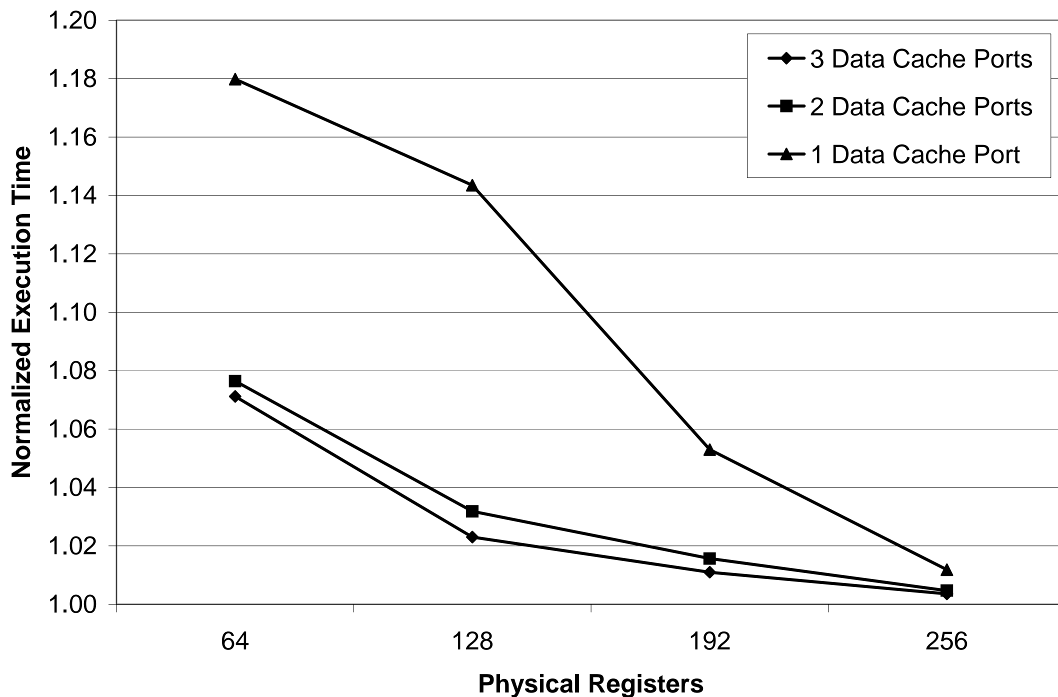



**Figure 6.15: Eight Issue VCA Execution Time Versus Ideal**
The execution time of the first virtual context architecture configuration normalized to the execution time of the ideal register window architecture with the same number of physical registers. The results are given for an eight issue pipeline with three data cache ports, two data cache ports and one data cache port. A range of physical registers is shown.

as the number of physical registers is decreased to 128. At this point, vca1 with one port is over 14% slower than ideal while with two ports it's still within 3% of ideal and three ports is just over 2% slower. The performance begins to get a little closer when the register file shrinks further. With 64 physical registers, one port is 18% slower while two ports is 7.5% slower and three ports is 7%. With so few physical registers, the lack of rename registers begins to dominate the performance and therefore the influence of the cost of spills and fills is somewhat mitigated.

The data cache accesses have the same behavior as seen in the previous sections. The baseline and ideal architectures generate fewer data cache accesses as the number of physical registers is decreased, while the conventional register window architecture and virtual context architecture generate more. The rate at which the VCA generates additional data cache accesses is greater with

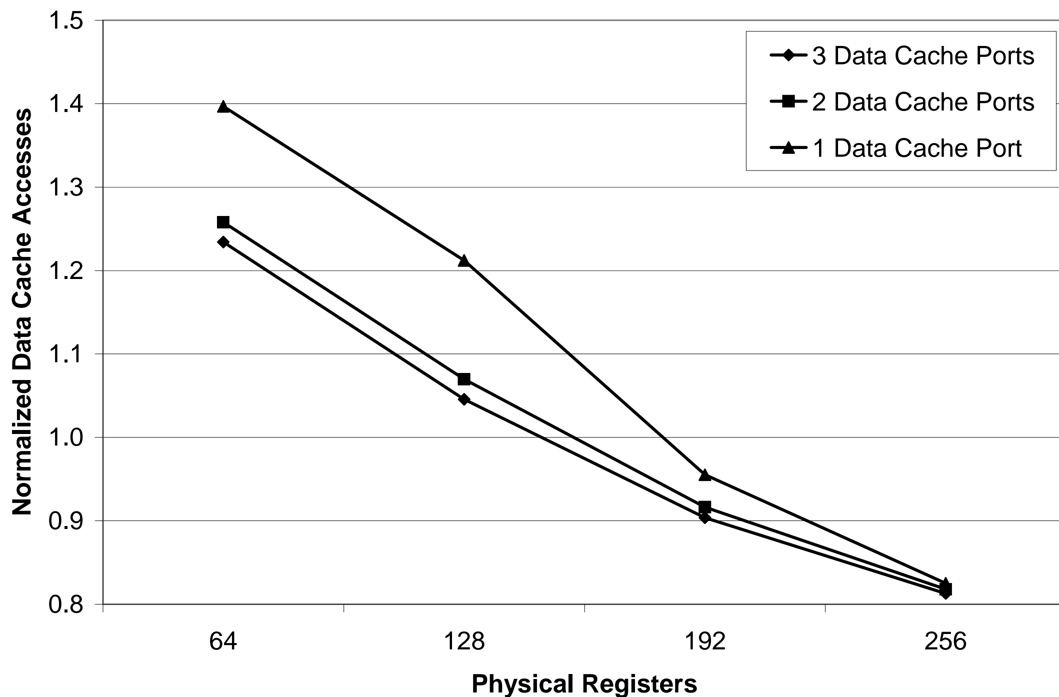



**Figure 6.16: Eight Issue VCA Data Cache Accesses**

The data cache accesses of the first virtual context architecture configuration normalized to the data cache accesses of the baseline architecture with 256 physical registers. The results are given for an eight issue pipeline with three data cache ports, two data cache ports and one data cache port. A range of physical registers is shown.

one data cache port than was seen with two or three. Figure 6.16 presents the data cache accesses of vca1 normalized to the data cache accesses of the baseline architecture with 256 physical registers. The results are given for all three eight issue pipelines studied. The results show that decreasing the number of data cache ports results in the virtual context architecture generating more spills and fills as the number of physical registers is decreased. While the three and two port results are relatively close, the VCA in a pipeline with a single port generates significantly more than the other two pipelines. With fewer data cache ports, the pipeline does not have as much cache bandwidth to execute the spills and fills. Therefore, the pipeline slows down. As mentioned previously, when the pipeline slows, the pressure is increased on the physical register file. When the pressure is increased on the physical register file it results in the generation of more spills and fills. In an eight issue pipeline with a single data cache port, this will result in the

generation of large numbers of spills and fills. For example, with 128 physical registers, the three port pipeline generates 4.5% more cache accesses than the baseline with 256 physical registers. In the two port pipeline this number increases to 7%. The one port is much higher. It generates over 21% more cache accesses than the baseline with 256 physical registers. These results provide a further factor why the execution time of the virtual context architecture behaves differently in a pipeline with only one data cache port versus a pipeline with two or three. Not only is the cost of a data cache access higher, but the virtual context architecture is also generating more accesses relative to the two and three port pipelines as the number of physical registers is decreased.

### 6.2.4 Summary

The results in this section show that similar to the four issue pipeline, the virtual context architecture in an eight issue pipeline is able to improve performance over the baseline architectures at all physical register file sizes and with between one and three data cache ports. The increased issue width causes the virtual context architecture to try to balance the physical register file so that it has more registers to use for rename, especially with smaller physical register file sizes. This is necessary to try to maintain large numbers of instructions in the instruction queue to maximize the number of independent instructions available to issue every cycle. The relationship between the cost of a data cache access and the performance of the virtual context architecture is very apparent in the eight issue pipeline. When the pipeline has several data cache ports, the cost of a cache access is relatively low. Under these conditions the improvement possible by using register windows is modest. However, the virtual context architecture is able to better maintain its performance as the size of the physical register file shrinks and can provide nearly ideal performance even with a small number of physical registers. If the cost of a data cache access is high, the virtual context architecture behavior changes. Under these conditions the improvement possible by using register windows is much greater. Therefore, the VCA shows a much larger improvement with large physical register files. However, the greatly increased cost of spills and fills

|  | Rename Table Size | Rename Table Ports | Extra Logic Cost |
|---|---|---|---|
| vca1 | 64x4 | 6 | none |
| vca2 | 64x3 | 6 | extra stage |

**Table 6.3: VCA Configurations For The Two Issue Pipeline**
The virtual context architecture configurations studied for the two issue pipeline. The configurations represent a range of implementations. The configurations are composed of two things. The first is the size and ports of the rename table. The second is the cost assumed for the extra logic in the rename table.

causes the virtual context architecture to lose performance rapidly as the size of the physical register file is decreased.

## 6.3 Two Issue Pipeline

This section presents the performance of the virtual context architecture in an two issue out-of-order pipeline. The purpose of this study is to examine the differences that a narrower pipeline makes in the performance of the virtual context architecture. As in the eight issue pipeline section, two different VCA configurations were studied, see Table 6.3. The configurations are equivalent to the first two configurations for the four issue pipeline. The only difference is that a fully ported rename table is used for both configurations. The results are presented for a range of physical register file sizes. The reorder buffer is half the size of the four issue pipeline; therefore, a smaller number of physical registers is needed to guarantee that no stalls occur in the baseline architecture. Thus, the range of physical registers tested only extends to 160 physical registers. Although the number of physical registers is less, the various numbers represent the equivalent to the four and issue eight pipelines of the physical register file size based on the relative number of registers available for rename versus the full number needed for no stalls. Thus, 160 represents a full complement of rename registers. 128 represents only two thirds of the full complement of rename registers. 96 represents only one third of the full complement of rename registers. Finally, 64 represents no rename registers.

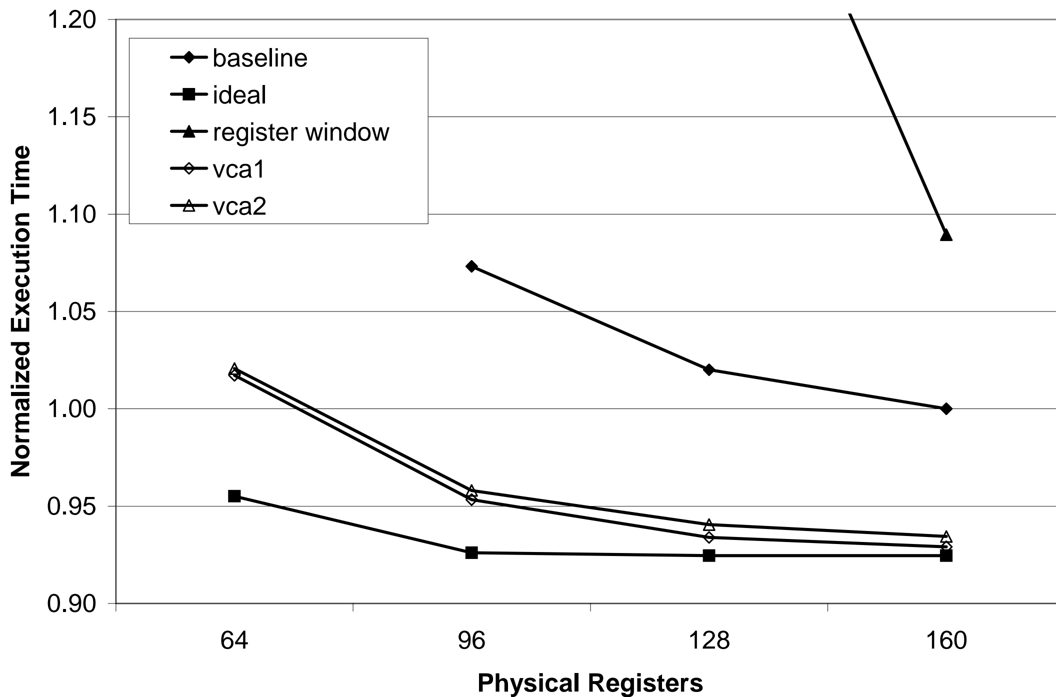



**Figure 6.17: Two Issue Execution Time Comparison**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and two different virtual context architecture configurations. The execution times are normalized to the execution time of the baseline architecture with 160 physical registers. The results are given for a range of physical register file sizes.

The results are presented in two subsections. The first subsection presents the execution time results. The second subsection presents the data cache results.

### 6.3.1 Execution Time Results

Figure 6.17 presents the execution time results for the two issue pipeline. The general trends seen in the two and four issue pipeline still exist. As the number of physical registers is decreased, all of the architectures lose performance. The baseline architecture loses 2% of its performance with 128 physical registers. With 96 physical registers, the performance drop is 7%. The equivalent drops for the four issue pipeline are 2% and 10%. In a two issue pipeline, a smaller number of instructions are needed to find enough independent instructions to fully issue every cycle. The baseline architecture is able to provide better performance with

the equivalent physical register file size because the probability of finding two instructions ready to execute requires a much smaller number of instructions than trying to find four.

The ideal register window architecture has a trend similar to the baseline architecture. Specifically, although the performance drops as the number of physical registers decreases, the drop is much less than the equivalent physical register file sizes of a four issue pipeline. In fact, the two issue ideal architecture only slows down 3% when the number of physical registers shrinks from 160 to 64. The four issue ideal architecture with two data cache ports slows down almost 11%. This is somewhat misleading. The ideal register window architecture only needs to maintain the actual in use registers in the physical register file. This is composed of the registers used for rename plus any architectural state currently being used by instructions in the pipeline. These values are fixed for any given set of instructions. In contrast, the baseline architecture must maintain the complete set of architectural state in the physical registers at all times (preventing it from running with fewer than 64 physical registers). This means that for the baseline, the number of instructions that can be renamed is approximately a function of the number of physical registers minus the number of physical registers needed for architectural state. For the ideal architecture on the other hand, the number of instructions that can be renamed is purely a function of the number of physical registers. Thus, the physical register files sizes used for the two issue are really only equivalent to the four issue in terms of the baseline architecture.

The virtual context architecture is also able to take advantage of it's independence on the number of architectural registers. Therefore, the VCA is also able to show a relatively small decrease in overall performance over the range of physical register file sizes. In the two issue pipeline, the vca1 execution times increase by only 9% when the number of physical registers is decreased from 160 to 64. The four issue pipeline experiences a change of nearly 18% when the number of physical registers is decreased from 256 to 64. Besides the smaller decrease in performance, the virtual context architecture also starts at a relatively large performance improvement over the baseline architecture. The cost of a cache access instruc-

tion in this pipeline will be relatively high. Although the ratio of data cache ports to issue width is equivalent to two ports in the four issue, the overall cost of an instruction is also increased in a two issue pipeline relative to a four issue pipeline. Therefore, the performance is closer to that seen in the four issue one data cache port pipeline. With 160 physical registers, vca1 is over 7% faster than the baseline. In comparison, vca1 in the four issue pipeline with two data cache ports was only 5% faster than the baseline with 256 physical registers, while in the four issue pipeline with one data cache port it was 7.5% faster. Combined with the slow decrease in performance as the physical register file is shrunk, this leads to the VCA being between 7% and 12% faster than the baseline at all the physical register file sizes studied.

The execution time of the virtual context architecture in a two issue pipeline does compare favorably to the execution time of the ideal register window architecture. With 160 physical registers, vca1 is within 0.5% of the ideal execution time. The difference steadily increases as the number of physical registers is decreased. With 128 physical registers, vca1 is 1% slower than ideal. With 96 physical registers, vca1 is 2.5% slower than ideal. Even with only 64 physical registers, the execution time of vca1 is still within 6% of the execution time of the ideal register window architecture. This is very close to the behavior seen in the four issue pipeline with two data cache ports. In this pipeline, vca1 is within 0.2% of ideal at 256 physical registers and slows as you decrease the physical register file size, until with 64 registers it is 7% slower than ideal. The two issue pipeline slows much less than the four issue pipeline with one port, which while within 0.6% with 256 physical registers, slows to 12% longer execution time than ideal with 64 physical registers.

One potential concern with the virtual context architecture on a narrower pipeline was the decrease in extra issue that could be used by the architectural state transfer queue to execute spills and fills. The priority mux is set to always give priority to instructions issuing from the instruction queue. Therefore, spills and fills will only be executed when less than the full complement of instructions is issued on a given cycle. It stands to reason that this is much more likely to occur in a four

issue pipeline than in a two issue pipeline. The statistics confirm this suspicion. The fills in the two issue pipeline take between 12.5 and 21.5 cycles to issue on average. The range reflects the rates at different physical register file sizes. In contrast, the fills in the four issue pipeline with two data cache ports take between 4 and 10 cycles to issue. In the four issue pipeline with one data cache port, the fills take between 7.5 and 16 cycles to issue. The spills show similar results. The average time to issue a spill in the two issue pipeline is between 8 and 51 cycles while the range in the four issue pipeline with two data cache ports is 2.5 to 23 cycles. The average time to issue a fill or spill is longer in the two issue pipeline, but the execution time results show that the increased issue time does not play a large factor in the results. Specifically, the virtual context architecture in the two issue pipeline maintains a similar performance gap to the ideal register window architecture as the four issue pipeline with two data cache ports.

### 6.3.2 Data Cache Results

The data cache results for the two issue pipeline are shown in Figure 6.18. Like the execution time results just discussed, the data cache results are also very similar to the results of the previous pipelines. The baseline and ideal register window architectures both decrease the number of data cache accesses as the number of physical registers is reduced. As noted earlier in this section, the execution time degradations in the two issue pipeline are slow compared to the other pipelines. Thus, for these architectures, the difference in data cache accesses over the range of physical register file sizes studied is also small. Unlike in the four and eight issue pipelines, the conventional register window architecture does not provide any data cache access savings. Even with 160 physical registers, it generates almost 18% more data cache accesses than the baseline architecture. With the smaller physical register file sizes studied, only a few register windows can be kept in the physical register file. Therefore, the data cache accesses caused by overflow and underflow events overshadow any savings due to the register window binary. The virtual context architecture is able to maintain a significant cache savings with both 160 and 128 physical registers. Even with as few as 96 physical

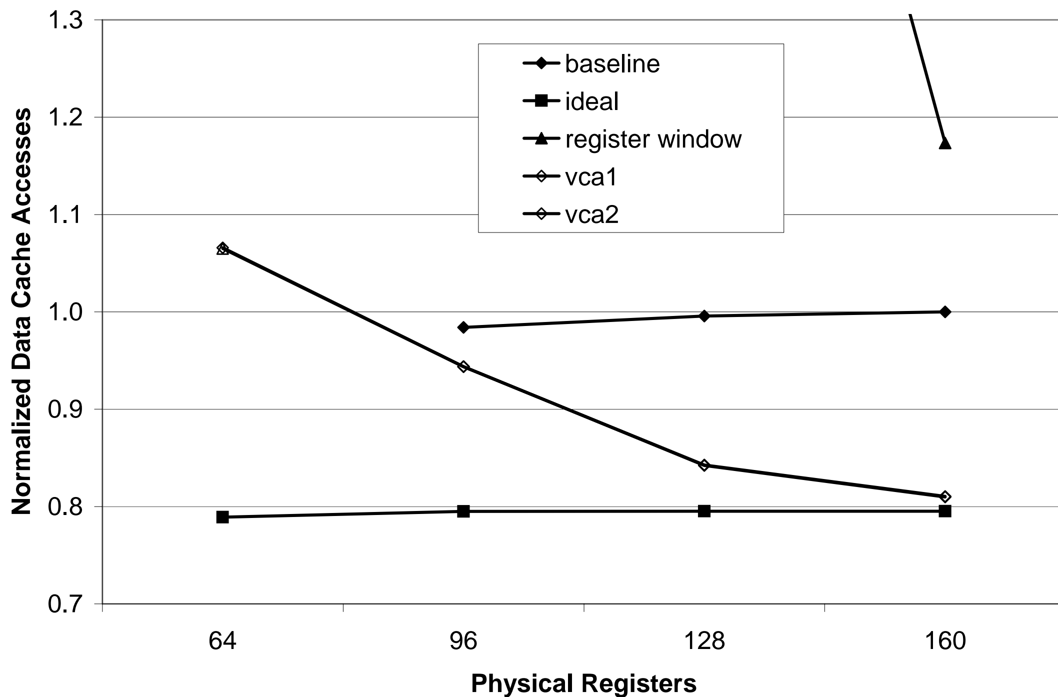



**Figure 6.18: Two Issue Data Cache Accesses Comparison**
The data cache accesses of the baseline architecture (baseline), ideal register window architecture (ideal), conventional register window architecture (register window) and two different virtual context architecture configurations. The data cache accesses are normalized to the data cache accesses of the baseline architecture with 160 physical registers. The results are given for a range of physical register file sizes.

registers, the virtual context architecture is generating nearly 4% fewer cache accesses than the baseline architecture with the same number of physical registers. The relative cache performance of the virtual context architecture in the two issue pipeline is better than that seen in both the four and eight issue pipelines, when compared to the baseline architecture with a full set of physical registers. The four issue pipeline with two data cache ports has 19% fewer cache accesses with 256, 12% fewer with 192, 3% fewer with 128 and 14% more with 64 physical registers. The two issue pipeline starts the same with 19% fewer with 160, but only drops to 16% fewer with 128, 6% fewer with 96 and 7% more with 64 physical registers. As the virtual context architecture compensates for the decreased physical register file size, it is forced to generate fewer spills and fills with the narrower pipeline.

The results in this section show that the virtual context architecture is able to provide good performance even on a narrow out-of-order pipeline. It provides a 7% to 12% improvement of the baseline architecture over the range of physical registers studied. On this pipeline the degradation of the execution time performance is smaller than any other pipeline as the physical register file size is decreased. The smaller issue width does limit the unused issue bandwidth that the architectural state transfer queue can use to issue spills and fills. Thus, the delay until a fill or spill can be issued is longer in this pipeline than the four issue, even the four issue with a single data cache port. However, the VCA does not generate as many fills and spills as the number of physical registers is reduced. This enables the virtual context architecture to still provide as close to ideal performance as it could in the four issue pipeline with two data cache ports.

## 6.4 Single Issue In-order Pipeline

This section presents the performance of the virtual context architecture in a single issue in-order pipeline. The purpose of this study is to evaluate the performance of the virtual context architecture in an in-order processor. The pipeline and VCA implementation are somewhat different from the other pipelines studied. The data cache for this pipeline is assumed to have a single cycle hit latency. This corresponds to a more conservative embedded processor. The virtual context architecture implements spills and fills by inserting operations directly into the pipeline instead of using an architectural state transfer queue. The in-order nature of the pipeline made this a more natural fit. As with normal instructions, the pipeline was constrained to dispatch only one spill or fill each cycle instead of a single instruction. Two virtual context architectures were studied, see Table 6.4. A conventional in-order pipeline does not require any renaming. Therefore, an additional stage is always going to be necessary. The two virtual context configurations only differ in the number of rename stages.

The execution time results of the single issue in-order processor are presented in Figure 6.19. The results are normalized to the execution time of the baseline in-

| | Rename Table Size | Rename Table Ports | Extra Logic Cost |
|---|---|---|---|
| vca1 | 64x3 | 3 | one extra stage |
| vca2 | 64x3 | 3 | two extra stages |

**Table 6.4: VCA Configurations For The Single Issue In-order Pipeline**
The virtual context architecture configurations studied for the single issue in-order pipeline. The configurations represent a range of implementations. The configurations are composed of two things. The first is the size and ports of the rename table. The second is the cost assumed for the extra logic in the rename table.

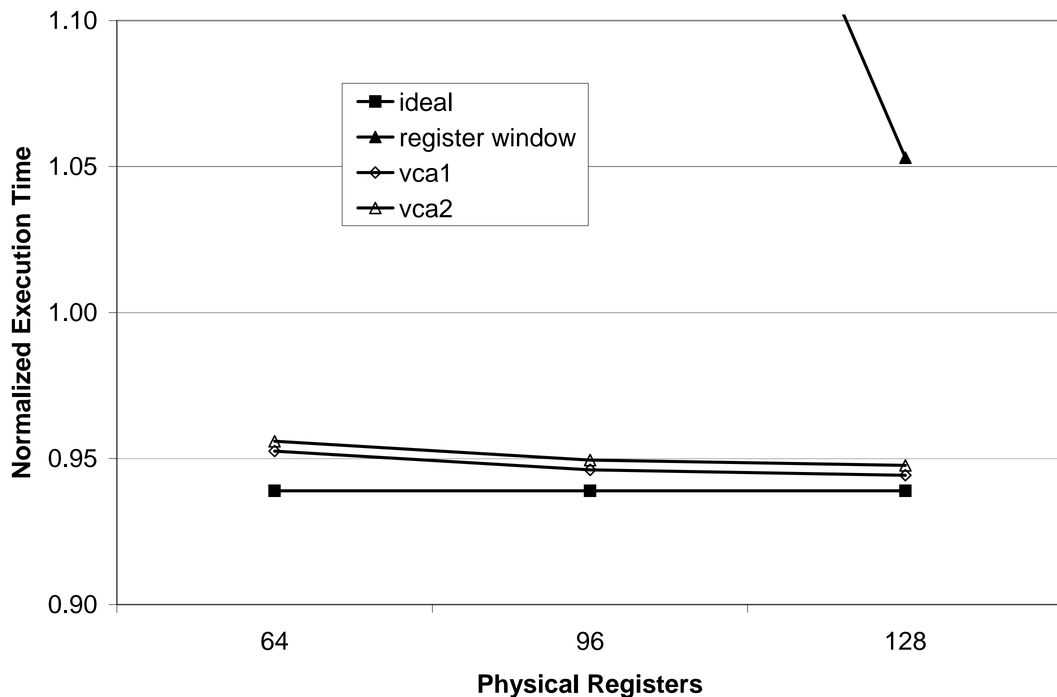



**Figure 6.19: Single Issue In-order Execution Time Comparison**
The execution time of the ideal register window architecture (ideal), conventional register window architecture (register window) and two different virtual context architecture configurations. The execution times are normalized to the execution time of the baseline architecture. The results are given for a range of physical register file sizes.

order processor. The baseline in-order processor does not require any extra register for rename purposes therefore it has a fixed physical register file size of 64 registers, one physical register for each logical register. The execution time of the ideal register window architecture does not change within the physical register file

sizes studied. To maintain maximum performance, this architecture only requires enough registers to fully rename instructions. In the case of an in-order machine, this would be only enough registers to rename a single instruction. As seen in the previous pipelines, the conventional register window architecture is not able to provide good performance with this few physical registers. Its performance is 5% slower than the baseline with 160 physical registers, and rapidly declines from there.

The virtual context architecture is able to provide very good performance for the full range of physical register file sizes studied. While the performance of the VCA declines as the number of physical registers is decreased as in the other pipelines, the rate of decrease in almost negligible for this range of physical registers. With 128 physical registers, vca1 is 5.5% faster than the baseline. When the size of the physical register file is dropped to 64, vca1 is still 4.5% faster than the baseline. Thus, even with one extra stage and no extra physical registers, the virtual context architecture is able to provide a 4.5% improvement in performance. The results for adding to additional stages (vca2) are almost as good. This configuration is 0.3% slower than vca1 at all physical register file sizes. The virtual context architecture is very near the ideal performance over the entire range of physical registers. With 128 physical registers it is within 0.5%. Even with only 64 physical registers, vca1 is still within 1.4% of the performance of an ideal register window implementation.

As discussed previously in this chapter, the execution time benefit of register window is very dependent on the relative cost of executing a load or store instruction. This cost is composed of the general expense of executing an instruction plus the added expense of a data cache access. In this pipeline, the cost of executing an instruction is high. The pipeline is only single issue and there is no extra issue width that can hide an instructions cost. In this particular implementation, the first level data cache was chosen to have only a single cycle latency. Therefore, the additional cost of a cache access is low. If this latency was increased, the performance improvement of the virtual context architecture would become much greater.

The data cache characteristics of this pipeline are similar to the previous studied pipeline. As was the case in the two issue pipeline study, the conventional register window architecture does not fare well with this few physical registers. Only a small number of register windows are held in the physical register file. This results in a large number of overflow and underflow events, yielding very poor data cache performance. In this case, with 128 physical registers the conventional register window design generates nearly 20% more data cache accesses than the baseline architecture, explaining its poor performance. In contrast, the virtual context architecture has very good data cache performance. With 128 physical registers, it has a savings of nearly 20% relative to the baseline. With 96 physical registers, this only drops to a 19% savings. Even with the same number of registers as the baseline architecture, with just enough physical registers to hold all the logical registers, the virtual context architecture generates 14% fewer data cache accesses than the baseline.

## 6.5 Summary

The results in this chapter show that the virtual context architecture is a very efficient implementation of register windows for a variety of different pipelines. In every pipeline studied, the virtual context architecture is able to provide nearly ideal performance when the physical register file is sized to guarantee no stalls for the baseline architecture. In almost every case, the performance of the virtual context architecture versus the baseline architecture improves as the number of physical registers is decreased. The VCA is able to dynamically balance the number of physical registers it is using for renaming to maximize the performance. In contrast, the conventional register window design using the same instruction set and modified binaries as the VCA was unable to provide good performance for the range of physical registers studied. Although the modified instruction set has a high number of windowed registers, this has the advantage of minimizing the saves and restores that the compiler may be forced to insert at register allocation.

The studies showed that the performance of the virtual context architecture is very dependent on the relative cost of a load/store instruction and the cost of a data cache access. When these costs are high, the architectures that support register windows have a higher potential performance improvement due to the removal of the save and restore instructions from programs. With a large physical register file, the virtual context architecture is able to achieve very near this ideal performance. However, when the number of physical registers is decreased, the VCA is forced to generate more spills and fills. If the cost of a data cache access is high, these spills and fills will degrade performance quickly.

# Chapter 7

# Simultaneous Multithreading Studies

This chapter evaluates the virtual context architecture in a simultaneous multithreading processor. The chapter is composed of three sections. The first and second sections present the results for a pipeline which support two and four threads respectively. The virtual context architecture using register window binaries is evaluated against the baseline non register window architecture and an ideal register window architecture. The conventional register window architecture was not studied because its physical register requirements were much higher than all the other architectures. The final section pulls together the results from the previous two sections, as well as the previous chapter, to compare the performance of the virtual context architecture in a one, two and four thread pipeline to determine the cost of adding threads.

All of the studies in this chapter were done with a four issue out-of-order pipeline with two data cache ports, see Table 4.2 for a description of this pipeline. The workloads were created by combining two and four benchmarks from the SPEC 2000 benchmark suite. A description of the methodology used to create the workloads can be found in Section 4.2. The two metrics used to measure the performance of the architectures are the weighted execution times of the workloads and the weighted data cache accesses. These metrics are similar to the single thread execution time and data cache accesses but weigh each of the individual benchmark statistics in the workload to the value of the benchmark when running as a single thread. This normalizes the contributions of the individual benchmarks in the workload and ensures they are given equal weight. A more thorough discussion of this can be found in Section 4.4.2. In the rest of this chapter, the weighted

|  | Rename Table Size | Rename Table Ports | Extra Logic Cost |
|---|---|---|---|
| vca1 | 64x6 | 10 | none |
| vca2 | 64x5 | 8 | extra stage |

**Table 7.1: VCA Configurations For Two Thread SMT**
The virtual context architecture configurations studied for the two threaded simultaneous multithreaded four issue pipeline. The configurations are composed of two things. The first is the size and ports of the rename table. The second is the cost assumed for the extra logic in the rename table.

terminology will be dropped and these will simply be referred to as execution time and data cache accesses.

The most important performance characteristic of a simultaneous multithreaded architecture is the performance improvement achieved by allowing more than one thread to execute at the same time. If there is no speedup associated with running more than one thread, then the extra hardware and design complexity needed to support more than one thread is wasted. Therefore, in the sections that discuss execution time, the speedup is also reported. Speedup is calculated as the total time to execute both benchmarks on a single thread machine divided by the total time to execute both benchmarks on a multithread machine. The speedup is calculated using the weighted execution times to ensure that each benchmark in the workload contributes equally to the results. For consistency across all the architectures and register file sizes, the speedups are always calculated with a single thread pipeline with enough physical registers to never stall. For the four issue pipeline studied, this is 256 physical registers.

## 7.1 Two Thread SMT

This section presents the performance of the virtual context architecture in a two thread simultaneous multithreading four issue pipeline with two data cache ports. The results are presented in two subsections. The first subsection presents the execution time results. The second subsection presents the data cache access results. Two different VCA configurations were studied, see Table 7.1. The

configurations are equivalent to the first two configurations for the single threaded four issue pipeline studied in the previous chapter. The only difference is the increase in the associativity of the rename table. As discussed in Section 5.2.1, a larger rename table is required in pipelines that support multiple threads.

The results are studied over a range of physical register file sizes. The smallest number of physical registers examined is 64 physical registers, which is just enough to hold the architectural state for a single thread without rename registers. Only the virtual context architecture and ideal register window architecture are able to run with this few physical registers. The baseline architecture requires its full architectural state be kept in the physical register file. With the addition of rename registers, this constrains them to a physical register file larger than 128 physical registers for a two thread pipeline. The largest number examined is 320 physical registers. This is enough registers to guarantee that the baseline architecture will never need to stall rename because of a lack of physical registers. To ensure this, the processor must have one physical register for each architectural register plus one physical register for each reorder buffer entry (assuming instructions only have a single destination register).

### 7.1.1 Execution Time

This section examines the execution times of the baseline architecture, ideal register window architecture and virtual context architecture in a two thread simultaneous multithreading pipeline. Figure 7.1 presents the execution time results. The execution times are normalized to the execution time of the baseline architecture with 320 physical registers. As seen in the results for a single thread, all three architectures slowdown as the number of physical registers is decreased. The virtual context architecture and ideal register window architecture are both faster than the baseline at every level of physical register file size. As the number of physical registers is decreased, the relative performance advantage of these two architectures over the baseline architecture increases. With a decrease in physical register file size from 256 to 192, the execution time of the baseline architec-

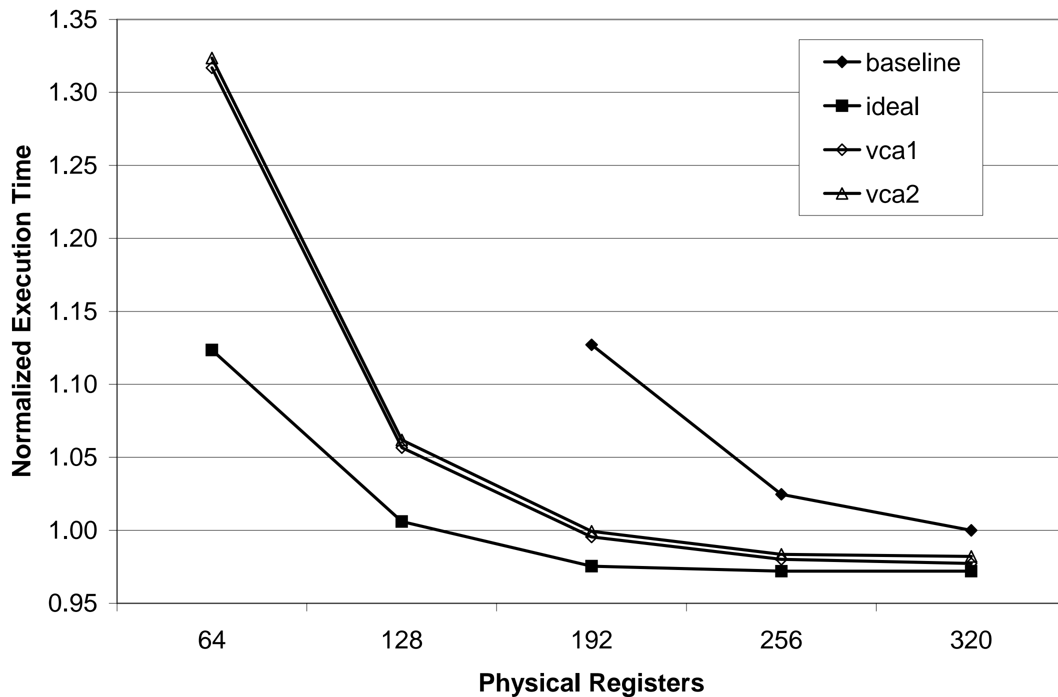



**Figure 7.1: Two Thread Execution Time Comparison**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations. The execution times are normalized to the execution time of the baseline architecture with 320 physical registers. The results are given for a range of physical register file sizes.

ture increases by over 10%, while the virtual context architecture only has a 1.5% increase.

The virtual context architecture has lower execution time than the baseline at every level of physical register file. With 320 physical registers, the performance difference is relatively small with only a 2.3% improvement. With 256 physical registers, the improvement almost doubles to 4.5%. With 192 physical registers, the difference becomes very large. The execution time of the virtual context architecture is over 13% faster than the baseline architecture. The virtual context architecture is able to provide a similar level of performance with 192 physical registers as the baseline with 320 physical registers.

The virtual context architecture is able to provide a nearly ideal implementation of register windows. This is especially true with the larger physical register file sizes. The difference in execution time between the VCA and the ideal register

window architecture is directly attributable to the virtual context architecture's generation of spills and fills. The ideal register window architecture is able to instantaneously transfer register values to and from memory, without generating any cache accesses. In contrast, the virtual context architecture generates spills and fills to perform these transfers. With a large enough physical register file, the virtual context architecture generates only a small number of spills and fills. As the number of physical registers is reduced, the number of spills and fills generated increases. The increased time to execute these spills and fills causes the virtual context architecture to slow in relation to ideal. With 320 physical registers, the execution time difference is only 0.5%. With 256 physical registers, the difference is still only 0.8%. As the number of physical registers is decreased further, the number of generated spills and fills increases rapidly and differences become larger, with a 2% difference with 192 physical registers, 5% with 128 and nearly 20% with 64 physical registers.

Register windows are used to reduce the save and restore overhead necessary for a function call. The benefits of register windows are only realized when there is a high frequency of function calls. In the single thread experiments, we only gathered results for those benchmarks that made a function call at least once every 500 instructions. Unlike the single thread experiments, in the creation of the multithread workloads we did not limit the benchmarks to those with a high frequency of function calls. Instead, we wanted to evaluate the VCA in as diverse a multithreaded environment as possible. In Figure 7.2, we present the execution time results of only those workloads where both benchmarks make at least one function call every 500 instructions. Out of the 43 workloads, 28 of them are composed only of these benchmarks. The results show that with workloads able to take advantage of register windows, the performance improvement of the virtual context architecture is even greater compared to the baseline architecture. This is true at every physical register file size. With 320 physical registers, the virtual context architecture is 4.3% faster than the baseline versus only 2.3% faster with the full set of workloads for a difference of 2%. With 256 physical registers, the differ-

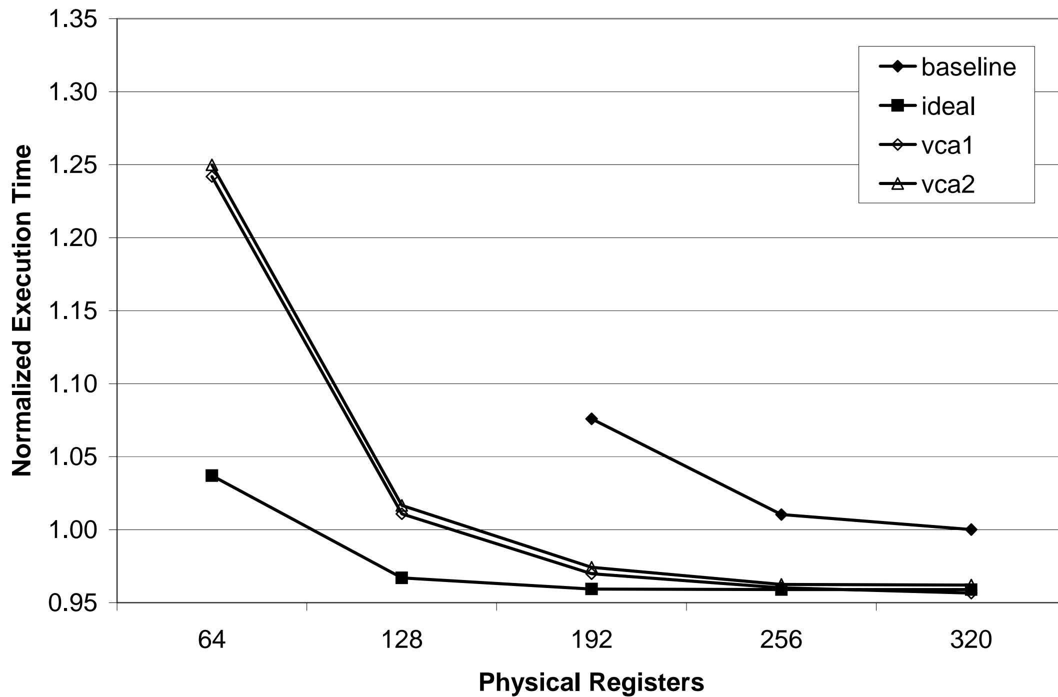



**Figure 7.2: Two Thread High Call Rate Execution Time Comparison**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations. The workloads are restricted to those in which both benchmarks make at least one function call every 500 instructions. The execution times are normalized to the execution time of the baseline architecture with 320 physical registers. The results are given for a range of physical register file sizes.

ence remains the same. However, as the number of physical registers is decreased further, the difference increases. With 192 physical registers, the difference grows to 2.5%. With 128 physical registers, the difference is 4.5%. Finally, with 64 physical registers the difference is 7.5%. The results show that even when all of the threads have a high frequency of function calls the virtual context architecture is still able to provide a high level of performance.

Finally, the most important characteristic of a multithread pipeline is the speedup over a similar single thread pipeline. Figure 7.3 presents the speedup of the two thread simultaneous multithreading pipeline over a single thread pipeline. Since speedup and execution times are inversely proportional, the speedups decrease as the number of physical registers is decreased. The rates of decrease are similar to the rates of increase of the execution times reported earlier in this

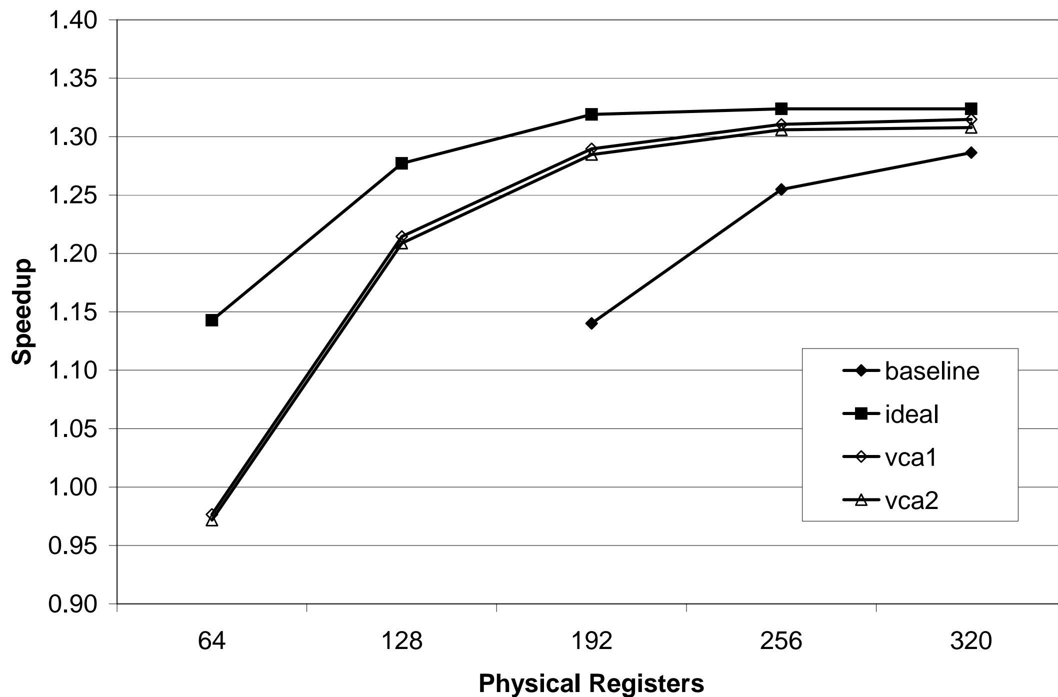



**Figure 7.3: Two Threaded SMT Speedups**

The speedups of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations in a two threaded simultaneous multithreading four issue pipeline. The speedups are relative to a single threaded four issue pipeline. The results are given for a range of physical register file sizes.

section. The results show that all three architectures are able to significantly speedup execution over a single thread pipeline. With a full set of 320 physical registers, the two thread pipeline is on average 30% faster than a pipeline with a single thread. With this full set of registers, the virtual context architecture provides a speedup of nearly 31.5% versus a speedup of 28.5% for the baseline architecture. With 256 physical registers, the vca1 still achieves a speedup of 31%, a 6% improvement over the baseline with this physical register file size. Even with as few as 192 physical registers, the virtual context architecture is able to provide the same speedup as the baseline has with 320 physical registers. With 128 physical registers, which is only enough to hold the architectural state of the two threads, vca1 is still able to achieve a 21% speedup. The baseline architecture is unable to run with this few physical registers. Finally, with 64 physical regis-

ters, the virtual context architecture has slowed so much that it is unable to provide any speedup over the single thread pipeline.

The virtual context architecture is able to achieve a 2%-13% improvement in execution time over the baseline architecture. With 256 or more physical registers, the VCA provides very close to ideal register window performance. In the case when all of the threads have a high frequency of function calls, the virtual context architecture is still able to provide a high level of performance and the improvement in execution time over the baseline increases. The two thread pipeline is able to achieve a 30% increase in performance over the single thread pipeline. The speedup of the virtual context architecture is 3% to 15% higher than the speedup of the baseline architecture with the same number of physical registers. The VCA is able to achieve the same speedup with 192 physical registers as the baseline with 320 physical registers.

### 7.1.2 Data Cache Accesses

This section examines the data cache accesses of the baseline architecture, ideal register window architecture and virtual context architecture in a two thread simultaneous multithreading pipeline. Figure 7.4 presents the data cache accesses of the two thread pipeline. The results are normalized to the data cache accesses of the baseline architecture with 320 physical registers. The data cache access behavior of the architectures is the same as was seen in the single thread results. The baseline architecture and ideal register window architecture generate a decreasing number of cache accesses as the number of physical registers is reduced. In these architectures, the number of data cache accesses is solely affected by the number of speculatively executed instructions. Therefore, as the pipeline slows, fewer instructions are executed speculatively and there are fewer data cache accesses.

The virtual context architecture generates more data cache accesses as the number of physical registers is reduced. The additional accesses are the result of the spills and fills generated by the pipeline. However, the virtual context architecture is also executing binaries that support register windows. These binaries have

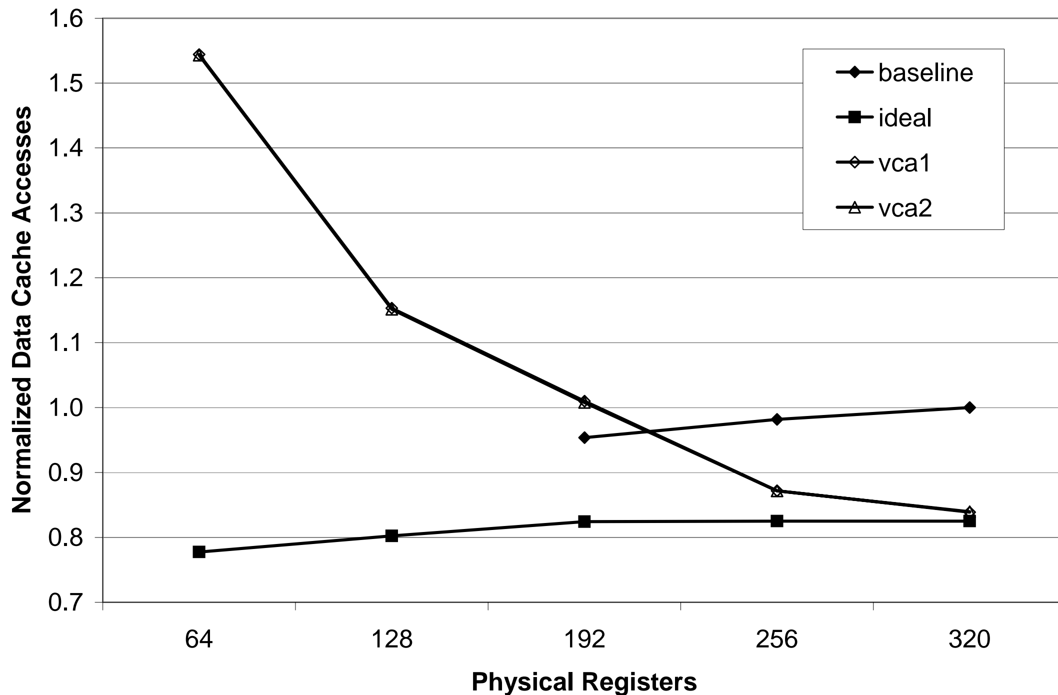



**Figure 7.4: Two Thread Data Cache Accesses Comparison**
The data cache accesses of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations. The data cache accesses are normalized to the data cache accesses of the baseline architecture with 320 physical registers. The results are given for a range of physical register file sizes.

fewer load and store instructions than the binaries used by the baseline architecture. With 320 physical registers, the virtual context architecture generates very few spills and fills and accesses the data cache over 16% fewer times than the baseline architecture. With 256 physical registers, the data cache difference shrinks to 11%. With 192 physical registers, the VCA is generating 5.5% more accesses than the baseline architecture with the same number of physical registers. This is the same number of data cache accesses as the baseline with 320 physical registers. With this number of physical registers, the number of spills and fills being generated is approximately equal to the number of load and store instructions removed by using register windows. With fewer physical registers, the spills and fills increase even further. With 128 physical registers, the virtual context architecture is now generating 15% more data cache accesses than the base-

line binary with 320 physical registers. With 64 physical registers, this difference becomes nearly 55%.

The virtual context architecture is able to provide a nearly ideal implementation of register windows with a large enough physical register file. With 320 physical registers, the VCA generates only 1.3% more data cache accesses than the ideal. As the number of physical registers is decreased, the number of spills and fills generated by the virtual context architecture increases. These spills and fills are the only difference between the VCA and the ideal register window architecture. The results clearly show this, with the data cache difference increasing rapidly as the size of the physical register file is reduced. With 256 physical registers, the virtual context architecture is still within 5% of the ideal. With 192 physical registers, the difference grows to almost 19%. The difference continues to increase drastically from here, with a difference of 35% with 128 physical registers and 77% with 64.

As mentioned in the execution time section, these results are for workloads composed of all the benchmarks. If we restrict the results to those workloads composed of only benchmarks that make a high frequency of function calls, the benefit obtained from using register windows will be greater. The data supports this. With 320 physical registers and the restricted workload set, the virtual context architecture generates 19% fewer data cache accesses. With the full set of workloads, the difference is 16%. The difference grows as the number of physical registers is decreased. With 256 physical registers, the restricted workloads have 15% fewer than the baseline while the full set is 11%. With 128, the virtual context architecture with the restricted set of workloads still generates 2.5% fewer data cache accesses than the baseline with the same number of physical registers. In contrast, with the full set of workloads, the VCA generates 5.5% more than the baseline.

Confining the results to this restricted set also affects the virtual context architecture's behavior versus ideal. With 320 physical registers, the VCA with restricted workloads generates 2% more data cache accesses than the ideal register window architecture using the restricted set. This compares to a difference of

1.3% will the full set of workloads. One might expect that this difference would grow as the number of physical registers is reduced. Benchmarks that make a high frequency of function calls generate a much larger set of architectural state than benchmarks that rarely call functions. The expectation would be that this added pressure would result in a relative increase in the generation of spills and fills. The data, however, shows the opposite affect. As the number of physical registers is decreased, the additional data cache accesses generated by the VCA over the ideal register window architecture is less with the restricted set of workloads than with the full set. With 256 physical registers, the difference between the virtual context architecture and ideal is the same for both sets of workloads. With 192 physical registers, the difference with the restricted set is 14% versus 18% with the full set. With 128 physical registers, these become 25% and 35%. Finally, with 64, the restricted set has a difference of 61% compared with 77% for the full set. The behavior is most likely a secondary effect of the benchmarks that make a high frequency of function calls. If the benchmarks were in general slower to execute, than the pressure on the physical register file would not increase as rapidly as the number of physical registers is reduced. Another possibility is that these benchmarks may tend to use fewer different logical registers, and therefore the number of in use physical registers would be smaller.

The virtual context architecture is able to provide a data cache access savings over the baseline architecture with a large physical register file. With 320 physical registers, the virtual context architecture generates 16% fewer data cache accesses than the baseline architecture and is within 1.3% of ideal. The savings decreases as the number of physical registers is decreased, until with 128 physical registers the virtual context architecture generates 5.5% more cache accesses than the baseline. Similar to the execution time results, if the workloads are restricted to those composed solely of benchmarks which make a high frequency of function calls, the advantage of the virtual context over the baseline increases.

| | Rename Table Size | Rename Table Ports | Extra Logic Cost |
|---|---|---|---|
| vca1 | 64x7 | 10 | none |
| vca2 | 64x6 | 8 | extra stage |

**Table 7.2: VCA Configurations For Four Thread SMT**
The virtual context architecture configurations studied for the four threaded simultaneous multithreaded four issue pipeline. The configurations are composed of two things. The first is the size and ports of the rename table. The second is the cost assumed for the extra logic in the rename table.

## 7.2 Four Thread SMT

This section presents the performance of the virtual context architecture in a four thread simultaneous multithreading four issue pipeline with two data cache ports. The results are presented in two subsections. The first subsection presents the execution time results. The second subsection presents the data cache access results. Two different VCA configurations were studied, see Table 7.2. The configurations are equivalent to the first two configurations for the single threaded four issue pipeline studied in the previous chapter and the configurations used for the two thread experiments. The only difference is the increase in the associativity of the rename table. As discussed in Section 5.2.1, a larger rename table is required in pipelines that support multiple threads.

The results are studied over a range of physical register file sizes. The range of physical registers is larger than the range used for two threads. The smallest number of physical registers examined is still 64. This is just enough registers to hold the architectural state for a single thread. The large number of simultaneous threads in this section requires a large number of physical registers. Four threads require 256 physical registers to hold their architectural state. Therefore, the minimum number of physical registers we use for the baseline is 320 physical registers. To guarantee the baseline architecture will never stall requires 192 registers for rename for a total of 448 physical registers. This is the upper limit used in our experiments in this section.

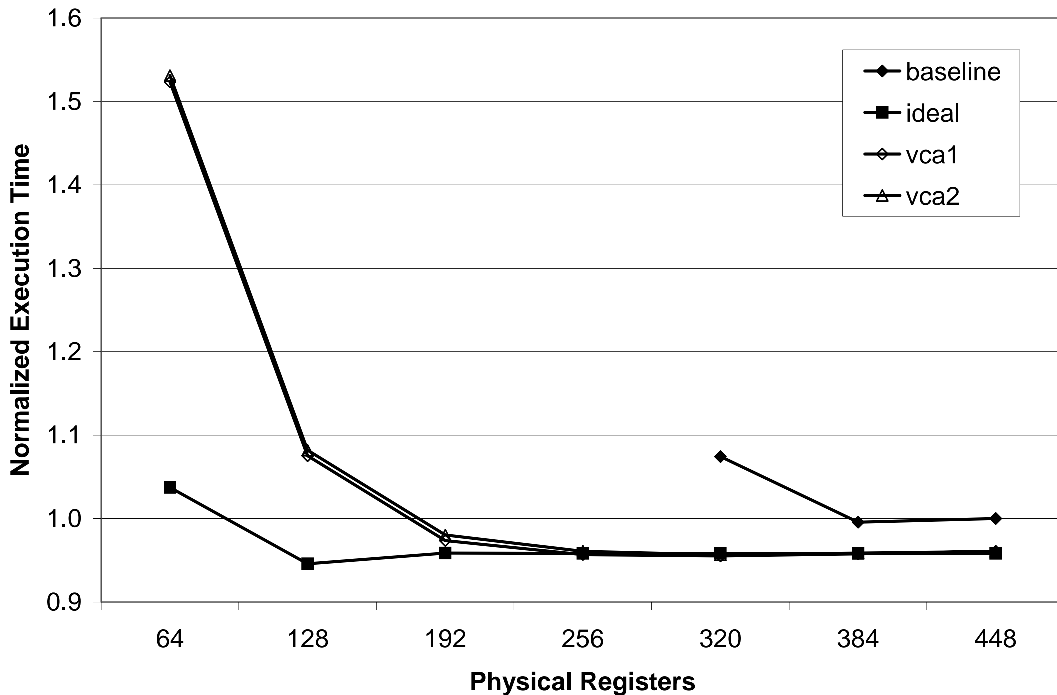



**Figure 7.5: Four Thread Execution Time Comparison**
The execution time of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations. The execution times are normalized to the execution time of the baseline architecture with 448 physical registers. The results are given for a range of physical register file sizes.

### 7.2.1 Execution Time

This section examines the execution times of the baseline architecture, ideal register window architecture and virtual context architecture in a four thread simultaneous multithreading pipeline. Figure 7.5 presents the execution time results. The results are normalized to the baseline architecture with 448 physical registers. The results in general are similar to those seen in all the other pipeline configurations. As the number of physical registers is decreased, the execution time tends to increase. However, the results are more complex than seen previously. In particular, all of the architectures at some point actually decrease their execution times with a decrease in physical registers. The baseline architecture is 0.4% faster with 384 physical registers than with 448 physical registers. The ideal register window architecture is 1.3% faster with 128 physical registers than any larger

physical register file size. The virtual context architecture is 0.3% faster with 256, 320 and 384 physical registers than with 448 physical registers.

The difference between this experiment and previous experiments is the greater number of simultaneous threads. In particular, in this experiment we have four threads on a four issue machine. This has two potential repercussions: scheduling complexity and independent instructions. First, the scheduling complexity is greatly increased. Each cycle the fetch stage will fetch from the thread with the least number of instructions in the pipeline. Each cycle the instruction queue determines which instructions to issue, giving priority to the oldest instructions. At any given point a small change in the execution can drastically change the execution for the rest of the run. For example, say the lack of a rename register prevents a load instruction from being dispatched. The next cycle instructions are dispatched from a different thread and a store to the conflicting cache line is dispatched. The store is issued first, replacing the cache line. Later the load from the first thread misses, causing the thread to stall. This in turn allows the other threads more resources to execute. This complexity allows a small change to the pipeline to have unexpected results on execution.

Second, the large number of threads greatly increases the chances of finding independent instructions with a given number of instructions in the instruction queue. Specifically, instructions from different threads are always independent of each other. In a pipeline with a single thread, the chance of finding four independent instructions with eight instructions in the instruction queue is very small. However, with two instructions each from the four threads, the chance of four of those instructions being independent is greatly increased. This is the theory behind simultaneous multithreading, increasing the instruction level parallelism. This has the added effect of making the execution time less dependent on the number of physical registers. Therefore, the potential performance difference with 128 registers available for rename versus 192 could become quite small. The execution time results of this section indicate that this is the case in this situation. The baseline architecture with four threads loses only 7.5% of its performance when

dropping from the full set of physical registers to only 64 rename registers. With two threads, the same drop results in a 13% decrease in performance.

The virtual context architecture provides better performance than the baseline architecture at every size of physical register file. With 448 physical registers, the virtual context architecture is 4% faster than the baseline. The performance difference remains the same with 384 physical registers. With 320 physical registers, the baseline has reached the limit of where it can run. At this point, the VCA is 12% faster. The relative performance of the virtual context architecture to the baseline is better with four threads than two with a large register file. With a full set of physical registers (448 for four thread, 320 for two thread), the VCA with four threads is 4% faster than the baseline, with two threads it is only 2.3% faster. While the baseline architecture loses performance over its range of physical register files in both experiments, the virtual context architecture's performance remains approximately constant over the same range. As stated previously, with decreasing physical register file sizes the performance loss is greater with two threads than with four. With 64 physical registers fewer than the full complement, the relative performance of the virtual context architecture to the baseline is 4% faster with four threads and 4.5% faster with two. With 128 fewer physical register, the percentages are 12% and 13%.

The virtual context architecture is able to maintain its 4% improvement in performance over the baseline architecture with 448 physical registers with as few as 256 physical registers. This is just enough registers to hold the architectural state of four threads. With 192 physical registers, the virtual context architecture is still nearly 3% faster than the baseline with a full set of registers. This is with less than half the physical registers of the baseline and not enough to even hold the full architectural state of the four threads. The ability of the virtual context architecture to dynamically manage the architectural state of the threads provides a huge advantage in this situation. The results dramatically demonstrate that the VCA can efficiently manage the movement of this state between the physical registers and the data cache. This allows the virtual context architecture to provide better performance with a much smaller set of physical registers.

The virtual context architecture is once again able to provide very near ideal performance with large physical registers files. In these results, the VCA is able to maintain very close to ideal performance with as few as 256 physical registers. With between 256 and 448 physical registers, vca1 is never slower than 0.2% of ideal and at some physical register file sizes is up to 0.3% faster than ideal. As stated previously in this section, the results show instances when a decrease in physical register file size actually improves performance. The virtual context architecture experiences such a drop with between 256 and 384 physical registers. Over this same range, the performance of the ideal register window architecture remains the same (its decrease in execution time occurs at 128 physical registers). The drop in performance of the virtual context architecture is enough to bring the execution time below that of ideal with the same number of physical registers, although the execution time is still higher than the lowest execution time that ideal ever has.

These results are for workloads composed of all the benchmarks. As mentioned in the section on two threads, if we restrict the results to those workloads composed of only benchmarks that make a high frequency of function calls, the benefit obtained from using register windows will be greater. Of the 127 four thread workloads, 61 are composed solely of benchmarks that make a high frequency of function calls. Restricting the workloads to these shows results similar to what was seen with two threads. With this restricted set, the virtual context architecture is up to 6.8% faster than the baseline architecture (also running the restricted set) with 448 physical registers. With the full set of workloads, the VCA is only 4.5% faster. Even with the larger number of function calls and increased pressure on the physical register file, the virtual context architecture is still able to maintain a very high level of performance with as few as 192 physical registers.

Supporting four simultaneous threads on a pipeline is only worth it if it can provide a performance advantage over a pipeline supporting either a single thread or two threads. Figure 7.6 presents the speedup of the four thread pipeline over a pipeline which supports only a single thread. The results show that there is a large

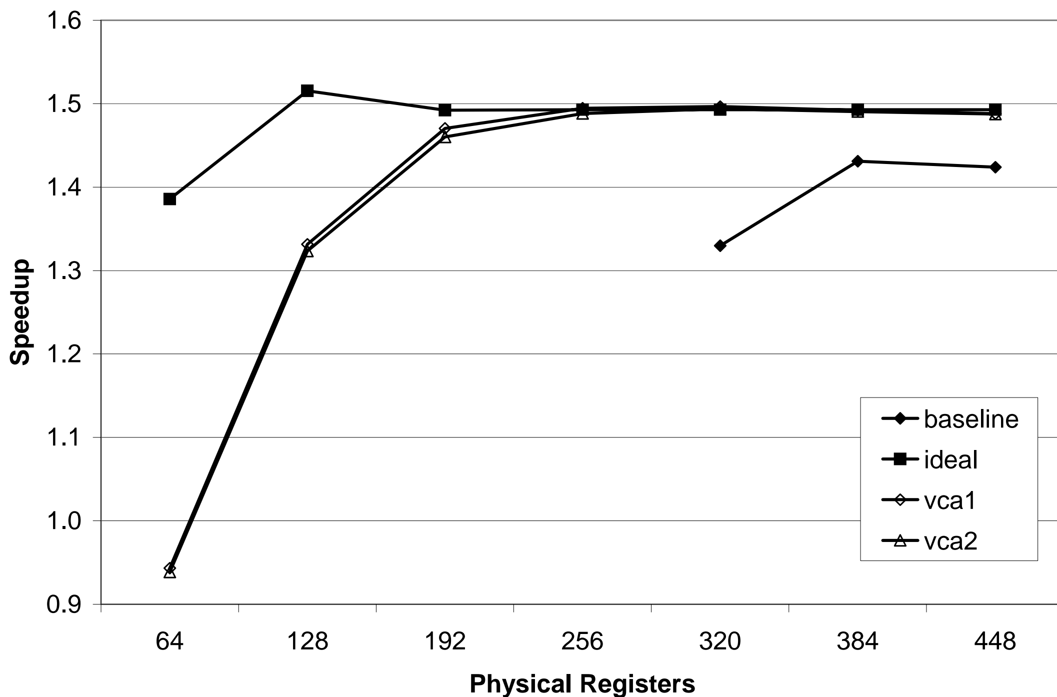



**Figure 7.6: Four Threaded SMT Speedups**

The speedups of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations in a four threaded simultaneous multithreading four issue pipeline. The speedups are relative to a single threaded four issue pipeline. The results are given for a range of physical register file sizes.

performance advantage in supporting four threads. The baseline architecture provides a 43% speedup over a single thread, compared to a 29% speedup for two threads. The virtual architecture provides an even larger speedup. With four threads, vca1 is 50% faster than a single thread while only 32% with two threads. The virtual context architecture is able to take advantage of the register window binaries to provide a 6.5% increase in speedup compared to the baseline with 448 physical registers. More importantly, as was seen in the execution time results, the virtual context architecture is able to maintain this speedup over a large range of physical register files. The speedup remains approximately constant with decreasing numbers of physical registers, all the way to 256 physical registers, the number of registers required to hold just the architectural state of four threads. The baseline architecture requires more than 256 physical registers to run. With 192 physical registers, there is only a slight drop in speedup of 2.5%. With 128

physical registers, the virtual context architecture begins to slow down rapidly. At this point the speedup is reduced to only 33% compared to the 50% we see with a larger register file. With 64 physical registers, the virtual context architecture with four threads is finally slower than the single threaded baseline architecture with 256 physical registers.

In a four thread simultaneous multithreading pipeline, the virtual context architecture is able to achieve a 4-12% improvement in execution time over the baseline architecture. The VCA is able to maintain a high level of performance over a large range of physical register file sizes. The virtual context architecture provides a nearly ideal implementation of register windows with a physical register file size of 256 registers or more. The four threaded virtual context architecture is able to achieve a 50% increase in performance over a single thread pipeline compared to a 30% increase for two thread SMT. The speedup of vca1 is 6% to 17% higher than the speedup of the baseline architecture with the same number of physical registers. The virtual context architecture with 192 physical registers provides a 5% greater speedup than the baseline architecture with 448 physical registers.

### 7.2.2 Data Cache Accesses

This section examines the data cache accesses of the baseline architecture, ideal register window architecture and virtual context architecture in a four thread simultaneous multithreading pipeline. Figure 7.7 presents the data cache access results. The results are normalized to the data cache accesses of the baseline architecture with 448 physical registers. The trends of the three architectures are consistent with those seen with the single thread pipeline and two thread pipeline. Unlike the execution time results, there are no exceptions to the basic trends. Both the baseline architecture and ideal register window architecture generate fewer data cache accesses as the number of physical registers is decreased. The virtual context architecture generates more data cache accesses as the number of physical registers is decreased.

The virtual context architecture is able to provide a data cache accesses savings compared to the baseline architecture with large register file sizes. The sav-

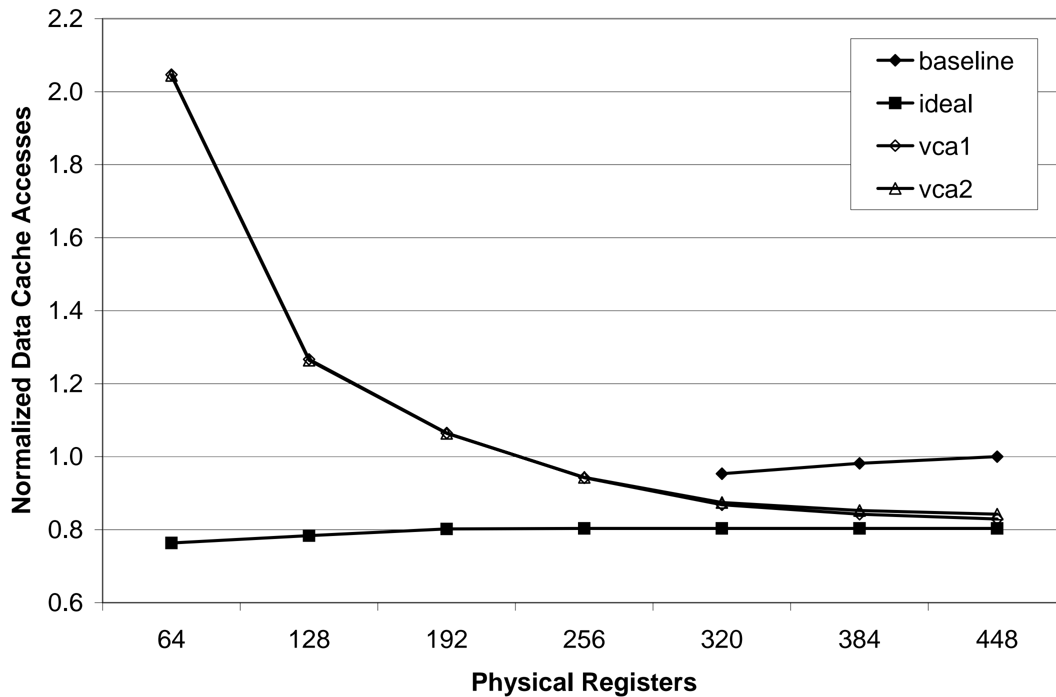



**Figure 7.7: Four Thread Data Cache Accesses Comparison**
The data cache accesses of the baseline architecture (baseline), ideal register window architecture (ideal), and two different virtual context architecture configurations. The data cache accesses are normalized to the data cache accesses of the baseline architecture with 448 physical registers. The results are given for a range of physical register file sizes.

ings are a result of the removal of explicit save and restore instructions from the register window binaries. With 448 physical registers, the savings is 17%. As the number of physical registers is decreased, the VCA generates more spills and fills which in turn cause more data cache accesses. Thus, with 384 physical registers, the savings has decreased to 14% fewer than the baseline with the same number of physical registers. With 320 physical registers, the difference becomes 8.5%. These savings are approximately the same as was seen with the three largest physical register file sizes studied in the two thread experiments. In that case, the savings started at 16% and dropped to 5.5%. With 256 physical registers, the generation of spills and fills has not overcome the benefit of register windows. The virtual context architecture generates 6% fewer data cache accesses than the baseline did with 448 physical registers. With 192 physical registers, the spills and fills start to dominate the register window savings. At this point, the virtual context

architecture is generating nearly 6% more data cache accesses than the baseline did with 448 physical registers. The generation continues to grow to 26.5% with 128 physical registers and 105% with 64.

In a four threaded simultaneous multithreading pipeline with a large physical register file, the virtual context architecture is still able to provide nearly ideal register window performance with respect to the data cache. With 448 physical registers, vca1 generates 2.5% more accesses than ideal. This is slightly greater than the 1.3% generated with the largest physical register file in the two thread experiments. The additional threads introduce additional architectural state, which needs to be managed and which in turn leads to more spills and fills. The virtual context architecture steadily generates more data cache accesses compared to ideal as the number of physical registers is decreased. With 384 physical registers, the difference between vca1 and ideal becomes 4%. With 320, the difference is 6.5%. With 256, it rises to 14%. The trend continues until with just 64 physical registers, the virtual context architecture is generating nearly 130% more data cache accesses than ideal. For a set physical register file size, the four thread pipeline always generates more data cache accesses relative to ideal than the two thread pipeline. For example, with 256 physical registers, vca1 in the four thread pipeline generates 14% more data cache accesses than ideal, while in the two thread pipeline the difference is only 4.5%.

Restricting the workloads to those which contain only benchmarks which have a high frequency of function calls has the same affect on the data cache accesses with four threads, as was seen in two threads. In relation to the data cache accesses generated by the baseline architecture, the cache savings provided by the virtual context architecture is increased with the restricted set of workloads. This is true at every size of physical register file. With 448 physical registers, vca1 generates 20% fewer data cache accesses using the restricted workloads, versus 17% with the full set. With 384 physical registers, the restricted has a difference of 17% versus 14% for the full. In relation to ideal, with large physical register files, the virtual context architecture running the restricted set of workloads generates more relative data cache accesses than with the full set of workloads. However,

as the number of physical registers is decreased, this trend reverses itself until the opposite is true. These effects were also seen in the two thread experiments. With 448 physical registers, vca1 running the restricted workloads generates 3.5% more data cache accesses than ideal, while with the full set of workloads the difference is 2.5%. With 384 physical registers, restricted is 5%, the full set is 4%. With 320, the numbers are 7% and 6.5% respectively. With 256 physical registers, the situation is reversed. The virtual context architecture running the restricted workloads generates 12% more than ideal while the full set generates 14%. The relative difference continues to increase as the size of the physical register file is decreased. With 192 physical registers, restricted is 19.5% while the full set is 26%.

Even with four threads, the virtual context architecture is able to provide a data cache savings over the baseline architecture at every physical register file size. With 448 physical registers, the savings is 17%. The virtual context architecture provides approximately the same level of savings for the largest three register file sizes studied with four threads as was seen with the largest three files with two threads. The VCA is close to the data cache accesses of ideal with a large physical register file, within 2.5% with 448 physical registers. The rate at which the difference between the virtual context architecture and ideal grows is less with four threads than with two. However, at any given physical register file size the VCA will generate more data cache accesses relative to ideal with four threads than with two. Restricting the workloads to those in which all the benchmarks have a high frequency of function calls has the same affect as it did with two threads.

## 7.3 Threads Study

This section examines the virtual context architecture's performance with regard to the number of threads supported in the pipeline. It combines the results obtained in this chapter with the results in the previous chapter to investigate the affects of adding threads to a virtual context architecture pipeline. All of the results in this section are based on a four issue pipeline with two data cache ports. The

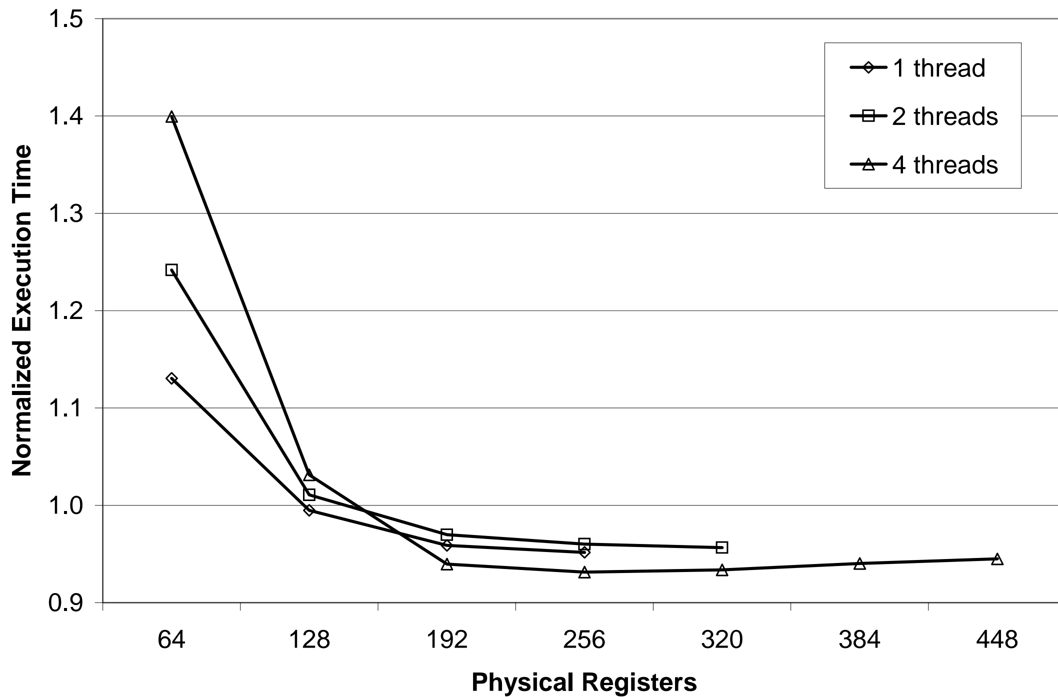



**Figure 7.8: Threads Execution Time Comparison**
The execution time of the virtual context architecture in a single thread pipeline, two thread pipeline and four thread pipeline. The results are normalized to the execution time of the baseline architecture with a full set of physical registers on a pipeline with the same number of threads. The multithread workloads are restricted to those composed solely of benchmarks which make a function call at least once every 500 instructions. The results are presented for a range of physical register file sizes. The results for each pipeline have an upper limit of physical registers equal to the full set required for the baseline architecture.

vca1 configuration from each experiment is used in the results in this section. Three things are investigated. First, the relative execution time of the virtual context architecture to the baseline architecture with a full set of physical registers is examined. Second, the relative number of data cache accesses of the virtual context architecture is compared to the baseline architecture with a full set of registers. Third, the potential speedups of the virtual context architecture and baseline architectures are compared

### 7.3.1 Execution Time Study

This section examines the execution time of the virtual context architecture relative to the baseline architecture with a full set of physical registers. Figure 7.8

presents the relative execution time results for a single thread, two threads and four threads. The results are normalized to the execution time of the baseline architecture with a full set of physical registers on a pipeline with the same number of threads. The results show the loss of performance of each pipeline as the number of physical registers is decreased. The multithread workloads are restricted to those used in the single thread register window studies. These are benchmarks which make at least one function call every 500 instructions.

The virtual context architecture is able to provide a performance advantage in all three pipelines. The maximum performance advantage of each pipeline is relatively similar. The single thread pipeline has a maximum advantage of 5%, the two thread pipeline has a maximum advantage of 4.5%, and the four thread has a maximum advantage of 7%. The one and two thread pipelines provide nearly the same benefit, while the four thread is 2% better. As mentioned in Chapter 6, the performance benefit obtained from register windows is relative to the removal of load and store instructions. The performance improvement is dependent on the relative cost of executing these instructions. The cost of executing an instruction is inversely related to the probability of there being another independent instruction that could be executed instead. In other words, the cost of executing an instruction is in some respects zero if there are no other instructions that could be issued instead that cycle. The probability of this occurring is much higher in a one or two thread pipeline than in a four thread pipeline. However, the results of each pipeline were obtained running a completely different set of workloads. The workloads are composed of the same set of benchmarks, but as was shown in Section 6.1.1, the benefits obtained from register windows has a large variation across the benchmarks. Therefore, it is possible that the maximum performance difference is actually a result of an over representation of high benefit benchmarks in the four thread workloads.

The decrease in execution time of all three pipelines as the number of physical registers is decreased is remarkably similar. The execution time of the virtual context architecture remains relatively constant with 192 or more physical registers. The data suggests that decrease in performance is independent of the amount of

architectural state and instead is solely a factor of the pipeline configuration. Each of the three pipelines is able to achieve near its maximum performance with as many physical registers as reorder buffer entries. In contrast, the baseline architecture must reserve one physical register for each architectural register and have additional registers for renaming. With 128 physical registers, the execution time of all three pipelines is approximately equal to the execution time of the pipeline using the baseline architecture. At this point, an equilibrium has been reached between the benefit of register windows and the cost of generating spills and fills. The performance also suffers from the lack of physical register available for rename.

With fewer physical registers, the execution time of all three pipelines increases dramatically. With so few physical registers, two factors are governing the performance: a lack of rename registers and the generation of many spills and fills. The lack of rename registers limits a pipeline's ability to keep many instructions in the instruction queue, which in turn decreases the probability of independent instructions. As mentioned previously in this chapter, multiple threads increases the probability of finding independent instructions in a fixed number of instructions. Therefore, the four thread pipeline should be least affected by this, followed by the two thread and finally the single thread. The generation of spills and fills increases the number of operations that need to be executed to complete a given set of code. This factor will affect the pipeline based on the relative cost of an additional operation. As mentioned previously in this section, the greater the probability of finding independent instructions, the greater the cost of additional operations. Therefore, this will affect the single thread pipeline the least, followed by the two thread and finally the four thread. The results show that the second factor dominates. With 64 physical registers, the relative execution time of the four thread pipeline is much higher than the two thread which is much higher than the single thread.

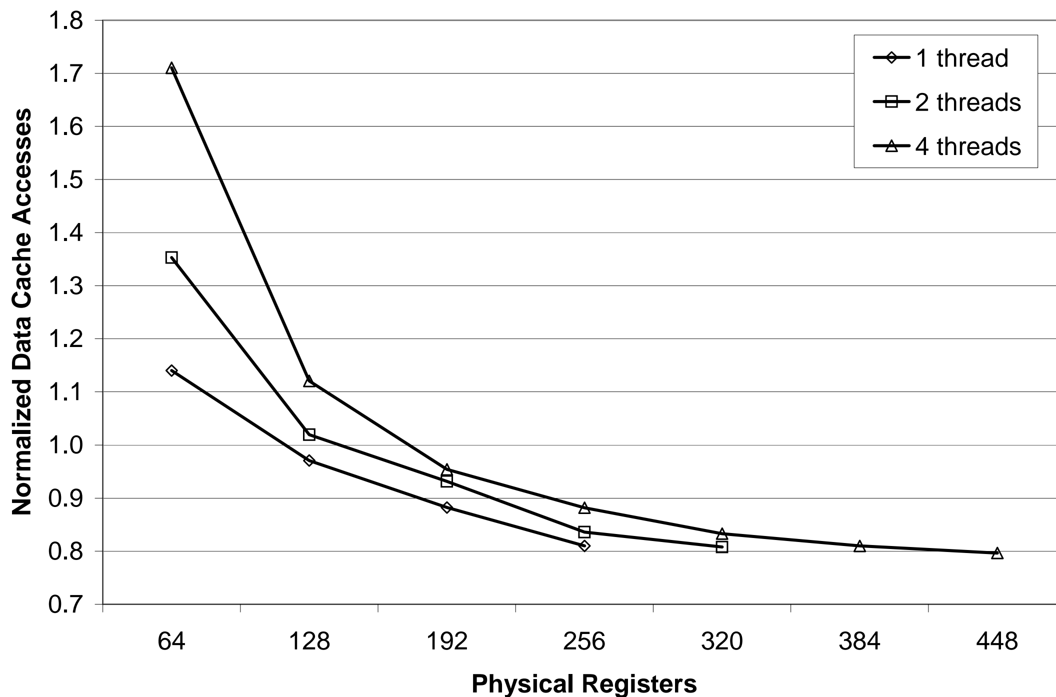



**Figure 7.9: Threads Data Cache Access Comparison**
The data cache accesses of the virtual context architecture in a single thread pipeline, two thread pipeline and four thread pipeline. The results are normalized to the data cache accesses of the baseline architecture with a full set of physical registers on a pipeline with the same number of threads. The multithread workloads are restricted to those composed solely of benchmarks which make a function call at least once every 500 instructions. The results are presented for a range of physical register file sizes. The results for each pipeline have an upper limit of physical registers equal to the full set required for the baseline architecture.

### 7.3.2 Data Cache Accesses Study

This section examines the data cache accesses of the virtual context architecture relative to the baseline architecture with a full set of physical registers. Figure 7.9 presents the relative data cache accesses for a single thread, two threads and four threads. The results are normalized to the data cache accesses of the baseline architecture with a full set of physical registers on a pipeline with the same number of threads. The multithread workloads are restricted to those used in the single thread register window studies. These are benchmarks which make at least one function call every 500 instructions.

The virtual context architecture is able to decrease the number of data cache accesses for a wide range of physical register file sizes, regardless of the number of threads the pipeline supports. In each case, the VCA provides approximately a 20% savings in data cache accesses with a full set of physical registers. As the number of physical registers is decreased, all of the pipelines generate more data cache accesses. Unlike the execution time results of the previous section, the data cache results show a very strong ordering based on thread. At every physical register file size, the four thread pipeline generates more data cache accesses relative to its baseline architecture than the two thread pipeline does to its baseline. Similarly, two threads always generates more than single thread. In all three pipelines, the virtual context architecture generates fewer data cache accesses than the baseline architecture if there are at least 192 physical registers. With 128 physical registers, the single thread generates slightly fewer accesses than the baseline. The two thread pipeline generates about the same number of accesses. The four thread pipeline generates over 10% more data cache accesses than the baseline. With 64 physical registers, all three pipelines generate more data cache accesses than the baseline architecture. The results show that unlike the execution time, the data cache savings are affected by the threads. For a given number of physical registers, the more architectural state that the pipeline has, the more spills and fills it will generate to move this state between the physical register file and data cache.

### 7.3.3 Speedup Study

This section examines the potential speedup associated with the virtual context architecture and baseline architecture. Figure 7.10 presents the speedup results. Results are presented for the virtual context architecture and baseline architecture in a single thread pipeline, two thread pipeline and four thread pipeline. The speedups are relative to the baseline architecture with a full set of physical registers (256) in a single thread pipeline. The results for each pipeline have an upper limit of physical register file size equal to the full set required for the baseline. This many physical registers allows the baseline architecture to achieve

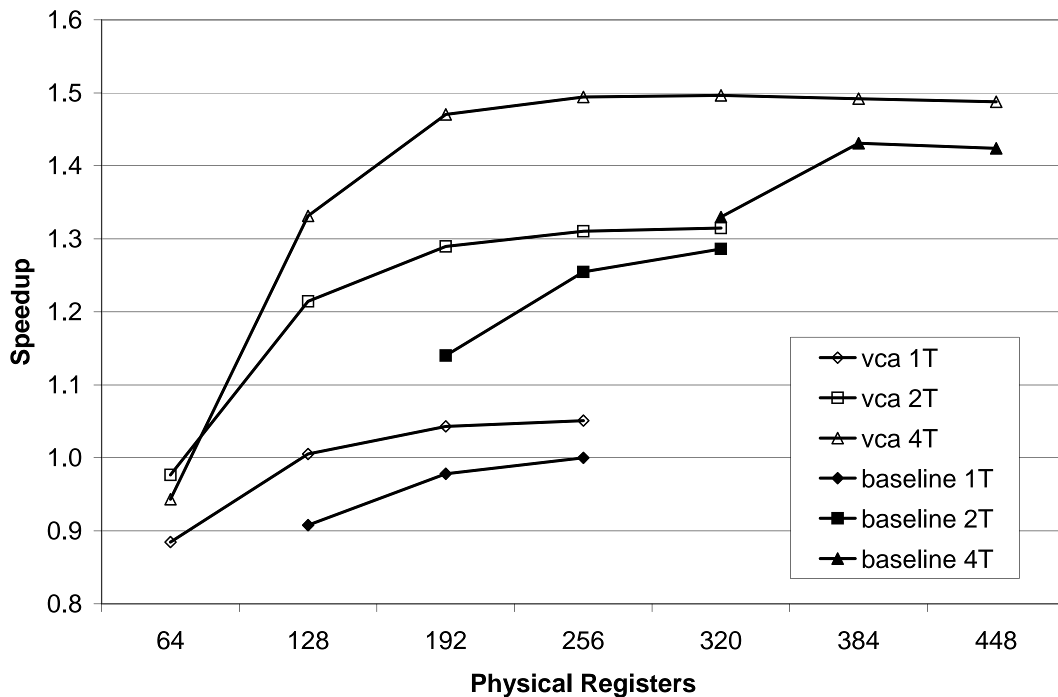



**Figure 7.10: Thread Speedup Comparison**

The speedup of the virtual context architecture and baseline architecture in a single thread pipeline (1T), two thread pipeline (2T) and four thread pipeline (4T). The speedups are relative to the baseline architecture with 256 physical registers in a single thread pipeline. The results are presented for a range of physical register file sizes. The results for each pipeline have an upper limit of physical registers equal to the full set required for the baseline architecture. This upper limit is also the upper limit on the performance for that configuration.

its maximum performance, and previous experiments have shown that the virtual context architecture achieves very near ideal performance also.

The virtual context architecture is able to provide a higher speedup at every size of physical register than the baseline architecture. The ability of the four thread virtual context architecture to maintain its performance even with a small register file greatly increases the potential speedup that can be achieved at each physical register file size. With 384 or 448 physical registers, the four thread baseline architecture is able to provide a speedup of 43%. The four thread virtual context architecture can provide a slightly higher speedup of 50%. With 320 physical registers, both the two and four thread baseline architectures provide a speedup of approximately 30%. While the two thread VCA can provide a similar speedup,

the four thread still provides a speedup of 50%. This is 20% higher than any baseline architecture can provide. With 256 physical registers, the four thread baseline can no longer run. The two thread baseline provides a speedup of 25%. The virtual context architecture can provide a speedup with any of its pipelines at this point. With a single thread, the speedup is only 5%. With two threads, a 31% speedup is possible, 6% higher than any baseline architecture. With four threads, the speedup is still 50% or double that achievable by any baseline architecture. With 192 physical registers, the single thread baseline architecture itself begins to slow while the two thread baseline architecture slows down dramatically. The two thread baseline is 16% faster than the single baseline with the same number of physical registers. The single thread virtual context architecture is 6% faster than the baseline with the same number of physical registers. The multithread virtual context architectures are still able to maintain most of their performance. The two thread VCA is 32% faster than the single thread baseline or double the speedup achieved with the two thread baseline. The four thread virtual context architecture is 50% faster than the baseline. This is over three times the speedup achieved by the two thread baseline. With 128 physical registers, the two thread baseline can no longer run and the performance of the single thread baseline drops even further. The single thread virtual context architecture is 10% faster than the baseline, the two thread is 33% faster and the four thread is 47% faster.

## 7.4 Summary

The results in this chapter showed that the virtual context architecture performs very well in a simultaneous multithreading pipeline. The VCA enables the use of register windows in an SMT processor with no additional physical registers, providing better performance than the baseline architecture. Both two and four thread SMT pipelines provide significant speedups of 30% and 50% respectively in a four issue pipeline over a single thread. The performance of the virtual context architecture relative to the baseline architecture is virtually independent of the number of threads. Thus, while multiple threads in a conventional architecture

require much larger physical register files, the virtual context architecture is able to achieve the same performance with no more physical registers than a single thread pipeline. This greatly increases the potential speedup achievable with moderate size register files. For example, with 192 physical registers, the best achievable performance for the baseline architecture is with two threads for a 16% increase over the single thread. The virtual context architecture with this size register file provides a 32% improvement with two threads or by using four threads it can achieve a 50% speedup.

The virtual context architecture is able to achieve nearly ideal register window performance with a large physical register file. With a smaller number of physical registers, the VCA must generate more spills and fills. The virtual context architecture's performance suffers with respect to ideal, which never generates spills and fills. In the next chapter, we investigate several techniques for reducing the number of generated spills.

# Chapter 8

# Spill Optimization

A study of the results obtained in Chapter 6 shows that up to 80% of the spills generated by the virtual context architecture are not needed. These extra spills occur when the value of an architectural register is spilled out of the physical register file to the data cache, but the architectural register is rewritten without the spilled value being brought back from the data cache. In other words, the value is spilled after its last use. This chapter evaluates three techniques that can be used to eliminate some of these unnecessary spills. All of the studies in this chapter were done with a four issue out-of-order pipeline, see Table 4.2 for a description of this pipeline. The experiments are done with physical register file sizes of 64, 128 and 192. These smaller sizes are where the performance of the virtual context architecture begins to diverge from ideal. The first three sections of this chapter describe the three techniques used to optimize spills: delay queue, register window deallocation and dead value information. These sections also determine the optimum size of any structures needed to implement the optimization. The final section evaluates the effectiveness of these techniques at reducing the number of spills generated by the virtual context architecture.

## 8.1 Delay Queue

Examining the results of our previous experiments shows that for a large fraction of the extra spills, the time between when the spill is executed and when the value is overwritten is relatively short. This section presents a completely architectural technique that makes use of this observation to eliminate some of the unneeded spills. We call this technique the delay queue. The idea of this tech-

nique is to delay the completion of these spills for a short time. If the value is overwritten during this time, the spill can be squashed.

The delay queue is a small queue that is placed between the architectural state transfer queue and the store buffer. Normally, when a spill is executed, the value being spilled and its address are placed directly into the store buffer. Instead, the spill is now placed into the delay queue. The operation of the delay queue requires two things: a delay and squashing.

First, the value is held in the delay queue for some set amount of time. If the value has not been squashed in that amount of time, the value and address are moved from the delay queue into the store buffer. The time that values are held in the delay queue is a parameter that needs to be determined. If the time is too long, a very large queue will be needed to hold the values. If the time is too short, many of the opportunities to squash will be lost. Instead of a fixed delay, the delay queue can operate based on its size. In this case, the delay queue is set up as a fixed size queue. As long as there is at least one empty entry in the queue, all the spills are held indefinitely. When the queue becomes full, the oldest entry is moved to the store buffer. This is a more practical implementation and will ensure that values are held as long as possible.

Second, the operation of the delay queue involves catching opportunities to squash the spills. A spill can be squashed when a new value mapped to the same location (same register in the same context) is committed by the pipeline. To accomplish this, when a value is committed by the pipeline, its address is sent to the delay queue. The delay queue cams this address. Any delay queue entries that match the address are squashed. A squashed delay queue entry is never moved to the store buffer and therefore does not cause a data cache access. For a four issue pipeline, this involves checking up to four address each cycle. However, this check need not be on the critical path. It can be pipelined arbitrarily. A missed opportunity to squash a spill costs performance, but does not affect the correct execution.

The removal of these spills will result in a decrease in the data cache accesses generated. For the delay queue to be effective, the majority of extra spills must be

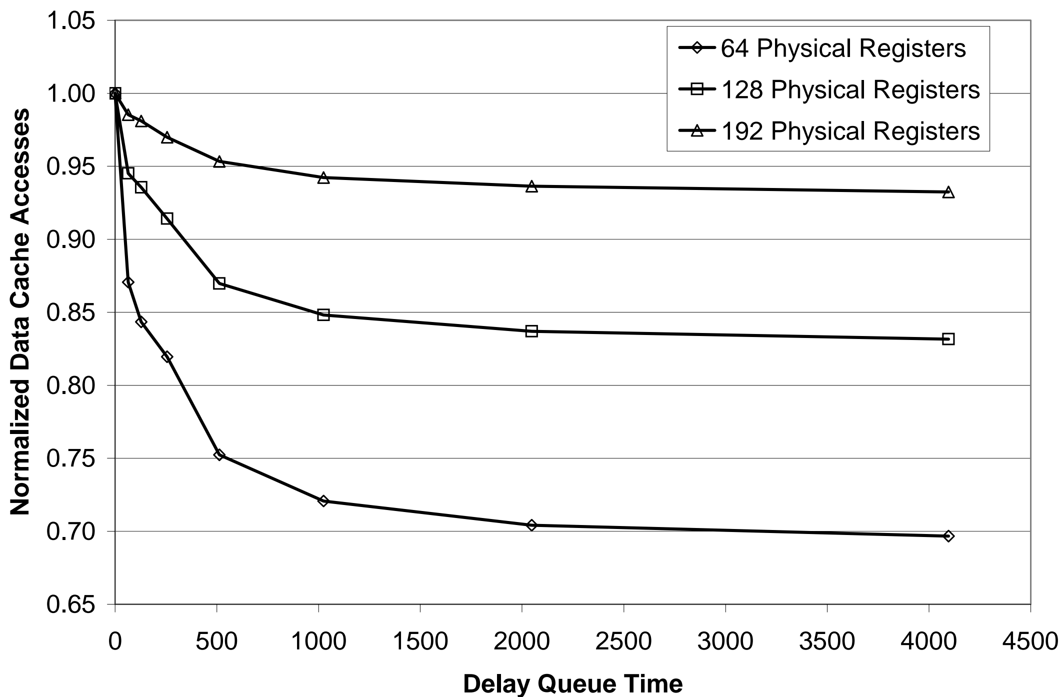



**Figure 8.1: Delay Queue Time Study**

The normalized data cache accesses of the virtual context architecture with a delay queue with different delay times. The delay queue time is studied from 0 cycles to 4096 cycles. The results are normalized to the 0 cycle (or no delay queue) data cache accesses. The results are provided for the three different physical register file sizes.

overwritten in a short amount of time. Otherwise, the delay needed will be too large and the queue size will be impractical. Figure 8.1 presents the relative data cache accesses with a range of delay times. No limit is imposed on the size of the queue. The results are presented for the three different physical register file sizes. The data shows that the majority of extra spills are overwritten in a relatively short amount of time. With a delay queue time of 512 cycles, the relative data cache accesses are already within 5% of their maximum decrease. The extent of the cache savings is very dependent on the size of the physical register file. With fewer physical registers, a larger percentage of the data cache accesses are the result of a spill. The percentage of extra spills remains relatively constant with the number of physical registers. Therefore, there is a higher potential gain with fewer physical registers.

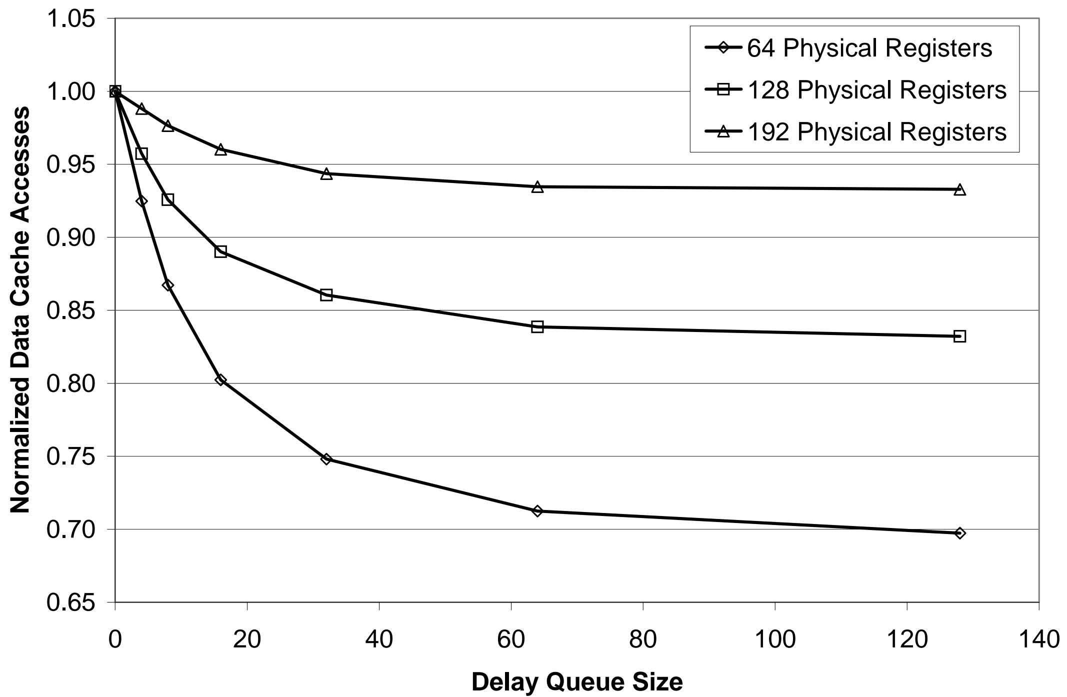



**Figure 8.2: Delay Queue Size Study**

The normalized data cache accesses of the virtual context architecture with a delay queue of different sizes. The delay queue size is studied from 0 to 128 entries. The results are normalized to the no delay queue data cache accesses. The results are provided for the three different physical register file sizes.

The previous results showed that the delay needed to remove a high percentage of extra spills is relatively low. The practicality of this technique however, is dependent on the size of the delay queue needed. Figure 8.2 presents the relative data cache accesses with various sizes of delay queue. The larger the physical register file the smaller the delay queue needed to realize most of the optimization. The results show that a 64 entry queue is large enough to delay the spill long enough to squash most of those that can be squashed. This is the size queue used in the evaluation of this technique.

## 8.2 Register Window Deallocation

The application binary interface we have defined for our architecture specifies that all of the communication between functions uses the non window registers.

Therefore, when a register window is deallocated, any physical register mapped to one of the windowed registers in that window can be freed. This section explores a technique for implementing this optimization.

The register window deallocation optimization is purely architectural and requires no changes to the existing instruction set architecture. The implementation of this optimization requires that when a return instruction is executed, all of the windowed registers in the deallocated register window be freed. Therefore, the hardware must keep track of which physical registers are mapped to windowed registers and which register window each of these registers is in.

One possible way to track this is to maintain a circular buffer of bit vectors with one bit for each physical register. A current index is kept, specifying which vector is currently active. As an instruction is committed, if the destination register is windowed, it sets the bit in the current vector. Thus, the vector specifies which physical registers are assigned to windowed registers in the current window. When the instruction deallocating the window is committed, the bit vector is ored with the free list vector to make the new free list (all windowed registers from the deallocated window immediately become free). The vector is then cleared. When an instruction allocating a new window is committed, the current index is incremented and the new current bit vector is cleared. Clearing the bit vector on allocations and deallocations naturally handles the circular nature of the buffer and the fact that it's a finite resource.

With n bit vectors, it is possible to maintain the information for the last n windows allocated. If the call depth is greater than n, the implementation will lose information about which physical registers were assigned to the oldest function's register window. For example, with a n of 2, say function A calls function B. At this point, the implementation is able to track which register window each windowed register belongs to. Now say function B calls function C. The implementation will lose the information on function A. Instead, that vector will now start tracking function C. When function C returns, the implementation can clear its registers. Similarly, when function B returns it also has the information to clear its registers. However, at this point there is no information for function A. However, once again

the bit vector will start to track the register usage of function A. When function A returns, only those registers defined between the return of B and the return of A will be cleared. Therefore, the only registers affected by the limited number of vectors will be those physical registers used before the call to function B and subsequently not spilled or overwritten. This implementation requires very little space and very little additional logic.

A study of the possible values of n shows that a very small value will capture most of the potential savings. The number of data cache accesses generated with a single bit vector is only 0.3% higher than with an infinite number of bit vectors. Two bit vectors is within 0.12% of infinite. Four bit vectors is within 0.04% of the data cache accesses generated with an infinite number of vectors. Eight bit vectors is enough to capture all of the information and yields the same number of data cache accesses as an infinite number of vectors. We choose to use four bit vectors because it is a good compromise between size and performance.

## 8.3 Dead Value Information

As part of the compilation process, register lifetime information is generated. This information is used throughout the compilation process for such things as register allocation and optimizations. The lifetime information can be used to determine the last use of any given register value. This is called dead value information. This is exactly the information required to remove the extra spills. This section describes how the instruction set architecture can be modified to allow this information to be inserted into the instructions.

The idea of using this information in the architecture has been explored before. Martin et al. [20] and Lo et al. [18] both used dead value information to reclaim physical registers as soon as the last use was committed. This is similar to what our optimization is trying to accomplish. However, although we will benefit from reclaiming the physical register faster, the main purpose is to prevent these dead values from being spilled.

We insert dead value information into the instruction set architecture in two ways. The first way is two mark the source registers of instructions in which this is the last use of the register. The way in which an instruction was marked was dependent on its binary layout. Specifically, if there were unused bits in the instruction itself, these bits were used to specify the last use of the source operand. If there were no unused bits in the instruction, a new opcode was used to specify the same operation but with a dead source register. In the case of a two source instruction, this required three new opcodes to represent the possible combination of dead source registers: only the first source dead, only the second source dead, or both sources dead. The Alpha ISA as used by our compiler and simulator had several unused opcodes that could be used for this purpose. The majority of common instruction types were modified to allow for dead value information. The notable exceptions were the load/store address operand and conditional branch instructions.

The second way in which dead value information was inserted into the pipeline was with the addition of special instructions. Two types of special instructions were used. The first special instruction was a modified call instruction. Any call to a function outside the compilation unit is done by loading the address of the function in a specific register ($27 or the procedure value register). The call instruction always uses the return address register as the destination. We added a new call instruction to the instruction set architecture. The instruction used these fixed source and destination registers and did not encode them in the binary. Instead, it encoded the live status of all 12 argument registers. Any call instruction that used these registers was instead replaced by the special instruction. The second type of special instruction was a dead value information only instruction. The sole purpose of this instruction is to communicate register lifetime information to the hardware. The instruction used one bit to specify either floating point or integer register file. The live status of all the windowed registers for the register file was then encoded into the instruction. The compiler inserted a pair of these instructions (one for floating point and one for integer) before every call instruction.

To implement this optimization, the compiler was modified to generate the conservative lifetime of all the registers used in a function just before the assembly was generated. As the compiler generated the assembly, each instruction referenced this lifetime information to determine if this instruction was the last use of any of the registers it was sourcing. If a register was on its last use, the register was marked in the assembly by adding 100 (for example $103 or $f129) to the register number. The compiler also used the special call instruction in all the appropriate cases and inserted the dead value information only instructions. The assembler was modified to recognize both the new instructions and the new register identifiers. If an instruction had one or more marked registers, the assembler determined if an encoding existed to specify the information. If an encoding existed, the new encoding was output. If no encoding was possible, the unmodified instruction was output. Finally, the simulator was modified to recognize the new instructions and all of the modified encoding.

## 8.4 Evaluation

This section evaluates the effectiveness of these three techniques in removing the extra spills. The three techniques were run individually and in two combinations. The first combination is composed of both the register window deallocation optimization and the dead value information optimization. Neither of these techniques requires very much hardware and they are completely compatible. The second combination consists of all three techniques. This represents the limit of what we could achieve. The results are presented in two subsections: data cache results and execution time results.

### 8.4.1 Data Cache Accesses

This section presents the data cache results. We use this to evaluate how many data cache accesses were optimized away. This is an indication of the effectiveness of the technique. The results are presented for a four issue pipeline with two data cache ports and for a four issue pipeline with one data cache port.

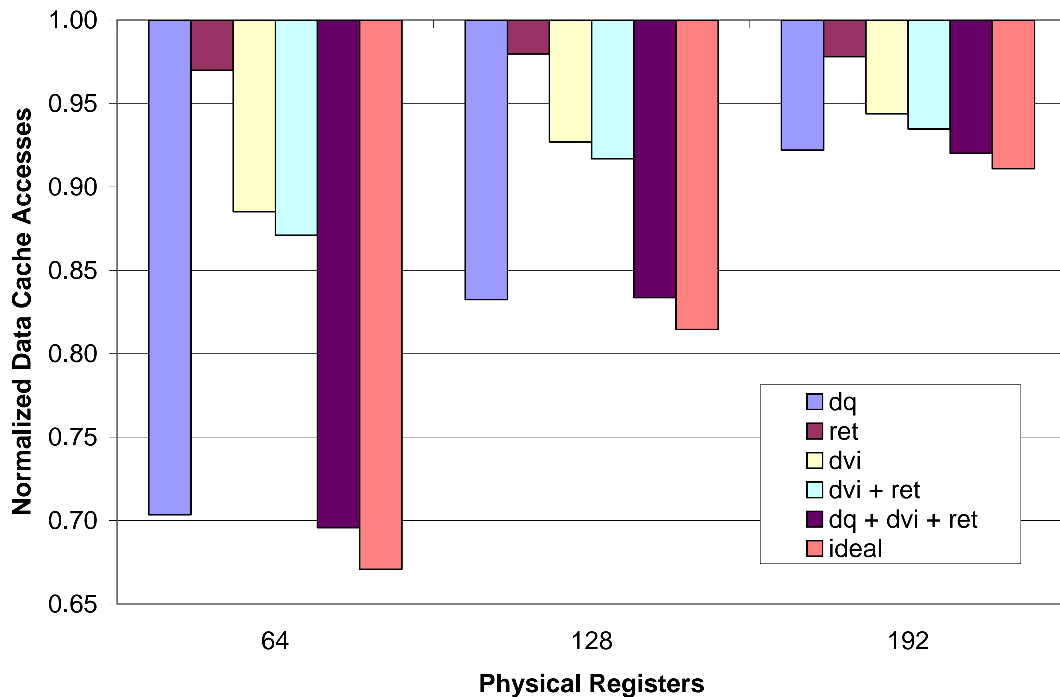



**Figure 8.3: Spill Optimization With Two Data Cache Ports**

The data cache accesses generated by the delay queue optimization (dq), register window deallocation (ret), dead value information optimization (dvi) and combinations of these techniques. The results are normalized to the data cache accesses generated by the unoptimized virtual context architecture with the same number of physical registers. The data cache accesses generated by the ideal register window architecture (ideal) are also included to provide a limit to the potential optimization.

The normalized data cache accesses that the two port pipeline generates with each technique investigated is presented in Figure 8.3. The results are normalized to the data cache accesses generated by the unoptimized virtual context architecture. The data cache accesses generated by the ideal register window architecture are also included. The ideal data cache accesses represent the theoretical limit that our optimizations can achieve.

The delay queue optimization proves to be very effective at removing the extra spills. At every size of physical register file studied, this optimization reduces the number of data cache accesses to within 3% of those generated by the ideal register window architecture. This is a decrease in data cache accesses of between 8% and 30%. The percentage of data cache accesses removed grows as the number of physical registers is reduced. However, the percentage of removed

data cache accesses compared to the difference between the unoptimized VCA and ideal remains relatively constant as the number of physical registers is reduced. These experiments were run with a 64 entry delay queue. Smaller queue sizes would reduce the savings.

The register window deallocation optimization proves to be the least effective at removing extra spills. Unlike the other two techniques, this is not a general purpose technique. Instead, it specifically targets a certain type of extra spill. Although all of the register windows become invalid when a function returns, even without any optimization it is possible for these registers not to be spilled. If another function call is made (even to a different function), as registers are written to by instructions in that function, they will overwrite and free the physical registers allocated by the previous function. This combination of factors leads to a relatively modest decrease of 3% in the data cache accesses. The percentage remains relatively constant with the number of physical registers. This would be expected because of the strong dependence of this optimization on the program behavior.

The dead value information optimization proves to be moderately effective, reducing the number of data cache accesses by up to 12%. The percentage of data cache accesses removed increases as the number of physical registers is decreased. However, unlike the delay queue, the increase is less than the increase in the number of data cache accesses generated. Therefore, the percentage of removed accesses relative to the difference between the unoptimized VCA and ideal decreases. With 192 physical registers, this optimization reduces the increase over ideal by over 50%. With 64 physical registers, the reduction is only 35%. The dead value information inserted by the compiler is conservative. A register is only marked dead if the instruction must be its last use, taking into account all the possible control flow of the program. In addition, the dead value information could not be encoded in every instruction type. Our results show that about 27% of committed instructions contain dead value information. Therefore, the dead value information is not capable of marking all the last uses, especially in the last use instructions. A full dead value specification is inserted at every call site. With more physical registers there is less pressure on the physical register

file, fewer spills and fills are generated, and there is a greater chance that the compiler will have been able to mark a register as dead. Thus, this technique is not as effective as the delay queue and is less effective with a smaller physical register file.

Two combinations of techniques were also evaluated. The first combination consists of the register window deallocation optimization and the dead value information optimization. These two techniques are low cost in terms of hardware and easily combined. The results show that the combination is more effective than either separate optimization. However, the combination is only slightly better than dead value information by itself, reducing the number of data cache accesses by an additional 1%. This can be explained by the overlap between the two techniques. Specifically, the dead value information will free some of the windowed registers before the function returns. These would also be freed by the register window deallocation optimization. The second combination employs all three techniques. In this case, there is virtually no difference between the combination and the delay queue by itself. The delay queue is able to remove almost every possible extra spill by itself.

These optimization techniques were also tested in a four issue pipeline with one data cache port. See Figure 8.4 for the results. The results are very similar to those seen in the two data cache port pipeline. The difference between the unoptimized virtual context architecture and ideal has increased with the loss of the data cache port. The delay queue removes a higher percentage of accesses, but the percentage removed relative to the difference between unoptimized and ideal remains the same. Thus, the delay queue is still able to remove almost all of the extra spills. The percentage of data cache accesses removed by the register window deallocation technique and dead value information technique remains the same. Therefore, the percentage relative to the difference has decreased. These two techniques are dependent on the benchmark themselves. In particular, they have the capability of removing a fixed number of spills based on the program itself. Therefore, they are less affected by the loss of the data cache port. The

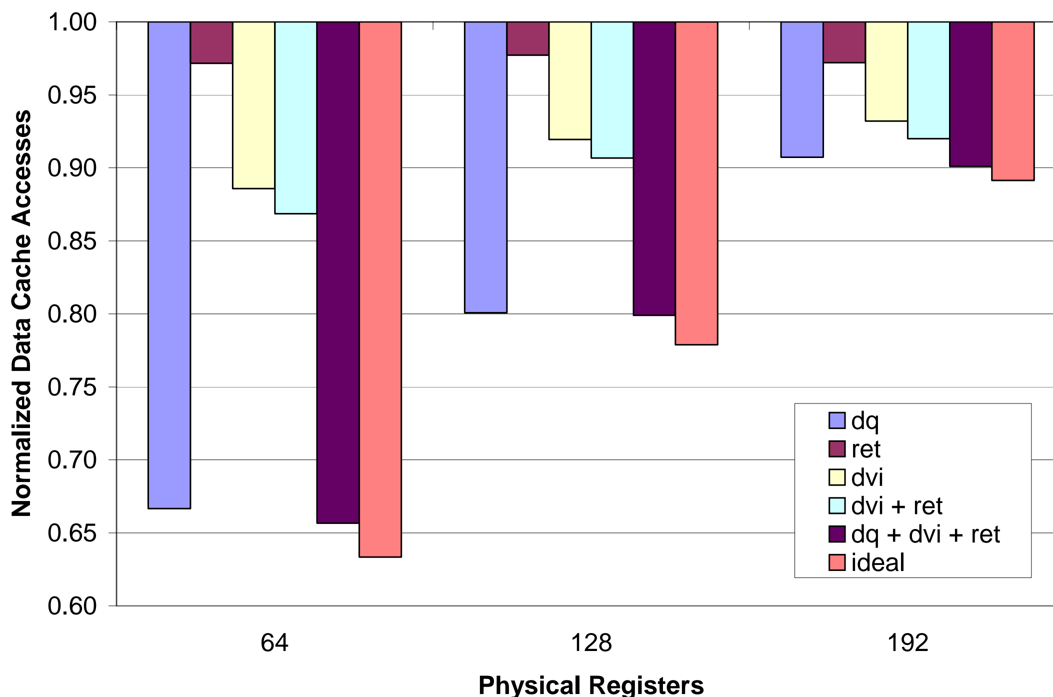



**Figure 8.4: Spill Optimization With One Data Cache Port**

The data cache accesses generated by the delay queue optimization (dq), register window deallocation (ret), dead value information optimization (dvi) and combinations of these techniques. The results are normalized to the data cache accesses generated by the unoptimized virtual context architecture with the same number of physical registers. The data cache accesses generated by the ideal register window architecture (ideal) are also included to provide a limit to the potential optimization.

dvi+ret combination behaves the same as the two techniques it is composed of, while the behavior of the combination of all three techniques behaves like the delay queue which dominates its performance.

### 8.4.2 Execution Time

This section evaluates the execution time effects caused by the use of the three optimization techniques. The overhead associated with the virtual context architecture is the cost of generating spills and fills. The previous section showed that these optimization techniques can significantly reduce the data cache accesses associated with this overhead. The results are presented for the same configurations tested in the previous section in a pipeline with two data cache ports and a pipeline with one data cache port.

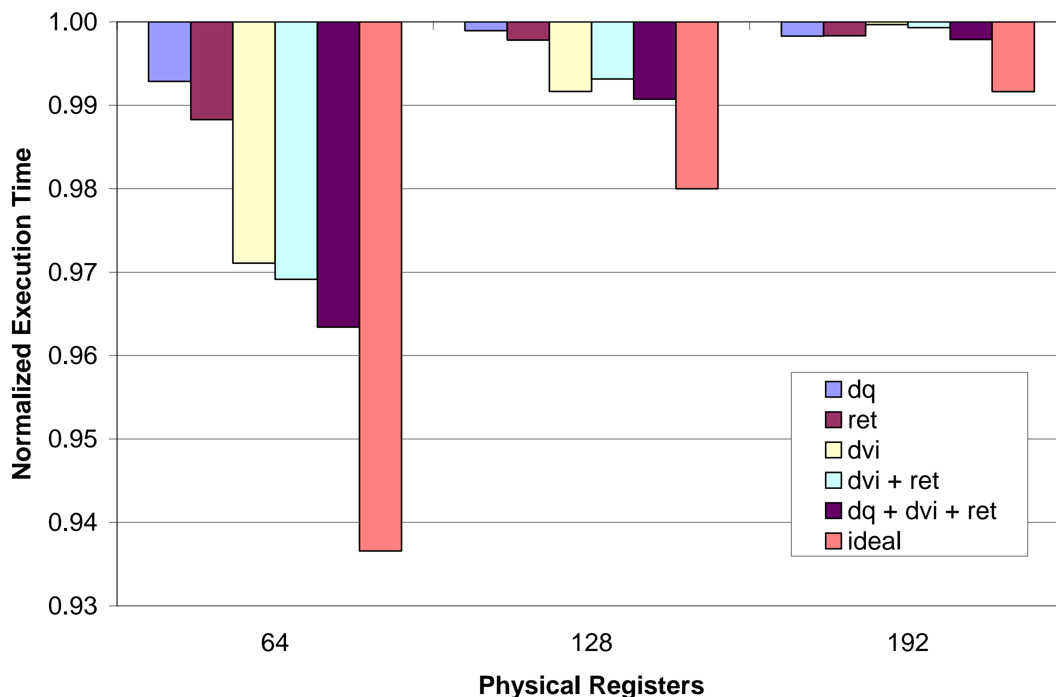



**Figure 8.5: Optimization Execution Time With Two Data Cache Ports**
The relative execution time of the delay queue optimization (dq), register window deallocation (ret), dead value information optimization (dvi) and combinations of these techniques. The results are normalized to the execution time of the unoptimized virtual context architecture with the same number of physical registers. The execution time of the ideal register window architecture (ideal) is also included to provide a limit to the potential optimization.

The normalized execution time of the virtual context architecture with these three optimization techniques is presented in Figure 8.5. The results are normalized to the execution time of the unoptimized virtual context architecture. The execution time of the ideal register window architecture is also included. The ideal execution time represents the theoretical limit that our optimizations can achieve.

Although the delay queue has the largest effect on the cache accesses, it has the smallest effect on the execution time, with a difference in execution time of less than 0.8% relative to the unoptimized execution time. The cost associated with a spill is composed of two factors. The first factor is the amount of time that the pipeline stalls while waiting for the spill to issue so that it can reuse the physical register. The second factor is the overhead of additional entries in the store buffer and their eventual writeback to the data cache. Unlike the other two optimi-

zations, the delay queue only alleviates the second factor. Spills are still issued, but they are sent to the delay queue where the majority are eventually squashed. In a pipeline with two data cache ports, the cost of the second factor is relatively small.

The register window deallocate optimization has a greater effect on the execution time than the delay queue. With 64 physical registers, there is a reduction in execution time of 1.2%. Although this optimization only affects a small number of spills, the physical registers are immediately freed. This removes the entire cost of the spill. The optimization also occurs when the pipeline is transitioning from one function context to another. This is one of the points in the execution when extra pressure is exerted on the physical register file. It is forced to try to hold the register state of multiple active contexts. Thus, the freeing of a large number of physical registers from one of these contexts has a greater benefit to the execution time than the removal of the data cache accesses would necessarily indicate.

The dead value information optimization provides the most benefit to the execution time. Although the reduction is very small with 192 physical registers, the smaller registers files experience a nearly 50% reduction in their execution time relative to the ideal execution time. This corresponds to a decrease of 3% for 64 physical registers and 1% for 128. All of the techniques show a larger impact as the register file is decreased. With the larger register file, the cost of the spills is more easily absorbed in the execution time. With the smaller register files, the large number of spills generated begins to have a much greater performance impact and the execution time impact of these optimizations increases.

The combination of dead value information and register window deallocation optimizations has similar performance to just the dead value information. As was seen with the data cache access results, the spills that these optimizations affect overlap. The combination of all three optimizations does show a significant decrease in execution time relative to the other combination. With 64 physical registers there is a further 0.6% reduction in execution time. The addition of the delay queue does provide savings. In fact, the savings are almost additive. The modest performance improvement gained by the removal of so many data cache

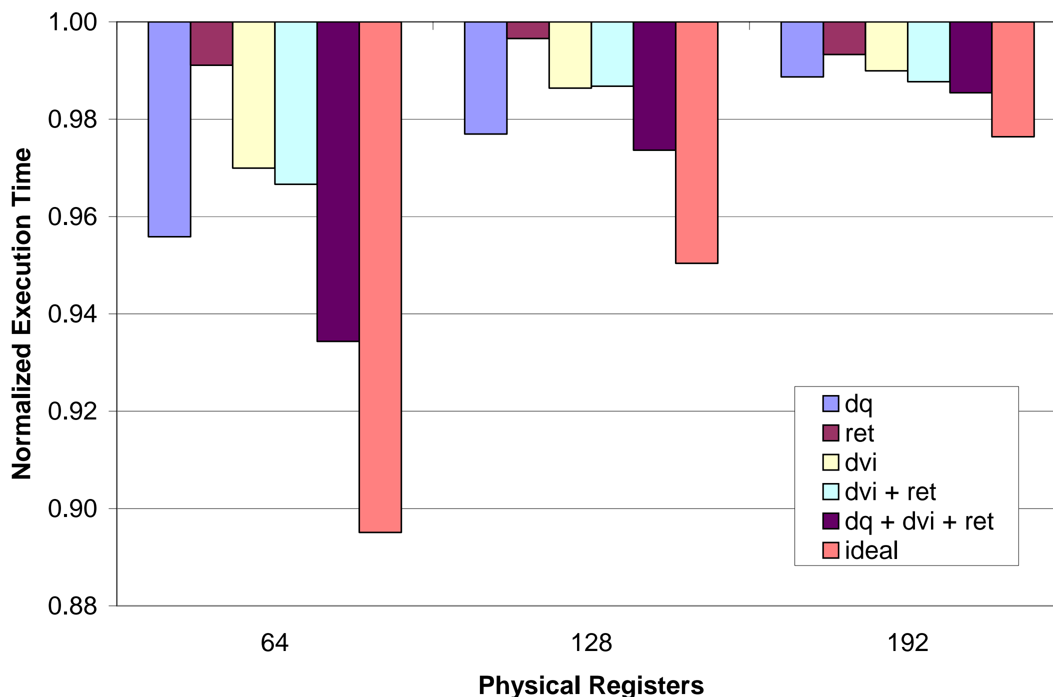



**Figure 8.6: Optimization Execution Time With One Data Cache Port**
The relative execution time of the delay queue optimization (dq), register window deallocation (ret), dead value information optimization (dvi) and combinations of these techniques. The results are normalized to the execution time of the unoptimized virtual context architecture with the same number of physical registers. The execution time of the ideal register window architecture (ideal) is also included to provide a limit to the potential optimization.

accesses is not diminished very much from the much smaller reduction in data cache accesses caused by the other two optimizations.

The execution time results of these optimizations in a one data cache port pipeline are presented in Figure 8.6. The configurations and statistics are the same as for the two data cache port results. With a single data cache port, the relative cost of a data cache access increases and the difference between the unoptimized virtual context architecture and ideal register window architectures increase. We would expect to see a more significant contribution by the optimizations. The results do show a significant decrease in the execution time with 192 physical registers. In contrast, the two data cache port results show almost no change in the performance with 192 physical registers. This shows that the cost of

spills is much more significant with 192 physical registers with only a single data cache port that when there are two data cache ports.

The delay queue has a much different effect on the execution time in the pipeline with only a single data cache port. This optimization provides the largest performance benefit at every physical register file size, reducing the execution time by between 1.2% and 4.5%. This is in contrast to the two port pipeline where it provided the least benefit. As mentioned in the two port execution results, the delay queue only removes the store buffer and cache access costs of a spill. With only a single data cache port, these become a much more significant percentage of the overall cost. With every size of physical register file, the delay queue is able to decrease the difference between the virtual context architecture and ideal by almost half.

The register window deallocation optimization with a larger register file has approximately the same percentage reduction in execution time as it did in the two data cache port pipeline. This is true, even with the increased overall cost of a spill. With the increase in the difference between unoptimized and ideal, this represents a smaller percentage of the difference. Like the other optimizations, the execution time benefit with 192 physical registers is much greater than in the two data cache port pipeline.

The dead value information optimization has very similar results to those achieved with two data cache ports. Like the other optimizations, it also experiences an increase in benefit with the larger register files. With 192 physical registers, this optimization reduces the execution time an additional 1%. With 128 physical registers, the percentage drops to only 0.5%. Finally, with 64 physical registers, the execution time is only reduced an additional 0.2% compared to the two data cache port pipeline. With the increased difference between the unoptimized virtual context architecture and the ideal register window architecture, this translates to a smaller percentage of the difference. While with 64 physical registers in the two data cache port pipeline this optimization was able to remove almost 50% of the overhead associated with the VCA, with one data cache port the dead value information optimization removes under 30% of the difference.

The dead value information and register window deallocation combination provides the same performance improvement as it did with two data cache ports. Since both of these techniques behave the same individually, it makes sense that the combination also does. The combination of all three provides the best performance. With 64 physical registers, the execution time of this configuration is 6.5% less than the unoptimized execution time. Across the whole range of physical register files studied, it reduces the difference between the virtual context architecture and the ideal register window architecture by between 53% and 63%. Although the delay queue is not quite additive with the first combination, there is still a substantial performance improvement.

## 8.5 Summary

This chapter examined three techniques for reducing the number of spills generated by the virtual context architecture. The first technique examined was the delay queue. This was an architectural solution which delayed the transfer of spills values to the store buffer. If the value was overwritten before being transferred, the spill could be squashed. Of the three techniques, this one reduced the data cache accesses by the most, to within 3% of ideal. However, because it only eliminates the cache access itself and not the generation of the spill, a significant performance savings was only observed in a pipeline with one data cache port. In this context, it reduced the difference between the virtual context architecture and the ideal register window architecture by half.

The second technique examined was an optimization to register window deallocation. The application binary interface specifies that all the windowed registers in a function are dead when the function returns. Our implementation used a small set of four bit vectors to track which physical registers belong to which context. This technique had very modest hardware costs, but also very modest improvements, both in terms of performance and reducing the data cache accesses.

The third technique was the introduction by the compiler of dead value information into the instruction set. Requiring virtually no new hardware, this technique

however required changes to the instruction set. Results indicated that nearly 27% of all committed instructions contained dead value information. This optimization produced good results in all the situations. With 64 physical registers, it reduced the performance difference between the virtual context architecture and the ideal register window architecture by almost 50% in a pipeline with two data cache ports and 30% in a pipeline with one data cache port.

The techniques provide additional improvements when combined. The combination of all three techniques produced the greatest reductions in data cache accesses and execution time.

# Chapter 9

# Related work

This chapter briefly describes the research that is related to this dissertation. First, we discuss other research that has used memory as a backing store to the register file. Next, we examine the various ways other researches have tried to reduce the save and restores imposed by register contexts. We then describe other work that has more aggressively managed the physical registers. We then look at other research that has enabled the use of large numbers of logical registers by creating efficient large physical register files. Finally, we examine work that has been done to enable more threads in a simultaneous multithreading processor without increasing the size of the physical register file.

## 9.1 Memory As A Backing Store To The Register File

The virtual context architecture maps the logical register to memory. The physical register file is treated as level in the cache hierarchy. It is backed up by the traditional data cache hierarchy, which is in turn backed by memory. Several other researches have used memory as a backing store to the register file.

Nuth et al. [23] proposed the Named State Register File, which like the VCA treats the physical register file as a cache and memory maps the logical registers. They explored the use of this new register file design for both block multithreading and as a more efficient implementation of register windows. The architecture was designed around a single issue in-order pipeline. Their design required a CAM of the entire register file on every read and write access. A miss in the register file stalled the pipeline until an entry for that register was allocated. This involved the location of a free entry. If entries were free, an existing entry is spilled to memory.

In the case of a read access, a miss also requires that the value be read from memory and placed into the register file (similar to our fill operation). They estimated that their design would increase the access time of the register file by 5%. This could potentially increase the cycle time of the processor. Unlike the rename table access, the register file access cannot be easily pipelined. Similar to pipelining the rename stage, adding an extra stage to the register file access would increase the branch misprediction penalty. Any additional stages would also add extra levels to the bypass logic which is a critical component of the backend of out-of-order processors. These extra stages would also increase the lifetime of physical registers. The register file access would be further complicated by the logic necessary to handle potential misses. In an out-of-order processor, this would either require a stall on the entire issue/execute loop or complex logic to handle issued instructions dependent on the stalled instruction. The virtual context architecture requires no changes to the backend of the pipeline. Our design requires no modifications to the physical register file. The VCA guarantees at rename that every logical register used by instructions in the backend of the pipeline will be mapped to a valid physical register. Therefore, the processor register file access does not need to modified. Unlike the Named State Register File, when the virtual context architecture stalls to handle register state transfers, only the frontend of the pipeline is stalled. The backend can continue to execute and commit instructions.

Idea of pointers associated with registers and background save/restore was proposed by Huguet and Lang [16]. This architectures uses a register mask to track live values for the current function context. The mask is updated as registers are used. The architecture also keeps a table with the save location of all the currently live registers. Save locations for all the registers are reserved in each function activation record. When a register is written to, the save location for that register in the current function is entered into the table. Only one location is specified for each register, although the location can be from any function activation record. The compiler was also modified to insert a mask of all registers that are live within the function at function entry and exit. On function entry, an intersection

of the current live register mask is done with the mask of all registers that are live in the function. This intersection specifies which registers need to be saved. The architecture stalls the pipeline and saves all the registers that need to be saved. The live register mask is updated to mark these registers as no longer live. On function exit, the live register mask is updated with the mask of all registers that are live in the function to clear the registers defined in this function. When a register is sourced, if the corresponding bit is not set in the live register mask, the pipeline is stalled while the register is read from its save location in the function activation record (current frame pointer plus register number). This architecture is able to take advantage of both static and dynamic liveness information to minimize the amount of saving and restoring done. While this technique does minimize the save and restore traffic, it only applies to a single register window. It is unable to take advantage of a larger physical register file. The original research was done using a simple single issue in-order pipeline. Like Nuth's architecture, it also has the potential of servicing a miss in register access with the same inherent complexities.

Ditzel et al. [10] proposed replacing the register file with a large circular buffer in the C machine stack cache. The instruction set for this architecture did not use registers, but instead all the values were stored in memory (similar to the mem-machine we proposed), with local values specified as offsets in the stack. The stack was mapped onto this buffer (they allowed byte level addressing into the cache, as long as you don't cross word boundaries). On a call or return, an enter or catch instruction was used to guarantee that as much of the stack as possible was mapped onto the circular buffer. On an underflow or overflow, the buffer was backed up by memory. The instruction cache held decoded instructions that contain full addresses instead of stack offsets. The lower order bits of these addresses could be used to access the cache entries. Any addresses that fall outside the circular buffer were accessed using memory. Unlike the virtual context architecture, this machine required a completely new instruction set similar to the one proposed in Chapter 2 (they do not specify how they encode the instructions). One of their arguments in favor of this new ISA was the complexity and speed of

multipass optimizing compilers and register allocation. Now, multipass optimizing compilers and register allocation are mature and well understood fields. The virtual context architecture is much more efficient in its resource utilization. In the VCA, the physical register file is managed on single register granularity. In the C machine stack cache, the buffer contains a large contiguous section of the stack, which is likely to contain a large fraction of unused values. Unlike the virtual context architecture, this architecture was designed for a single issue in-order processor and does not lend itself to out-of-order processing due to the complexity of dependence checking with memory addresses.

This section briefly described three other architectures that used memory as a backing store. The first architecture used memory as a backing store of the register file to implement register windows and block multithreading. This architecture required a CAM of a fully associative physical register file on every read and write leading to either increased cycle time or multiple stages for register access. In contrast, the virtual context architecture does not modify the physical register file and therefore its access time is unchanged. The second architecture takes advantage of both static and dynamic liveness information to minimize the amount of saving and restoring done. The virtual context architecture is able to efficiently implement register windows and remove most of the save and restore traffic. The third architecture implemented a new instruction set and replaced the physical register file with a cache of a contiguous section of the stack. The virtual context architecture works with existing instruction sets and more efficiently manages the cache using single register granularity. These architectures were all implemented in an in-order processor. In contrast, the virtual context architecture is designed around a higher performance superscalar out-of-order processor.

## 9.2 Reducing Save/Restore

The virtual context architecture can provide an almost ideal implementation of register windows. By providing each function with its own set of logical registers, register windows reduces the save and restore overhead associated with a func-

tion call. A lot of research has been done on the reduction of save and restore traffic.

Register windows provides each function with its own set of general purpose registers, pushing the responsibility of spilling and filling down to hardware[10, 24, 30]. In these architectures, a large register file capable of handling several windows is provided in the core. Windows are allocated on function calls and deallocated on returns without significant overhead, but some method of handling overflow or underflow is required. Two commercial architectures that use register windows are the SPARC and Itanium[37,8]. SPARC processors use a software approach to handling this problem by trapping to the operating system on an underflow or overflow condition. The Itanium uses a hardware approach known as the register stack engine (RSE). When an overflow or underflow occurs, the processor halts the execution of the program and the RSE proceeds to move registers to or from memory. In either case, the handling of an underflow or overflow halts the execution of the program and adds a significant amount of overhead. The virtual context machine is able to provide a nearly optimal implementation of register windows by spilling/filling on single register granularity instead of entire register windows. It also does not require the entire register window to be resident in the physical registers and thus uses the register file much more efficiently than most previous schemes. As the experiments in Chapter 6 showed, the virtual context architecture is able to implement register windows in an out-of-order processor without any additional physical registers. In contrast, a conventional register window architecture requires a much larger physical register file. In the VCA, spilling and filling are handled by the frontend, and the backend of the pipeline can continue execution in parallel with these operations. The out-of-order nature enables it to tolerate the extra latency of the infrequent spills/fills just like it enables conventional systems to tolerate long memory latencies.

Researchers have also explored techniques for using liveness information to minimize the amount of saving and restoring done. As mentioned in the previous section, Huguet and Lang [16] explored the use of liveness information and use masks to dynamically determine which registers needed to be saved on function

entry. Their technique also delayed the restore of a register value until its actual use. Martin et al. [20] also explored the use of liveness information. In their research, dead value information is inserted into instruction set architecture. The architecture could make use of this information to efficiently determine the set of live registers on function entry. The hardware used this set to squash any save instructions that were saving dead values. They also proposed maintaining a stack of live register sets. By saving a snapshot of the live register information on function entry, restore instructions that were restoring dead registers could be squashed at function exit. This technique can also be expanded to reduce the saves and restores necessary on a context switch.

The virtual context architecture has two advantages over Martin's solutions. First, they require that all the save and restore instructions still exist in the binary, even though they may eventually be squashed. The virtual context architecture does not require these instructions, thus reducing the static size of the binary and improving the fetch efficiency. Second, the VCA is able to take advantage of its ability to keep registers from multiple contexts in the physical register file to actually eliminate some saves and restores. Martin's solution minimizes the number of saves and restores, but if a register is live at function entry, it still must be saved and later restored. In the virtual context architecture, the register is only saved (spilled) if its physical register needs to be reclaimed.

Previous work has also focused on reducing the memory bandwidth of save and restore traffic by splitting out part of the memory stream and directing it to a separate pipeline and cache. Cho et al. [7] proposed the data decoupled architecture. In this architecture the memory stream is split into two streams - local variable accesses and the rest. The architecture provided a small separate cache and load store queue for local variable accesses. They also explored a fast data forwarding (using the offset field) and access combining. Lee et al. [17] proposed the Stack Value File. Like the data decoupled architecture, this architecture split the memory stream into two separate streams, in this case stack accesses and non stack accesses. The easily identified stack based accesses (those that are stack pointer relative) are moved into a simple pipeline that renames them into register

operations. The stack value file is a simple direct mapped cache that represents a contiguous piece of memory at the top of the stack. Both implementations reduce the bandwidth to the first level data cache, but only by directing it to a separate cache. Programs still require the same number of cache accesses and dynamic instructions. The virtual context architecture actually eliminates cache accesses and reduces the number of instructions required to execute the program. Both architectures could easily be combined with the virtual context architecture. However, because of the reduction in accesses in the redirected stream, their effectiveness would be diminished.

In this section, we briefly described previous research done to reduce the save and restore overhead. Register windows is a well known technique for reducing this overhead. Unlike other architectures, the virtual context architecture is able to provide a nearly ideal implementation of register windows in an out-of-order processor without increasing the number of physical registers. Liveness information has also been used to reduce the save and restore traffic. These techniques required the save and restore instructions to still be inserted into the binary, but squashed them. Although they are effective at minimizing the number of saves and restores, they cannot remove them. The virtual context architecture, by efficiently implementing register windows, allows the complete removal of save and restore instructions from the binary and the elimination of almost all of the overhead associated with saving and restoring. Finally, two techniques were described that reduce the bandwidth to the first level cache by directing save and restore traffic to a different pipeline and cache. As mentioned previously, the virtual context architecture is able to eliminate most of these accesses and therefore truly removes the bandwidth overhead.

## 9.3 Aggressive Physical Register Management

The traditional management of the physical register file is very conservative. A logical register is renamed to a physical register at the rename stage. The execution and writeback of the instruction is all accomplished using the physical register

tags. When an instruction is committed, the value of the destination operand is committed and becomes architectural state. If a physical register was currently holding the old architectural state of this register, it was freed. By waiting to free a physical register until it is overwritten, the pipeline guarantees it can recover from any branch misprediction. The virtual context architecture is able to manage the physical registers more aggressively. The VCA is able to move architectural state out of the physical register file and reclaim the physical registers before they are overwritten. This allows the VCA to keep only the most active registers in the physical register file. Several researches have investigated more aggressive management of the physical register file.

Martin et al. [20] and Lo et al. [18] proposed software identification of dead register values to free physical registers early. The dead value information is inserted into instruction set architecture either as modifications to the instructions or as special instructions. When an instruction containing dead value information commits, the physical register mapped to the dead logical register can be immediately freed. Our design targets similar goals, and also achieves them in part by moving dead values out of the register file into memory. Our scheme has two advantages: it does not require software annotations, and it also seamlessly manages the efficient saving and restoring of values that are not dead but have not been accessed recently (e.g., live values from calling procedures or from active but stalled threads). The proposed dead-value annotations would be a useful addition that should integrate easily into our design, allowing us to avoid spilling dead values to memory and to reclaim dead registers preferentially over live but inactive ones. Dead value information was integrated into our design in Chapter 8 and the results showed that it was effective at reducing the number of extra spills.

Akkary et al [1] also examined reclaiming physical registers early in their proposed Checkpoint Processing and Recovery architecture. Branch recovery was done using checkpoints. However, the checkpoints were not taken on every branch, instead they used a confidence estimator to pick branches likely to be mispredicted. They also checkpointed every 256 instructions. A use counter was used to keep track of the number of uncompleted instructions in each checkpoint

and, as in the virtual context architecture, the number of instructions in the pipeline currently using the register. To handle a misprediction, when a checkpoint is created, it incremented the use counter for all the currently mapped physical registers. This ensured that the checkpoint can always restore the correct logical register values. The combination of their new checkpointing scheme and register use counters allowed them to implement a more aggressive management of physical registers. They reclaim the register as soon as the use counter drops to zero and the register has been renamed. A traditional architecture must wait until the renaming instruction commits; this guarantees it is not speculative. Otherwise the physical register may be needed for branch recovery. This architecture was able to free the physical register as soon as the register was renamed. By having the checkpoint increment the use counter, they guaranteed that the physical registers required for branch recovery will always be available. The virtual context architecture could potentially be combined with this architecture. Both track the use of physical registers using counters, so the hardware cost is shared. However, the virtual context architecture does require larger checkpoints, so that may make the combination impractical.

In contrast to the previous research, Monreal et al. [21] proposed an architecture to delay the allocation of a physical register instead of reclaiming it early. Their architecture was called the virtual-physical register scheme. In this architecture, a physical register is not allocated until writeback. This was accomplished by using a larger set of virtual physical tags to track dependency. A second renaming occurred at the writeback stage. This renamed a virtual physical tag to a free physical register. One of the drawbacks to this scheme was the potential deadlock that can occur if there are no free physical registers to use when the oldest instruction is trying to commit. Their proposal for avoiding this deadlock was to make sure there was always a physical register for the oldest instruction with a destination in the reorder buffer. This allowed the oldest instruction to always be able to commit. In a conventional pipeline, the commit of an instruction with a destination will always free a physical register (the physical register holding the old architectural state). Their virtual physical register scheme could more efficiently

use the physical registers, either for better performance with the same sized file, or the same performance with a smaller file. This work is orthogonal to ours. Although they decreased the lifetime of physical registers, they do not increase the number of logical registers that can be handled, or allow physical registers to be reclaimed by moving their state out to memory. The virtual context architecture may be compatible with this scheme. There are two potential problems. The first problem is their deadlock solution. Because we can spill architectural state, it is possible that committing an instruction with a destination will not free a physical register. The second problem is potentially doing spills at writeback. It is much more expensive and harder to stall at writeback versus rename.

This section briefly described several techniques used to allow more aggressive management of the physical registers. Two architectures were described that use dead value information to reclaim physical registers more quickly. In Chapter 8, we showed how this could be integrated into our design. Unlike these techniques, the virtual context architecture also manages the efficient saving and restoring of values that are not dead but have not been accessed recently. The Checkpoint Processing and Recovery architecture [1] could also reclaim physical registers more quickly. This architecture made use of its checkpointing scheme to reclaim physical registers as soon as an overwriting instruction was renamed, even though it was speculative. The integration of dead value information into the virtual context architecture accomplishes the same thing. Finally, we discussed the virtual-physical registers scheme and its ability to delay the allocation of a physical register and thereby decrease its lifetime. Although they decreased the lifetime of physical registers, they do not increase the number of logical registers that can be handled, nor allow physical registers to be reclaimed by moving their state out to memory. The integration of dead value information into the VCA also allows it to reduce the lifetime of physical registers.

## 9.4 Efficient Large Physical Register Files

The virtual context architecture is able implement a large number of logical registers (register windows) without increasing the number of physical registers. A large number of logical registers is impractical if it requires a large physical register file. The large multiported physical register file can make it difficult to meet cycle time goals. There has been a lot of research into this problem. The two main research areas are register caches and register banking.

Register caches [2, 5, 9, 25, 39] have been proposed as a way to support a large number of logical registers with the access time of a small physical register file. These architectures are similar to the virtual context architecture in that they treat the physical register file as a cache. Specifically, they break the physical register file into a hierarchy of caches. The first level of cache is generally small but fully ported. The higher levels of cache have more entries but fewer ports and serve as a backing store to the first level of register cache. The architectures differ in how they organize the hierarchy, implement the first level, transfer values between levels, and rename the registers; their insertion policy into the first level; and their replacement policy for the first level. In these designs, the full architectural state is still kept in the register file; thus die area continues to limit the number of supported contexts. In contrast, the virtual context architecture is able to provide the appearance of an almost infinite number of logical registers. The VCA could be combined with some of these other architectures. In this case, the spills and fills would occur for the second level of register cache. Some of these techniques modified the rename stage, and this would further complicate a combined architecture.

Register banking [2, 9] has been proposed as a technique to reduce the access time of a given size physical register file. Register banking is similar to register caching in that it breaks the monolithic register file into smaller componenents. In this case, the components are multiple banks. A register bank is a slice of the register file. Each bank supports a smaller number of ports, with a resulting decrease in latency and size. The cost of register banking occurs in the case of a bank conflict. Ideally, the register read and writes will be spread out over all of the

banks and each bank will have enough ports. However, a bank conflict can occur when multiple reads or writes are made to a bank in a single cycle and the bank does not have enough ports to service them. The pipeline must stall one of the problem instructions and execute it the next cycle. The virtual context architecture is completely orthogonal to register banking. The changes due to register banking are usually confined to the execute stages of the pipeline. The virtual context architecture makes no modifications to these stages and could therefore be very easily combined with register banking.

## 9.5 Simultaneous Multithreading

Tullsen et al. [35] explored a realistic model of an simultaneous multithreading processor. They specifically stated that a large register file was required, and they assumed a two cycle access time. They listed several ramifications to the increased access time. It increases the misprediction penalty (extra stage). The longer access time requires an extra level of bypass logic because it takes multiple cycles to write. While the two cycle access time does not increase the inter-instruction latency for fixed latency instructions, it might for loads unless they are speculatively issued. This also adds a couple extra stages between rename and commit, increasing the lifetime of a physical register. They conclude that “register file access time will likely be a limiting factor in the number of threads an architecture can support“. The virtual context architecture is able to support a large number of simultaneous threads with no increase in the size of the physical register file over a single thread pipeline. Several other researchers have investigated techniques for supporting a larger number of threads without increasing the size of the physical register file.

Waldspurger et al. [36] proposed register relocation, a technique that allowed static or dynamic partitioning of a register file into variable size contexts for use by a multithreading processor. They introduced a special register, the register relocation mask (RRM). At decode, the register offsets were combined using a bitwise OR with the RRM to yield the actual register number. The scheduling of the con-

texts was managed in software. Each context maintained the RRM mask for the next context. This formed a linked list of contexts, and each thread could always jump to the next context. Simple modifications to this scheme could yield different scheduling policies. Dynamic allocation was possible by maintaining an allocation bitmap and using shift and mask operations to search it. Their research assumed that all the threads are from one application. The compiler was responsible for figuring out context sizes for each thread and for managing memory. This scheme was targeted at a much older in-order style course grained multithreaded processor. It would be possible to use the virtual context architecture to implement something similar if user code was allowed to modify the base register. This would allow the user code to switch between contexts whenever it wanted, without having to deal with actually storing or loading contexts. The VCA could also very easily implement a variable size context if it was desired. However, the virtual context architecture is able to allow a large number of threads in a modern out-of-order SMT processor without compiler support or increasing the size of the physical register file.

Redstone et al. [27] proposed minithreads to increase the number of contexts in an SMT processor without increasing the register file size. They provided full contexts for some number of threads. The minithread architecture also provided the additional hardware needed to run some number of minithreads within each context (PC, control registers, return stacks, etc.). The minithreads were like normal threads, except they share the same set of architectural registers. They investigate a static partitioning. The minithreads required a lot of operating system and compiler support. In particular, to support a mixed environment, the OS was modified to block the other minithreads when a minithread makes an OS call. They would probably also need to provide both standard libraries and libraries compiled with the minithread convention of only using half the logical registers. The virtual context architecture could easily be used to do something similar. In this case, the same hardware is duplicated to provide however many contexts you want. To get the same effect, you can simply compile the programs so that they only use half the normal number of registers. The virtual context architecture makes it easier

though, because the threads can actually use the full set of registers if they want to. In other words, the operating system and libraries could be allowed to use the full set of registers; it would just cause more spill and fill traffic. A simple change could be made to mark as dead all those registers not used by the user mode programs when the OS code returns to user code. This would prevent these registers from causing extra spills. In Chapter 7, the experiments showed that the virtual context architecture was able to provide a full set of logical registers, including using register windows, with four threads without increasing the size of the physical register beyond that used by the single thread pipeline.

## 9.6 Summary

This chapter briefly described the research that was related to the virtual context architecture. The previous work covered five different areas. First, we described the research related to the virtual context architecture's use of memory as a backing store. Second, we outlined the research related to the virtual context architecture's reduction of save and restore traffic. Third, we presented research on architectures that also aggressively managed the physical register file. Fourth, we provided a brief overview of other research that has enabled a large number of logical registers. Finally, we examined work that also enabled a larger number of threads in a simultaneous multithreading processor without increasing the size of the physical register file.

The virtual context architecture is able eliminate much of the register save and restore overhead by providing a nearly ideal implementation of register windows. The VCA allows physical registers to be reclaimed by moving their state out to memory, while the integration of dead value information also allows it to reduce the lifetime of physical registers. The virtual context architecture is able to provide the appearance of an almost infinite number of logical registers without increasing the size of the physical register file. The virtual context architecture is able to support a large number of simultaneous threads with no increase in the size of the physical register file over a single thread pipeline. Although the other architectures

described in this chapter may be able to provide some of these benefits, none of them provide all of these benefits. While many of the other architectures are implemented in a single issue in-order pipeline, the virtual context architecture provides all of these benefits in an aggressive superscalar out-of-order pipeline without modifying the critical backend of the pipeline.

# Chapter 10

# Conclusion

Multiple logical register contexts can increase CPU efficiency by storing several function contexts (i.e., register windows) or storing the state of several independent threads (multithreading). Register windows reduce the number of load and store instructions by removing many of the spill and fill operations resulting from function calls. Simultaneous multithreading (SMT) improves throughput by using processor resources more efficiently. These techniques have been difficult to integrate, because each requires a large number of logical registers, and their combination would have a multiplicative effect. We sought to bypass the trade-off between multiple context support and register file size by mapping registers to memory, thereby decoupling the logical register requirements of active contexts from the contents of the physical register file. This dissertation proposed a new register-file architecture that virtualizes logical register contexts. We call this architecture the virtual context architecture.

## 10.1 Summary

First, we looked at mapping compiler virtual registers to memory. By mapping the compiler virtual registers directly to memory, we completely remove registers from the instruction set architecture. This created a memory to memory instruction set architecture which we called the mem-machine. We detailed the design of the mem-machine instruction set architecture and application binary interface. In general we found that the disadvantages of the mem-machine outweighed its advantages. The performance improvements were relatively mixed on the small set of benchmarks we examined. The instructions size is also much larger. Although the

compression results showed these offsets can be compressed into approximately the same size as a register specifier in a RISC ISA, this would require the addition of complex decompression hardware. Finally, the mem-machine would be difficult to realize in hardware, especially a superscalar out-of-order pipeline.

Next, we looked at mapping ISA logical registers to memory. By delaying the memory mapping to the frontend of the pipeline, this architecture gets all of the advantages of registers and the automatic context management of virtual memory. We called this scheme the virtual context architecture. Although the architecture requires extensive changes to the rename stage, it requires no changes to the physical register file design and the performance-critical schedule/execute/writeback loop. The changes necessary to the rename stage are the addition of tags and multiple ways. We presented an optimization that greatly reduces the size of the tags in comparison to a full memory address.

The virtual context architecture has a number of parameters that can be used to optimize performance. These parameters include the implementation of spills and fills and the rename stage configuration. Our results showed that an overwrite last physical register replacement policy combined with a four entry architectural state transfer queue with two ports provided the best implementation of spills and fills. We showed that the configuration of the rename table is dependent on the number of threads supported in the pipeline. For a single thread, a 64x3 rename table provided good performance, while two threads required a 64x5 and four threads required a 64x6. Several solutions were examined to compensate for the extra logic required in the rename stage. We found that the best solution was adding an extra stage. If the rename logic is still too complicated to meet cycle time constraints, we determined that a delay on rename table entry replacement would not impact performance very much.

Our experiments showed that the virtual context architecture is a very efficient implementation of register windows for a variety of different pipelines. In every pipeline studied, the virtual context architecture provided nearly ideal performance when the physical register file was sized to guarantee no stalls for the baseline architecture. In contrast to a conventional register window design, the VCA pro-

vides a performance advantage over the baseline architecture even with small physical register file sizes. The virtual context architecture was between 3% and 15% faster than the baseline architecture with the same number of threads. The VCA also reduced the data cache accesses by up to 20%. The studies showed that the performance of the virtual context architecture is very dependent on the relative cost of a load/store instruction and the cost of a data cache access. When these costs are high, the architectures that support register windows have a higher potential performance improvement due to the removal of the save and restore instructions from programs. With a large physical register file, the virtual context architecture is able to achieve very near this ideal performance. However, when the number of physical registers is decreased, the VCA is forced to generate more spills and fills. If the cost of a data cache access is high, these spills and fills will degrade performance quickly.

The virtual context architecture enables the use of register windows in an SMT processor with no additional physical registers, providing better performance than the baseline architecture. Both two and four thread SMT pipelines provide significant speedups of 30% and 50% respectively in a four issue pipeline over a single thread. The performance of the virtual context architecture relative to the baseline architecture is virtually independent of the number of threads. Thus, while multiple threads in a conventional architecture require much larger physical register files, the virtual context architecture is able to achieve the same performance with no more physical registers than a single thread pipeline. This greatly increases the potential speedup achievable with moderate size register files. For example, with 192 physical registers, the best achievable performance for the baseline architecture is with two threads for a 16% increase over the single thread. The virtual context architecture with this size register file provides a 32% improvement with two threads, or by using four threads it can achieve a 50% speedup.

The virtual context architecture is able to achieve nearly ideal register window performance with a large physical register file. With a smaller number of physical registers, the VCA must generate more spills and fills. The virtual context architecture's performance suffers with respect to ideal, which never generates spills and

fills. We proposed three techniques for reducing the number of spills generated by the VCA: the addition of a delay queue, a register window deallocation optimization and dead value information. Our experiments showed that with these techniques it was possible to reduce the performance difference between the virtual context architecture and the ideal register window architecture by almost 50% for pipelines with small register files.

The virtual context architecture provides a hardware scheme for efficiently managing register context. The VCA enables support for both register windows and simultaneous multithreading without increasing the size of the register file while still providing nearly ideal performance. This unifying framework provides support for both techniques in an out-of-order pipeline.

## 10.2 Future Work

The work begun in this dissertation can be continued in two ways. The first way is improving the virtual context architecture implementation. The second way is exploring new uses for the virtual context architecture.

### 10.2.1 Improving The VCA Implementation

The virtual context architecture requires modifications to the rename table. The table is modified to be set associative and requires tags. As described in Section 3.2.1.1, the tags can be optimized to reduce their size. The table itself is still large, and the set associative nature complicates the rename logic. The access pattern of the table is not random. The table will generally be accessed using memory addresses from a small number of contexts representing the active register windows of each thread. The maximum number of simultaneous entries is also fixed. The maximum number of different registers that can ever be renamed simultaneously is equal to the number of physical registers. This indicates that some type of caching scheme could be used to minimize the cost of the average type of access. Although a penalty may result when transitioning from one window to the next, the potential savings may be worth it if the penalty is small enough.

Although the virtual context architecture does not require any changes to the physical register file itself, it does tend to put more pressure on it. This would merit investigating some previously studied techniques on optimizing the physical register file itself. In particular, banking and caching can be easily incorporated into the VCA design. Previous work has also focused on more aggressively managing physical registers. This work should be compatible with the VCA and allow a more aggressive reclamation of physical registers (freeing them).

### 10.2.2 Exploring New Uses

The virtual context architecture has three main advantages over a traditional out of order architecture. First, it completely decouples the physical registers from the logical registers. This allows the compiler and operating system to work with an unlimited set of logical registers and allow the architecture to handle them. Second, it allows for fast context switching. The operating system is only responsible for changing the base pointer. The architecture will move architectural state into and out of the physical registers as needed. Third, the architecture can directly map addresses into physical registers.

#### 10.2.2.1 Decoupling The Physical Registers

Decoupling the physical registers from the logical registers can have several potential applications. The normal limitations on the number of active thread contexts is relaxed. This allows the architecture to naturally support multiple contexts simultaneously. Within one thread, each function activation record can be given its own context (register windows). Also, multiple threads can have active contexts in the processor at once (SMT). The initial results in these areas are promising. However, a more aggressive approach is possible in both areas.

Register windows are used to reduce some of the memory traffic imposed by register management. By modifying the compiler to mark loads/stores used for this management, the remaining overhead could be measured. If significant, a new ABI could be developed to try to alleviate these costs. Each function could be allowed to allocate a larger pool of registers and his pool could be used to hold all the local variables, removing any necessary spill or fill traffic. Instead, the memory

traffic would be solely dependent on the size of the physical register file. A simple move instruction could be introduced to move a value from the pool into the standard logical registers or from a logical register back into the pool. A similar move could also be used to allow a function to access the registers in the register window of the calling function. This would remove the need for non windowed registers.

By decoupling the number of logical registers and physical registers, the cost of supporting additional thread contexts is greatly reduced. This has the potential of allowing large numbers of simultaneous threads. A limit study could be used to determine the performance improvement achieved by adding additional threads. This could be done in both the ideal physical register file case, then repeated with more practical register file sizes. A scheduling algorithm could also be introduced. The processor could support a very large number of active threads, but an internal hardware scheduling algorithm could be introduced to schedule them. The algorithm would have to take into account both resource utilization and the cost of register spilling and filling. The study may reveal how close to 100% resource utilization it is possible to get.

#### 10.2.2.2 Fast Context Switching

Fast context switching also has several potential benefits. The performance/ simplicity of an operating system can be enhanced by allowing the hardware to manage some of the context switch overhead. Very efficient interrupts are possible by using a separate context to handle the interrupts. In fact, completely different contexts could be used for kernel code, simplifying it and enhancing security. This has the potential for allowing efficient microkernels and virtual machines.

#### 10.2.2.3 Directly Mapping Addresses To Physical Registers

Directly mapping addresses to physical registers also has several potential applications. In particular, it has the potential of allowing very efficient emulation of other instruction sets. This would allow for an efficient implementation of a stack based architecture (Java) without placing any limits on the size of the stack. Systems with vastly different logical register requirements (x86 and Itanium) could

coexist on the same system. The backend of the pipeline would implement a micro-op architecture with enough flexibility to support both systems. Separate decodes could supply the backend of the pipeline with micro-ops that use addresses for sources and destinations. The virtual context architecture's rename stage and backend could then efficiently implement this.

The virtual context architecture has shown great promise in these initial studies. There is potential in the future for simplifying its implemenation and for exploring new uses.

# Appendix A

# Mem-Machine Supported Operations

jal - jump and link

jail - jump and link indirect

b(cond)* - conditional branch

j - jump

ji - jump indirect

omov - offset move

mov - move

cvt - convert type

add - add

sub - subtract

mul - multiply

div - divide

mod - modulus

sll - shift left logical

srl - shift right logical

sla - shift left arithmetic

sra - shift right arithmetic

neg - negation

and - logical AND

or - logical OR

xor - logical XOR

comp - logical complement

s(cond) - set conditional where cond: eq(=), ne(!=), lt(<), le(<=), gt(>), ge(>=)

# Appendix B

# Mem-Machine Assembly Language

C code:

```
#include <stdio.h>

int main()
{
int i;
  for (i=0;i<10;++i) {
    printf("test = %d\n",i);
  }
  return 0;
}
```

Assembly code:

```
.data
        .align 2
$__mirv_pack.m1.200:
        .ascii "test = %d\n\0"

.text
.ent main
.globl main
main:
$L46:
          // Prologue sizes local=12 call=16 frame
size=32
        !mov.uw -4(%sp), %fp
        !mov.uw %fp, %sp
        !sub.uw %sp, %sp, #32
$L47:
        !mov.uw -12(%fp), #0
$L51:
        !bge.w -12(%fp), #10, #$L50
$L49:
        !mov.uw 8(%sp), #$__mirv_pack.m1.200
        !mov.uw 12(%sp), -12(%fp)
        !jal 0(%sp), #printf
        !add.w -12(%fp), -12(%fp), #1
        !j #$L51
$L50:
        !mov.uw 4(%fp), #0
        !j #$L48
$L48:
        !mov.uw %sp, %fp
        !mov.uw %fp, -4(%sp)
        !ji 0(%sp)
.end main
```

‘#’ - 0 level
‘ ‘ - 1 level
‘*’ - 2 level

Indirection Level

%0 - zero register
%sp - stack pointer
%fp - frame pointer

Register Specifier

s - single precision float
d - double precision float
b - byte
h - half word
w - word
l - long word
ub - unsigned byte
uh - unsigned half word
uw - unsigned word
ul - unsigned long word

Type Specifier

# References

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *36th Ann. Int'l Symp. on Microarchitecture*, pages 423–434, December 2003.

[2] Rajeev Balasubramonian, Sandhya Dwarkadas, and David Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *34th Ann. Int'l Symp. on Microarchitecture*, pages 237–248, December 2001.

[3] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[4] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[5] J. Adam Butts and Gurindar S. Sohi. Use-based register caching with decoupled indexing. In *Proc. 31st Ann. Int'l Symp. on Computer Architecture*, page 302. IEEE Computer Society, 2004.

[6] Steve Chamberlain, Roland Pesch, Jeff Johnston, and Red Hat Support. The red hat newlib c library, July 2002. Red Hat, Inc.

[7] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *Proc. 26th Ann. Int'l Symp. on Computer Architecture*, pages 100–110, May 1999.

[8] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual*. Santa Clara, CA, 2000.

[9] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proc. 27th Ann. Int'l Symp. on Computer Architecture*, pages 316–325, June 2000.

[10] David R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, March 1982.

[11] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5, February 2003.

[12] Free Software Foundation. GNU Compiler Collection. http://gcc.gnu.org.

[13] David Anthony Greene. *Design, Implementation, and use of an Experimental Compiler for Computer Architecture Research*. PhD thesis, April 2003.

[14] Linley Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–16, Oct. 28, 1996.

[15] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[16] Miquel Huguet and Tomás Lang. Architectural support for reduced register saving/restoring in single-window register files. *ACM Trans. Computer Systems*, 9(1):66–97, February 1991.

[17] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Gary S. Tyson, and Chris J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. 7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 5–14, January 2001.

[18] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Trans. Parallel and Distributed Systems*, 10(9):922–933, September 1999.

[19] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[20] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting dead value information. In *30th Ann. Int'l Symp. on Microarchitecture*, pages 125–135, December 1997.

[21] Teresa Monreal, Antonio Gonzlez, Mateo Valero, Jos Gonzlez, and Victor Vinals. Delaying physical register allocation through virtual-physical registers. In *32nd Ann. Int'l Symp. on Microarchitecture*, pages 186–192, November 1999.

[22] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of MICRO-26*, 1993.

[23] Peter R. Nuth and William J. Dally. The named-state register file: Implementation and performance. In *Proc. 1st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 4–13, January 1995.

[24] David A. Patterson and Carlo H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proc. 8th Intl. Symp. Computer Architecture*, volume 32, pages 443–457, Nov. 1981.

[25] Matt Postiff, David Greene, Steven Raasch, and Trevor N. Mudge. Integrating superscalar processor components to implement register caching. In *Proc. 2001 Int'l Conf. on Supercomputing*, pages 348–357, 2001.

[26] Steven E. Raasch and Steven K. Reinhardt. The impact of resource partitioning on smt processors. In *Proc. 12th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, September 2003.

[27] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. Mini-threads: Increasing tlp on small-scale smt processors. In *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 19–30, February 2003.

[28] Yiannakis Sazeides and Toni Juan. How to compare the performance of two SMT microarchitectures. In *Proc. 2001 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, November 2001.

[29] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, October 2002.

[30] Richard L. Sites. How to use 1000 registers. In *Caltech Conference on VLSI*, pages 527–532. Caltech Computer Science Dept., 1979.

[31] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 3 edition, 1998.

[32] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. In *Proc. IEEE*, volume 83, pages 1609–1624, December 1995.

[33] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 234–244, November 2000.

[34] John A. Swenson and Yale N. Patt. Hierarchical registers for scientific computers. In *Proc. 1988 Int'l Conf. on Supercomputing*, pages 346–353, July 1988.

[35] Dean Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd Ann. Int'l Symp. on Computer Architecture*, pages 191–202, May 1996.

[36] Carl A. Waldspurger and William E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proc. 20th Ann. Int'l Symp. on Computer Architecture*, pages 120–130, May 1993.

[37] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[38] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[39] Robert Yung and Neil C. Wilhelm. Caching processor general registers. In *Proc. 1995 Int'l Conf. on Computer Design*, pages 307–312, October 1995.