Implementing and Exploiting Static Speculation
On Multiple Instruction Issue Processors

Mayan Moudgill
Ph.D Thesis

TR 94-1440
August 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
IMPLEMENTING AND EXPLOITING STATIC SPECULATION
ON MULTIPLE INSTRUCTION ISSUE PROCESSORS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Mayan Moudgill
May 1994
IMPLEMENTING AND EXPLOITING STATIC SPECULATION ON MULTIPLE INSTRUCTION ISSUE PROCESSORS

Mayan Moudgill, Ph.D.
Cornell University 1994

Trends in processor architecture and design suggest that static speculation will become a candidate for implementation on future high-performance processors. In this dissertation, we shall examine issues related to the implementation and exploitation of static speculation. There are four primary results:

Precise Exceptions Prior work in static speculation has not examined the interaction between exception handling and speculative instructions in any great detail. We investigate this interaction, exhibiting certain problematic subtleties that arise, and show how they can be overcome.

Speculative Tagging Earlier proposals for implementing speculative instructions tended to have several drawbacks, including restricted applicability. We introduce speculative tagging, a new, more general, mechanism for specifying static speculation, and show it is possible to optimize exception recovery through this mechanism.

Whole-DAG Scheduling Recently, there has been some work on scheduling regions of acyclic code larger than a basic block so as to take advantage of static speculation. We describe another such algorithm, known as whole-DAG scheduling, that contains innovations that make it more flexible, and allow it to use better heuristics.

Dynamic Speculation The work on static speculation and exceptions suggested an alternative approach to implementing dynamic speculation. This approach results in simpler hardware than prior schemes, and is consequently cheaper to implement and potentially has a lesser impact on cycle-time.

Additionally, we report results of experimental studies measuring the effectiveness of whole-DAG scheduling. In it, we show, among other things, that our scheduling technique can result in near-optimal schedules, and that the selection heuristics we adopt are superior to those used in earlier algorithms.
Biographical Sketch

Mayan Moudgill was born on 11th June, 1967 in the city of Bhopal in India. After a migratory childhood, during the course of which he attended 7 schools, he finally graduated in 1984 from La Martiniere College, Lucknow. He received the Bachelor of Technology in Computer Science and Engineering in 1988 from the Indian Institute of Technology, Kanpur. He joined the graduate program of the Department of Computer Science at Cornell University in August, 1988.
This dissertation is dedicated to my parents and my wife, for all the support they have given me over the years.
Acknowledgements

I wish to thank Keshav Pingali and Stamatis Vassiliadis for the help and guidance they have given me over the years. It is difficult to quantify how much their advice and support have meant to me, and how much they have contributed to this dissertation; suffice it to say that, without them, I could never have achieved what I have.

I would like to thank my thesis committee, i.e. Bard Bloom, Geoff Brown and Keshav Pingali for their guidance and support.

I have been fortunate to work several summers at IBM. The people I worked with there—Bob Iannucci, Paul Suhler, Kei Hiraki, Katamudhari Eknadham, Ragunathan Rajkumar, Mike Hale, Jim Chambers, Bill Virun—made my stays there enjoyable and educational. I would especially like to thank Steven Gregor for teaching me about system architecture. He and Tom Jeremiah evaluated the design in Chapter 7 and were instrumental in producing a gate-count for it.

I would like to thank my fellow students, especially Micah Beck, Richard Johnson, Wei Li, Paul Stodghill, Vladimir Kotlyar, Sudeep Gupta, Indu Prakas Kodukola and Richard Huff, for interesting conversations.

A special thanks to my various house-mates, V. V. Bhaskar, Sandip Bose, Navin Budhiraja, Suresh Chari, Radha Jagadeesan, Jayesh, and Tushar Deepak Chandra for the many stimulating discussions that have influenced me and hence this thesis.

Finally, I’d like to thank my wife, Priti Srivastava, for marrying me, for her support, and for keeping me sane.
Table of Contents

1 Introduction
   1.1 Problems, Previous Approaches, and Our Approach .................................................. 4
   1.1.1 Mechanisms for Static Speculation ................................................................. 4
   1.1.2 Static Speculation and Exceptions ...................................................................... 5
   1.1.3 Static Scheduling for Multiple Instruction Issue ............................................... 5
   1.1.4 Performance of Static Speculation ...................................................................... 6
   1.2 Significance and Contributions ................................................................................ 7
   1.3 Organization ............................................................................................................. 8

2 Background and Related Work
   2.1 Instruction Level Parallelism .................................................................................... 9
      2.1.1 Pipelining ......................................................................................................... 10
      2.1.2 Multiple Instruction Issue ................................................................................. 12
   2.2 Constraints on ILP .................................................................................................. 12
      2.2.1 Resource .......................................................................................................... 12
      2.2.2 Data Dependence ............................................................................................ 14
      2.2.3 Control Dependence ....................................................................................... 15
      2.2.4 Increasing Exploitable ILP Dynamically ......................................................... 17
   2.3 Scheduling .............................................................................................................. 19
      2.3.1 Basic Block ....................................................................................................... 19
      2.3.2 Cyclic Code ...................................................................................................... 20
      2.3.3 Acyclic Code ................................................................................................... 21
   2.4 Speculation .............................................................................................................. 22
      2.4.1 Vocabulary ....................................................................................................... 22
      2.4.2 Mechanisms ..................................................................................................... 24
   2.5 Summary .................................................................................................................. 29

3 Exception Handling and Static Speculation ................................................................. 30
   3.1 Introduction ............................................................................................................. 30
      3.1.1 Precise Exceptions ........................................................................................... 30
      3.1.2 Exception Handlers and Precise Exceptions ..................................................... 31
      3.1.3 Abstractions ..................................................................................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.4 Origin vs. Commit: Producing Speculative Code</td>
<td>33</td>
</tr>
<tr>
<td>3.1.5 Assumptions</td>
<td>34</td>
</tr>
<tr>
<td>3.2 Precise Exceptions &amp; Static Speculation</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 The Breakdown of Precision</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Equivalence</td>
<td>37</td>
</tr>
<tr>
<td>3.2.3 Sequentialization</td>
<td>37</td>
</tr>
<tr>
<td>3.2.4 Sequentialization Without Shadow Registers</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Implementing Precise Exceptions</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1 Constraining Speculation</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2 A Proof of Equivalence</td>
<td>41</td>
</tr>
<tr>
<td>3.3.3 Restart</td>
<td>41</td>
</tr>
<tr>
<td>3.3.4 Writing Exception Handlers for Static Speculation</td>
<td>44</td>
</tr>
<tr>
<td>3.3.5 Discarding Precision</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Conclusions</td>
<td>45</td>
</tr>
<tr>
<td>4 Speculative Tagging</td>
<td>47</td>
</tr>
<tr>
<td>4.1 Speculation Tags</td>
<td>47</td>
</tr>
<tr>
<td>4.1.1 Success and Commits</td>
<td>47</td>
</tr>
<tr>
<td>4.1.2 Failure and Flushing</td>
<td>48</td>
</tr>
<tr>
<td>4.1.3 Register Files</td>
<td>49</td>
</tr>
<tr>
<td>4.1.4 Memory Operations</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Multi-Path Speculation</td>
<td>51</td>
</tr>
<tr>
<td>4.3 Implementing Exceptions</td>
<td>53</td>
</tr>
<tr>
<td>4.3.1 Register Pressure</td>
<td>54</td>
</tr>
<tr>
<td>4.3.2 Restart Blocks</td>
<td>55</td>
</tr>
<tr>
<td>4.3.3 Lazy Restart</td>
<td>56</td>
</tr>
<tr>
<td>4.4 Miscellanea</td>
<td>56</td>
</tr>
<tr>
<td>4.5 Design</td>
<td>57</td>
</tr>
<tr>
<td>4.5.1 Summary</td>
<td>57</td>
</tr>
<tr>
<td>4.5.2 Cost Comparison</td>
<td>58</td>
</tr>
<tr>
<td>4.6 Conclusions</td>
<td>60</td>
</tr>
<tr>
<td>5 Scheduling with Speculative Instructions</td>
<td>61</td>
</tr>
<tr>
<td>5.1 Background</td>
<td>61</td>
</tr>
<tr>
<td>5.1.1 Information Representation</td>
<td>61</td>
</tr>
<tr>
<td>5.1.2 List Scheduling</td>
<td>64</td>
</tr>
<tr>
<td>5.1.3 Selection Heuristics</td>
<td>65</td>
</tr>
<tr>
<td>5.1.4 Global Code Motion</td>
<td>66</td>
</tr>
<tr>
<td>5.1.5 Probabilities in Scheduling</td>
<td>69</td>
</tr>
<tr>
<td>5.2 Whole-DAG Scheduling</td>
<td>69</td>
</tr>
<tr>
<td>5.2.1 Frontiers</td>
<td>69</td>
</tr>
<tr>
<td>5.2.2 Readiness and Ready Lists</td>
<td>70</td>
</tr>
<tr>
<td>5.2.3 Miscellanea</td>
<td>74</td>
</tr>
</tbody>
</table>
5.2.4 Summary ........................................... 74
5.3 Selecting Instructions ................................ 76
  5.3.1 Computing Late .................................... 76
  5.3.2 Selection Heuristics ............................... 78
5.4 Comparison with Prior Work ......................... 80
5.5 Summary ............................................ 81

6 Results .............................................. 82
  6.1 Preliminaries ........................................ 82
    6.1.1 The Pidgin Compiler ........................... 82
    6.1.2 Measuring DAGs ................................ 84
    6.1.3 Benchmarks ..................................... 85
  6.2 The Benefits of Speculation ......................... 85
    6.2.1 The Importance of DAGs ....................... 85
    6.2.2 Non-numerical Programs ....................... 86
    6.2.3 Dynamic Speculation ............................ 88
  6.3 The Scheduling Algorithm ......................... 89
    6.3.1 The Variables .................................. 89
    6.3.2 The Scenarios .................................. 90
    6.3.3 Evaluating Strategies ......................... 93
    6.3.4 Evaluating Heuristics ......................... 96
    6.3.5 Other Concerns ................................ 96
  6.4 Tag Usage ......................................... 104
    6.4.1 The Influence of Strategies .................. 104
    6.4.2 Architecture .................................. 107
  6.5 Conclusion ....................................... 107

7 Dynamic Speculation ................................ 109
  7.1 Prior work ......................................... 109
  7.2 Our approach ...................................... 112
  7.3 Reclaiming physical registers ..................... 114
  7.4 Branches and speculation .......................... 117
  7.5 Precise interrupts ................................ 120
  7.6 Summarizing the design ............................. 122
  7.7 Optimization ...................................... 123
  7.8 Implementation ................................... 124
  7.9 Comparison ....................................... 127
  7.10 Conclusion ...................................... 128

8 Conclusions and Future Work ......................... 129
  8.1 Conclusions and Contributions ..................... 129
  8.2 Future Work ..................................... 130
    8.2.1 Non-Speculative Architectures ............... 130

viii
# List of Tables

6.1 Input Programs & DAGs .......................................................... 86
6.2 Dynamic Execution Frequency .............................................. 87
6.3 Overall Improvement ............................................................. 87
6.4 Dynamic Frequency of DAGs in C Code ................................. 88
6.5 Scheduling Algorithms ......................................................... 90
6.6 Result Latencies ................................................................. 91
6.7 Actual Performance: Balanced Probabilities ......................... 94
6.8 Actual Performance: Profiled Probabilities .............................. 94
6.9 Selection Heuristic Performance .......................................... 97
6.10 Code Expansion ................................................................. 98
List of Figures

1.1 Original Code Sequence ............................................ 1
1.2 Optimal Code Sequence ............................................ 2
1.3 Sample Loop ..................................................... 2
1.4 Loop Unrolling .................................................. 3
1.5 Static Speculation ................................................. 3
1.6 Exceptions ....................................................... 5

2.1 Instruction Level Parallelism ....................................... 9
2.2 Pipelined Execution ............................................... 11
2.3 Multiple Instruction Issue ........................................ 13
2.4 Data Dependencies ............................................... 14
2.5 Removing False Dependencies .................................... 15
2.6 Control Dependencies ............................................ 15
2.7 Preserving Data Dependencies ................................... 16
2.8 Control-Independent Code Motion ............................... 17
2.9 Register Renaming ............................................... 17
2.10 3–1 ALU .......................................................... 18
2.11 Speculation ........................................................ 18
2.12 Software Pipelining ............................................... 20
2.13 Speculation ........................................................ 22
2.14 Speculation ........................................................ 24
2.15 Boosting ........................................................... 25
2.16 Boosting with “don’t cares” ...................................... 26
2.17 Speculation Inexpressible with Boosting ....................... 26
2.18 Poison-Bit .......................................................... 27
2.19 Register Manipulation in Boosting ............................... 28
2.20 Original Code for Fig. 2.19 ....................................... 28
2.21 Speculation with Renaming ....................................... 29

3.1 Pipelined Exceptions ............................................... 31
3.2 Representing Speculation .......................................... 32
3.3 Origin vs. Commit ................................................ 34
3.4 Speculation Changes Inputs ...................................... 35
3.6 Producing a Non-Sequentializable Program .................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5 Problems with Exceptions</td>
<td>36</td>
</tr>
<tr>
<td>3.7 Sequentialization</td>
<td>38</td>
</tr>
<tr>
<td>3.8 Restarting</td>
<td>42</td>
</tr>
<tr>
<td>3.9 Re-executing Speculative Instructions</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Speculative Tagging</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Grouping using Speculative Tagging</td>
<td>49</td>
</tr>
<tr>
<td>4.3 Speculation with Flushes</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Renaming Memory</td>
<td>50</td>
</tr>
<tr>
<td>4.5 Tagging with Renaming</td>
<td>52</td>
</tr>
<tr>
<td>4.6 Tagging with Copying</td>
<td>52</td>
</tr>
<tr>
<td>4.7 Multi-Path Memory Speculation</td>
<td>53</td>
</tr>
<tr>
<td>4.8 Multiple Memory Speculation</td>
<td>54</td>
</tr>
<tr>
<td>4.9 Illegal Speculation</td>
<td>54</td>
</tr>
<tr>
<td>4.10 Restarting after an Exception</td>
<td>55</td>
</tr>
<tr>
<td>5.1 Control-Flow Graph</td>
<td>62</td>
</tr>
<tr>
<td>5.2 Def-Use Chains</td>
<td>62</td>
</tr>
<tr>
<td>5.3 Dependence Flow Graph</td>
<td>64</td>
</tr>
<tr>
<td>5.4 List Scheduling</td>
<td>65</td>
</tr>
<tr>
<td>5.5 Safe Code Motions</td>
<td>67</td>
</tr>
<tr>
<td>5.6 Unsafe Code Motions</td>
<td>68</td>
</tr>
<tr>
<td>5.7 Predication</td>
<td>68</td>
</tr>
<tr>
<td>5.8 Frontier</td>
<td>70</td>
</tr>
<tr>
<td>5.9 Whole-DAG Scheduling: Top-Level Structure</td>
<td>71</td>
</tr>
<tr>
<td>5.10 Computing Ready</td>
<td>71</td>
</tr>
<tr>
<td>5.11 Controlling Code Motion</td>
<td>72</td>
</tr>
<tr>
<td>5.12 Implementing Code Motion</td>
<td>75</td>
</tr>
<tr>
<td>5.13 Eliminating Redundancies</td>
<td>76</td>
</tr>
<tr>
<td>5.14 Whole-DAG Scheduling: Detailed Breakdown</td>
<td>77</td>
</tr>
<tr>
<td>5.15 Computing late</td>
<td>78</td>
</tr>
<tr>
<td>5.16 Selection Heuristic—Case I</td>
<td>79</td>
</tr>
<tr>
<td>5.17 Selection Heuristic—Case II</td>
<td>79</td>
</tr>
<tr>
<td>5.18 Selection Heuristic—Case III</td>
<td>80</td>
</tr>
<tr>
<td>6.1 The Pidgin Compiler</td>
<td>83</td>
</tr>
<tr>
<td>6.2 Instrumentation</td>
<td>83</td>
</tr>
<tr>
<td>6.3 Scheduling and Simulation</td>
<td>84</td>
</tr>
<tr>
<td>6.4 Computing Average Execution Time</td>
<td>85</td>
</tr>
<tr>
<td>6.5 Distribution of Branch Probabilities</td>
<td>92</td>
</tr>
<tr>
<td>6.6 DAG vs. Register Pressure: 4 issue, best 5</td>
<td>98</td>
</tr>
<tr>
<td>6.7 DAG vs. Register Pressure: 8 issue, best 8</td>
<td>99</td>
</tr>
<tr>
<td>6.8 Weighted DAG vs. Register Pressure: 4 issue, best 5</td>
<td>100</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Modern microprocessors are increasingly being built with the ability to issue multiple instructions per cycle. Current generation microprocessors, such as the Motorola MC88110 [11] and the IBM PowerPC [5], can issue 2 to 4 instructions every cycle. In the future, 8 or 16 may not be uncommon. This raises the question: how can this capability be fully utilized?

One way of finding and issuing the necessary number of instructions every cycle is for the processor to include dynamic scheduling hardware. Every cycle, this hardware scans a large portion of the instruction stream, from which it extracts the issuable instructions. For instance, given the code fragment shown below in Fig. 1.1, the dynamic scheduler for a 2-issue processor would examine all 3 instructions. It would recognize that (1) and (3) could be issued in the current cycle, but that (2) used the result produced by (1) and so could not be issued till (1) had completed. It would therefore issue (1) and (3) immediately, and then issue (2) in the next cycle.

\[
\begin{align*}
(1) & \quad r1 := r2 + r3 \\
(2) & \quad r7 := r1 + 8 \\
(3) & \quad r5 := r9 - r2
\end{align*}
\]

Figure 1.1: Original Code Sequence

To keep the complexity within realistic bounds, the dynamic scheduling hardware examines a “window” of only 16 to 32 instructions every cycle. This size appears to be practical for a 2-issue processor. However, even with this small a window, the dynamic scheduling hardware required is complex and expensive, and could potentially increase the cycle time. To support issue rates of 8 to 16, the window will have to be substantially larger. The complexity of the dynamic scheduling hardware goes up at least as the square of the window-size; thus, the dynamic scheduling hardware for higher issue rates could prove to be prohibitive to implement, and, even if possible, will potentially be the bottleneck in reducing cycle times.

An alternative is to use less ambitious, and therefore simpler hardware. For instance, the processor could be designed to issue instructions in the order they appear in the original program each cycle, till it has issued as many as it can process in that cycle, or till it encounters an instruction
that can not be issued. In this case, if the processor is to be fully utilized, the compiler must order instructions in the code it produces so that adjacent instructions can be issued simultaneously. This approach is known as static scheduling.

As an example, consider the code in Fig. 1.1. Using the in-order issue technique described above, a 2-issue processor would issue (1) in the first cycle. (2) uses the result of (1), so it could not be issued until the second cycle. Finally, (3) would be issued in the third cycle. If, however, the compiler had ordered the instructions as shown in Fig. 1.2 with (1) and (3) adjacent, the processor would have been able to issue (1) and (3) in the first cycle, followed by (2) in the second cycle—a saving of a cycle.

\[
\begin{align*}
(1) & \quad r1 := r2 + r3 \\
(3) & \quad r5 := r9 - r2 \\
(2) & \quad r7 := r1 + 8
\end{align*}
\]

Figure 1.2: Optimal Code Sequence

This leads to another question: is it possible for the compiler to arrange instructions so that the processor can find the necessary number of instructions to issue every cycle? This is clearly dependent on the program; for instance, in the code shown in Fig. 1.2 at most 2 instructions can be issued in the first cycle, no matter what the compiler does. Several studies have shown that the average number of instructions in a program that can be issued simultaneously under ideal conditions is fairly large, possibly greater than 8 [67,52,8,35]. However, they have also indicated that when the compiler is restricted to reordering instructions solely within basic blocks, this number drops to between 2 and 3 [62,17,30]. Clearly, the compiler can exploit the multiple-issue capability of processors, but to do so it must schedule regions larger than basic blocks.

One obvious extension is loops. A loop body may not contain enough simultaneously-issuable instructions to use a processor to full capacity. However, it has been observed that instructions from different iterations of the same loop can usually be issued simultaneously. A simple-minded way to exploit this observation is to unroll the loop till the new loop body contains enough instructions to fully utilize the processor [15]. The example shown in Fig. 1.4 illustrates the procedure for a 2-issue processor using the loop shown in Fig. 1.3.

```plaintext
DO i = 1, 16
   x[i] = x[i] + 1
ENDDO
```

Figure 1.3: Sample Loop

Clearly, a straightforward compilation of the original loop (Fig. 1.4(a)) will issue only one instruction per cycle most of the time. In particular, only (3) and (4) can be overlapped. Thus, it requires 4 cycles to execute one iteration of the loop. However, if the loop was unrolled once, as shown in Fig. 1.4(b), then instructions from different iterations of the loop could be arranged to execute simultaneously. In this case, 2 instructions could be issued every cycle most of the time, resulting in an execution time of 5 cycles for every two iterations.
Lo0:
(1) r1 = ld x[r0]
(2) r1 = r1 + 1
(3) st x[r0], r1
(4) r0 = r0 + 1
(5) br<= r0, 16, L0

Lo0:
(1) r1 = ld x[r0]
(1') r9 = ld x[r0+1]
(2) r1 = r1 + 1
(2') r9 = r9 + 1
(3) st x[r0], r1
(3') st x[r0+1], r9
(4) r0 = r0 + 2
(5) br<= r0, 16, L0

(a) 
(b)

Figure 1.4: Loop Unrolling

There has been a large amount of research on techniques that overlap instructions from several iterations of the same loop, especially on a more sophisticated method known as software pipelining [51,59,34,2,13,44,68,24]. These techniques can build code sequences that issue as many instructions per cycle as the processor is capable of issuing. While it may be premature to say that scheduling loops to fully utilize the capabilities of multiple issue processors is a mature technology, software pipelining does appear to produce fairly good results.

The remaining problem area is the scheduling of acyclic codes. In such codes, it is not possible to build a good schedule by simply “borrowing” instructions from other iterations of a loop. Instead, it becomes necessary to speculate—execute an instruction before it is known whether the instruction should have been executed.

Consider the code in Fig. 1.5(a) below. A 2-issue processor would first execute (1), use its result to evaluate the branch, and then, if the branch was not taken, execute (2).

(1) r1 := r2 + r3
    br== r1, 0, L0
(2) r5 := r9 - r2

(1) r1 := r2 + r3
    (2*) r5 := r9 - r2
    br== r1, 0, L0

(a) 
(b)

Figure 1.5: Static Speculation

Thus, the processor only issues one instruction per cycle, leaving some free capacity. One way the compiler could utilize this excess capacity is to “speculate” that the branch would not be taken and schedule (2) from below the branch so it could be issued simultaneously with (1), thereby producing the code shown in Fig. 1.5(b)\(^1\). This may sometimes be useless; in the case that the branch was taken, the speculation would have been futile. If, however, the speculation was correct, and the branch not taken, the execution time of the program would have been decreased.

\(^1\)In this dissertation, speculated instructions will be indicated by adding a * to the instruction label, or at the end of the instruction.
For a compiler to be able to use speculation in a general fashion requires that the hardware provide an additional mechanism for static speculation. To understand why, consider a program execution where (2) in the Fig. 1.5(b) caused an exception such as overflow. Clearly, this exception can not be reported immediately, since it is not known whether (2) should have been executed at all. Instead, the hardware must delay reporting the exception till it is known that the instruction should have been executed, i.e., the branch is not taken. Thus, the static speculation mechanism must allow the compiler to inform the processor whether an instruction has been speculatively scheduled, and if so, when its exceptions should be reported.

This dissertation deals with both the mechanisms necessary to implement static speculation, and the techniques that enable the compiler to take advantage of static speculation when scheduling acyclic code. It shows that such a static speculation mechanism, in combination with the appropriate scheduling algorithm, can result in acyclic code being executed on multiple issue architectures in a near-optimal manner.

1.1 Problems, Previous Approaches, and Our Approach

This dissertation focuses on 4 problem areas in acyclic code scheduling using static speculation: a mechanism for static speculation, the interaction of static speculation with exception handling and recovery, static scheduling for multiple-issue processors, and the performance of static scheduling & speculation. We shall briefly review these areas; detailed discussion shall be deferred to later chapters.

1.1.1 Mechanisms for Static Speculation

As indicated above, a speculated instruction needs to be treated specially, since it is now being executed in some program runs where it would not otherwise have been executed. This becomes a particular problem when the instruction could cause an exception.

The simplest solution, which was adopted in the early static scheduling compilers, is to speculate only those instructions that are guaranteed not to cause an exception. This category includes operations such as unsigned addition in C that are guaranteed by the language to never cause exceptions. It also includes instructions where the compiler can guarantee no exceptions through analysis.

Of course, this solution considerably restricts opportunities for speculation. However, with additional hardware, it is possible to overcome these restrictions, and speculate instructions that can cause exceptions. A common approach is to add special speculative instructions. These instructions are identical to normal instructions except that exceptions caused by them are not reported immediately; instead, through the co-operation of the hardware, any such exception is buffered, and reported only when (and if) it is determined that the speculative instruction should actually have executed. In Chapter 2, we shall discuss in detail two approaches to implementing speculative instructions, boosting [56,57] and safe-bit [14,53]. It shall be seen that though these approaches do allow fairly general speculation, they still impose some restrictions. We shall propose another approach that is more general than these and is more amenable to use by a compiler.
1.1.2 Static Speculation and Exceptions

Speculative instructions eliminate spurious exceptions. However, there are other problems caused by the interaction of static speculation and exceptions, related to exception handling and recovery. They shall be discussed at length in Chapter 3. We shall illustrate one such problem using the code shown below in Fig. 1.6.

(1) \[ r1 := r2 + r3 \]
    \[ \text{br} = r1, 0, \text{L0} \]
(2) \[ r5 := r9 - r2 \]
    \[ \text{print } r5 \]

(a)

(2*) \[ r5 := r9 - r2 \]
(1) \[ r1 := r2 + r3 \]
    \[ \text{br} = r1, 0, \text{L0} \]
    \[ \text{print } r5 \]

(b)

Figure 1.6: Exceptions

Assume that the non-speculative instruction (1) caused an exception. At this point, the program would start executing an exception handler. Assume that this exception handler, as part of its recovery action, changed the value of \( r2 \) to 0. In non-speculative programs, an exception handler resumes execution after processing the exception by branching to the instruction following the excepting instruction. If this approach is taken, the value of \( r5 \) as printed by the speculative and non-speculative versions of the program will differ. This may be acceptable, or it may be necessary to come up with a mechanism which will ensure that after recovery the values are consistent. In either case, the problem must be addressed.

Previous work on static speculation has either ignored exception recovery, or treated it informally. This has resulted in restrictions being implicitly placed on the kinds of exception recovery actions allowed, and on the kinds of speculation permissible. We shall treat the interaction of exceptions and static speculation formally, and, in particular, derive the restrictions that exception recovery places on static speculation.

1.1.3 Static Scheduling for Multiple Instruction Issue

The earliest schedulers for multiple issue processors were derived from those for pipelined processors, and, like them, concentrated on scheduling within a basic block. As mentioned above, only 2-3 instructions can be issued simultaneously with this approach. However, this was adequate for the early multiple issue processors, since they were not capable of issuing more instructions than this every cycle.

The first scheduling algorithms that processed acyclic regions larger than a basic block considered sequences of basic blocks. These regions were generalized to trees, and then to arbitrary acyclic graphs. Each increase in the region being considered causes a major increase in the complexity of engineering the scheduling algorithm. Unfortunately, it has never been determined whether this additional complexity results in a concomitant performance boost.
A similar issue arises when comparing static scheduling against dynamic scheduling. In static scheduling there is no reason to restrict speculative movement of instructions to only one side of a branch; however, because of the cost of fetching instructions past both sides of multiple branches in hardware, dynamic speculation is tends to speculate past exactly one side of a branch. Thus, static speculation theoretically has an advantage over dynamic speculation in that it can speculate past both sides of multiple branches; it is not clear, however, how much of a performance gain this results in.

Most scheduling algorithms work by repeatedly computing the set of instructions ready for code motion within the region, selecting some of them to be scheduled based on available resources, and then performing any necessary code motion on the selected instructions. While selecting instructions, at times there will be a choice between whether or not to speculate an instruction, or between speculating one of two instructions. A simple selection heuristic is: when scheduling a group of simultaneously issuable instructions, first schedule the non-speculative instructions. If there are more resources available, then schedule speculative instructions, preferring those that are most likely to be executed. The likelihood of execution is determined by some heuristic, such as the number of branches the instruction is moved past. The selection heuristic used by previous algorithms tend to be similar to the one described, or even simpler. However, more complex heuristic may yield better performance.

We have developed a new scheduling algorithm, whole-DAG scheduling, for scheduling general acyclic regions for execution on multiple-instruction issue processors. Whole-DAG scheduling, like other algorithms, computes a set of instructions ready for code motion; however, unlike other algorithms, the set of basic blocks from which the ready instructions are computed is easily controlled. This allows whole-DAG scheduling to control code-motion, take into account restrictions of the static speculation mechanism, and to simulate the effect of algorithms that schedule less general regions. Further, whole-DAG scheduling uses selection heuristics more sophisticated than those used in previous algorithms.

1.1.4 Performance of Static Speculation

There are two varieties of performance studies: so-called limit studies, and attained performance studies. Limit studies address the question:

What is the best performance possible (under a certain set of assumptions) for a program, assuming ideal (possibly unrealizable) hardware and compilation?

Attained performance studies measure the performance that is actually achieved on realistic hardware for code produced by an existing compiler. Typically, the numbers reported are measured for a set of representative programs, such as the SPEC benchmarks.

The earliest limit studies [62,17] ignored the possibilities of speculation and found that, at best, 2-3 instructions could be issued simultaneously, a result duplicated in [30]. However, limit studies that did allow for speculation [52,67,8,35] found that the performance was much higher, with an average of around 7 instructions per cycle even under fairly restrictive assumptions.

Attained performance studies measure the performance of code compiled for a particular multiple-issue processor by running it on the processor (or on a simulation) [29,40,6,25]. One
way these studies are used is to measure the benefits of a particular architectural feature. Another use for them, of more interest to us, is used to measure the performance improvements caused by using a more sophisticated compiler. The performance numbers are reported as a speed-up, i.e., the run-times of programs compiled using the old compiler versus the run-times of the same programs compiled using the new compiler.

These studies have one serious drawback, from our perspective—they report performance for the program as a whole. This makes it impossible to isolate the benefits of specific features, such as speculation or acyclic region scheduling. Consider the limit studies that report large speedups once speculation is added. Much of this speedup comes from the fact that speculation makes it possible to overlap of several iterations of loops. However, this is still possible to do without speculation, by using software pipelining. So, it is not clear how much additional performance can be achieved solely through the addition of speculation. A similar criticism can be made of attained speculation studies that report speedups achieved by a compiler that uses static speculation over a base-line compiler does not implement software pipelining.

Another problem with attained performance studies is that they are incomparable. The compilers used in the studies target different architectures, with radically different timings, ruling out any quantitative comparison. Further, compilers contain many different optimizations and phases, which interact in some non-obvious ways. Thus, when a performance improvement is reported for some compiler by the addition of some feature, it is not clear how much improvement would result by adding the same feature to another compiler, even for the same architecture. The situation is even worse for studies that compare two radically different compilers, rather than modifications of the same compiler.

We seek to isolate the performance benefits of static scheduling of acyclic regions, and of speculation. To achieve this, we perform both limit and attained performance studies that are restricted to acyclic regions. These studies seek to measure the performance improvement achievable and achieved by adding static speculation. We report our results using a new path-averaged metric that may make it easier to isolate the benefits of different scheduling algorithms, even when implemented on different compilers.

1.2 Significance and Contributions

This dissertation shows that it is possible to implement static speculation in a such manner that it can be used by a compiler to achieve significant performance gains on acyclic code, even while taking error recovery issues into account. More specifically, the contributions are as follows:

- We formalized the treatment of error recovery in code containing speculative instructions, and derived from it a set of constraints on speculation. (Chapter 3)

- We developed a new mechanism for specifying static speculation, one of whose benefits is that it allows the compiler greater flexibility while scheduling. We have also showed how it can be used to optimize error recovery. (Chapter 4)
• We developed a novel acyclic region scheduling methodology, called whole-DAG scheduling, that allows for more control than existing techniques, and uses more sophisticated heuristics. This algorithm was implemented on an existing FORTRAN compiler. (Chapter 5)

• We measured both the limits of performance of acyclic regions and the performance attained by the code produced by our scheduler. These measures are presented using a novel path-averaged metric. They show that, given sufficient resources, results close to optimal can be achieved; moreover, multi-path speculation out-performs single-path speculation both in the limit and in achieved performance. (Chapter 6)

• We used some of the insights that arose from our work on speculative execution and exception recovery to develop a technique that allows us to implement speculation, exception handling and register renaming in hardware. This technique is less complex than existing techniques, with accompanying benefits in hardware costs and cycle-time penalties. (Chapter 7)

1.3 Organization

Chapter 2 of the dissertation reviews the background for this dissertation and reviews prior work in the area. Chapter 3 describes the implications of exception handling for static speculation. It formalizes concepts that were previously left to intuition, and shows how how exception handling can constrain speculation. Chapter 4 describes our mechanism for implementing static speculation. It includes a detailed comparison with prior proposals. Chapter 5 introduces our scheduling algorithm. Chapter 6 reports performance numbers for various programs, including both limit and attained performance on various target architectures. Chapter 7 describes a dynamic speculation mechanism developed as spin-off of the work on the static speculation, that can be implemented at low cost. Finally, Chapter 8 contains our conclusions and suggestions for future work.
Chapter 2

Background and Related Work

This chapter surveys related work in static speculation and scheduling for general acyclic regions. Section 2.1 describes hardware mechanisms that can simultaneously execute multiple instruction. Section 2.2 focuses on factors that can inhibit the ability of such hardware to execute several instruction simultaneously. Section 2.3 discusses the development of compiler techniques that seek to overcome these inhibiting factors. Section 2.4 defines the vocabulary of speculation, and discusses the mechanisms used to implement static speculation. Finally, a summary is provided in Section 2.5.

2.1 Instruction Level Parallelism

The area of Instruction Level Parallelism (ILP) studies the set of hardware and compiler techniques that improve performance by overlapping the execution of instructions. This area includes multiple instruction issue hardware, and the compiler techniques required to exploit it. For a detailed survey of issues and work in ILP, refer to [50]. In this section, we shall focus on the hardware required to exploit ILP.

Two instructions in a program are said to be parallel if they can be executed in any order without changing the result of the program\(^1\). For instance, consider the following code fragment shown in Fig. 2.1. In it, instructions (1) & (2), (1) & (4) and (2) & (3) are parallel, and their execution could be overlapped.

\[
\begin{align*}
(1) \quad r1 & := r2 + r3 \\
(2) \quad r4 & := r5 * r6 \\
(3) \quad r7 & := r1 + r8 \\
(4) \quad r9 & := r4 - r0
\end{align*}
\]

Figure 2.1: Instruction Level Parallelism

\(^1\)Note that instructions do not need to be parallel for their execution to be overlapped; we shall show examples of this later in the chapter.
2.1.1 Pipelining

Pipelined processors [32] issue only a single instruction per cycle. However, the execution of an instruction is broken up into several pipeline stages. A pipeline architecture exploits ILP by having several instructions executing simultaneously, each in a different pipeline stage, as shown in Fig. 2.2. There are usually at least 4 stages, known as:

Instruction Fetch This stage is responsible for fetching the next instruction(s), usually from an instruction cache.

Operand Fetch During this stage, the instruction is decoded, and the values to be operated on are fetched from the register file.

Execute An instruction may go through more than one execute stage. It is in this stage (or stages) that the result is actually computed. In Fig. 2.1, for example, the multiply takes 3 execute stages.

Write-Back During this stage, the result produced by the instruction is actually written back to the register file.

If an instruction is waiting for a result produced by another instruction, then it cannot be issued. Such an instruction stalls the pipeline, halting the fetching of any further instructions till its input becomes available, at which point it is issued and execution can proceed. As can be seen, instruction (4) forces the pipeline to stall for a single cycle.

It might appear that the example in Fig. 2.1 is wrong; that an instruction’s result should be written to the register file before it is read by any instruction that needs it. However, a mechanism called bypassing obviates the need for this, by providing the value produced by the last execute stage directly to the instruction in the operand fetch stage, if that instruction needs the value.

Two numbers are used to characterize the execution of instructions in a pipeline: the result latency and the issue latency.

Result Latency Also simply called latency, it is the number of cycles that must elapse before an instruction that uses the result can be issued. In the example, it is 3 for the multiply and 1 for all the other instructions.

Issue Latency This is the number of cycles that must elapse before the next operation of a similar kind can be issued. For instance, in an architecture with a load issue latency of 2, a load instruction cannot be immediately followed by a load instruction. If there are two adjacent loads, then the pipeline will stall.

Instructions with multi-cycle result latencies provide an opportunity for pipelined processors to exploit ILP. If a multi-cycle latency operation is followed immediately by an instruction which uses its result, then the processor would stall. However, if there were instructions that could be issued in parallel with the second instruction, the compiler could schedule them to execute before it. This would allow the processor to do some useful work instead of stalling. The code sequence
Figure 2.2: Pipelined Execution
in Fig. 2.2 illustrates both these situations. The multiply in (2) is a 3-cycle latency operation. This implies that the instruction that uses its result, (4), cannot be issued for 2 cycles. If the compiler could find two instructions that could be issued in parallel with (4), it could schedule them between the two, and thus avoid all stalls. However, it could only find one such instruction, (3), and thus only one of the two stalls can be avoided.

2.1.2 Multiple Instruction Issue

Even single-issue processors are usually implemented with several distinct execution pipelines, e.g., a floating point pipeline and an integer pipeline. Each execute pipeline is capable of starting off a new execution every cycle. However, a pipelined architecture is capable of only sending one instruction for execution every cycle. Thus, all but one of the pipelines must idle. Performance could be increased (at some extra cost) by allowing an operation to be issued to all pipelines every cycle.

Multiple Instruction Issue machines, also known as multiple issue machines, issue more than one instruction per cycle. Fig. 2.3 shows an example of multiple instruction issue. This example assumes that the processor had a multiplier as well as an adder. In such a case, the multiple issue architecture would be able to issue instructions (1) and (2) simultaneously, thereby executing the code fragment in one less cycle.

The issue width of such a processor is the maximum number of instructions that the processor can issue per cycle. Typically, a processor with an issue width of $N$ cannot issue any $N$ instructions. For instance, if a processor can issue an add and a multiply instruction every cycle, it has an issue width of 2. However, it cannot issue 2 multiplies simultaneously.

2.2 Constraints on ILP

There are several factors that prevent two instructions from being scheduled in a way that maximizes parallelism. Some of these constraints on parallelism occur because the processor does not have enough resources to exploit available parallelism.

Dependencies, on the other hand, force instructions to be ordered, so that one instruction can not be issued until some prior instruction has been issued or completed. This reduces the amount of instruction level parallelism available to be exploited. Dependencies, as will be shown in the following sections, are a function of the program.

In this section we shall review the factors that can inhibit instruction-level parallelism.

2.2.1 Resource

Resource constraints arise when the machine does not have enough resources to exploit the available parallelism. There are several kinds of resources that can bottleneck performance.

Functional Units If there are 3 add operations which could be issued simultaneously, but only 2 integer units, then only two instructions of the three instructions can be issued.
Figure 2.3: Multiple Instruction Issue
**Issue Latency** If two instructions attempt to use the same functional unit in successive cycles, but the first instruction has an issue latency of 2, then the second instruction cannot be issued in the next cycle.

**Issue Width** If the processor has an issue width of 2, then the maximum number of instructions that can be issued in parallel is 2. Even if there were 3 instructions that were ready to execute on 3 different functional units, only 2 of them could be issued.

**Registers** A program may need to keep 60 values in registers to optimally execute a program. If the machine has only 32 registers available, the additional values will have to be kept in memory, and moved back and forth when needed, decreasing performance.

The availability of resources may constrain instructions from being issued in parallel. The amount of resources available vary from processor to processor. Accordingly, any scheduling algorithm should take into account the resource availability on the processor for which it is producing code.

### 2.2.2 Data Dependence

A *data dependence* arises when two instructions access the same location. This location can be a memory location or a register. These two accesses may need to be serialized, and thus prevent the instructions from executing in parallel. There are several kinds of data dependencies, which are illustrated by the code in Fig. 2.4.

\[
\begin{align*}
(1) & \quad r_1 := r_2 \div r_3 \\
(2) & \quad r_3 := r_1 \times 5 \\
(3) & \quad r_1 := r_7 + r_9 \\
(4) & \quad r_4 := r_1 - 4
\end{align*}
\]

Figure 2.4: Data Dependencies

Instruction (2) gets one of its inputs from \( r_1 \), which is written to by \( r_1 \). Clearly, (2) cannot start execution till (1) has written its results, otherwise it would read the wrong value. This is an example of a *flow* dependence.

Instruction (3) cannot write its results before instruction (1) has written its results. Otherwise, (1) would overwrite the value of (3), and instruction (4) would read the wrong value from register \( r_1 \). This is called a *output* dependence.

Further, instruction (3) cannot write its result until instruction (2) has read the value produced by instruction (1). Otherwise, (2) would read the result of (3)—the wrong value. This is called a *anti* dependence.
anti and output dependencies are also known as false dependencies. This is because if there were enough registers available, these dependencies could be removed by using different output registers. Fig. 2.5 shows how, by renaming the output registers, instructions that were anti and output dependent in Fig. 2.4 could be made independent. The technique of changing the output registers to remove false dependencies will be revisited later in this chapter.

2.2.3 Control Dependence

Branches create another kind of dependence, called control dependence. Every branch alters control flow, and thereby makes it impossible to determine which instruction should be issued next. Consider the code fragments shown in Fig. 2.6.

\[
\begin{align*}
(1) \ r1 &= r2 \ast r3 \\
(2) \ br &= r1, 0, L1 \\
(3) \ r4 &= r5 \ast r3 \\
&\ldots
\end{align*}
\]

\[
\begin{align*}
(1) \ r1 &= r2 \ast r3 \\
(3)* \ r4 &= r5 \ast r3 \\
(2) \ br &= r1, 0, L1 \\
&\ldots
\end{align*}
\]

In Fig. 2.6(a), clearly (3) can be executed in parallel with (1). However, until it is determined whether the branch will be taken or not, it is not possible to execute it. There is said to be a control dependence between the branch and the instructions that follow it—they must be issued after the branch direction is determined, thereby reducing the amount of parallelism available. A similar phenomenon occurs when the branch direction is known (as it is in an unconditional branch), but the address is not.

One might be tempted to ignore control dependencies, and execute instructions from below the branch before the branch direction is determined, as is shown in Fig. 2.6(b). This can be dangerous. Consider the situation where \( r3 \) has value 0. Executing (3) would result in a divide-by-zero error. However, this error would not have arisen in the code in Fig. 2.6(a); there, \( r1 \) would have had value 0, the branch would have been taken, and consequently (3) would never have been executed. Thus, circumventing control dependencies requires care. It can be done with the help of special speculation techniques, which shall be described below.
(0) read r4
(1) br== r1, 0, L1
(2) r4 := r5 * r3
(3) r4 := r4 - r2
L1:
(4) r1 := r4 + 1

(a)

(0) read r4
(2) r4 := r5 * r3
(1) br== r1, 0, L1

(3) r4 := r4 - r2
L1:
(4) r1 := r4 + 1

(b)

(0) read r4
(2) r9 := r5 * r3
(1) br== r1, 0, L1

(3) r4 := r9 - r2
L1:
(4) r1 := r4 + 1

(c)

Figure 2.7: Preserving Data Dependencies

This is not the only problem with moving code past branches. One must take care to preserve data dependences. Consider the code in example Fig. 2.7(a). Blindly Speculating (2), as shown in Fig. 2.7(b), will result in the value of r4 read by (0) being overwritten whether the branch is taken or not. Clearly, this is incorrect in the situation where the branch is taken. This problem can be easily solved by judicious renaming of registers, as shown in Fig. 2.7(c).

It is important to note that the fact that an instruction occurs after a branch does not necessarily imply that it is control-dependent on it. An instruction is control dependent on a branch only if the branch direction can determine whether or not the instruction is to be executed. Consider the situation shown in Fig. 2.8(a). Here, (3) is below an if-then. Clearly, it will be executed no matter which branch target is taken. Such an instruction is control-independent of the intervening branch. Some compilers [6] take advantage of this fact to optimize code by moving instructions past non-dependent branches, as shown in Fig. 2.8(b). Note that this movement is not speculative.
2.2.4 Increasing Exploitable ILP Dynamically

Several hardware techniques have been developed that increase ILP, letting the hardware issue more instructions per cycle. We shall briefly consider some of them.

Register Renaming

As mentioned above, using the same register name to hold two distinct values creates anti and output dependencies that can force otherwise independent instructions to be issued sequentially. An implementation can provide additional registers, and, in hardware, rename one of the two values; i.e. uses a different register for it, as shown below in Fig. 2.9.

(1) \textcolor{red}{r1 := \ldots} \quad (1) \textcolor{red}{r1 := \ldots} \quad \ \ 
(2) \textcolor{red}{\ldots := r1 \ldots} \quad (2) \textcolor{red}{\ldots := r1 \ldots} \quad \ 
(3) \textcolor{red}{r1 := \ldots} \quad (3) \textcolor{red}{p5 := \ldots} \quad \ 
(4) \textcolor{red}{\ldots := r1} \quad (4) \textcolor{red}{\ldots := p5} \quad \ 

Figure 2.9: Register Renaming

By renaming, the hardware removes the false dependencies, permitting the instructions to be issued in parallel. Thus, in the example, instead of all 4 instructions having to be issued serially, now (1) & (3) and (2) & (4) can be issued in parallel.

3-1 ALUs

There is no way to remove flow dependences. However, it is possible to build hardware that can execute pairs of instructions with a flow dependence between them in the same time as it would take to do one instruction. The most common of these is a floating point multiply-add [43], which can be used to combine floating-point multiply and add instructions into a single instruction as shown below, in Fig. 2.10.
(1) f1 := f3 * f7
(2) f8 := f1 + f2

(a)

(1) f1 := f3 * f7 + f2

(b)

Figure 2.10: 3-1 ALU

There are many other combinations that can be implemented, including almost all pairs of simple (i.e. not including shift, multiply or divide) integer instructions [66,65]. These 3-to-1 ALUs effectively remove a flow dependence, by allowing two 2-1 instructions serialized by a flow dependence to execute simultaneously.

Dynamic Speculation

Speculation is used to execute instructions before it is known whether they should be executed—i.e., before an intervening branch has been evaluated. Speculation is usually used in conjunction with some static or dynamic mechanism for predicting branch targets [37]. The issue hardware will fetch and issue instructions from the predicted target. If the prediction turns out to be incorrect, the result is discarded. Consider the example in Fig. 2.11.

(1) r3 := . . (1) r3 := . .
(2) r1 := . . (2) r1 := . .
(4*) r3 := . .
(3) br==1 r1, 0, L0 (3) br== r1, 0, L0
(4) r3 := . .

(a) (b)

Figure 2.11: Speculation

(4) is along the predicted (not-taken) path, and is executed before it is known whether or not the branch will be taken. If it turns out that the branch is taken, the result will be discarded; in particular, the value in r3 will be restored to the value produced by (1). Otherwise, if the prediction was successful and the branch was not taken, the value of r3 would be that produced by (4). Thus, dynamic speculation removes the control dependence that would otherwise have forced (4) to be executed after the branch.

Out-of-Order Execution

The techniques outlined above would provide only a limited amount of additional performance if the processor was constrained to issue instructions in the order that they appear in the instruction stream. Consider the code obtained through register renaming in Fig. 2.9. If the processor was restricted to in-order issue, it would issue (1), then have to wait till it completed before issuing any other instructions. An out-of-order issue implementation, would however examine and issue
from several instructions every cycle. In particular, such a processor could notice that (3) was ready to issue, and execute it in parallel with (1).

**Prior Work**

Out-of-order issue was implemented on the CDC 6600 [60]. The earliest implementation of ILP-increasing hardware dates back to the 1960s. Register-renaming combined with out-of-order execution was implemented in the IBM 360/91, using the Tomasulo algorithm [63]. A survey of the early work in this area can be found in [31].

Some work has concentrated on implementing exactly one of these techniques (such as [12,26, 58] on out-of-order issue or [20] on register renaming); however, most recent work integrates several of these features. The more ambitious of these proposals choose to integrate register renaming, out-of-order issue and dynamic speculation [39,49]. [29] explores the costs and tradeoffs of implementing and integrating these proposals.

3–1 ALUs are a more recent innovation. Typically, the only time they are used is when the compiler produces special 3–1 instructions. In this form, they can be combined with any of the other ILP-increasing proposals. There exists at least one proposal [66,42] that attempts to dynamically detect when adjacent pairs of 2–1 instructions can be combined to use a 3–1 ALU. However, we are not aware of any proposal that integrates such a combining mechanism with any other dynamic ILP increasing mechanism.

### 2.3 Scheduling

#### 2.3.1 Basic Block

The earliest work in scheduling to exploit ILP was for pipelined processors. These processors need to exploit only minimal amounts of instruction-level parallelism for full utilization, since they only need to cover the delay of instructions with multiple-cycle result latencies. The amount of ILP required is less than 2 for most processors [67]. Since a basic-block typically has an ILP of 2-3, this provided little incentive for scheduling regions larger than a basic block.

Early theoretical work includes proofs that various flavors of the optimal scheduling for pipelined processors are NP-complete [10]. A summary of such work can be found in [36]. There are several heuristic scheduling techniques that tend to work fairly well in practice [21,18, 3]. Later work in the area focuses on the interaction with other phases of compilation, especially register allocation [19,7]. Much of this work is surveyed in [33].

The advent of multiple-issue processors that exploit more ILP than available in a basic block motivated research into scheduling regions bigger than basic blocks. This research is discussed next.
2.3.2 Cyclic Code

Cyclic code should be scheduled differently from acyclic code, since it has an additional degree of freedom. In an acyclic code fragment, the only source for parallelism is that between instructions in the body of the code. In cyclic code, on the other hand, one can also overlap instructions from different iterations of the loop.

Software pipelining [59,34,2,13,51] attempts to find a sequence of instructions, which may include instructions belonging to different iterations, that completely utilizes available resources. It may not be possible to schedule the original loop body so as to use all available resources. These "gaps" are filled by instructions from some succeeding iteration. This, in turn, may produce gaps. These gaps are filled in turn. Finally, a repeating pattern is obtained. This pattern becomes the new loop body. Fig. 2.12 illustrates this process.

It is interesting to note that, for DO/for loops, it is not necessary to use speculation to obtain a schedule that uses all available resources efficiently, even in those cases where the loop body contains branches. There exist techniques, such as hierarchical scheduling [34] or reverse if-conversion [68], that can yield fairly dense schedules when such branches are present.

Speculation does pay off when attempting to software pipeline while loops [61]. In a DO/for loop, the number of iterations that will be executed can be computed before the loop body is entered. This makes it safe to produce code that overlaps execution of several iterations—the software pipelined loop is executed only as long as it is safe to execute all instructions in the body, then an epilogue is executed, which finishes executing the remaining instructions from the last few iterations. In a while loop, there is usually no way of determining a priori how many times the loop will execute. Therefore, any gaps between the beginning of the loop and the determination of whether the next iteration will be executed cannot be safely filled with instructions from the next iteration. However, if static speculation is available, these gaps can be filled, by issuing instructions speculatively from the next iteration speculatively.
2.3.3 Acyclic Code

Acyclic code scheduling techniques can be distinguished on the basis of how big a region of the control flow graph (CFG) they pick to schedule, and on the kinds of code motions they allow.

*Trace scheduling* [15] considers paths in the CFG. These paths, known as *traces*, are selected using information about branch probabilities. The scheduler selects a trace and schedules it. It then repeats the process till the entire CFG is scheduled. During the scheduling process, instructions may be moved past forks in the control flow graph both ways, and up through merges. During this process additional *bookkeeping* code needs to be inserted to preserve semantic correctness. Trace scheduling results in fairly good schedules for numerical code. However, it can lead to excessive code expansion. Also, it is inherently restricted to considering a single path at a time.

*Superblock scheduling* [25] is based on trace scheduling. After trace selection, all joins on a trace are removed by *tail duplication*, i.e., by copying all code below the join. This simplifies the bookkeeping associated with moving an instruction up past a join.

*SRDAG scheduling* [38] extended trace scheduling to regions that form trees of basic blocks. Results reported in that paper suggested that it could perform much better than trace scheduling; however, it has apparently never been implemented in a compiler\(^2\).

Work is underway on an extension to trace scheduling, known as *trace scheduling-2* [16]. This work will attempt to schedule general acyclic regions.

*Hyperblock scheduling* [41] extends superblock scheduling to deal with multiple paths. Initially, the CFG is converted into a basic block by if-converting the instructions that are conditionally executed. During if-conversion, the conditional instructions are associated with a predicate or guard that is true only if the instruction should be executed. The resulting basic block is scheduled normally. Then, reverse if-conversion is applied: basically, the guards are converted into if statements.

*Percollation scheduling* [2] provides mechanisms for scheduling instructions on all paths, rather than just one path. However, the policy used to apply this mechanism is not so clear. One particular strategy, *greedy scheduling*, can extract massive parallelism, but is very complex, and results in uncontrolled production of speculative operations.

Instructions can be safely moved from a basic block to a basic block with equivalent control dependence, without generating any copies. For instance, an instruction can be moved from below an if-then-else to the basic block above it, as long as the data dependences permit it. Both the *boosting optimizer* [57] and *global scheduling* [6] use this code motion optimization.

*Selective scheduling* [45,44] is a general multiple-path method. This scheduling method is not restricted to acyclic code; it can be applied to schedule cyclic code, producing a software pipelined schedule. It, like global scheduling and the boosting optimizer, was designed to implement speculation. We shall discuss these techniques at greater length in Chapter 6.

\(^2\)The original paper dealt with microcode compaction
2.4 Speculation

2.4.1 Vocabulary

An instruction is executed speculatively if it is issued and may complete before it is known whether control flow will actually reach the instruction. For instance, consider the code shown in Fig. 2.13. (4) is speculated, since it is executed, and should complete, before it is known whether the branch (2) is not taken.

Note that speculation has been used to denote other, less aggressive, behaviors (usually in the context of hardware speculation). In "fetch" speculation, instructions are fetched from the predicted path, but not issued. In "execute" speculation, the predicted instructions are allowed to execute, but their results are not written back till the branch is executed.

The speculation can either be dynamic, i.e., performed by the hardware, or static, i.e., performed by the compiler. In the case of static speculation, the architecture must provide instructions that are the speculative equivalents of normal instructions. The compiler will produce a program in which an instruction that should be speculated has been converted to its speculative equivalent and moved (by the compiler) past one or more branches.

A speculated instruction must be treated specially in several ways.

- Any exceptions caused by a speculated instruction must not be reported until it is known that the instruction should have been executed. Only after that must the exception be reported. For instance, if (4*) in the Fig. 2.13(b) overflowed, the overflow must not be reported if the branch is taken.

- Register updates can be dealt with in two ways: the output register of a speculative instruction such as r9 can either be updated immediately, or later, after it is known whether the instruction should have been executed. In the second case, speculative instruction register updates must be handled specially.

- There must be some way of indicating to the hardware where the speculative instruction originally came from, and thus at which point its exceptions should be reported.

Before proceeding, let us define some terms and phrases that shall be used in this dissertation to discuss speculation. Some of these definitions have two flavors. First, there is a definition exclusively in terms of a speculative program. Then, there is an intuitive description which
assumes that the speculative program was derived from a non-speculative program, and appeals to the original program and the transformation. It is important to recognize that there is no necessity for there to be an original non-speculative program.

**Origin** The origin of a speculative instruction is the position of its equivalent instruction in the original non-speculative program. The origin is defined only in those cases where the speculative program was derived from a non-speculative program.

**Commit Point** The commit point of a speculative instruction is the position in the speculative program where any interrupt caused by the speculative instruction is reported. The commit and origin points can be different. Further, it is possible for an instruction to have multiple commit points. An instruction is said to have been *committed* when control flow reaches its commit point, and all actions associated with the commit point have been accomplished.

**Speculated Past** A speculative instruction is said to be speculated past a branch if there is a path containing the branch from the speculative instruction to some commit point for that instruction. In the context of a transformation from a non-speculative program, an instruction is said to be speculated past a branch if the instruction used to be "below" a branch (in the control flow sense), and was moved "above" it. Thus, (4) was speculated past branch (2).

**Fail** A speculative instruction is said to fail if control flow reaches a point from which it is impossible to reach the commit point without first executing the speculative instruction again. This happens when control flow reaches a branch which the instruction was speculated past, and then branches in the "wrong" direction. The instruction is said to have *failed at that branch*. In the example, (4) would fail at (2), if the branch was taken.

**Succeed** An speculated instruction is said to succeed if control flow reaches its commit point. An instruction is said to *succeed at a branch* if it does not fail, i.e., control flow at a branch goes in a direction from which it is still possible to reach the instructions commit point. For instance, (4) succeeds at branch (2) if the branch is executed and not taken.

**Resolve** A speculative instruction is resolved when the instruction either succeeds or fails.

**Outstanding** A speculative instruction is said to be outstanding at a point if it has been executed but not resolved at that point.

The control flow graph in Fig. 2.14 illustrates the movement of non-speculative I from its origin point (which is also its commit point) above branches B1 and B2, converting it into speculative instruction I*. In the process, the instruction has been speculated past branches B1 & B2. Clearly, the speculation will fail at B1 if control flows to the true side of the branch, similarly at B2. I* will succeed only if control flows to the false side of both branches, thereby reaching its commit point. It will be resolved if, after executing I*, either control reaches the commit point, or if control flows to the true side of either B1 or B2. The instruction will be outstanding as long as control is at some intermediate point.
2.4.2 Mechanisms

We have briefly described dynamic speculation, and proposals for implementing it in Section 2.2.4. A more detailed study of such mechanisms can be found in Chapter 7. In this section, we shall introduce the two main proposals for implementing static speculation. These are boosting [55, 57, 56] and safe-bit [14, 53]. These schemes differ, as will be seen below, both in the way the commit points of the speculative instructions are determined, and in the way register updates are handled.

Boosting

Boosting defines the commit point for a speculated instruction by annotating the speculated instruction with the path to the commit point. The path to the commit point is described by listing the number of conditional branches on the path to the commit point, and whether the path is along the taken/not-taken side of the branch.

Consider the example in Fig. 2.15. The speculated instruction, \((1^*)\) has been moved past the two branches, first past the left \((L)\) side of a branch, and then past the right \((R)\) side of a second branch. The path from the speculated instruction to the actual instruction is simply the reverse of this path. Thus, as shown, the additional information added to the speculated instruction in boosting will be that the path to the commit point is \(LR\), indicating that the instruction will succeed if the first branch encountered goes to the left, and the second branch encountered goes to the right.

A modification made to boosting was the addition of the ability to use \(X\) or "don't care". This is used when an instruction has been speculated past both sides of a branch, as is shown in Fig. 2.16.
A point worth noting is that not all speculations can be described by boosting, even after this addition. One such speculation is shown in Fig. 2.17.

An instruction's commit point is necessarily immediately after the last branch in its path. Thus, any exception caused by the instruction will be initially reported at the first branch it was speculated past. Notice also that several instructions can be simultaneously committed at branch. This allows the commit actions to be performed en bloc, which may be more efficient than committing each instruction separately.

Safe-Bit

In the safe-bit approach, every register has an added safe-bit. If a speculative instruction causes an interrupt, its output register is unsafe, i.e. the safe-bit for its output register is set, and some interrupt information is stored. Should some speculative instruction have a safe value as an input, then it, too, is made unsafe. If a non-speculative instruction reads an unsafe value, the interrupt is reported. Thus, committing a speculative instruction requires an explicit "commit" instruction. This is any non-speculative instruction which reads the output register for the speculative register.

The safe-bit approach defines the commit point for a speculated instruction using another instruction, which reads the output register of the speculated instruction. Consider the example in Fig 2.18. The instruction, \((1)\), has been speculated to \((1^*)\). \((1^*)\) will be committed at the first read of its result register, \(r5\), by a non-speculative instruction, in this case \((2)\). This assumes, of course, that there are no intervening writes to \(r5\).

Notice that the safe-bit approach allows very fine-grain control of commit point placement. This comes at the cost of having to add an extra instruction that reads the output register of the speculative instruction at the point where we wish to commit it\(^3\).

\(^3\)Of course, sometimes there will already be an instruction which reads the output register; in that case it comes for
Figure 2.16: Boosting with “don’t cares”

Figure 2.17: Speculation Inexpressible with Boosting
The use of output registers to link speculative instructions with their commit points means that a different mechanism has to be used for instructions that have no output register, such as stores. [53] has proposed such a speculation.

In safe-bits, a speculation fails when the output register of a speculative instruction is written to by a non-speculative instruction without having been read. At this point, the safe-bits are cleared. Of course, there is still the problem of detecting failure of instructions with no output register.

**Register Access**

Another way in which boosting and the poison-bit scheme differ is in their treatment of registers. In the poison-bit scheme, there is exactly one register file, and all register names refer to entries in this file. Thus, a register update is visible to all instructions immediately. In boosting, register updates made by speculative instructions are hidden; in particular, a non-speculative instruction may not read a value written by a speculative instruction till that instruction commits.

To implement this hiding, boosting uses multiple register files. One of these is the “regular” register file, which is manipulated by non-speculative instructions in the usual fashion. The others are known as shadow register files. They are used to buffer the results of speculative instructions. Each shadow register file is associated with a certain commit point. All instructions are associated with some file, the non-speculative instructions with the regular file, and the speculative instructions with the shadow file that has the same commit point as the instruction.

The rules for accessing and manipulating registers are somewhat complicated, and will be illustrated using Fig. 2.19. The files are arranged in a hierarchy based on the distance to the commit point.
### Table: Register Manipulation in Boosting

<table>
<thead>
<tr>
<th>Regular Registers</th>
<th>Shadow Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 18</td>
<td>17 18</td>
</tr>
<tr>
<td>24 24</td>
<td>24 24</td>
</tr>
</tbody>
</table>

1. \( r_0 := r_1 + 1 \)
2. \( r_1 := r_0 > 8 \)
3. \( r_1 := r_0 + 2 \)
4. \( .. := .. r_1 .. \)

#### Figure 2.19: Register Manipulation in Boosting

Point, with the regular file at the top of the hierarchy. In the example, there are only two files, and the shadow file is necessarily below the regular file. Every instruction reads from its associated register file. Thus, (3*) from Fig. 2.19 reads from the shadow file. It writes to its associated register file. Thus (3*) modifies \( r_1 \) in the shadow file. It will also modify that register in all register files lower in the hierarchy, unless the register has previously been written to by some instruction associated with some file lower in the hierarchy. Thus, (1) modifies \( r_0 \) in both register files, while (2) only modifies \( r_1 \) in the regular file, since \( r_1 \) in the shadow file has already been written to by (3*).

If a speculation succeeds, its shadow file is committed by copying its values into the regular file. After committing, or after the speculation fails, the shadow register file can be reused. This requires resetting it, which is done by copying values from the lowest active register file.

The original program that was transformed into the speculative version used in Fig. 2.19 might have been the one shown below in Fig. 2.20.

1. \( r_0 := r_1 + 1 \)
2. \( r_1 := r_0 > 8 \)
3. \( r_1 := r_0 + 2 \)
4. \( .. := .. r_1 .. \)

#### Figure 2.20: Original Code for Fig. 2.19

It is not possible to blindly perform the same speculation using poison-bits, as is obvious from
the code in Fig. 2.21(a). In this case, instruction (4*) incorrectly reads the value written to r1 by (2), instead of that written by (3*). The fix, shown in Fig. 2.21(b), is to use register renaming. Here, the compiler simply chooses different registers (in this case r4) to connect (3*) and (4*). Notice that any other instruction that read the value written to r1 by (3*) must now instead read from r4.

\[
\begin{align*}
(1) \quad & r0 := r1 + 1 \\
(3*) & r1 := r0 + 2 \\
(2) & r1 := r0 > 8 \\
(4*) & \ldots := \ldots r1.. \\
\text{branch} & \\
\text{(a)} & \\
\end{align*}
\quad
\begin{align*}
(1) \quad & r0 := r1 + 1 \\
(3*) & r4 := r0 + 2 \\
(2) & r1 := r0 > 8 \\
(4*) & \ldots := \ldots r4.. \\
\text{branch} &
\end{align*}
\quad
\text{(b)}
\]

Figure 2.21: Speculation with Renaming

In general, a scheme that uses \( M \) register files containing \( N \) registers as proposed in boosting\(^4\) makes less efficient use of registers than a scheme that uses a single register file containing \( M \times N \) registers. A compiler can always use renaming in the single file scheme to implement any speculation possible using the multiple file scheme. However, it has more flexibility in how it uses registers. For instance, it could choose to use all the registers for non-speculative operations, and thereby avoid spilling registers to memory. Actually, as shall be seen in the next chapter, it may be necessary to use a multiple file scheme if it is necessary to support a certain strict model of exception recovery.

2.5 Summary

This chapter introduced the concept of instruction-level parallelism, showing how it can be exploited in hardware. We also showed how ILP can be inhibited, and techniques for increasing it, both dynamically in hardware, and statically, via the compiler. We described efforts that have been made in the past to generate good schedules for processors that can exploit ILP. Finally, we defined the vocabulary we shall use in this dissertation while discussing static speculation. We also described two schemes for static speculation.

\(^4\)It must be noted that the register structure is somewhat orthogonal to the commit point specification mechanism; for instance it would be possible to combine the annotations of boosting with a single register file.
Chapter 3

Exception Handling and Static Speculation

In this chapter, we discuss the interaction of exceptions and speculative code. In Section 3.1, we introduce concepts used in the rest of the chapter, including an abstract mechanism for specifying speculation. In Section 3.2, we describe the issues that arise when writing exception handlers for programs that contain speculative exceptions. We extend the precise exception model so that it simplifies the task of writing exception handlers for such programs. In Section 3.3, we describe the details of implementing this extended model. This includes constraints that the speculative program must satisfy, and certain other requirements imposed by exception handling requirements. Finally, a summary is given in Section 3.4.

3.1 Introduction

3.1.1 Precise Exceptions

Normally, a processor fetches and executes instructions from some body of code, under the control of the program counter for that program. Occasionally, however, this regular execution sequence is interrupted by an exception, and control is transferred to a piece of code known as the exception handler, whose purpose is to process the exception. The exception handler takes action appropriate to the exception, and then, possibly, resumes normal execution of the program.

As an example, consider a low-cost implementation of an architecture that includes an integer divide instruction. The implementation is designed with no integer divide hardware. Instead, it emulates the divide instruction in software, by interrupting normal execution whenever the processor encounters a divide instruction (via an exception such as "unimplemented instruction") and transferring control to an exception handler. The handler contains code which performs the divide using implemented instructions such as shifts and subtructs.

Handling any exception after it has been reported requires that the hardware satisfy certain constraints. Typically, these are satisfied on most non-speculative architectures by implementing precise exceptions [54]. An exception is said to be precise if, at the time the exception is reported, the machine state (i.e., registers, memory, etc.) is that which would be obtained if the following conditions were satisfied:
• All instructions occurring before the excepting instruction in the program order have completed.
• No instruction occurring after the excepting instruction has been issued.
• The program counter points to the excepting instruction.

It is possible to add dynamic speculation to an implementation, and still satisfy these conditions. Unfortunately, as will be shown, this definition of precise exceptions breaks down when applied to architectures with speculative instructions. Further, no adequate extension or replacement has been proposed in the literature; the interaction between exceptions and static speculation has usually been treated informally.

3.1.2 Exception Handlers and Precise Exceptions

In this section, we shall discuss the implementation of exception handlers on a pipelined processor, since the problems encountered in implementing exceptions on a pipelined processor are the similar to those encountered in implementing exceptions on an architecture with static speculation.

On a pipelined processor, by the time the hardware recognizes that an instruction will cause an exception, it is possible for several instructions issued after the excepting instruction to have completed and updated the machine state. Further, there may be instructions from before the excepting instruction that are still in execution. Such a situation can arise in the code fragment shown in Fig. 3.1. Assume that the latencies of the divide, multiply, and add are 11, 6, and 4 respectively. Further, assume that the multiply, (2), detects an exception, such as overflow, at the last stage of execution. At this point, instruction (3) will have completed and modified \( r_5 \), while (1) will still be executing.

\[
\begin{align*}
(1) & \quad r_1 := r_2 / r_3 \\
(2) & \quad r_4 := r_5 * r_6 \\
(3) & \quad r_5 := r_8 + r_9
\end{align*}
\]

Figure 3.1: Pipelined Exceptions

The exception cannot be reported immediately; in particular, the processor must wait for all instructions prior to the excepting instruction to complete, in case they also cause an exception. If one of them does except, then that is the exception that must be reported. Thus, in Fig. 3.1, reporting the exception caused by (2) must be delayed till (1) completes. If it happens to cause an exception, then that is the exception that must be reported.

Generally, the exception handler must be able to read the inputs to the excepting instruction to determine the appropriate corrective action. This requires that the effects of any instruction issued after the exception be undone before executing the exception handler. For instance, in the example, (3) overwrites the input to (2) stored in \( r_5 \). Without the ability to restore the original value of \( r_5 \), it may be impossible for the exception handler to determine the appropriate action.

Thus, in general, an exception handler requires that, at the point an exception is reported, the machine state should reflect the execution of all preceding instructions and no succeeding
instructions—requirements that are satisfied if precise exceptions are implemented.

Notice that the complicating factor in the handling of exceptions is out-of-order completion, i.e., instructions finishing execution while instructions occurring prior to them in the program are still executing. Thus, (2) had to wait for (1) to complete before reporting its exception, since it finished before (1). Similarly, the effects of (3) had to be undone because it finished, and updated the machine state, before (2) completed. Out-of-order completion, as will be seen, occurs naturally in architectures with static speculation.

After processing the exception, the exception handler typically resumes normal program execution by branching to the excepting instruction (or the instruction after it). Note that this may cause instructions to be re-executed in a modified state; for instance, if the handler for the overflow of (2) modified r5, (3) would produce a result different from that produced the first time it was executed. This has repercussions when dealing with restart in speculative architectures, as shall be seen later.

3.1.3 Abstractions

Instead of choosing to discuss the interaction between exception handling and speculation using either boosting or safe-bits, we shall use an abstract approach: each speculative instruction's commit point will be explicitly represented. This allows us to ignore the mechanism used to link a speculative instruction and its commit point. Thus, the speculation of (3) in Fig. 3.2(a) will be represented abstractly, as shown in Fig. 3.2(b), instead of using either boosting (Fig. 3.2(c)) or safe-bit (Fig. 3.2(d)).

Speculation performed using either boosting or safe-bits can be represented using this abstract approach. Clearly, the only concern is finding the commit point for each speculative instruction. In boosting, each speculative instruction is committed immediately after the last branch in its path; in the abstract representation, we add a commit for that instruction after the branch. In the
safe-bit approach, a speculative instruction is committed when its output register is read by some non-speculative instruction. This is represented in the abstract approach by adding a commit for the instruction before its first use by a non-speculative instruction.

Another distinction between the two speculation mechanisms is in the way they handle the output of speculative instructions; in boosting the output values are buffered till commit, while in safe-bits the output registers are committed immediately. However, we can abstract this difference in behavior to some degree. Let us define the following terms:

**Speculative State:** The values produced by uncommitted speculative instructions, and the locations in which they are stored. In boosting, this would include the shadow registers; in safe-bits, this would include all registers written to by "uncommitted" speculative registers.

**Non-Speculative State:** The values produced by non-speculative instructions and by committed speculative instructions, and the location in which they are stored. In safe-bits, a register would be logically moved from the speculative state to the non-speculative state by committing it, i.e., by testing its safe-bit.

**Non-Speculative Location:** For a value in the speculative state, the location to which the value is "copied" when the instruction that produced it is committed. For a value in the non-speculative state, the location in which it is stored. In boosting, this is the regular register with the register number specified by the output register for the instruction.

Thus, in both schemes, a speculative instruction first writes its output to the speculative state. After the instruction is committed, the non-speculative location is updated with this buffered output value. In boosting, this update copies the value from the speculative state to the non-speculative state. In safe-bits, no value is physically copied; instead, the register in which the value resides is now considered part of the non-speculative state.

### 3.1.4 Origin vs. Commit: Producing Speculative Code

In the examples discussed above, we have speculated code so that the commit point of an instruction is the same as its origin point. Fig. 3.3 illustrates, using boosting, a case where it is not. In the example, (2) is moved from below (1) to a point above the branch. However, boosting always commits instructions immediately after the branch. Thus, as indicated in Fig. 3.3(c), the instruction is committed before (1), rather than after it, i.e., the commit point of (2 *) is different from its origin point.

In practice, however, this distinction is vacuous. Typically, the code for a speculative architecture will be produced by a compiler from some high-level language. The compiler will produce only the speculative code, as in Fig. 3.3(b). Since there is no non-speculative code to compare against, there is no way of defining the origin point. Further, if there was a mechanism to force the compiler to produce both a speculative and non-speculative version of the same program, it is probable that the non-speculative code produced by the compiler would have identical commit and origin points. Thus, in our example, the non-speculative code produced would be that of Fig. 3.3(d).
3.1.5 Assumptions

We shall, as far as possible, use the abstract representation of speculation. Unless otherwise mentioned, the examples used can apply to either scheme or we shall explicitly mention which approach is being used. Also, we shall defer distinguishing between cases where the speculative instruction results are written directly to the regular register file, or are first buffered.

Further, since the focus of the remainder of the chapter is the impact of static speculation on exceptions, unless otherwise mentioned, we shall ignore complications introduced by pipelining, parallel issue, etc. The implementation we shall be assuming is a serial (i.e., sequential, in-order, single-issue, non-pipelined) implementation augmented by speculative instructions.

3.2 Precise Exceptions & Static Speculation

3.2.1 The Breakdown of Precision

Our goal is to make it possible to write exception handlers for programs with speculative instructions. On a non-speculative architecture, this can be achieved by implementing precise exceptions; i.e., at the time an exception is reported, all prior instructions must have executed, and no succeeding instructions must have executed.

The words “prior” and “succeeding” imply some order on the instructions. In the case of non-speculative programs the order was the program order. Now, a speculative instruction actually has two positions in the program—its issue point and its commit point. Intuitively, if two instructions exception, and the exception of one instruction is reported before the other, then that instruction should precede the other; thus, the natural order in the case of speculative instructions would be to assume that the speculative instructions are ordered by the position of their commit point in the program. In that case, the instructions preceding a speculative instruction are those that precede its commit point, and the instructions that occur after a speculative instruction are those that occur after its commit point.
One reason behind the rule that no instructions after the excepting instruction should have executed was to ensure that the inputs to the excepting instruction were unchanged. Unfortunately, this goal is not satisfied by the rule when it comes to speculative instructions, as can be seen from Fig. 3.4. In this example, if instruction (3\*) exceptions, even though no succeeding instruction (under the order defined) has executed, the input \(r0\) to (3\*) has been overwritten by (2). Furthermore, as shall be shown later, it is perfectly reasonable for a compiler to produce such code.

\[
(1) \quad r0 := \ldots \\
(3\*) \quad r1 := r0 + r2 \\
(2) \quad r0 := \ldots \\
\text{br} = r2, 0, L0 \\
\text{commit } (3\*)
\]

Figure 3.4: Speculation Changes Inputs

Assume, for the sake of argument, that we could restore the state to that which was present at the time (3\*) was issued, in which case the value in \(r0\) is the value used by (3\*). This will let the exception handler read the necessary values. However, now we are faced with the problem of restart. Exception handlers can modify arbitrary state; assume that the exception handler for (3\*) set the value of \(r2\) to 0. If execution resumed at the point (3\*) was issued, then the branch would be taken, resulting in the paradoxical situation where the excepting instruction (3\*) should not have been executed, and therefore should not have caused an exception.

So the alternative is to resume execution at the commit point. This in turn implies that all modifications must be done on the commit state. Consider the base case, where the exception handler makes no change to the state: intuitively, any succeeding instructions must see the value \(r0\) as defined by (2), not the value at the issue point. Thus, an exception handler is expected to read from the issue state and write to the commit state—a somewhat paradoxical situation.

Previous authors have tried, informally, to sidestep this dilemma by considering only speculative programs derived from non-speculative programs via a set of structured transformation rules that preserve the input-output semantics of the original, non-speculative program. They then define a precise state for the exception of a speculative instruction to be the state obtained if the equivalent instruction in the original non-speculative program had caused the exception. Thus, if speculative instruction (4\*) in Fig. 3.5(b) causes an exception, its exception state is said to be precise if the state is identical to that obtained if (4) in the original non-speculative program of Fig. 3.5(a) had caused a precise exception.
x := ...

y := x * 2

if(...) {
  z := x + 1
}

x := ...

(1) vr0 := ...

(2) vr1 := vr0 * 2

breq ... L0

(3) vr2 := vr0 + 1

L0:

(4) vr4 := ...

L0:

L0:

(4) r4 := ...

(4) r9 := r5 + 1

(4) r9 := r5 + 1

(1) r4 := r2 * r7

(2) breq r4, 0, L0 L1:

(3) r1 := r2 / r3

(3) r1 := r2 / r3

(3) r1 := r2 / r3

commit (4*)

commit (3*)

commit (3*)

commit (3*)

Figure 3.6: Producing a Non-Sequentializable Program

Figure 3.5: Problems with Exceptions

The approach of appealing to a non-speculative equivalent program is attractive; it appears to model our intuitive notion of what it means to, and where we should, speculate. Unfortunately, it is as yet informal. For instance, what should we do in the case where we do not start by transforming from a non-speculative program? It should be clear from Fig. 3.4 that we can write speculative programs for which there exist no obvious non-speculative equivalent. Note that, in that example, any exceptions caused by the speculative instruction (3*) will be reported after the branch, at the commit point. At this point the input to (3*), r0, has been over-written by (2). Now, any equivalent non-speculative program must execute (3) after the branch. However, at that point, the input to (3) is unobtainable, since it has been over-written. It is possible to find an equivalent non-speculative program, but only by rearranging the registers.

As mentioned earlier, it is entirely reasonable for an optimizing compiler to produce such a program. Consider the compilation sequence in Fig 3.6, which yields the code similar to that in Fig 3.4. Fig. 3.6(a) shows an example of some high-level code. Note that x is defined above the if, and that its last use is inside the if statement. The compiler converts it into assembly, using "virtual registers" to represent the values computed in the program. Then, it schedules the code, as
shown in Fig 3.6(c), introducing speculation. Finally, it allocates processor registers to the values represented in the virtual registers. At (1) it allocates $r0$ for $vr0$, and at (3*), $r1$ for $vr2$. At (2), the compiler notes that the last use of the value in $r0$ has occurred, and so it is free to reuse $r0$ for $vr1$. This action makes it impossible to directly convert back to a sequential program; in such a program, the value of $vr0$ and therefore $r0$ would need to be available at the speculated instructions origin point.

### 3.2.2 Equivalence

In the previous section, we introduced the concept of “equivalence” where, intuitively, a non-speculative program is said to be “equivalent” to a speculative program if an instruction in the speculative program causes an exception exactly when the “equivalent” instruction in the non-speculative program does, the machine states of the two programs are identical at the point the exception is reported, and exceptions in the non-speculative program are precise. Before proceeding to formalize the concept, let us consider why the notion of having such an “equivalent” non-speculative program is so attractive.

Assume that there exists an equivalent non-speculative program for some speculative program. Since exceptions in the non-speculative program are precise, the machine state will be such that all preceding instructions have executed, and no succeeding instruction will have modified the state. Thus, it is possible to write an exception handler that can examine the machine state, including the inputs to the excepting instruction, determine the cause of the exception, make arbitrary modifications to the state, and resume execution, with the expected behavior. Now, by our (informal) definition of equivalence, the machine state in the speculative program, when the “equivalent” exception is reported, is identical to the machine state in the non-speculative program; hence, it is possible to use the same handler to process the exception. Thus, by ensuring the existence of an “equivalent” non-speculative program for some speculative program, we ensure that we can write exception handlers for that speculative program; further, these exception handlers will be identical to those written for non-speculative programs.

Note that this notion of equivalence is stronger than equivalence by input-output behavior. For instance, the speculative program in Fig. 3.6(d) has the same input-output behavior as the high-level program and any non-speculative code generated from it. The only time the behavior of the speculative assembly code can be distinguished from that of some non-speculative assembly code is if an intermediate state of the program is examined.

### 3.2.3 Sequentialization

It is clear that there are some speculative programs for which there exist no equivalent non-speculative programs. However, we have still not defined when a non-speculative program is equivalent to a speculative program. Nor have we shown, other than at an intuitive level, how to apply this notion of equivalence to defining exception models. In this section, we shall attempt to formalize the approach of using a non-speculative program to define precise exceptions for a speculative program.
A speculative program for some architecture with static speculation and a non-speculative program for the same architecture are said to be equivalent if they satisfy the following conditions:

- The two programs have the same input-output behavior.
- There is a bijective mapping between those instructions, speculative and non-speculative, of the speculative program, and those instructions of the non-speculative programs that can cause an exception.
- Under the same input, one program has an exception exactly when the other program does; further, the instructions that cause the exception are equivalent to each other under the previous mapping.
- At an exception, the machine state presented to the exception handler is identical for both the programs.

A speculative program is said to be sequentializable if there exists an equivalent, non-speculative, program. Fig. 3.7 gives an example of a sequentializable speculative program. Equivalent instructions are shown connected with dashed lines.

Of course, this is true only if the architecture specifies that all speculative register updates are buffered, as in boosting. If so, it is easy to verify that if the speculative and sequentialized programs are run, and one of them exceptions, the other will do so. Further, the two excepting instructions will be equivalent, and the machine states will be identical. Notice that the equivalent non-speculative program is not the same as the original non-speculative program.

Exceptions in a program with static speculation are precise if the program is sequentializable, and the exceptions in the non-speculative program are precise. We call such a program precise.
3.2.4 Sequentialization Without Shadow Registers

Formally, it is necessary for all effects of a speculative instruction to be buffered from the machine state till the instruction is committed. This includes writes to registers. For instance, if, in example Fig. 3.7, instruction (3*) caused an exception, then the state should not reflect the execution of speculative instruction (1*); in particular, register r5 must not have been modified by (1*). This implies that an approach similar to that adopted by boosting is necessary—speculative instructions must write their results to auxiliary storage such as shadow registers, which will be copied into the machine state only when the instructions are committed.

This is inefficient: the auxiliary registers are equivalent to the normal registers except that they can be accessed only by speculative instructions. There are situations in which performance would be improved if the non-speculative instructions had access to all available registers. Further, there is an overhead for copying at the commit point: while it is possible to achieve the effect of a commit without copying, it complicates the register access hardware. Lastly, either spilling speculative registers (by explicit load instructions or on a context switch) is not possible, with the attendant inefficiency, or special instructions are required to access the speculative registers.

For reasons of efficiency, the idea of using exactly one register file, and having the results of speculative instructions update the register file is extremely attractive. If we use such an architecture, it is clear that exceptions will be imprecise. However, we can define a weaker notion of precision. We call an exception of a speculative program on such an architecture precise upto speculative writes.

Consider an architecture in which speculative instructions modify their output registers immediately: a speculative program and a non-speculative program for the same architecture are said to be equivalent upto speculative writes if they satisfy the following conditions:

- The two programs have the same input-output behavior.

- There is a bijective mapping between those instructions, speculative and non-speculative, of the speculative program and those instructions of the non-speculative programs that can cause an exception.

- Under the same input, one program has an exception exactly when the other program does; further, the instructions that cause the exception are equivalent to each other under the previous mapping.

- At an exception, the machine state presented to the exception handler is identical for both the programs ignoring those registers that were last written to by uncommitted (i.e. outstanding or failed) speculative instructions at that point in the speculative program.

Exceptions in the speculative program are precise upto speculative write if there exists a non-speculative program that is equivalent upto speculative writes, and all exceptions in the non-speculative program are precise. Such a speculative program will be called precise upto speculative writes.

Under this definition, the speculative program in Fig 3.7 is precise upto speculative writes, with the equivalent program being the sequentialized program shown in the example. This assumes that
in the underlying architecture all speculative output register writes occur as soon as the corresponding instruction completes, and that exceptions for non-speculative instructions are precise. If (3 *) causes an exception, then all the registers but r5 will be the same in both programs. However, r5 was the output register for (1 *), which is an outstanding speculative instruction at the point of the interrupt. Therefore, the two register states are the same but for those registers that were written to by outstanding speculative instructions, and (at least with respect to (3 *)) the two programs are equivalent up to speculative writes.

### 3.3 Implementing Precise Exceptions

#### 3.3.1 Constraining Speculation

The burden of ensuring that a program with static speculation satisfies either of the two definitions of precision is mostly the responsibility of the compiler. The architecture determines whether it will be possible to achieve precision, or just precision up to speculative writes. After that it is the compiler which must ensure that the code is sequentializable.

Proving that an arbitrary program is sequentializable is undecidable. However, it is possible for a compiler to produce speculative programs that are guaranteed to be sequentializable, and whose equivalent program can be found using some simple transform. One intuitive transform is as follows:

- Each speculative instruction is assumed to be *similar* to some non-speculative equivalent, where a non-speculative instruction is said to be similar to some speculative instruction if they have exactly the same behavior except that the non-speculative instruction except that
  - reports exceptions immediately and
  - refers to the non-speculative locations of all values read/written by the speculative instruction.

For each speculative instruction, add at its commit point a similar non-speculative instruction.

- If two (or more) instructions share the same commit point, arrange their non-speculative instructions in the order in which any exceptions caused by them are ordered, i.e. if the exceptions of one of the two speculative instructions will be reported first, put the non-speculative instruction similar to that instruction first.

- Delete all speculative instructions, and all instructions that are used only to support speculation.

The bijection under which the two programs should be equivalent is the obvious one: each speculative instruction is mapped to the non-speculative instruction that replaces it, and each non-speculative instruction is mapped to itself. Such non-speculative and speculative programs are said to be *similar* to each other.
The compiler can ensure that the code it produces is sequentializable, and that the equivalent program can be found using the simple transformation rule mentioned above, by ensuring that the produced code satisfies the following constraints:

1. A non-speculative instruction cannot access values in the speculative state; i.e., it cannot read the result of an uncommitted speculative instruction.
2. If control flow reaches an instruction's commit point, then it must have passed through the instruction; specifically, a speculative instruction must dominate its commit points, and lie on every cycle that contains a commit point.
3. Any value read by a speculative instruction must be available at the instructions commit point at its non-speculative location. Note that this implies that if a speculative instruction reads a value produced by another speculative instruction, then its commit point must be preceded on all paths by the commit point of that other instruction.
4. At the commit of a speculative instruction, the non-speculative location must be updated by the value produced by the instruction, and not some other value.

All of these constraints can be easily satisfied with current compiler technology. In particular, Constraint 3 and Constraint 4 can be modeled by assuming that there is a “use” of all the inputs and outputs of a speculative instruction at the instructions commit point. The other conditions can be satisfied by speculating instructions in a controlled manner.

Of course, it is necessary that the implementation give the same effect as a serial issue architecture. Thus, the implementation must implement precise exceptions for all non-speculative instructions. Further, for architectures that are precise up to speculative writes, it may be desirable to ensure that all speculative instructions issued before the exception is reported must have updated the register state, and that none issued after that point should have modified the register state.

3.3.2 A Proof of Equivalence

A formal proof that a non-speculative program similar to speculative program will be speculatively equivalent to it if the compiler constraints outlined above are satisfied is deferred to Appendix A. There, we shall provide a formal model of execution of speculative and non-speculative programs. We shall define similarity, compiler constraints and speculative equivalence in the context of that model, and show that similarity and compiler constraints imply speculative equivalence.

3.3.3 Restart

Restarting in the presence of static speculation is somewhat complicated. After processing an exception, the exception handler may have modified the state in an arbitrary fashion. Normally, under the model of serial execution that we have assumed, no further action needs to be taken. However, static speculation introduces out-of-order execution. For instance, if (2*) in Fig. 3.8(a) exceptions, then (3*) will have been executed, out-of-order. Intuitively, (3*) should be re-executed in the modified context. Thus, if the exception handler modified the value of $r3$, (3*)
Restart (2*):
(1) r3 := r7 - r9
(3*) r1 := r3 + 1
(5) jmp (5)

Restart (3*):
(3) r1 := r3 + 1
(6) jmp (6)

(b) Speculative Code

should be re-evaluated using this new value. In fact, should any instruction exception, then
all speculative instructions that are outstanding at the point the exception is reported must be
re-executed in the new context.

To re-execute all outstanding speculative instructions requires that the exception handling
system must be able to identify them. The direct approach, keeping a hardware log of all such
instructions, can become prohibitively expensive if we allow unlimited speculation. If, for instance,
the hardware can keep track of the 32 outstanding speculative instructions, what happens to
programs which has 33 outstanding speculative instructions at some point? Also, this log becomes
additional state that must be saved on context switches. If this approach is used, it is reasonable
to restrict the compiler to produce code with no more than, in this case, 32 outstanding speculative
instructions at any point.

There is an interesting software-based solution to this problem. In most cases the compiler
(or programmer) that produces the program will be able to identify, for each point in the program,
the speculative instructions that must be re-executed should an exception occur at that point. So,
the compiler adds additional information to the program. This information enables the exception
handler (perhaps with some minor additional hardware support) to identify and re-execute
the outstanding speculated instructions after processing the exception. The exact details of this scheme
depends on the architecture it is applied to; it was first developed for boosting, and a description of
it can be found in [55].

Basically, adapting it to the abstract scheme we have been using requires that the compiler
do the following: for each commit point, the compiler produces a block of code known as the
restart block for that commit point. It contains the non-speculative version of the instruction that
is committed at the point, followed by all the speculative instructions outstanding at the commit
point. The restart block is terminated by a jump to the instruction following the commit point. The
compiler also builds an auxiliary table that maps each commit point to the beginning of its restart
block. This is probably a hash-table, indexed by the address of the commit points.

The restart blocks for (2*) and (3*) in Fig. 3.8(a) are shown in Fig. 3.8(b). Note that the
restart block for (2*) contains the non-speculative equivalent for (2*), and (3*), which is
outstanding at the commit of (2*). This restart block is terminated by a jump back to (5), the
instruction following the commit point.

When an excepting speculative instruction is committed, the corresponding branch handler is re-executed. Instead of jumping back to the commit point after processing the exception, the exception handler finds the beginning of the restart block for the commit point. It then branches to the first instruction in the restart block (or to the instruction after the first one), and resumes normal execution. Thus, before the original program is re-entered, all the outstanding speculative instructions will have been re-executed.

In this example, if (2*) causes an exception, the exception handler is entered at the commit point (4). Assume that during the execution of the handler, it changes the value of r7. After completing execution, it resumes execution at Restart (2*). Thus, it re-executes (2), changing the value of r3. It also re-executes the outstanding instruction (3*). Finally, it jumps back to the original program.

If some non-speculative instruction causes an exception, then all outstanding speculative instructions need to be executed. It is too expensive to build a restart block for every instruction. Fortunately, there is a way to avoid this. This optimization, which was developed for the speculation mechanism technique described in the next chapter, depends on the following observation: an instruction that is outstanding at some point continues to be outstanding till either its speculation fails, or till it is committed.

Now, assume that it was possible for the exception handler to determine the next commit point to be executed. Then, the instructions in that commit block include all instructions that were outstanding at the excepting instruction, less those whose speculations failed, plus some other speculative instructions issued in the interim. So, what the exception handler does is set a flag, called the restart flag, which causes the restart block for the next commit point to be executed in its entirety. This will have the desired effect, as is indicated by the following observations:

- All instructions in the restart block that were outstanding at the time of the original exception will get re-executed in the state as modified by the exception-handler.

- Those instructions that were outstanding at the original exception, but are not in the restart block are those that failed; they did not need to be re-executed, since their results could not be used by any instruction that did not itself fail.

- Those speculative instructions that are added in the interim need to be executed in the state created by the re-execution of prior speculative instructions in the state as modified by the exception handler.

Of course, these observations are applicable only if the code was compiled to obey the rules mentioned above.

Thus, assume that in our example, (1) causes an exception. The exception handler, while processing the exception, changes the value of r7. Now, it is necessary to re-execute (2*) with this new value, and therefore the value in r3 is invalid. To ensure that (2*) gets re-executed in the new state if necessary, the exception handler sets the restart flag, and resumes execution. The program executes (3*) using the invalid value for r3. It then executes the branch. If the branch is taken, the speculation failed. Therefore, the results of (2*) and (3*) are never going to be
used, and there was no need to re-execute them. If, however, the branch is not taken, then these instructions must be re-executed. If the branch is not taken, the commit for \((2^*)\) is executed. Due to the restart flag being set, the entire restart block is executed, resulting in \((2)\) and \((3^*)\) being re-executed. This time, \((2)\) uses the value of \(r7\) as modified by the exception handler, and produces the correct value for \(r3\). Then \((3^*)\) executes using this corrected value. Thus, exceptions of the non-speculative instruction \((1)\) is handled correctly, by using the restart flag.

There is another problem: how are context-switches to be implemented? There will be state in the processor that keeps track of the speculative instructions, including details such as the shadow register values and which, if any, speculative instruction has caused an exception. One approach is to dump out all of this state. The other approach is to recreate it, using the restart technique outlined above. After a context switch, the first time a commit point is reached, the restart flag is set. This causes the speculative state to be rebuilt from the saved non-speculative state.

### 3.3.4 Writing Exception Handlers for Static Speculation

Putting the pieces together, it is clear that writing and executing exception handlers for speculative programs that are precise by our definition is the same as writing exception handlers for non-speculative programs with precise exceptions\(^1\). Basically, the exception handler should written with respect to the equivalent non-speculative program. It will not be able to distinguish between a precise speculative program and its non-speculative equivalent. However, before resuming execution, it will have to use one of the techniques discussed above to re-execute all outstanding speculative instructions.

In the cases the underlying architecture restricts the program to being precise upto speculative writes, the exception handler will not be able to distinguish between the speculative program and its equivalent as long as it does not examine those registers last written by an uncommitted speculative instruction. Thus, as long as the exception handler is transparent to (e.g., does not read) the values in those registers, it can be written with respect to the non-speculative equivalent program, and ignore the fact that it is actually being written for a program with speculative instructions. The handler must, however, restart differently, by re-executing all outstanding speculative instructions.

### 3.3.5 Discarding Precision

Implementing precise exceptions in the presence of speculative instructions creates two problems:

- **Register lifetimes are unnecessarily extended.** If a register value is read or written by a speculative instruction, the register cannot be reused after the last time the value is read; instead, the value must be preserved till all speculative instructions that read the value have been resolved.

- **Restarting requires re-executing all the outstanding speculative instructions.**

\(^1\)Note that in the case of non-speculative architectures, precision is usually enforced by the hardware, whereas in speculative architectures, it requires the co-operation of software and hardware.
A technique that would allow us to implement exceptions imprecisely might allow us to get around these problems. There are various approaches to weakening precision for non-speculative architectures [46]. These schemes preserve the ability to execute the exception handlers for those exceptions critical to the operation of a machine, such as virtual memory exceptions, but exchange the ability to handle all exceptions for simpler hardware. Unfortunately, in the case of exceptions on architectures with static speculation, these approaches do not solve the problem. It is still necessary to re-execute all outstanding speculative instructions after returning from an exception, and this in turn requires that all register values read by a speculative instruction be preserved. For instance, consider a TLB-miss exception on the speculative load (1\textsuperscript{*}) in Fig. 3.9. There may be speculative instructions that would have used the result of the load, such as (2\textsuperscript{*}). So after the exception is issued and handled, at the commit of (1\textsuperscript{*}), there may be outstanding speculative instructions which used the value of the load, and need to be re-executed.

\[
\begin{align*}
(1\textsuperscript{*}) \quad r9 & := \text{ld} \ldots \\
(2\textsuperscript{*}) \quad r11 & := r9 + r10 \\
(3) \quad \text{breq } r4, 0, L0 \\
L1: & \\
& \quad \text{commit (1)} \\
& \quad \text{commit (2)}
\end{align*}
\]

Figure 3.9: Re-executing Speculative Instructions

Fortunately, it may be that all critical exceptions can be processed immediately; for instance, the TLB-miss of (1\textsuperscript{*}) could have been processed before executing any of the remaining instructions. This would do away with the necessity of restarting any instructions. Of course to get any benefit from this approach one must save all the state that was associated with speculative instructions at a context switch; otherwise, recreating it would still require the re-execution of all outstanding speculative instruction.

### 3.4 Conclusions

In this chapter, we examined the interaction of static speculation and exceptions. We showed that the standard model of exception handling on non-speculative architectures needs to be extended to apply to architectures with speculative instructions. We formalized two such extensions, one for architectures which buffered a speculative instruction’s result register updates till the instruction was committed, and the other for architectures in which speculative instructions updated their result registers immediately.

It is possible to produce speculative code in a fashion such that it is impossible to satisfy the extended precise exception model we have proposed; thus, it is necessary for speculative code to satisfy certain properties if it is to comply with the exception model. We describe these properties.

We discuss the problems associated with restarting a program after handling an exception. The major problem is that the outstanding speculative instructions in a program must be re-executed.
We show why hardware solutions are inappropriate, and discuss a software approach.

Thus, we arrive at a framework for implementing exception handlers for programs with speculative instructions. The programs are compiled so as to be precise up to the limits of the underlying architecture. The exception handlers are written as though for non-speculative programs with precise exceptions. The only difference between these handlers, and those that are written for non-speculative programs is that they must re-execute outstanding speculative instructions.

Finally, we show how to weaken the extended precise exception model, so that it is still possible to handle those exceptions such as virtual-memory faults that are essential to the functioning of a "correct" program, but in which all exceptions cannot be handled. This approach makes it possible to make better use of available resources.
Chapter 4
Speculative Tagging

In this chapter, we describe an alternate mechanism for specifying speculative instructions. This mechanism, known as speculative tagging, is introduced in Section 4.1. Section 4.2 investigates the problems associated with multi-path speculation, and shows how speculative tagging can be adapted to overcome them. In Section 4.3 we show how speculative tagging can be optimized for exception handling and recovery in speculative programs. These modifications, as well as some others discussed in Section 4.4, are summarized in Section 4.5. Finally, we conclude in Section 4.6.

4.1 Speculation Tags

In Chapter 2, we described two mechanisms that had been proposed in the literature for implementing speculative instructions—boosting and safe-bit. We showed that they both had certain strengths and weaknesses. In particular, safe-bits allowed instructions to be speculated in a fairly general manner. Boosting, on the other hand, could not represent all possible speculations. However, boosting could allow all instructions to be speculated, while safe-bits had trouble speculating instructions with no result registers, such as stores. Further, boosting allowed the hardware to group identically speculated instructions. This grouping made it possible optimize various actions; for instance, boosting can simultaneously commit a group of instructions.

Neither boosting and safe-bits are completely satisfactory. Ideally, we would like to come up with an approach that combines the uniformity and grouping of boosting with the flexibility of safe-bits. Our mechanism, speculation tagging, attempts to do so.

4.1.1 Success and Commits

In the speculative tagging mechanism, which we shall usually abbreviate to tagging, every speculative instruction is annotated with some tag. A speculative instruction is committed when its tag is committed. Thus, the speculation desired in Fig. 4.1(a) will be represented using speculative tagging as shown in Fig. 4.1(b).

Notice that we use a special commit instruction to commit (the speculative instructions associated with) a tag. The combination of tag and commit provides the same flexibility as the
output-register/register-read mechanism used in safe-bits. Further, since tags are independent of output registers et. al., they can be applied uniformly to all speculative instructions.

Multiple speculative instructions can have the same tag. These instructions are considered to be in the same group. This allows us to specify and take advantage of grouping; for instance, identically speculated instructions can be given the same tag, and all of the speculative instructions with the same tag are committed when the tag is committed. In Fig. 4.2, both (1) and (2) have been speculated from the same region. They are given the same tag; hence, when (3) is executed, they will both be committed together. Notice that it is straightforward to determine if two speculative instructions are in the same group; simply examine the tags.

4.1.2 Failure and Flushing

In a processor that implements speculation, certain effects of a speculatively executed instruction are buffered. These effects include exceptions, memory updates, and possibly, as in boosting, register updates. If the instruction is successful, then these effects are committed, and the buffer space can be reused. If the instruction fails, the hardware must discard, or flush, the buffered effects, so as to free the buffer space for later reuse. Therefore, to allow reuse of buffer space, it is important to indicate failure as well as success in an architecture that implements static speculation.

Failure of speculation is fairly straightforward to detect in boosting—if control flows off the path specified by the boosted instruction, then the instruction has failed. In safe-bits, a speculation fails when the output register of a speculative instruction is written to by a non-speculative instruction without having been read.

We indicate failure of speculation by means of explicit flush instructions. When a tag is flushed, all speculative instructions with that tag are assumed to have failed, and the resources
being used to buffer their effects are reclaimed. A flush instruction is typically added right after a branch direction that, if taken, causes the speculation to fail. Thus, the code produced for the speculation shown in Fig. 4.1(a) would actually be that shown in Fig. 4.3.

Notice that one of the resources that is reclaimed is the tag itself. When a tag is flushed (or committed) it, too, is reset for reuse; among other things, all speculative instructions that were associated with the tag cease to be so associated, and a new group is started.

### 4.1.3 Register Files

It is possible to use speculative tagging with either a single register file, or with some mechanism that buffers speculative register updates, such as the shadow register scheme used in boosting. As has been shown, the single file approach tends to make better use of the available registers. We shall, in this chapter, and through the rest of the dissertation, assume a single file approach, unless otherwise stated. This, of course, restricts us to being precise up to speculative writes.

### 4.1.4 Memory Operations

Upto this point, we have not considered the problems associated with speculating memory operations, such as stores and loads. It turns out to be necessary to add additional hardware to support such speculation. To understand why, consider Fig. 4.4(a): assume that it is desirable to speculate (3) up past the branch and issue it before (2). Clearly, doing this naively will result in (3) overwriting the value to be read by (2).

---

1A note about terminology: \((r1)\) denotes the contents of the address stored in \(r1\). Thus \((r1) := \ldots\) is a store to \((r1)\), and \(. := (r1)\) is a read from \((r1)\).
A possible solution is to use different memory locations, as shown in Fig. 4.4(b). Originally both stores wrote to the address in r1. In the transformed version, the speculative store, (3*) is to some other memory location, whose address is stored in r7. The instructions that used to read the value stored by (3), such as (4), must be modified to now use the address in r7.

(1) (r1) := ..
(2) .. := (r1)
(3) (r1) := ..
(4) .. := (r1)
(5) .. := (r8)

(a)

(1) (r1) := ..
(2) .. := (r1)
(3*) (r7) := ..

(b)

Figure 4.4: Renaming Memory

Unfortunately, there are situations where this approach will fail. Consider instruction (5) in Fig. 4.4(b). Clearly if r8 has the same value as r1 when the program is executed, (5), too, should be modified to use the new address. However, if r8 and r1 are different, it should be left untouched. Unfortunately, is not always possible for the compiler to determine which of the two behaviors is correct. Assume (4) was a read of x[i] and (5) of x[j], where x is some array. Then, r1 will be the same as r8 exactly when i is equal to j. Obviously, there will be situations when it is impossible for a compiler to determine whether this is so. Worse still, there may be situations in which the two are sometimes equal, and sometimes not. This is another problem caused by aliasing.

Aliasing forces us to add extra hardware to support the speculation of memory operations. The
mechanism used is an adaptation of the store-buffers used in non-speculative processors. In such processors, when a store operation is executed, the results of the store must be written to the cache (and possibly to memory, such as when the cache is a write-through cache). It may be the case that it is not possible to write the store into cache immediately, either because of a cache-miss, or because of the cache organization. In this case, the store is buffered in the store-buffer till it is possible for it to be released to the cache. All loads must look in the store-buffer, as well as the cache. If there is an address match both in the store-buffer and cache, then the value from the store-buffer is returned.

The modifications that must be made to the store-buffer to support speculative memory operations seem fairly straight-forward. Briefly:

- A speculative store is first buffered in the store-buffer.
- If the speculation succeeds, the store is written to the cache.
- If the speculation fails, the store is discarded.
- A load looks in the store-buffer for a match.
- If there is a match, then the matched value is returned.

Notice, however, that these modifications do not cover the case when there are multiple matches in the store-buffer. In the original, non-speculative, store-buffer the solution was to return the value of the match added last to the buffer. This solution will work in the presence of speculation only if the order of memory references to the same address is preserved during the speculation. In the presence of aliasing, this condition can result in having to preserve during speculation the order of all memory operations.

Notice also that the maximum number of speculative stores that can be simultaneously outstanding is limited by the size of the store-buffer. This size must be known to the compiler, which must ensure that this limit is not exceeded while speculatively scheduling stores.

### 4.2 Multi-Path Speculation

Previous work on speculation mechanisms has mostly dealt with speculation past exactly one side of a branch; it has generally ignored the complications caused by speculating instructions from both the taken and not taken sides of a branch.

One complication that arises in multi-path speculation occurs when the same destination register is being used for instructions being speculated from both sides of a branch. Obviously, one of the two has to be renamed. All the uses of this register have to be renamed, too. This is possible as long as the uses can access only one of the two values, as is the case in Fig. 4.5. If, however, there exists an instruction that can read both values, such as (3) in Fig. 4.6, then it becomes necessary to "undo" the renaming by adding a copy operation at the appropriate point.

Memory operations have a similar problem. If memory operations are moved past both sides of a branch, then they may interfere with each other. Further, because of aliasing, it may not be possible to determine whether or not the operations interfere. Nor is it possible to solve this
Figure 4.5: Tagging with Renaming

Figure 4.6: Tagging with Copying
Figure 4.7: Multi-Path Memory Speculation

problem using the store-buffer. Consider the case illustrated in Fig. 4.7. Clearly, (2*) should read the value at address r3. When r3 = r1, this should be the value written by (1*). However, the store-buffer as described above will incorrectly return the value stored by (3*). In fact, unless r3 is always different from or always the same as r1, no order of the four speculative instructions is correct.

This problem arises because a store (3) that would not have been executed simultaneously with the load (2) in the sequential program is visible to the load in the speculative program. So, the speculative load instruction must somehow be able to specify those instructions that are visible to it. One simple technique is to add a set of tags to each speculative load instruction, called the path mask. When a speculative load is matched in the store-buffer, all stores whose tags are not in the path mask are ignored. In the example, (2*) could be written ..:3 : = (r1)[1]. Since (3*)'s tag, :7, is not in the mask, it will not be considered. Thus, if r3 = r1, the value stored by (1*) will be returned.

Path masks in effect declare all stores that should be visible to the load operation. This property makes path masks useful for other purposes as well. For instance, path masks can be used to allow stores from different basic blocks to be speculated past memory operations from other basic blocks, as is shown in Fig. 4.8. Each load simply masks out the stores that were speculated from below the load; as a special case, non-speculative loads ignore all speculative stores.

4.3 Implementing Exceptions

The previous chapter showed how exception recovery issues imposed certain constraints on the way the compiler could speculate instructions. In this section we shall discuss hardware modifications tailored for speculative tagging that help alleviate some of these restrictions.
4.3.1 Register Pressure

One constraint on the compiler is that it has to ensure that, for each speculative instruction, the instruction's inputs are available at the commit point. Since more than one instruction can be committed with the same instruction, this implies that a register can be used as the output register for at most one instruction with a given tag. Consider the speculation shown in Fig. 4.9, which uses r1 as the output register for both (1*) and (2*). Even though this code has the correct input-output behavior, it is illegal, since it will result in imprecise exception for (2*). In particular, the value in r1 that is used by (2) would have been overwritten by (3). It is still possible to perform this speculation by using a different register as the output register of (1*); however, this increases the register pressure.

```
(1*) r1:3 := ..
(2*) ..:3 := .. r1
(3*) r1:3 := ..
branch
(1) r1 := ..
(2) .. := .. r1
(3) r1 := ..
```

(a) branch

```
branch
(1) r1 := ..
(2) .. := .. r1
(3) r1 := ..
```

(b) commit :3

Figure 4.9: Illegal Speculation

It is possible to modify the hardware to make this sequence legal. The hardware keeps track of whether any instruction associated with a tag has caused an exception; if so, it treats all subsequent
speculative instructions with that tag as no-ops (until the tag gets committed or flushed, of course). This can be implemented by adding an exception flag for each tag. With this modification, the above sequence would be legal. If (2*) caused the exception, (3*) would never be executed, and the value in r1 would be the necessary input to (2*).

Note that if the values were written by instructions with different tags it is always possible for the compiler to relieve register pressure. The compiler simply spills the registers, i.e., stores the register value in memory, and loads it into the necessary register where appropriate, such as before the commit of the instruction’s tag.

4.3.2 Restart Blocks

One possible exception recovery mechanism that can be used is the restart block. Let us review the use of the restart block: when an exception is reported at a commit point, the exception handler is entered. After the exception handler finishes execution, the code branches to the restart block for that commit point. In particular, it branches to that instruction corresponding to the excepting
speculative instruction. In Fig. 4.10, if $1^*$ causes an exception, then the restart block should be entered at $(1')$.

Finding the beginning of the restart block will probably involve using the address of the commit instruction to hash into a table that will provide the address of the corresponding restart block. Thus, the address of the instruction commit :3 would be used to determine the address Restart:3.

However, finding the address of the instruction within the restart block at which execution is to resume is more of a problem. The hardware can aid this process: specifically, it can keep track of the number of successful instructions issued for each tag. This is implemented by adding an issue counter to each tag. The instruction in the restart block corresponding to the instruction that caused the exception is offset from the beginning of the restart block by the value of the issue counter. If $(1^*)$ caused the exception, only 1 instruction, $(2^*)$ would have been issued. Thus, the excepting instruction is Restart:3 + 1.

4.3.3 Lazy Restart

Restart blocks can result in a significant amount of code replication. Part of expansion can be avoided by using a scheme called lazy restart. In this scheme, the restart block for a commit instruction contains only those instructions corresponding to the ones that are committed by that commit; thus, it does not contain copies of instructions that are outstanding at the commit point. Now, if a commit point reports an exception, all instructions with the same tag issued subsequent to the instruction that caused the exception will be re-executed using the restart block. All the outstanding speculative instructions still need to be re-executed. Notice that these instructions are available in the restart blocks corresponding to their commit points. So, the hardware simply forces the entire restart block to be executed when (and if) control reaches the commit point.

To implement lazy restart, a restart flag is added to every tag. When an commit reports an exception, it sets the restart flags for all tags other than the current one. Before trying to commit a tag, the instruction examines the tag's reset flag. If it is set, the commit branches to the beginning of its restart block, and starts executing it. The restart flag is cleared at each commit or flush instruction.

In the example above, instruction (30) would not be part of the restart block. Instead, if (1) should report an exception, the restart flag of tag :1 would be set. Then, if control reached commit :1, the program would start executing the corresponding restart block, and in particular would re-execute (3). Of course, if control never reached the commit instruction, it would eventually encounter a flush, and the restart flag would be cleared.

4.4 Miscellanea

Just as registers can be saved to memory and later restored, so should the information associated with tags. This would be be useful in the same way that register spilling is useful:

- When the number of tags required exceeds the number available.
- To preserve tag values across function calls.
• To preserve machine state on a context switch.

This would need additional instructions which moved the state associated with tags, including exception information, the various flags, and store buffer entries, to and from memory.

As mentioned above, one way of dealing with tags at a context switch is to save their values, and restore them when the process is swapped in. The contents of the store-buffer must also be saved.

However, it is not necessary to save this "speculative" state. An alternative mechanism, based on lazy restart, saves only the state that would be saved in a non-speculative architecture (i.e., registers, PC, etc.). Instead, the restart flag for all tags is set when a process is swapped in. This will cause the program to regenerate the speculative state by re-executing all speculative instructions that were outstanding at the time of speculation. During this re-execution, any exception that was caused by a speculative instruction, and thereby lost on a context switch, will be reported when that instruction's equivalent in a restart block is executed.

4.5 Design

4.5.1 Summary

The architecture we propose has various mechanisms for implementing static speculation with a single register file. These mechanisms are built around a new resource, the speculation tag. A processor has several speculation tags, which can be uniquely identified (written as :n, where n is some number). Each tag has the following state associated with it:

Issue Counter keeps track of the number of speculative instructions with that tag that have been issued. Is incremented every time such an instruction is issued, unless some previous instruction with that tag would have caused an exception.

Exception Flag is set when some speculative instruction with that tag would have caused an exception. Once set, all further instructions with that tag are converted to no-ops.

Restart Flag is used to signal that the entire restart block associated with the tag needs to be executed.

Additionally, there may be some state keeping track of exception information.

Each speculative instruction has a tag. This tag is used to buffer and the instruction's "speculative" state, i.e., information about any possible exceptions caused by the instructions, and about the store-buffer.

Tags can be manipulated in various ways. They can be committed, flushed or restarted. Further, they can be spilled to and loaded from memory. There may exist special instructions that perform some combination of these tasks simultaneously. A good example would be an instruction that takes two sets of tags as inputs, committing one, and flushing the other. Further, this instruction could be combined with some other instruction, such as a branch.
Loads use tags, in the path-mask, to mask out unwanted stores while searching the store-buffer. Typically, the path-mask for all loads with the same tag will be the same. Thus, instead of specifying a path-mask with each load, a special instruction can be used that associates a path-mask with each tag. Then, every load instruction with that tag will implicitly use that associated path-mask.

One important question that we have not addressed in this chapter is the number of tags needed. Every tag takes up bits in the instruction word; we would therefore like to make provision for as few tags as possible. Note that the ability to spill tags allows us to reuse tags and therefore perform more speculation than would be otherwise possible; however, this solution comes at the cost of extra spills. We would like to keep the number of spills to a minimum. One way of doing that is to suitably restrict the amount of speculation performed. The scheduling algorithm described in Chapter 5 incorporates techniques for doing so.

We shall see in Chapter 6 that the amount of speculation required to fully exploit a processor depends on the issue-width of the processor. The results in that chapter indicates that, for a 4 issue processor, the best performance is obtained when instructions are speculated into a basic block from its 5 most likely descendant blocks, while for a 8 issue processor, the best performance is obtained using the 8 most likely descendants. The results described in that chapter will indicate that, under these circumstances, one can avoid spilling 98% of the time by providing 5 and 8 tags respectively. Further, to capture most of the performance, we will need only 3 and 5–7 tags to get most of the performance from 4 and 8 issue processors.

4.5.2 Cost Comparison

There are several ways in which static speculation increases the cost of building the architecture. It requires extra bits in instruction words. Some hardware is required to buffer the exceptions caused by speculative instructions. Finally, extra hardware is needed to "shadow" or buffer the effects of speculative instructions. We shall roughly estimate these costs for the three static speculation mechanisms we have discussed: boosting, safe-bit and speculative tagging.

Instruction Cost

The safe-bit approach adds an extra bit to each instruction, indicating whether it is speculative or not. However, this is not the only additional bits added to the stream by the addition of speculation. In the safe-bit approach, it is necessary to read the output of a speculative instruction to trigger the reporting of any exception caused by it. For some speculative instructions, the result may be read at the commit point as part of natural program execution. Otherwise, dummy read instructions will need to be added to commit the result. This additional overhead may be significant. Even if only 1 in 10 instructions is a speculated instruction that requires an additional (32 bit)\(^2\) dummy read instruction, it still results in an additional 3 bits per speculated instruction.

The number of additional bits required by boosting depends on the maximum number of branches that need to be speculated past, and how flexible the path specification should be. Assume

\(^{2}\)Clearly, there is no need to use a full 32 bit instruction for the dummy read; however, this is the method used in the published literature on safe-bit speculation.
that we pick the most flexible variant of boosting, which includes "don't-care" branch specification. That variant requires 2 bits per level. Our results suggest that, in the symmetric 4-issue case, restricting speculation so that instructions are moved past at most 2 branches is adequate. Thus boosting requires an additional 4 bits per instruction. Note that boosting needs no additional bits at the commit point.

In speculative tagging the additional bits required depend on the number of tags required. Results for symmetric 4-issue processors suggest that 3 tags will be adequate. In that case, we need an additional 2 bits per instruction. Further, we will need to commit/flush these tags. Assuming that the commit and flush operations are folded into the branches, as described above, we add an additional 6 bits per branch instruction. Assuming that conditional branches form 20% of the total instructions, and that all branches incur this 6 bit penalty, we add an average of 1.2 bits per instruction to the instruction stream, for a total overhead of around 3 bits per instruction.

Exception State

The safe-bit approach potentially needs to save exception information for every register. Thus, there needs to be one exception vector for per register. Most processors today have 64 registers (32 integer and 32 floating-point). Thus, safe-bit requires at least 64 exception vectors.

Boosting requires one exception vector per path. Assuming the 2 level boosting described above, 4 exception vectors are needed.

Tagging requires an exception vector per tag. Assuming 3 tags, for the reasons described above, 3 exception vectors are needed.

Each exception vector holds roughly 64 bits of information. Implemented as single-ported SRAMs, this would require roughly 400 transistors per vector. Thus, the safe-bit approach will require 24000 transistors more than the other two.

Other Buffer State

The other buffer state added because of speculation that we have considered includes shadow register files and store-buffer. Note that neither is necessary, and the existence of either is orthogonal to the technique chosen to specify speculation. For instance, if there is no store-buffer, then stores cannot be speculated. This restriction is already present in the original safe-bit proposal; we could add it to either of the other two, and do away with the need for a store-buffer. Similarly, though boosting as originally proposed used shadow registers, it is possible to combine immediate register update by speculative instructions with the path-to-commit technique used by boosting to specify speculative instructions.

An store-buffer is basically a hash-table. It contains 32 (or 64) bits of data and a 32 (or 64) bit address that acts as the key. Implementing an 8-entry 32-bit store buffer using associative storage

---

3Earlier safe-bit proposals had suggested that it would be possible to save hardware by saving some of the exception information in the register; however, this introduces complications. For instance, extra data-paths will be required, from the exception producing "unit" to the register file. This in turn requires either more ports in the register file or additional arbitration logic. In light of these complications, we shall assume that exception vectors are stored in locations distinct from the register file.
would require roughly 10000 transistors. Note that [53] state that adding such a buffer can improve performance of the safe-bit technique by 3%–8% by permitting the speculation of stores.

As argued above, replicating the register file to create shadow register files is not as beneficial as simply doubling the number of registers. However, if one chose to do so (perhaps for reasons related to exception recovery), each copy of a register file containing 32 32-bit registers would require 7000 transistors. However, this estimate significantly underestimates the actual costs of replicating the register file, since it ignores the fact that the registers will be multi-ported. It may be off by as much as a factor of 5.

4.6 Conclusions

In this chapter, we proposed a novel mechanism for implementing static speculation, called speculative tagging. Speculative tagging combines some of the strengths of prior proposals. It provides the flexibility of the safe-bit approach, thereby enabling it to perform several types of multi-path speculation not expressible in boosting. It allows speculative instructions to be grouped, which minimizes the number of commit points in the program, thereby allowing exceptions to be implemented with fewer and smaller restart blocks.

Grouping has other advantages, which stem from the fact that instructions with the same tag belong to the same group. Two uses of this were shown while explaining paths; first, instead of having to specify all speculative stores visible to a speculative load, we simply specified the tags. second, instead of defining a path for each load, we specified a path per tag.

We investigated the problems associated with multi-path speculation. We showed how to specify multi-path speculation using speculative tagging. Finally, we showed how to modify speculative memory operations, so as to support multi-path speculation of memory operations.

Lastly, we considered implementing precise exceptions. We showed how our scheme could be modified to somewhat reduce the register pressure caused by the requirements of implementing precise exceptions on a statically speculative architecture. We showed how to modify our scheme to optimize restart. For instance we introduced the instruction counter that allowed speculative tagging to quickly identify the instruction in the restart block where execution should resume after exception handling. We also described a technique called lazy restart that enables us to reduce the size of the restart blocks.
Chapter 5

Scheduling with Speculative Instructions

In this chapter, we describe our scheduling algorithm, called whole-DAG scheduling. Section 5.1 provides the details of prior work from which our algorithm was developed. Section 5.2 details the whole-DAG scheduling algorithm. A more detailed description of the selection heuristic used in the algorithm is given in Section 5.3. A brief comparison with prior work is given in Section 5.4. Finally, we conclude with a summary in Section 5.5.

5.1 Background

In this section we shall discuss certain aspects of scheduling: information representation, list scheduling and selection heuristics, global code motion, and branch probabilities.

5.1.1 Information Representation

Typically, the code to be scheduled is presented to the scheduler as a directed graph, the control-flow graph (CFG) [1]. We shall briefly review some terms associated with the CFG. The nodes of a control-flow graph are basic blocks. Basic blocks are linear sequences of instructions. There is a directed edge between two basic blocks if it is possible for control flow to reach the second basic block after passing through the first basic block. A fork in the CFG occurs when a basic block has more than one successor. A join in the CFG occurs when a basic block has more than one predecessor. Fig. 5.1(b) shows the CFG for the code in Fig. 5.1(a).

Scheduling algorithms make extensive use of dependence information. One way of representing data dependences between instructions is by using def-use chains. Def-use chains connect two dependent instructions by a directed edge. Thus the def-use chains for the flow dependences in the example in Fig. 5.1 would be represented by the def-use edges shown in the dependence graph shown in Fig. 5.2¹. There should also be an edge from (1) to (7), representing the output dependence between them caused by r1.

We list some terms associated with the data dependences of a program below.

¹Note that, for clarity, we have not shown the branch instruction.
(1) r1 := ...
(2) r2 := ...
(3) br== ... L1

L0:
(4) ... := r2 ...
(5) goto L2

L1:
(6) ... := r2 ...
(7) r1 := ...

L2:
(8) ... := r1 ...

(a)

Figure 5.1: Control Flow Graph

(b)

Figure 5.2: Def-Use Chains
Dependence Path A dependence path is a path in the dependence graph, i.e. a sequence of instructions each of which is dependent on the previous instruction in the sequence.

Path Length The length of a path is the time it would take to execute the instructions in the path.

Critical Path The critical path in a basic block is the dependence path with the greatest length. It is a lower bound on the amount of time to execute the instructions in a basic block.

Critical Instruction An instruction on the critical path is known as a critical instruction.

Distance The distance between two instruction is the length of the longest path joining them.

Start The start instructions are those instructions that are not dependent on any other instruction. For engineering reasons, a dummy instruction called the start instruction is often added to the CFG, and all otherwise independent instructions are made dependent on it.

End The end instructions are those instructions that have no instruction dependent on them. Again, a dummy instruction called the end instruction is often added to the CFG, and it is made dependent on all instructions that otherwise have no other instructions dependent on them.

Early The early of an instruction is the distance between start and the instruction. It represents the earliest time after which an instruction can be issued in any execution.

Late The late of an instruction is the distance between the instruction and end. It represents the least interval possible between the execution of the instruction and the execution of the last instruction in the block.

While def-use chains represent the data dependences between instructions adequately, they ignore the effects of control dependences. Therefore, we use an alternative dependence representation technique, the dependence-flow graph (DFG) [48,28]. In the DFG, dependence edges that cross basic block boundaries are “pinned”. In particular, a data dependence edge that crosses a fork is split into several edges by the addition of a dummy node associated with the fork known as a switch. Similarly, a data dependence edge that crosses a join is split into several edges by the addition of a the merge node, which is associated with the join. Further, some of these newly created edges are combined. Consider the flow dependences for register r2 in the example of Fig. 5.1. In the DFG representation, the dependence edge between (1) and (4) is broken into two edges: between (1) and a switch, and between the switch and (4). The dependence between (1) and (6) is broken up similarly. Finally, the same edge (and switch) can be used for the (1) to switch component of both dependences. Thus, the two dependences are represented in the DFG by 3 dependence edges: (1) to switch, switch to (4) and switch to (6).

Note that there is no difference between representing just data dependences and using a DFG representation as long as the dependences are confined to a single basic block.

A detailed discussion of the properties, construction and uses of DFGs can be found in [27]; for our purposes it is enough to note that no dependence in the DFG can cross a basic block boundary without also passing through a switch or merge node, and we can discover the required control dependence information by examining these switch/merge nodes.
Our code typically computes properties of the program by setting up and solving data-flow equations on the DFG. Since the implementation relies fairly heavily on the properties of DFGs, we shall defer all such details to Appendix B. In this chapter, we shall attempt to explain the algorithms and heuristics used without reference to DFGs.

5.1.2 List Scheduling

In this section, we shall describe a technique used for scheduling within single basic blocks, known as list-scheduling. It forms the basis for several global scheduling techniques, including ours.

List-scheduling operates on a cycle-by-cycle basis. At each cycle, up to $N$ instructions are scheduled, where $N$ is the issue width of the processor. For correctness, all instructions that an instruction depends on must already have been scheduled before the instruction is itself chosen for scheduling.

Further, for optimal performance, the instruction(s) scheduled should be selected so as to minimize the likelihood of a stall. For instance, the processor will stall if it issues an instruction with result latency $l$ and then encounters an instruction which uses that instruction's result before $l$ cycles have elapsed. To minimize the chances of this happening, the scheduler will start considering instructions that use the result of a $l$ latency cycle for scheduling only after $l$ (scheduler) cycles have elapsed.

These observations form the basis for the ready-list. The ready-list at any cycle is a list of those unscheduled instructions that are ready at that cycle, where an instruction is considered ready if it satisfies both of the following conditions:
Figure 5.4: List Scheduling

- All instructions on which it is dependent have been scheduled.
- If it uses the result of an instruction with latency \( l \), then that instruction must have been scheduled \( l \) cycles (or more) earlier.

At each cycle, the scheduler schedules upto \( N \) instructions from the ready-list, and then increments the cycle. As a consequence of this, additional instructions may be added to the ready list.

This process is illustrated by the example in Fig. 5.4, in which the target processor is a single issue pipelined processor with a multiply result latency of 6. Initially, at cycle 1, the ready-list consists of those instructions which are independent of all instructions in the basic block, namely (1) and (2). Out of these, the scheduler picks one for scheduling, say (1). It then considers cycle 2. At this cycle, the only ready instruction is (2), which it then schedules. At cycle 3, (3) becomes ready: it uses the result of single latency operation (2), which was scheduled in the previous cycle. This is the only instruction on the ready list, and is scheduled. At cycles 4–6, the ready-list is empty, and no instruction can be scheduled. Finally, at cycle 7, (4) becomes ready, and is scheduled.

### 5.1.3 Selection Heuristics

While list scheduling, there will be some cycles in which the number of instructions that are ready exceeds the number of instructions that can be issued. Some heuristic has to be used to choose between the available instructions.

Clearly, the time taken to execute the instructions in a basic block can be no shorter than the length of the critical path. If this bound is to be met, critical instructions should be scheduled as
soon as they become ready. Otherwise, if a critical instruction is scheduled later, then executing the instructions as scheduled will necessarily take longer than the critical path length. Thus, critical instructions must be scheduled before non-critical instructions.

One can extend this idea further. An instruction cannot be scheduled before its early time without stalling the processor, and thus lengthening the time to execute the program. Similarly, an instruction cannot be scheduled less than late cycles away from the end. Thus, an instruction cannot be scheduled any later than critical path length – late without impairing performance. However, it can be scheduled in any cycle within these bounds. This is the slack of the instruction.

\[
\text{slack} = \text{critical path length} - \text{late} - \text{early}
\]

Not surprisingly, the slack of a critical instruction is 0.

In Fig. 5.4, the critical path was (1), (3), with a length of 7. (2) was independent of all other instructions, so its early was 0. (4), with a latency of 1, depends on it, so (2) had a late of 1. Thus, (2) could be scheduled at any cycle between 1 and 6 (inclusive) without hurting performance; consequently, it has a slack of 6.

Now, an instruction with greater slack has more cycles in which it can be scheduled without hurting performance. If a conflict for resources arises, it is better to allocate the resource to the instruction with less freedom. The instruction with more freedom will have more chances to find a cycle in which there is no conflict for resources. Thus, a scheduling heuristic that can be used is, when selecting between instructions, prefer those with less slack. For instance, in the example, (1) had a slack of 0, while (2) had a slack of 6. Thus, we scheduled (1) first. This still left 5 other cycles in which (2) could be scheduled without impairing performance. As it turned out, it was simple to find another cycle in which to schedule (2).

The slack selection heuristic can be greatly simplified when using list scheduling. List scheduling schedules cycle-by-cycle. In particular, when it is attempting to schedule an instruction, it does not matter what the instruction’s early was— the instruction cannot be scheduled any earlier than the current cycle. Thus, a more appropriate value for slack would be

\[
\text{slack} = \text{critical path length} - \text{late} - \text{current cycle}
\]

Of these terms, only late is instruction dependent. In particular, choosing the instruction with minimum slack is equivalent to choosing the instruction with maximum late. This simplified heuristic of choosing the instruction with maximum late is commonly used in list scheduling.

### 5.1.4 Global Code Motion

The scheduler is typically given a region of code. It rearranges the instructions within this region, so as to decrease the amount of time taken to execute the region. When the region is a basic block, the scheduler simply reshuffles the instructions within the basic block. However, if the region contains several basic blocks, then instructions may be moved between basic blocks. This is known as global code motion.

Global code motions can be divided into two categories: safe and unsafe. The safe code motions are illustrated in Fig. 5.5. Briefly, they include:
Movement down past fork (Fig. 5.5(a)): When an instruction is moved down past a fork, it is replicated and added to each destination basic block of the fork.

Movement up past join (Fig. 5.5(b)): When an instruction is moved up past a join, it is replicated and added to each source basic block of the join.

The unsafe code motions (Fig. 5.6) are those that add instructions to paths on which they would otherwise have not been present. If one of these instruction caused an exception, then the program would incorrectly have caused an error. Hence, some mechanism must be present to prevent this from occurring. The unsafe code motions, and possible hardware mechanisms that allow the compiler to safely perform these motions, are listed below:

Movement up past fork (Fig. 5.6(a)): To allow an instruction to be moved up past a fork, some support for speculative execution is required. The approach we are studying involves converting the instruction being moved into a speculative instruction.

Movement down past join (Fig. 5.6(b)): Predicated execution [47] allows the movement of instructions down past joins. It uses predicated instructions, i.e. instructions with a third input known as the predicate. Predicated instructions are executed only if the predicate is true. To safely move an instruction down past a join, it is converted into a predicated instruction with an appropriate predicate input, as shown in Fig. 5.7.

Our work ignores the possibility of movement down past joins. We shall use the terms movement up and down to indicate safe code motions up past joins and down past forks. We will use speculative movement to denote movement up past forks.
(1) ifnot r1, L0
(2) r3 := r5 + r7
L0:

(a) Up Past Fork

(b) Down Past Join

Figure 5.6: Unsafe Code Motions

(1) ifnot r1, L0
(2) r3 := r5 + r7 if r1
L0:

(a) Predication

(b)
5.1.5 Probabilities in Scheduling

Certain regions of code get executed more often than others; consequently, the effect of reducing their execution time on overall execution time is much greater. In fact, in certain cases it may be beneficial to increase the execution time of some region, if it decreases that of some other region, since the execution time of the program as a whole may still go down. Such considerations play a major role in most global scheduling algorithms.

The relative execution frequencies of various regions can be derived from the branch probabilities, i.e. the probability that a branch will be taken/not taken. Typically, the method used to identify branch probabilities is profiling. In profiling, an instrumented, unoptimized, version of the program to be compiled is executed using representative sample data. For each branch, the number of times it was taken and not taken is recorded. These numbers are used to estimate the branch probability. Then the program is compiled and optimized using these probabilities.

Of course this approach may not always be acceptable; there may be no representative sample inputs, the user may not wish to compile/execute the program twice etc. In this case, the compiler will need to estimate the probabilities from the structure of the program. Recent work has shown that the compiler can do this with reasonable accuracy [4].

5.2 Whole-DAG Scheduling

Whole-DAG scheduling extends list scheduling to general acyclic CFGs. Such regions form directed acyclic graphs or DAGs, hence the name. The extensions made by whole-DAG scheduling make it possible to use modified versions of heuristics such as late as selection criteria. Further, the extensions make it easy to control both inter-block code motion and the amount of speculation one can represent or is willing to allow.

5.2.1 Frontiers

Some global scheduling algorithms allow the movement of branches past branches, and other transforms that change the control structure of the program being scheduled. We, however, do not allow any such code motions, and shall preserve the shape of the original graph. Thus, we know the shape of the final DAG a priori—it is the same as that of the input DAG.

This suggests a scheduling strategy similar to that of list scheduling: we start with the first basic block, and start scheduling instructions in it, in a cycle-by-cycle fashion. When the block becomes “full”, i.e., when we decide not to schedule any more instructions in it, we start scheduling instructions in its successor basic blocks.

This set of blocks in which we are scheduling instructions at any one time is called a frontier, and the basic blocks in it are known as frontier blocks. The frontier blocks always constitute a node-cut of the DAG. Initially, the frontier consists of the entry nodes in the DAG. When all predecessors of a block become “full”, they are deleted from the frontier, and the block is added to the frontier. Thus, at any point the status of the DAG is as shown in Fig. 5.8—the frontier blocks
are partially scheduled, the blocks above the frontier have been completely scheduled, while the blocks below the frontier are unscheduled.

The top-level structure of whole-DAG scheduling is summarized in Fig. 5.9. As should be evident, this algorithm schedules instructions cycle-by-cycle. In each cycle, it schedules instructions in each frontier block. The instructions to be scheduled in a frontier block are chosen from a ready list unique to that block. Thus, conceptually, there is a ready list per frontier block. Note that the ready list may contain instructions from descendant basic blocks; further, such an instruction could belong to several ready lists. If an instruction from a descendant basic block is chosen, it must be moved into the current basic block. This may involve replicating the instruction, or conversion into a speculative instruction, or both. Note also that the ready list will never contain branches; they are handled separately.

5.2.2 Readiness and Ready Lists

Clearly, the ready lists used in whole-DAG scheduling differ from those used in list-scheduling in several ways. To reconcile these differences, we must answer the following questions:

- When is an instruction ready? An instruction's inputs may be available on some path through the graph, but not all.
- How are the ready lists for each frontier blocks maintained?
- How can we identify and use instructions that are ready only on some paths through the graph?

We shall consider these, and other questions, in this section.
frontier = entry blocks of DAG
cycle = 1
while( frontier is not empty ) {
    for( each block in frontier ) {
        if( block is not full ) {
            for( i = 0 to issue width of processor )
                from ready list for block
                select an instruction
                move selected instruction into block
                schedule selected instruction
        }
    }
    add/delete frontier blocks
    cycle = cycle + 1
}

Figure 5.9: Whole-DAG Scheduling: Top-Level Structure

Multi-Path Ready

Consider Fig. 5.10. If program execution was such that control entered the basic block on the right side of the join, then the multiply instruction that defines r2 would, presumably, be executed at cycle 3. Assuming that the multiply has a latency of 12, any instruction that uses r2 cannot be scheduled till cycle 15. Thus, (3) will not be issued till cycle 15. However, if control entered from the left side, (3) could be issued at cycle 6. So, when should we consider (3) ready? We can pick cycle 6, cycle 15, or some cycle in between.

As was seen earlier, we use readiness as a way of determining when an instruction can be scheduled so that it would not cause any stalls. While considering the instruction ready prior to

5 -> 6
(1) r2 := r1 + 3
(2) r2 := r1 * 9
3 -> 15
(3) ... := r2 + 5
?

Figure 5.10: Computing Ready
cycle 15 might not cause a stall if control entered from the left side of the join, it would certainly stall the processor when control enters from the right side. Clearly, we must choose the maximum value—in this case, cycle 15.

Between cycle 6 and cycle 15, the instruction is partially ready, i.e. ready on at least one control-flow path. We can take advantage of such partial readiness. Details of this are deferred till a later section.

The Ready List & Code Motion

The ready list of a frontier block, as mentioned earlier, can contain ready instructions from different basic blocks. In the most general case, the ready list will contain all ready instructions from all basic blocks that are descendants of the frontier block\(^2\). However, there is no reason to choose all descendants.

By restricting the descendants from which we choose ready instructions we can control the kinds and amount of code-motions allowed. For instance, if we choose to select a block that is connected to the frontier block via a join, then we may select an instruction from that block for scheduling. In that case, the instruction would have to be moved up past the join, as shown in Fig. 5.11(a). Thus, by choosing a descendant that is below a join, we have implicitly allowed code motion up past joins. Similarly, if we select a block below a fork, as shown in Fig. 5.11(b), we implicitly allow speculation. If we select blocks below both sides of a fork, we are allowing multi-path speculation. It is possible to first select an instruction from the block below one side of the fork, and then an instruction from the block on the other side of the fork, resulting in speculation past both sides of a branch.

A systematic policy for choosing descendants of frontier blocks so that their ready instructions can be considered for the ready list of the frontier block is called a code-motion strategy or, simply, a strategy. Possible strategies and their consequences on code motion are:

- Choose only descendants that are past joins—no speculation

\(^2\)It is incorrect to put an instruction from a non-descendant block into the ready list for the frontier block. Such an instruction can never be safely moved into the frontier block. Further, such a code motion would be of no practical value.
• Choose only descendants that along the most probable path to end—analogous to trace scheduling, results in single path speculation.

The choice of strategy shall turn out to be important in determining the performance of the scheduled code.

Maintaining the Ready List

Instead of maintaining a ready-list for each frontier block, where some instruction may be present in multiple ready-lists at any one time, we maintain distributed per-block ready-lists. These per-block ready-lists can be combined to provide the ready-list for a frontier block.

Every block has an owned-list. Initially, it contains the instructions in the block in the original DAG. As instructions are moved around in the DAG while scheduling, instructions may be added to/deleted from the owned-list of a block. The per-block ready-list of a block is simply those instructions in the owned-list for the block that are ready.

Clearly, the ready-list of a frontier block can be found by picking a set of descendant basic blocks according to the strategy being used, and then combining the per-block ready-lists for those blocks.

Partial Readiness

Partially ready instructions can be scheduled in the basic block(s) in which they are ready, but not in all basic blocks. A simple way of allowing the mechanisms described above to exploit a partially ready instruction is to move it up past the join. By doing so, the instruction will get replicated. Some of these copies will become ready, and will be handled appropriately by the scheduling mechanisms described above.

We still have to identify the partially ready instructions. To do so, we shall make use of late\textsuperscript{3} and the following observation: if an instruction is dependent on another instruction, then it’s late is less than that of the other instruction’s late. Consequently, if an instructions late value is greater than that of all unscheduled instructions in a (frontier) block, it is not dependent on any of those instructions, and is ready in that block. Actually, it is a little more complicated than this: there may be some $l$ cycle latency instruction that was scheduled less than $l$ cycles ago, on which the instruction could be dependent. Thus, for accuracy, we have to consider the late values of all instructions that are unscheduled, or have been not yet “completed”.

Based on this observation, we come up with the following technique to handle partial readiness. Basically, for each frontier block we find the maximum late value of the unscheduled and “incomplete” instructions. We select the minimum of these maximum late values. All instructions with late values greater than this get moved up past a join (if they are below a join).

---

\textsuperscript{3}Actually, the late used in whole-DAG scheduling is somewhat different from that used in list-scheduling, and is described later. However, the observation still holds for the modified late.
5.2.3 Miscellanea

Implementing Code Motion

Upward code motion of an instruction from a basic block to a frontier block is implemented as a series of moves along a path joining the frontier block to the basic block. At each step, the instruction is moved from its current block to the next basic block in the path, till it eventually reaches the frontier block. In the process, the instruction may be replicated or made speculative. This process is shown in Fig. 5.12.

Note that by moving this instruction in a step-wise fashion, we may have added redundant copies—i.e. copies of the instruction that recompute the value, even though it is available. After moving an instruction, we execute a total availability algorithm to eliminate these copies. Note that the algorithm is fairly restricted—it computes availability only for the instruction that was moved, and only for the region between the frontier block and the original basic block for the instruction. The result of this computation for our example can be seen in Fig. 5.13

Moving Below Forks

So far we have only considered upward movement of instructions, i.e. speculation and movement up past joins. We have not considered movement down past switches. Downward movement of code in whole-DAG scheduling is caused by the scheduling of a branch. When a branch in a basic block is scheduled, all unscheduled instructions in the blocks “owned-list” are moved down past the fork, and copied into the block’s immediate successors.

We have a choice as to when to schedule a branch. In our algorithm, we have chosen to schedule a branch as soon as possible, i.e. as soon as it becomes ready. This may cause code expansion, without necessarily improving performance. If this expansion turns out to be a problem, it may be desirable to choose a different policy.

5.2.4 Summary

A more detailed break-down of the whole-DAG scheduling algorithm is shown in Fig. 5.14. Note the use of the following policies:

- A frontier block is considered full when all instructions on its “owned-list” have been scheduled.

- A branch at the end of a frontier block is scheduled as soon as it becomes ready.

There are still two policies that have not been discussed. One, the selection heuristic, shall be discussed in detail in the next section. The other, the choice of code motion strategy, is a subject of study in the next chapter. As shall be shown, by varying the strategy, we can obtain different performance results.
Figure 5.12: Implementing Code Motion
5.3 Selecting Instructions

The selection heuristics we shall use shall be based on a extension of \textit{late} for DAGs, and on branch probabilities. We shall first describe how we compute the extended version of \textit{late} and then describe the heuristics.

These heuristics are aimed at improving the average performance of the DAG—i.e., improve the expected time to execute the DAG, based on the given branch probabilities. This means that, at times, we shall degrade the performance along a less likely path so as to improve the performance along a more likely path.

5.3.1 Computing \textit{Late}

Multi-Path Late & Early

\textit{late}, in list scheduling, is a measure of the distance to the end of the basic block. Extending this idea to a path in the CFG is simple: \textit{late} is the distance to the end of the path. In a more general DAG, however, there can be multiple paths to the end. One solution to computing \textit{late} for an instruction is to enumerate all paths through the DAG containing the instruction, compute the \textit{late} for each path and somehow combine them. We could use several methods to combine the \textit{late} values; one appealing method is to average the \textit{late} value for the paths, weighted by the relative probabilities of each path. This gives greater weight to the more likely \textit{late} values, but does not ignore the effect of the \textit{late} values on less likely paths. However, this is impractical; the number of paths through a DAG is exponential in the size of the DAG.

We can approximate this by a different approach, shown in Fig. 5.15. We average \textit{late} values at each fork, weighted by the branch probabilities at that fork. This averaged \textit{late} value is then used...
frontier = entry blocks of DAG

cycle = 1

while (frontier is not empty) {
    handle partial readiness
    for (each block in frontier) {
        if (block has ready branch) {
            schedule branch
            move instructions below fork
        }
        else if (block has unscheduled owned instruction) {
            /* not full */
            choose descendants according to strategy
            build ready list by combining per-block ready-lists of descendants
            for (i = 0 to issue width of processor) {
                from ready list for block
                select an instruction
                move selected instruction into block
                schedule selected instruction
            }
        }
    }
}

add/delete frontier blocks

cycle = cycle + 1

Figure 5.14: Whole-DAG Scheduling: Detailed Breakdown
to compute the late value of other instructions. Thus, (1) has a late of 2 on the 25% probability branch, and a late of 13 on the 75% probability branch. Its late is therefore 10.25.

We compute multi-path early similarly. In this case, we combine at joins, averaged by the relative probabilities of the blocks preceding the join.

Late & Branches

When computing late, instructions that have no other instruction depending on them are given late value of 0, indicating that they can be scheduled as close to the end as desired. This causes a problem with branches. A branch has no output; thus no instruction is data dependent on a branch. Consequently, branches have a late value of 0, and the late for the instructions that compute the inputs to the branch are also close to 0. This results in these instructions, and therefore the branch, getting scheduled late, resulting in a sub-optimal schedule.

To improve the quality of the schedule, it was found necessary to give more importance to scheduling branches, by initializing their late value to something other than 0. The way we compute this initial value is to compute the early for all instructions in the graph. The late values of branches is set to the difference between the largest early in the DAG and the early for the branch.

5.3.2 Selection Heuristics

The selection heuristics determine, in the case where two instructions are competing to be scheduled, which of the two instructions should be preferred. These heuristics can be broken into three cases:

1. The two instructions are in the same basic block.
2. There is a path containing both instructions.

3. There is no path containing both instructions.

Case I: Same Basic Block

If the two instructions come from the same basic block, as shown in Fig. 5.16, it suffices to compare their late values. Of the two instructions, prefer the one with the larger late. In the example, the numbers in parentheses indicate the late for that instruction. Thus, we would prefer J over I.

Case II: Common Path

When the two instructions have a common path between them, but come from different basic blocks, as shown in Fig. 5.17, it is not enough to compare the late values; the relative probabilities of execution of the two instructions must be taken into account. Consider the case shown in the example; even though J has a larger late, it is only executed 1/100 as often as I is. Thus, 99/100 of the time, scheduling J instead of I would be useless. So, instead of comparing just the late values, we compare:

\[ \text{late} \times \text{probability of execution} \]

In the example, if probability of I being executed is 0.2, then the probability of J being executed is 0.002. The products of late and probability are 4.0 and 0.08 respectively, indicating that I is to be preferred.
Case III: No Common Path

If there is no path through both instructions, as shown in Fig. 5.18, the situation becomes even more complicated. In fact, we have to go back to the original idea of slackness to come up with a heuristic for selecting between instructions. The reason should be apparent from the example. \( J \) has a larger \textit{late} than \( I \). However, that block also contains an unscheduled instruction, \( M \) with \textit{late} 80. Thus, \( J \) is not on the critical path; indeed it has a slack of at least 40. \( I \), on the other hand, has a slack of 0. Clearly, \( I \) is more “critical”, and should be preferred over \( J \).

The reason that \textit{late} cannot be used to estimate slack is that there are multiple paths to end, one through the block containing \( I \) and the other through the block containing \( J \), and these paths have different critical path lengths. To estimate the critical path lengths, we examine instructions in the respective blocks, and identify the instruction with the largest \textit{late}. We use this value as an estimate the critical path length. Of course, we must also account for probability. Thus the metric which we use when selecting an instruction is:

\[
(late - \text{max late}) \times \text{probability of execution}
\]

We pick the instruction with the largest such value. Note that this value will be non-positive.

In the example, \( J \) has value \((40 - 80) \times 0.5 = -20\), and \( I \) has value \((20 - 0) \times 0.5 = 0\). Thus, we will prefer to schedule \( I \) over \( J \).

5.4 Comparison with Prior Work

Our approach improves over prior global scheduling algorithms in several ways. The earliest such global algorithms, such as trace scheduling [15] and the boosting optimizer [57], scheduled only a single path at a time. This could result in the performance of that path being optimized at the cost of the performance of the DAG as a whole.

Another difference between our approach and other approaches is in the introduction of code motion strategies. In previously proposed algorithms, either all ready instructions were considered for scheduling, as in selective scheduling [44], or some fairly restricted region was considered, such as speculation past exactly one fork [6]. They did not have the ability to vary the amount of speculation possible. It shall be seen in the next chapter that the flexibility to choose different strategies is important, and that the best strategy to choose is not obvious.
The most prominent difference between our work and prior work lies in our selection heuristics. Typical heuristics [44,6] tend to schedule all non-speculative instructions first. Only after that do they consider scheduling a speculative instruction. Between two speculative instructions, the more probable instruction is selected. late for an instruction was computed is the end of its basic block. It is used only to break ties between instructions from the same basic block.

We, on the other hand, make our selections based on both a global late and probability. No one factor completes dominates the selection process. Thus, our approach can result in situations where a speculative instruction is scheduled in preference to a non-speculative instruction. As shall be shown in the next chapter, factors like this make our heuristic superior to the one described in the previous paragraph.

5.5 Summary

We have presented a scheduling algorithm known as whole-DAG scheduling that schedules an acyclic region of code, i.e. a DAG. It is derived from list-scheduling, and uses many of the same concepts and heuristics, extended to DAGs. The extensions make fairly extensive use of probability information to generate a schedule that optimizes the expected execution time of the DAG.
Chapter 6

Results

In this chapter, we will describe the results of various experiments we performed to measure the potential benefits of static speculation, and to gain some insight into appropriate methods of scheduling for static speculation.

We start off in Section 6.1 by describing the foundations for our experiments—the compiler, the benchmarks etc. The next three sections describe the inferences drawn from the results of the experiments. Section 6.2 examines the benefits to be gained from static speculation. Section 6.3 compares several variations in the scheduling algorithm, and shows which are better. This section also includes more details on the experiments performed. In Section 6.4, we use the performance information to indicate a good choice for the number of speculative tags an architecture should have. Finally, we summarize our results in Section 6.5.

6.1 Preliminaries

In this section, we describe the foundations on which our experiments are based. This includes a description of the compiler on which we implemented our scheduler, the benchmark programs we measured and the methods we used for measuring and reporting performance.

6.1.1 The Pidgin Compiler

Our scheduling algorithm is integrated into the Pidgin compiler, which was implemented as part of the Typhoon project. The compiler is designed to be re-targetable, so that one should be able to change either the language being compiled or the machine being targeted fairly simply. This is achieved by dividing the compiler into the following 3 modules:

Front-end Reads the high-level language input program and translates it into a compiler intermediate representation.

Optimizer Reads the compiler intermediate representation of a program and performs various optimizations on it. It outputs the optimized version of the program in a compiler intermediate representation, which may be the same as the input representation.
Back-end Read the output of the optimization phase and translates it into a form suitable for execution on the target architecture.

In theory, all that needs to be done to compile a new language is to write a new front-end for that language. Similarly, to compile all the existing languages that are supported for a new architecture, it should be sufficient to write a new back-end.

The modules we shall be using are shown in Fig. 6.1. As shown, the front-end translates FORTRAN to TIL, our compiler intermediate form. The optimizer reads the TIL representation of the program, performs the standard set of optimizations [1] on it, and outputs the optimized version in TIL. The back-end, instead of producing machine code for a particular architecture, converts the TIL into C code.

For our purposes, we modify the compiler in two ways. First, as shown in Fig. 6.2, we add an instrumentation module. This alters the C code produced by the back-end so that, on execution, it will collect various statistics, such as branch probability information. Second, as shown in Fig. 6.3, we add a scheduler that extracts and schedules DAGs from the optimized code. This TIL code is then directly simulated.
6.1.2 Measuring DAGs

Our work deals with the impact of static speculation on DAGs. So, we need to focus on the DAG code in programs. To this end, we selectively extract DAGs from the programs. Our criteria for selecting DAGs are:

1. It must be a single-entry, single-exit region [27].

2. It must not be a basic block, i.e. must contain at least one fork and one join.

3. There must be less than 1000 distinct control flow paths through it.

Criteria 1 is a consequence of our representation of control structure, which makes it easy to isolate DAGs that are single-entry single-exit regions. Criteria 2 filters out those DAGs that could not possibly benefit from the introduction of speculation. Criteria 3 is necessary to control the amount of time required to evaluate the performance of the DAG. It is not very restrictive; in particular, over the set of benchmarks we used, it caused the rejection of exactly 2 DAGs.

After isolating the DAGs in a program, we schedule them, and simulate the performance of the scheduled code. While simulating a DAG, we separately measure the time taken to execute each control flow path through it. We then compute a weighted average of the execution times for the paths, where the weight is the path probability\(^1\). This number gives the average execution time for the DAG.

To illustrate this process, consider the DAG shown in Fig. 6.4. There are two control-flow paths through the DAG, \texttt{ABD} and \texttt{ACD}, with execution times of 15 and 5 cycles respectively. From the branch probabilities, it follows that the two paths are executed 20\% and 80\% of the time respectively. Thus, the average execution time for the DAG is $15 \times .2 + 5 \times .8 = 7$ cycles.

We can obtain a lower-bound on the average execution time using a similar approach. Basically, we measure the critical path through each path in the DAG, ignoring the effect of branches. We then average the critical path lengths weighted by the path probabilities. This gives us the optimal average time to execute the DAG, i.e., the average time that it would take to execute the DAG on a machine with an infinite amount of resources which performs an infinite amount of speculation.

\(^1\)Perhaps it would be more appropriate to use the term path frequency, since the path “probability” is really the ratio of times path is executed to total number of times DAG is executed.
6.1.3 Benchmarks

The DAGs we measure are obtained from standard FORTRAN benchmark programs, including the linpack benchmark and programs from SPEC-89 and Perfect Club benchmark suites. A summary of the programs and the DAGs extracted from them is given in Table 6.1.

We chose the 4 out of the 13 programs from the Perfect Club benchmarks: APS, LGS, TFS and TIS. Of the 6 SPEC-89 FORTRAN benchmarks, 013.spice2g6 was not considered because its size prevented our compiler from processing it. 030.matrix300 was rejected because it is basically a simple, triply-nested loop, and would therefore yield no insight into the use of static speculation for DAGs.

We isolated a total of 234 DAGs from the 9 programs. Interestingly, only 183 of these are ever exercised when the programs are run using the standard input data provided with the benchmark. Control never enters the other 51 DAGs.

6.2 The Benefits of Speculation

In this section, we report the benefits to be had from using static speculation to improve the performance of DAGs. This includes measurements of the importance of DAGs in both FORTRAN and C programs, as well as estimates of the speedups to be obtained by using static speculation with our scheduling algorithm.

6.2.1 The Importance of DAGs

FORTRAN programs are notorious for spending most of their time executing inner-loop bodies. Worse yet, most of these loops are simple basic blocks. So, it would seem that a technique that improves the performance of DAGs would have little impact on the performance of the program as a whole. This appears to be borne out by the figures in Table 6.2, which measures the fraction of the
Table 6.1: Input Programs & DAGs

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>Total</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DAGs</td>
<td>Insns</td>
</tr>
<tr>
<td>APS</td>
<td>6105</td>
<td>86</td>
<td>9899</td>
</tr>
<tr>
<td>LGS</td>
<td>2389</td>
<td>24</td>
<td>2118</td>
</tr>
<tr>
<td>TFS</td>
<td>1986</td>
<td>30</td>
<td>4517</td>
</tr>
<tr>
<td>TIS</td>
<td>485</td>
<td>5</td>
<td>110</td>
</tr>
<tr>
<td>015.doduc</td>
<td>5336</td>
<td>64</td>
<td>9809</td>
</tr>
<tr>
<td>020.nasa7</td>
<td>1108</td>
<td>4</td>
<td>117</td>
</tr>
<tr>
<td>042.fpppp</td>
<td>2717</td>
<td>13</td>
<td>771</td>
</tr>
<tr>
<td>047.tomcatv</td>
<td>195</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>linpack</td>
<td>793</td>
<td>7</td>
<td>312</td>
</tr>
<tr>
<td>total</td>
<td>21114</td>
<td>234</td>
<td>27701</td>
</tr>
</tbody>
</table>

dynamically executed instructions that are in DAGs. Overall this number is 4% of all instructions executed. As we shall show in the next section, adding speculation improves the performance of DAGs by 13%–17%. At a first glance, it appears that the overall improvement in performance obtained by the addition of speculation in DAGs will be minor.

However, on closer examination, it becomes apparent that the data is badly skewed by 020.nasa7 and TFS. If we ignore them, 15.5% of the instructions executed by the remaining 7 programs are in DAGs. Thus, it is possible that static speculation can have a significant impact on some programs.

As we can see, from Table 6.3, which summarizes the improvement in overall performance for each function, static speculation does have a significant impact on some programs. The benchmarks, i.e., LGS, 015.doduc, and 047.tomcatv show a substantial improvement in performance. The overall improvement, averaged over all programs, is only around 1%. However, if we again ignore the effects of TFS and 020.nasa7, the average improvement for the remaining 7 programs is around 4%.

6.2.2 Non-numerical Programs

Our experiments have been done using numerical FORTRAN programs. It is believed that non-numerical C programs would have more DAGs than the programs we have considered, and thus static speculation would be more beneficial to such codes. Unfortunately, we did not have access to a C compiler. However, there exists an incomplete C front-end to the Pidgin compiler. By using it, we were able to estimate the fraction of the dynamic "instructions" executed that occur in DAGs for 3 of the 4 SPEC-89 C benchmarks. We could not process 001.gcc1.35, since it uses setjmp() / longjmp(). The numbers for the other 3 are summarized in Table 6.4.

Note that the “instructions” measured here are intermediate between a statement and a RISC instruction. In particular, they have not been converted to a form suitable for execution on a
### Table 6.2: Dynamic Execution Frequency

<table>
<thead>
<tr>
<th>Program</th>
<th>Total Insns (in millions)</th>
<th>DAG Insns (in millions)</th>
<th>%age</th>
</tr>
</thead>
<tbody>
<tr>
<td>APS</td>
<td>5905.2</td>
<td>701.7</td>
<td>11.9%</td>
</tr>
<tr>
<td>LGS</td>
<td>2482.4</td>
<td>527.8</td>
<td>21.3%</td>
</tr>
<tr>
<td>TFS</td>
<td>9673.2</td>
<td>16.5</td>
<td>0.2%</td>
</tr>
<tr>
<td>TIS</td>
<td>2.0</td>
<td>1.6</td>
<td>80.0%</td>
</tr>
<tr>
<td>015.doduc</td>
<td>1290.8</td>
<td>474.8</td>
<td>36.8%</td>
</tr>
<tr>
<td>020.nasa7</td>
<td>24302.1</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td>042.fpppp</td>
<td>143.0</td>
<td>3.0</td>
<td>2.1%</td>
</tr>
<tr>
<td>047.tomcatv</td>
<td>2273.4</td>
<td>195.5</td>
<td>8.6%</td>
</tr>
<tr>
<td>linpack</td>
<td>242.4</td>
<td>13.4</td>
<td>5.5%</td>
</tr>
<tr>
<td>total</td>
<td>46214.3</td>
<td>1934.3</td>
<td>4.2%</td>
</tr>
</tbody>
</table>

### Table 6.3: Overall Improvement

<table>
<thead>
<tr>
<th>Program</th>
<th>%age Improvement (4 issue)(^a)</th>
<th>%age Improvement (8 issue)(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APS</td>
<td>0.3</td>
<td>1.1</td>
</tr>
<tr>
<td>LGS</td>
<td>12.4</td>
<td>12.6</td>
</tr>
<tr>
<td>TFS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TIS</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>015.doduc</td>
<td>4.5</td>
<td>7.4</td>
</tr>
<tr>
<td>020.nasa7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>042.fpppp</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td>047.tomcatv</td>
<td>4.3</td>
<td>4.9</td>
</tr>
<tr>
<td>linpack</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>average(^c)</td>
<td>1.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>

\(^a\) profiled, best-5 over none  
\(^b\) profiled, best-7 over none  
\(^c\) Weighted by total instructions (see Table 6.2)
Table 6.4: Dynamic Frequency of DAGs in C Code

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Dags</th>
<th>DAG “Insns” (in millions)</th>
<th>Total “Insns” (in millions)</th>
<th>%age</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>240</td>
<td>343.1</td>
<td>1332.2</td>
<td>25.8</td>
</tr>
<tr>
<td>022.li</td>
<td>55</td>
<td>1155.3</td>
<td>5348.8</td>
<td>21.6</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>30</td>
<td>498.8</td>
<td>819.9</td>
<td>60.8</td>
</tr>
<tr>
<td>total</td>
<td>325</td>
<td>1997.2</td>
<td>7500.9</td>
<td>26.2</td>
</tr>
</tbody>
</table>

load-store architecture. They are in a form akin to CISC instructions. However, the numbers still suggest that C programs spend far more of their time executing instructions from DAGs—an average of 26.2%, or around double that for the top 7 FORTRAN programs.

From the numbers shown above and to be shown later, it appears that static speculation can improve the performance of DAG code by around 20%. Assuming that DAGs constitute 25% of all code executed by C programs, and assuming that this ratio holds good for C programs, static speculation could improve performance by around 5%.

Interestingly, [6] actually measured an improvement of 4.9% for 0.22i by adding static speculation to the RS/6000, and scheduling for it. The other programs showed little (less than 0.5%) improvement. Note that their scheduling technique constrains speculation to movement past at most one branch, and that the RS/6000 would, at best, correspond to a 2-issue architecture. Clearly, a technique that performed more speculation targeted for an architecture with more resources would show larger improvements.

6.2.3 Dynamic Speculation

Our measurements suggest that the introduction of static speculation can produce modest improvements in overall performance, of the order of a few percents. Studies of dynamic speculation, however, appear to indicate that adding dynamic speculation can improve performance by an order of magnitude more. For instance, [29] shows an improvement 36% by adding dynamic speculation.

The reason for this discrepancy is simple: dynamic speculation allows instructions to be speculated past loop back-edges as well as past branches in acyclic code. When instructions are speculated past a loop backedge, in effect several iterations of the loop are being executed concurrently. Thus, dynamic speculation improves the performance of loops as well as DAGs.

Similar performance improvements for loops can be obtained without speculation, by using a compilation technique known as software pipelining [24,51]. Software pipelining schedules the loop body so that instructions from different iterations of the loop are being executed concurrently, thereby increasing the utilization of resources and improving performance. Various studies of software pipelining [24,34] suggest that it can schedule 90%+ of loops so that they execute optimally.

Dynamic speculation studies typically do not use code containing loops that have been software pipelined, and thus the performance gains reported show additional improvements obtained through
the overlapping loop iterations. It would be interesting to see the performance improvement yielded by dynamic speculation on codes that have been software pipelined. It is our hypothesis that the improvements on such codes will be similar to the improvements we have shown.

6.3 The Scheduling Algorithm

In this section, we measure the performance of whole-DAG scheduling, and validate certain choices made by the algorithm. In particular, we focus on the choice of an appropriate basic-block selection strategy and the tie-breaking instruction selection heuristic. We will compare the consequences of using our techniques with the results obtained by prior work. We also measure the code expansion and register pressure caused by our scheduling algorithm.

6.3.1 The Variables

Most of the global DAG scheduling algorithms are fairly similar in that they are derived from list-scheduling. However, they differ in two major areas:

Basic-block strategy: When attempting to schedule instructions into a basic-block, the scheduling algorithm can choose instructions for the ready list from different basic blocks. The number of such basic blocks chosen and the rules for choosing them, i.e. the strategy, varies among algorithms.

Selection heuristic: When there are two or more instructions in the ready list competing to be scheduled, the algorithm must use some heuristic to break the tie. This heuristic is another point of difference among algorithms.

The basic-block strategies used can be divided up into the following:

Single-Path: Typically, these strategies make use of branch probability information. When scheduling instructions in a block, only instructions in descendant blocks that are on the most probable path to the end of the DAG are considered.

Unlimited: Here, when scheduling instructions in a block, instructions from all descendant basic blocks are considered.

Limited: Instead of examining all descendant basic blocks, only instructions from some subset are considered. For instance, [6] restrict the instructions to be those that are:

- In a basic block which is control dependence equivalent to the basic block in which instructions are being scheduled, or,
- Joined to it by at most one branch.

The selection heuristics proposed include:

Late: Use the distance to the end of the DAG, i.e. similar to our LATE.
Table 6.5: Scheduling Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Strategy</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Scheduling</td>
<td>Single-Path</td>
<td>Late</td>
</tr>
<tr>
<td>Boosting</td>
<td>Single-Path</td>
<td>Late</td>
</tr>
<tr>
<td>Superblock Scheduling</td>
<td>Single-Path</td>
<td>Late</td>
</tr>
<tr>
<td>Global Scheduling</td>
<td>Limited</td>
<td>DOS</td>
</tr>
<tr>
<td>SRDAG Scheduling</td>
<td>Unlimited^a</td>
<td>Late</td>
</tr>
<tr>
<td>Trace-2 Scheduling</td>
<td>Unlimited</td>
<td>Speculative Yield</td>
</tr>
<tr>
<td>Selective Scheduling</td>
<td>Unlimited</td>
<td>DOS</td>
</tr>
<tr>
<td>Whole-DAG Scheduling</td>
<td>Variable</td>
<td>Full</td>
</tr>
</tbody>
</table>

^aCould also be called Limited

Speculative Yield: Proposed in [16], this uses Late * probability as the tie breaking metric.

Depth of Speculation: Also known as DOS. In this heuristic, distance to end of block is used to break ties if the two instructions being compared are from the same block. Otherwise, the more probable of the two is picked.

Full: Our novel selection heuristic, as described in Chapter 5.

Table 6.5 summarizes (our interpretation of) the basic-block selection strategy and the instruction selection heuristic adopted by the various scheduling algorithms introduced in Chapter 2, including those used by our algorithm.

Note that we have, as yet, not picked a particular basic-block selection strategy to pick. In the Section 6.3.3, we shall investigate the behavior of various strategies under different scenarios. Further, in Section 6.3.4, we shall compare our instruction selection heuristic with those adopted by previous proposals.

6.3.2 The Scenarios

Target Machine Model

We are compiling for a load-store architecture. All operations are assumed to be fully-pipelined, with an issue latency of 1. The result latencies of those instructions with latencies greater than 1 are listed in Table 6.6.

We shall not be varying the latencies of the operations during the course of our experiments. Instead, we vary the number of functional units and issue-widths. We shall consider 2 machines, which are capable of any 4 and any 8 instructions per cycle respectively. These counts do not include the number of branches; the machines are assumed to be capable of executing any number of branches concurrently.
Table 6.6: Result Latencies

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>2</td>
</tr>
<tr>
<td>fp-add, fp-sub</td>
<td>4</td>
</tr>
<tr>
<td>fp-mul</td>
<td>5</td>
</tr>
<tr>
<td>int-mul</td>
<td>6</td>
</tr>
<tr>
<td>fp-div</td>
<td>11</td>
</tr>
<tr>
<td>int-div</td>
<td>12</td>
</tr>
</tbody>
</table>

Branch Probabilities

Branch probability information is used both while scheduling and for simulation. One way of obtaining these probabilities is to instrument each benchmark program so that it will record the number of times each branch direction is taken, and then execute the program. We derive the branch probabilities from the measured branch direction frequencies. We shall refer to these branch probabilities as *profiled* probabilities.

The branches in numerical FORTRAN benchmarks are notorious for being skewed, i.e., for taking one direction almost exclusively. Actual measurements, shown in Fig. 6.5, confirm this belief. Almost 80% of the branches choose one target over the other at least 90% of the time. However, some programs, particularly non-numerical codes, are believed to exhibit less skewed branching behaviour. We would, therefore, like to investigate the behavior of various strategies on code with more balanced branches. To this end, we consider an alternate set of branch probabilities in which all branches are perfectly balanced, i.e., a branch will pick each of its targets equally often. We call these branch probabilities *balanced* probabilities.

As mentioned earlier, we had isolated 234 DAGs from the benchmarks. Of these only 183 were ever executed. Clearly, the profiled probabilities make sense only when for these 183 DAGs. Thus, when considering profiled probabilities, we shall restrict ourselves to the 183 DAGs which are executed at least once. When considering balanced probabilities, we shall consider all 234 DAGs.

Strategies

Our scheduling algorithm is flexible in the basic block scheduling strategy we can use. Instead of choosing one strategy for selecting basic blocks from which to move instructions into a frontier block, we shall investigate the effects of different strategies. In particular, we shall look at the following 9 strategies:

- **best-N** choose $N$ most-probable (connected) descendant basic-blocks, $N$ between 2 and 8.
- **mpp** choose descendant basic-blocks that are on most-probable path to end.
Figure 6.5: Distribution of Branch Probabilities
none choose only descendant basic-blocks that are not past a fork, i.e. those that are connected to the frontier block through a sequence of joins.

It is interesting to see how the strategies we have picked compare with those used by previous work. Clearly, mpp captures the behavior of single-path fairly well. Both of them will allow code motion past at most one side of a branch, the side being determined by the probability. Similarly, of the strategies described above, best-8 is the closest to the behavior of unlimited strategies. It could be that a more accurate picture of the behavior of unlimited strategies could be found by extrapolating the behavior of best-9 and upwards from the behavior of best-2 through best-8; however, it turns out not to be necessary.

The strategy used in the limited case, in particular Global Scheduling, resembles best-3. However, there is one difference: Global Scheduling will pick both blocks below a branch, in the balanced probability scenarios, so will best-3 (or best-2). However, in the case of skewed probabilities, best-3 may pick only one of the blocks below a branch, if the descendants of that block are more probable to be executed.

Summary

There are 36 different scenarios for which each DAG is scheduled and simulated. These correspond to the cross-product of the 2 machine models, the 2 branch probability models, and the 9 strategies considered.

The DAGs are scheduled assuming no constraints on speculation. In particular, this means that all instructions, including stores, can be speculated. Further, instructions speculation is not constrained by the amount of resources required to represent the speculation. Thus, the results are independent of any particular method of implementing speculation.

We have already described how we determine the average actual and optimal execution times for a DAGs. The number we report for each scenario is the sum of these average execution times for over all the DAGs being considered. Note that we could have chosen to weight these by the relative importance (i.e. execution frequency) of the DAG. This does not really make sense when using our assumed balanced probabilities. Further, since we are trying to compare the behavior of the scheduling algorithms, rather than measure the gain in performance, we have chosen to report the unweighted sums.

6.3.3 Evaluating Strategies

Optimal Performance

The optimal performance will be different for the two branch probabilities, both because they have different numbers of basic-blocks, and because the optimal performance of a DAG is path-probability weighted average for the optimal performance for each path in the DAG. The sum of the optimal performance numbers for the DAGs being considered are:

---

2Global Scheduling also picks basic-blocks with equivalent control dependence, while we, in general, do not. This prevents a direct application of the numbers found by our algorithm to Global Scheduling.
Table 6.7: Actual Performance: Balanced Probabilities

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Cycles</th>
<th>%age over Optimal</th>
<th>Cycles</th>
<th>%age over Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>best 2</td>
<td>10799</td>
<td>26.3%</td>
<td>10055</td>
<td>17.6%</td>
</tr>
<tr>
<td>best 3</td>
<td>10653</td>
<td>24.6%</td>
<td>9833</td>
<td>15.0%</td>
</tr>
<tr>
<td>best 4</td>
<td>10599</td>
<td>23.9%</td>
<td>9776</td>
<td>14.3%</td>
</tr>
<tr>
<td>best 5</td>
<td>10535</td>
<td>23.2%</td>
<td>9671</td>
<td>13.1%</td>
</tr>
<tr>
<td>best 6</td>
<td>10589</td>
<td>23.8%</td>
<td>9651</td>
<td>12.8%</td>
</tr>
<tr>
<td>best 7</td>
<td>10583</td>
<td>23.7%</td>
<td>9636</td>
<td>12.7%</td>
</tr>
<tr>
<td>best 8</td>
<td>10687</td>
<td>25.0%</td>
<td>9617</td>
<td>12.4%</td>
</tr>
<tr>
<td>mpp</td>
<td>11258</td>
<td>31.6%</td>
<td>10549</td>
<td>23.3%</td>
</tr>
<tr>
<td>none</td>
<td>11951</td>
<td>39.7%</td>
<td>11475</td>
<td>34.2%</td>
</tr>
</tbody>
</table>

Table 6.8: Actual Performance: Profiled Probabilities

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Cycles</th>
<th>%age over Optimal</th>
<th>Cycles</th>
<th>%age over Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>best 2</td>
<td>8664</td>
<td>28.7%</td>
<td>8037</td>
<td>19.4%</td>
</tr>
<tr>
<td>best 3</td>
<td>8663</td>
<td>28.7%</td>
<td>7936</td>
<td>17.9%</td>
</tr>
<tr>
<td>best 4</td>
<td>8659</td>
<td>28.6%</td>
<td>7889</td>
<td>17.2%</td>
</tr>
<tr>
<td>best 5</td>
<td>8641</td>
<td>28.4%</td>
<td>7836</td>
<td>16.4%</td>
</tr>
<tr>
<td>best 6</td>
<td>8679</td>
<td>28.9%</td>
<td>7837</td>
<td>16.4%</td>
</tr>
<tr>
<td>best 7</td>
<td>8712</td>
<td>29.4%</td>
<td>7777</td>
<td>15.5%</td>
</tr>
<tr>
<td>best 8</td>
<td>8681</td>
<td>29.0%</td>
<td>7780</td>
<td>15.6%</td>
</tr>
<tr>
<td>mpp</td>
<td>8599</td>
<td>27.8%</td>
<td>8098</td>
<td>20.3%</td>
</tr>
<tr>
<td>none</td>
<td>9555</td>
<td>42.0%</td>
<td>9190</td>
<td>36.5%</td>
</tr>
</tbody>
</table>

Profiled Probabilities: 6731 cycles.

Balanced Probabilities: 8553 cycles.

Actual Performance

The actual performance numbers for the various scenarios are shown in Table 6.7 and Table 6.8. Interestingly, even without speculation, we can achieve performance within 35%–42% of optimal. This means that the best we can hope to do is improve the performance of the DAG portion of the code by about 40%; in practice we can improve the DAG performance by an about 16%. This means that for code which spends, say 25%, of its total cycles executing DAGs, adding speculation will yeild an average performance improvement of around 4%.

Not surprisingly, as we increase the issue-width of the processor, the performance goes up.
Note that the gap between the performance of speculation and no speculation also increases, by about 6% of the optimal DAG performance. Clearly, speculation can make better use of available processor resources.

As we increase the number of blocks considered in the best-N strategy, the performance of the scheduled code begins to level-off and then decreases. For the 4-issue scenarios, performance bottoms out at best-5, while for the 8-issue, unbalanced scenarios, there is a similar peak at best-7. This occurs because the scheduler is finding and scheduling more speculative operations than the hardware can actually usefully exploit. These numbers suggest that considering an unlimited number of blocks does not necessarily improve performance; a more restricted approach is more beneficial.

Multi-path speculation strategies out-perform mpp in all cases but the one where the issue-width is 4 and the branch probabilities are skewed. Even in this case the difference is marginal—0.6%. This is the best scenario for the mpp strategy. Because the issue width is low, the processor cannot exploit the additional parallelism to be found by speculating down the less probable paths. Further, since the branches are skewed, there is less benefit to be had even when it is possible to exploit this parallelism. If either of these assumptions is changed, multi-path strategies out-perform mpp. In the most extreme case, when neither assumption holds, it is possible for a multi-path strategy to be 11% closer to optimal than mpp.

Ideally, if we are constrained to pick one strategy, the strategy to pick would be some form of limited, best-N, strategy. A good choice for an 8-issue architecture would be best-7, while for a 4-issue architecture it would be best-5. As we shall see in the Section 6.4, issues of implementation cost may cause us to restrict the amount of speculation possible. In that case, a best-3 strategy may be a good choice for the 4-issue architecture, since it yields over 99% of the performance of a best-5 strategy. It may be an equally good strategy to pick for an 8-issue architecture as well, where it delivers about 97% of the performance of the best-7 strategy.

Conclusions
The experimental numbers suggest that:

- Single-path strategies are appropriate when compiling code with skewed branch probabilities for machines with a low issue-width. However, on non-skewed code, or when compiling for architectures with large amounts of resources, another strategy should be picked.

- Unlimited strategies are more appropriate for machines with large amounts of resources. However, even then they may cause some degradation in performance.

- Limited strategies appear to do reasonably well. In particular, best-3, while not optimal, is always within 3% of the optimal.

The optimal strategy to pick appears to be intermediate between limited and unlimited. The appropriate strategy is influenced both by the target architecture and the kind of code (more specifically, its branch behavior) being compiled.
6.3.4 Evaluating Heuristics

The Variables

We have used a selection heuristic more complex than those reported in earlier work. To actually justify this complexity, we shall compare it against the following other heuristics:

LATE  Prefers instructions with larger LATE values.

Weighted LATE  Uses the product of LATE and probability of execution for an instruction as the selection heuristic. Basically, our heuristic without Case III.

Probability  Compares instructions using LATE if they are from the same basic block, otherwise chooses the instruction with the greatest probability of execution.

There are close correspondences between these heuristics, and those proposed earlier in the literature. In particular:

- LATE resembles the Late heuristic used by trace-based schedulers. However, the distance to end they use is an unweighted distance to end along the most-probable path, while we use a weighted distance along all paths.

- weighted LATE is similar to speculative yield, in that it uses the product of probability and distance to end. Again, the distance to end that they propose to use is not the weighted distance to end we use; instead they use the maximum distance to end.

- probability resembles the DOS heuristic. There are two sources of difference:
  - Instead of using probability information directly, they estimate it by counting the number of branches between the instruction and the frontier block.
  - They use either a global maximum distance to end, or use the distance to the end of the block to break ties.

Performance

The results are summarized in Table 6.9. These results have been computed for 4-way issue using balanced probabilities. The percentage is the improvement in performance as a percentage of optimal performance that is gained by using our full heuristic.

Note that our heuristic tends to outperform the others in most cases. The exception occurs in the mpp strategy. Here, the probability heuristic tends to perform a little better than ours. However, to obtain best performance, it is still necessary to use our heuristic.

6.3.5 Other Concerns

There are two major concerns about the code produced by a scheduler:

- By how much does it expand the program?

- How many registers does it require?
Table 6.9: Selection Heuristic Performance

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Full Cycles</th>
<th>late Cycles</th>
<th>%age</th>
<th>Weighted late Cycles</th>
<th>%age</th>
<th>Probability Cycles</th>
<th>%age</th>
</tr>
</thead>
<tbody>
<tr>
<td>best 2</td>
<td>10799</td>
<td>11613</td>
<td>7.5%</td>
<td>11582</td>
<td>7.3%</td>
<td>11604</td>
<td>7.5%</td>
</tr>
<tr>
<td>best 3</td>
<td>10653</td>
<td>11129</td>
<td>4.5%</td>
<td>10951</td>
<td>2.8%</td>
<td>10983</td>
<td>3.1%</td>
</tr>
<tr>
<td>best 4</td>
<td>10599</td>
<td>10982</td>
<td>3.6%</td>
<td>10760</td>
<td>1.5%</td>
<td>10848</td>
<td>2.3%</td>
</tr>
<tr>
<td>best 5</td>
<td>10535</td>
<td>10990</td>
<td>4.3%</td>
<td>10771</td>
<td>2.2%</td>
<td>10821</td>
<td>2.7%</td>
</tr>
<tr>
<td>best 6</td>
<td>10589</td>
<td>10961</td>
<td>3.5%</td>
<td>10712</td>
<td>1.2%</td>
<td>10748</td>
<td>1.5%</td>
</tr>
<tr>
<td>best 7</td>
<td>10583</td>
<td>11010</td>
<td>4.0%</td>
<td>10745</td>
<td>1.5%</td>
<td>10851</td>
<td>2.5%</td>
</tr>
<tr>
<td>best 8</td>
<td>10687</td>
<td>11066</td>
<td>3.5%</td>
<td>10801</td>
<td>1.1%</td>
<td>10822</td>
<td>1.3%</td>
</tr>
<tr>
<td>mpp</td>
<td>11258</td>
<td>11255</td>
<td>-0.0%</td>
<td>11223</td>
<td>-0.3%</td>
<td>11188</td>
<td>-0.6%</td>
</tr>
<tr>
<td>none</td>
<td>11951</td>
<td>11961</td>
<td>0.1%</td>
<td>12002</td>
<td>0.4%</td>
<td>12100</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

Code Expansion

Table 6.10 shows the code expansions for the various strategies. As can be seen the code expands by around 50–70%.

Register Pressure

A more serious concern is register pressure. Fig. 6.6 shows a histogram of the number of DAGs versus maximum register pressure using the best 5 strategy on a 4 issue implementation, and Fig. 6.7 shows the same information for the best 8 strategy on an 8 issue implementation. Both consider schedules for balanced probabilities.

It seems, from Fig. 6.6, that most DAGs can be scheduled using a small number of registers; in particular, with 40 registers, one can directly use the produced schedule for 90% of the DAGs. However, this is deceptive. The schedules for the bigger DAGs tend to use more registers. Once we weight the DAGs with their size, as shown in Fig. 6.8 and Fig. 6.9, the number of DAGs schedulable at 40 registers with the best 5 strategy on 4 processors drops to 60%.

However, perhaps a more important figure of merit is the increase in the register pressure due to speculation. This is show in Fig. 6.10 and Fig. 6.11 for 4 and 8 issue processors respectively. The base-line is the maximum register pressure for the schedule produced using the none strategy for that issue width. Note that even this strategy is fairly aggressive, and may lengthen life-times more than some other non-speculative schedule; thus, the actual increase in register pressure may be greater.

As can be seen, in some cases the register pressure can rise dramatically, with 10% of the DAGs requiring more than 40 additional registers in the 8 issue case. However, the increase in register pressure is generally more controlled. 80% of the DAGs require less than 20 additional registers.

\[^3\text{More accurately, the 60% of all instructions are in a DAG that can be scheduled using less than 40 registers.}\]
Table 6.10: Code Expansion

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Balanced</th>
<th></th>
<th></th>
<th>Profiled</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 issue</td>
<td>8 issue</td>
<td>4 issue</td>
<td>8 issue</td>
<td>4 issue</td>
<td>8 issue</td>
</tr>
<tr>
<td>best 2</td>
<td>1.45</td>
<td>1.43</td>
<td>1.47</td>
<td>1.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best 3</td>
<td>1.54</td>
<td>1.51</td>
<td>1.56</td>
<td>1.53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best 4</td>
<td>1.54</td>
<td>1.52</td>
<td>1.56</td>
<td>1.53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best 5</td>
<td>1.58</td>
<td>1.54</td>
<td>1.59</td>
<td>1.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best 6</td>
<td>1.65</td>
<td>1.61</td>
<td>1.66</td>
<td>1.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best 7</td>
<td>1.65</td>
<td>1.62</td>
<td>1.70</td>
<td>1.65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>best 8</td>
<td>1.71</td>
<td>1.65</td>
<td>1.71</td>
<td>1.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree</td>
<td>1.60</td>
<td>1.57</td>
<td>1.61</td>
<td>1.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpp</td>
<td>1.50</td>
<td>1.47</td>
<td>1.52</td>
<td>1.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>1.44</td>
<td>1.42</td>
<td>1.46</td>
<td>1.43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.6: DAG vs. Register Pressure: 4 issue, best 5
Figure 6.7: DAG vs. Register Pressure: 8 issue, best 8
Figure 6.8: Weighted DAG vs. Register Pressure: 4 issue, best 5
Figure 6.9: Weighted DAG vs. Register Pressure: 8 issue, best 8
Figure 6.10: DAG vs. Increase in Register Pressure: 4 issue, best 5 over none
Figure 6.11: DAG vs. Increase in Register Pressure: 8 issue, best 8 over none
in the 8 issue case while in a 4 issue schedule, 85% of the DAGs require less than 10 additional registers.

This increase in register pressure is not as big a concern it would be on a single-issue implementation. On such an implementation, every extra load and store operation added to support spilling can cause performance degradation by displacing an instruction that would otherwise have been issued in that slot. In a multiple-issue implementation, it is fairly likely that the extra spill code will fit into existing gaps in the schedule. Thus, it seems fairly likely that a post-pass register allocator will perform adequately.

### 6.4 Tag Usage

In the previous sections, we have shown how much performance improvement is to be gained through the use of static speculation, and how much can be gained by using a particular strategy. In this section, we shall correlate number of tags required to represent the speculation with the various strategies for speculation, and thereby provide some insight into the number of tags that should be provided by an architecture.

#### 6.4.1 The Influence of Strategies

A best-N strategy moves code into a basic block from at most N other basic blocks. Assuming that one tag is needed to represent the speculation from each block, only N tags will be live within a basic block. However, there are two reasons why more than N tags can be needed.

First, even though only N tags are required to represent the speculations within a basic block, there may be a tag live through the basic block. An instance where this can happen is shown in Fig. 6.12. Assume that while scheduling basic-block A, an instruction was selected from E, and moved into A. Clearly, the tag used by A to represent the speculation from E cannot be reused till the branch at the end of E has been executed. Now, assume that while scheduling instructions in X, E was not picked by the selection strategy. Now, X may need up to N tags to represent its own speculative instructions; further, it will need an additional tag to represent the speculation from E into A.

Second, we may need more than one tag to represent the speculation of instructions from a single basic block. This arises when an instruction is speculated past at least two branches along different paths, and its inputs are different along the different paths. Such a situation is illustrated in Fig. 6.13.

In practice, neither of these cases arise very often, and consequently, in most cases, a best-N strategy can be implemented using N or less tags. This is illustrated in Fig. 6.14, which shows the number of tags required to implement the schedule produced by the best-5 strategy in the balanced, 4-issue scenario. As can be seen exactly 3 DAGs require more than 5 tags.

Further, it is impossible for either of these situations to arise when using either a best-2 or best-3 strategy with the whole-DAG scheduling algorithm as currently implemented\(^4\). Thus, if a

---

\(^4\)This will be shown in Appendix B.
Figure 6.12: Tag Live Through Block

Figure 6.13: Two Tags for a Basic-Block
Figure 6.14: Tag Usage: best-5, 4-issue, balanced
best-3 strategy is adopted, at most 3 tags will be required to obtain the speedups described in the previous section.

6.4.2 Architecture

The maximum number of tags available is determined by the architecture. Therefore, the architecture should be designed with a sufficiently large number of tags. From the numbers described in the previous section, the optimal strategies seem to be best-5 for a 4-issue architecture and best-7 for a 8-issue architecture. Further, best-5 and best-7 use at most 5 and 7 tags respectively, in about 99% of the cases. Thus 5 and 7 tags seems to be good numbers.

However, picking the largest number of tags required may be over-kill. Consider using 5 tags for a 4-issue architecture. Clearly, most instruction have to have extra bits attached indicating whether they are speculative, and if so, which tag they use. This information requires adding 3 bits to each instruction\(^5\). If, however, we chose to provide only 3 tags, the overhead would be 2 bits per instruction. This would force us to use the best-3 strategy. But the difference in DAG performance between best-3 and the optimal strategy is less than a percent; thus, for resource reasons, it makes sense to use only 3 tags on a 4-issue architecture.

The reasoning is not so clear-cut with an 8-issue architecture. Going from 7 to 3 tags represents about a 3% loss in DAG performance, which, depending on how important DAGs are overall, may or may not cause a significant drop in overall performance. There is little point in choosing some number between 3 and 7; they will all add an overhead of 3 bits per instruction.

6.5 Conclusion

In this chapter, we described the results of experiments we conducted to measure the performance of code scheduling for static speculation. Some of the major conclusions that we derived from the results we obtained include:

Static Speculation

- Static speculation can improve the performance of numerical programs by as much as 12% by increasing the performance of DAGs; however the performance gain is generally low, because DAGs are usually an insignificant part of the entire program.
- When used with an appropriate scheduling algorithm, static speculation improves the performance of DAGs by an average of 13%–17%.

Scheduling Algorithm

- Ideally, the scheduling algorithm should move instructions from a limited number of descendant DAGs, ranging between 3 and 7.

\(^5\) Actually, the overhead is \(\log_2(6)\) per instruction. It may be possible to achieve this, but it will complicate decoding.
• To gain full benefit from static speculation, the scheduling algorithm must move instructions past both sides of a branch.

• Our slack-based speculation heuristic is an improvement over other proposals; in particular, it improves DAG performance by about 3%.

Speculative Tags

• At lower issue widths, we can use 3 tags and not suffer a significant performance penalty. This necessitates an overhead of 2 bits per instruction.

• At higher issue widths, we may need up to 7 tags. This would require an overhead of 3 bits per instruction.
Chapter 7

Dynamic Speculation

In this chapter, we shall use the perspective gained in implementing static speculation to derive a dynamic speculation mechanism that also integrates register renaming and precise interrupts, and can be implemented at low cost. The rest of the chapter is organized as follow. Section 7.1 summarizes the state of the art of dynamic register renaming and speculation. Section 7.2 and Section 7.3 describe our approach to register renaming. Section 7.4 and Section 7.5 specify extensions for dynamic speculation and precise interrupts. Section 7.6 sketches a design. Section 7.7 discusses an optimized design that merges register renaming, dynamic speculation and precise interrupt mechanisms. Section 7.8 elaborates the performance-critical portion of the design, and argues that it should not impact cycle time. Section 7.9 compares our approach against previous approaches. Finally, Section 7.10 summarizes the results of the paper.

7.1 Prior work

Tomasulo's Algorithm: The earliest implementation of register renaming was for the floating-point unit of the IBM 360/91 [63]. The IBM 360 is a CISC architecture, but in the following description, we will restrict our attention to register-to-register operations.

Every register is augmented by a busy bit. If the bit is off, the register holds the value needed by any instruction that references it. If the bit is set, the register holds the tag of the instruction that will produce the need value.

When an instruction enters the decode stage, it is assigned a tag. It reads each source register, obtaining an input value or a tag, depending on the busy bit. The instruction writes its tag into its output register and sets that register’s busy bit. The instruction and its input values/tags are buffered in one of several reservation stations, depending on which functional unit it will execute in.

At each cycle, a functional unit scans the instructions stored in its reservation station. An instruction may be ready, i.e., have both input values available, or waiting, i.e., one or more of its inputs is a tag, and the instruction cannot execute until some other instruction produces that input. The functional unit will select a ready instruction and start executing it.
When an instruction completes, its result and tag are broadcast to all waiting instructions. Each waiting instruction compares this tag against the tags it is waiting on. If the tags match, the instruction overwrites the tag with the result. This may change the instruction’s status from waiting to ready. The result is also sent to the register file. If the tag in the output register is the same as that of the instruction, the value is written to the register file.

Tomasulo’s algorithm must be extended to implement precise interrupts [54]. Johnson [29] describes one such extension, based on the future file mechanism [54]. The registers described above correspond to the future file. There is also a duplicate register file called the in-order file, and a reorder buffer. Instructions issue as described, using the future file\(^1\). Additionally, results are written to the reorder buffer. The reorder buffer is a FIFO queue. When an instruction is issued, it receives a slot at the bottom of the queue. When the instruction completes, its result is written to the allocated slot. If the instruction at the top of the queue has completed, its result is used to update the in-order file, and the queue is advanced.

It is easy to see that the reorder buffer updates the in-order file with the results of the instructions in the order in which the instructions were issued. At the time an instruction’s results are written to the in-order file, any exception caused by it is signalled. Therefore, when an instruction causes an exception, the in-order file contains the register state obtained by executing all instructions prior to the excepting instruction—exactly that required by a precise interrupt.

**Johnson’s Algorithm:** Johnson, in [29], describes another approach to register renaming. This approach replaces the future-file and reorder buffer of the previous section with an associative reorder buffer. As in the previous method, results are written first to the reorder buffer, which releases them to the in-order file in program order. The reorder buffer is also extended in the manner described below.

When an instruction is allocated a slot in the reorder buffer, its output register name and its tag are stored in that slot. When the instruction completes, the tag is overwritten with the result value.

To obtain an input value, the in-order file and the reorder buffer are accessed in parallel. If the reorder buffer contains no entry whose register matches the input register, the value returned by the in-order file is used. If there is some entry in the reorder buffer with a matching register, the tag or value of the youngest such entry is returned. This, of course, requires an associative lookup prioritized by age, using the register name as the key.

**Metaflow’s DRIS:** The Metaflow architecture [49] implements out-of-order execution and dynamic speculation as well as register renaming. Most of the mechanisms required are integrated into the DRIS (deferred-scheduling, register-renaming instruction shelf). Unlike the approaches described above, but like [12], instructions are not deferred in reservation stations at each execute unit, but in one central structure. Every cycle, this centralized structure is searched for instructions which can be executed and which are then dispatched to the appropriate execute unit. Apart from that, register renaming using the DRIS is very similar in spirit to the previous approach.

\(^1\)For simplicity, we ignore the changes made in [29] to support speculation.
The DRIS is a queue, much like the reorder buffer. When an instruction is issued, it is allocated a slot at the bottom of the DRIS. When an instruction completes, its result is written to the allocated slot. If the top instruction in the DRIS has completed execution, it is popped from the DRIS, and the register file is updated with the instruction’s result.

Every time an instruction is added to the DRIS, the DRIS is searched for instructions whose result registers match the incoming instruction’s source registers. If none is found, the required value can be read from the register file. If some match is found, then the source register is associated with the youngest such instruction. When the associated instruction completes, the required value can be read from that instruction’s DRIS entry.

This mechanism differs from the tag-and-result broadcast of Tomasulo’s algorithm, in that only the tag (i.e., the DRIS entry) is broadcast. It indicates that the result is available. All instructions in the DRIS waiting for that result mark it as available, and may become ready. The actual value is read from the DRIS entry\(^2\) when the instruction is dispatched to an execute unit.

**RS/6000 Floating Point Renaming:** The register renaming mechanism in the floating point unit of the RS/6000 [20] is quite different from those discussed above, both in motivation and in implementation. One of the motives for removing dependences was to enable the fixed-point unit to perform floating-point loads. Instead of having the fixed-point unit keep track of floating-point false dependences, the architects chose to rename the destination register of the load to eliminate the dependences.

As implemented, the floating-point unit has more physical registers than register names. There is a map-table containing a mapping of register names to physical registers. There is a queue containing free physical registers (roughly, those that are not mapped). When a floating-point instruction, other than a load, is decoded, its registers are renamed according to the map. When a floating-point load is encountered, a free physical register is removed from the queue, and used as the target of the load. The map is updated to reflect the fact that all subsequent references to the target register name will use the newly allocated physical register. We will not discuss this mechanism in more detail since it is subsumed by the mechanism discussed below, and by the one we are proposing.

**ES/9000:** The IBM ES/9000 [39] extends the renaming mechanism described above to all registers for most instructions. It also implements dynamic speculation past at most 2 conditional branches.

As in the RS/6000, there are more physical registers than register names, and a map-table (called the decode-time register assignment list, or *DRAL*) is used to keep track of the mapping of logical register names to physical registers. In the decode stage, the source register names in every instruction are renamed using the map. The instruction’s output register is mapped to a new free physical register, and the map is updated appropriately, displacing the old physical register mapped by the output register.

\(^2\)Or from the register file, if the associated instruction has completed and been removed from the DRIS before the incoming instruction is issued.
Registers are controlled using information stored in an content-addressable structure called the ACL (array control list). Whenever all instructions prior to an instruction have completed, the ACL is associatively searched using the instruction's identifier, and the physical register displaced by that instruction is freed.

The value of the map at a branch that is speculated past is saved in structures called the BRAL (backup register assignment list). On a mispredicted branch, the DRAL is restored to the appropriate state by copying from the appropriate BRAL. When a branch is mispredicted and all instructions prior to the branch have completed, the ACL is associatively searched to identify physical registers allocated to instructions issued after the branch. These registers are then freed.

Finally, the ACL contains information for precise register state recovery. For every physical register, it keeps track of whether the physical register contains a value that is part of the precise state. On an interrupt, the precise state is recovered by searching the ACL for entries in the precise state, and using them to update the map appropriately.

7.2 Our approach

The approach we are proposing implements register renaming in a direct fashion, using a single storage area. There is no separation of storage locations into an architectured register file and some auxiliary storage. Instead, there is one set of physical registers, from which all storage is allocated and into which all values are written. The architectured registers are mapped onto some subset of the available physical registers. This mapping changes as instructions are issued. The association of architectured register name with physical register is maintained in a mapping table [31]. The mapping table has an entry for each register name, which contains the physical register currently associated with, or mapped by, that register name. There is a free pool of physical registers, used to provide new physical registers when necessary.

After an instruction is fetched from the I-cache, but before it accesses the physical registers, the instruction’s source register names are mapped to physical registers. This is accomplished by indexing into the mapping table using the source register name.

The output register name is mapped to some new physical register, allocated from the free pool. The mapping table is modified to reflect this new association. The old physical register in the output register name’s entry is replaced by the new physical register. The register name is said to have been remapped to the new physical register. The old physical register is said to have been unmapped.

The mapping process is illustrated in Fig. 7.1. The processor used in this example is a 4 stage pipelined processor with 4 register names r1-r4. The implementation uses 8 physical registers p1-p8. Free contains a physical register from the free pool.

After the first instruction, \( r1 := r3 + r2 \), is fetched, its source register names are translated using the map to p3 and p2 respectively. The result of this instruction will be placed in the physical register p5. The map is updated to reflect the fact that \( r1 \) is now mapped to p5. The remaining stages of the pipeline execute the instruction \( p5 := p3 + p2 \).

When the next instruction is fetched, it is translated using the modified map. Thus, the source register \( r1 \) is mapped to p5. The result is to be written to p6, the new physical register, and the
<table>
<thead>
<tr>
<th>Map</th>
<th>Physical Register</th>
<th>Insn-Fetch</th>
<th>Op-Fetch</th>
<th>Execute</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>r1 := r3 + r2</td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>Free</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r1 := r1 + r4</td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>r3 := r2 * r3</td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>r2 := r3 - r1</td>
<td></td>
<td></td>
<td>p5 := p5 + p2</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>p8 := p7 - p6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1: Mapping
map must be modified to show that \( r_1 \) is now associated with this physical register.

Notice that the mapping table has only to keep around the most recent physical register for each register name. This information is adequate since instructions are issued to the op-fetch stage in program order. A source register name, therefore, always refers to the value produced by the closest, and therefore most recently processed, instruction with the same (output) register name. Hence, the instruction uses the value stored in the physical register mapped most recently to that register name—which is exactly what the mapping table stores.

After the mapping, the rest of the stages in the processor use the physical registers. They never see the original register names. One way of viewing this transformation is to look at the mapping as a translation of instructions from an architecture with a small number (number of register names) of registers to an architecture with a larger number (number of physical registers) of registers, performed in a manner that increases instruction-level parallelism.

### 7.3 Reclaiming physical registers

Since there are only a finite number of physical registers, the processor must start reusing them at some point. A physical register can be reused when it is guaranteed that the value in it can never be used by any later instruction. This is true if all of the following hold:

1. The value has been written to the physical register.
2. All issued\(^3\) instructions that need the value have read it.
3. The physical register has been unmapped.

The motivation for the first two conditions should be obvious. The third condition is motivated by the following: as long as the physical register is mapped by some register name, some later instruction may have that (source) register name, and therefore require that value. However, once that register name has been remapped (and, consequently, the physical register unmapped), no instruction can access that value.

The process of reclamation is illustrated by the following code fragment:

\[
\begin{align*}
(1) \quad r_1 &:= \ldots \\
(2) \quad &:= r_1 + \ldots \\
(3) \quad r_1 &:= \ldots \\
(4) \quad &:= r_1 - \ldots 
\end{align*}
\]

When instruction (1) is executed, it is allocated a new physical register, say \( p_{11} \). This is the register which the result of (1) will be written to. \( p_{11} \) cannot be freed until (1) completes. The next instruction refers to \( r_1 \), so it will read out of \( p_{11} \). Obviously, \( p_{11} \) cannot be reused until (2) has been issued. Now, (3) again uses \( r_1 \) as an output. So, it will be allocated a new output register, say \( p_{23} \). \( r_1 \) will be mapped to \( p_{23} \), and \( p_{11} \) is unmapped. Now, any subsequent reference to

---

\(^3\)In this paper, an instruction is said to be issued when it enters the instruction-decode/operand-fetch stage.
r1, such as in (4), will use p23. Clearly, no further uses of p11 can be produced. So p11 can be freed once (1) writes to it and (2) reads from it.

These conditions are easily detected using an approach which waits until the physical register has been written to and has been unmapped, and then waits for all instructions that use the physical register to complete. A counter per physical register is used to keep track of issued but unexecuted instructions in the processor pipeline that read that physical register. Since the total number of issued but unexecuted instructions in the processor is bounded and small, the counter does not have to be very big. An instruction that uses the value increments the counter as soon as it is fetched, and decrements the counter when it has read the value. A physical register can be reused when it has been written to, unmapped and its counter is zero.

To summarize—the processor maintains the following state for each physical register:

Complete Flag: The complete flag is false if the physical register has not been written into. It is set to false when the physical register is removed from the free pool, and is set to true when it is written to.

Unmap Flag: The unmap flag is false as long as the physical register is mapped by some register name. It is set to false when the physical register is first added to the mapping table, and is set to true when it is unmapped.

Counter: The counter keeps track of the number of instructions issued that need the value stored (or to be stored) in the physical register, but which have not yet read it. Every time an instruction is issued that needs this value, the counter is incremented. Every time an instruction reads the value, it is decremented.

A physical register can be reclaimed if the complete and unmap flags are true, and the counter value is zero.

Initially, each architectured register name is mapped to the corresponding physical register. These registers are in state F 0 T. The remaining registers are in the free register pool, with states T 0 T.

Fig. 7.2 illustrates the process of reclaiming physical registers. The processor implementation has been extended by the addition of the update flag, the counter, and the complete flag for each physical register. If these fields have the value T 0 T, the register is reclaimable. One of the reclaimed register is inserted in the free slot.

The first instruction unmaps p1. p1 is now reclaimable, since it has been written to, and there are no outstanding reads to it. Register p5 is mapped by r1, so its unmap flag is set to false in cycle 1. The next instruction has r1 as a source register name, so the counter for p5 is incremented. In cycle 3, the fetched instruction has r1 as an output register. This means that p5 is now unmapped, and its unmap flag is set to true. Also, the result of the first instruction is forwarded to the second instruction. Since the second instruction has read the value for p5, it decrements the counter for p5. Finally, in cycle 4, the first instruction completes, writes its result to the p5, and sets the complete flag for p5 to true. p5 can now be reclaimed, and is placed in the free slot.
### Figure 7.2: Reclaiming Registers

<table>
<thead>
<tr>
<th>Complete Counter</th>
<th>Unmap</th>
<th>Insn-Fetch</th>
<th>Op-Fetch</th>
<th>Execute</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>T 0 T</td>
<td>r1 := r2 + r3</td>
<td>p5 := p2 + p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F 1 T</td>
<td>Map r1 to p5</td>
<td>Read p2, p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F 0 T</td>
<td>Unmap p1</td>
<td>Decr p2, p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Incr p2, p3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T 0 T</td>
<td>r3 := r1 x r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T 0 T</td>
<td>p5 := p2 + p3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F 1 T</td>
<td>Map r3 to p6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F 0 T</td>
<td>Read p2, p3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Unmap p3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Incr p5, p2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T 0 T</td>
<td>r1 := r2 - r3</td>
<td>p6 := p5 x p2</td>
<td>p5 := p2 + p3</td>
<td>Forward p5</td>
</tr>
<tr>
<td>2</td>
<td>F 1 T</td>
<td>Map r1 to p1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T 0 T</td>
<td>Read p5, p2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Unmap p5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Decr p5, p2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T 0 T</td>
<td>Incr p2, p6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T 0 F</td>
<td>r1 := r1 + r1</td>
<td>p1 := p2 - p6</td>
<td>p6 := p5 x p2</td>
<td>p5 := p2 + p3</td>
</tr>
<tr>
<td>2</td>
<td>F 1 T</td>
<td>Map r1 to p3</td>
<td></td>
<td></td>
<td>Write p5</td>
</tr>
<tr>
<td>6</td>
<td>T 0 T</td>
<td>Read p2, p6</td>
<td></td>
<td></td>
<td>Complete p5</td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Unmap p1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td>Decr p2, p6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T 0 T</td>
<td>Incr p1, p1</td>
<td>p3 := p1 + p1</td>
<td>p1 := p2 - p6</td>
<td>p6 := p5 x p2</td>
</tr>
<tr>
<td>2</td>
<td>F 0 T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F 0 F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F 0 T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T 0 T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.4 Branches and speculation

Branches are another cause of serialization. There may be an interval between the time a branch is issued and the time it is resolved, i.e., the branch's direction and target are determined. Halting instruction issue until a branch is resolved results in a serious performance loss. The alternative, predicting a target for the branch, and continuing to issue instructions from that target before the branch is resolved, can cause problems when the prediction proves to be incorrect. For best results, the processor should be able to deal with several simultaneously unresolved branches. To simplify the hardware, branches are assumed to be resolved in program order.

An instruction that is issued from below a branch before the branch is resolved is said to have been speculatively issued. If a speculative instruction causes an exception, then the exception must not be reported till it is determined whether the predicted target was correct. If it was not correct, i.e., the branch was mispredicted, the interrupt must be ignored. Otherwise the interrupt must be reported in a manner consistent with the program order.

Branches are mispredicted fairly often. The mechanism that recovers from a mispredicted branch must be efficient. Prior work on recovery treats a mispredicted branch as a kind of interrupt, and modifies the precise-interrupt recovery mechanism to implement mispredicted branches. But mispredicted branches do not require the full power of the precise-interrupt recovery mechanisms. Consider the following example:

\[
\begin{align*}
  r1 & := r2 / r3 \\
  b\text{r}gt0 & r5, r8
\end{align*}
\]

Assume that at the time the branch is resolved, the divide has not completed. Precise interrupt recovery mechanisms would halt further execution till the divide completed. However, it is not necessary to do so just to recover from a mispredicted branch.

What actions must be taken to recover from a mispredicted branch? Consider the following example:

(1) \( r1 := r2 + r3 \)
(2) \( r3 := r7 / r4 \)
    \begin{align*}
    \quad \text{b}\text{r}gt0 & r5, L10 \\
    \end{align*}
(3) \( r1 := r1 - r4 \)
L10:
(4) \( r1 := r8 * r1 \)

Assume that the branch was predicted (incorrectly) to be not taken. Then, to correctly execute instruction (4) after resolution, source registers \( r8 \) and \( r1 \) must contain the values they would have had, assuming none of the speculative instructions had executed. This means that the value read from \( r1 \) must not be the value produced by (3), but the value produced by (1). Further, all values produced by the speculative instructions should be discarded. Specifically, the value produced by (3) could be used only by other speculative instructions, and since they, too, are mispredicted, their reading the value is irrelevant. Lastly, all interrupts produced by speculative instructions must also be discarded.
Modifying our design to satisfy these aims turns out to be fairly simple. Since multiple occurrences of the same output register name are given different physical registers, values stored in the register are never overwritten until the register is reclaimed and reassigned to another instruction. Therefore, the design must be modified to ensure that a physical register is not reclaimed before the branch is resolved. When the branch is discovered to have been mispredicted, the logical register names must be set to point to the physical registers containing the old values. In our design, this means that the mapping table must be restored to the state it was in at the time the branch was issued. These changes are described in more detail below.

The mapping table is small enough that it can be saved each time we speculate past a branch. If the branch is correctly predicted, the saved map is discarded. If the branch was mispredicted, the saved map is used to restore the mapping table, and all other saved maps are discarded. The depth of speculation in the scheme being presented is constrained by the number of maps that can be saved. There is an alternative scheme, described later, that does not require the map to be saved at each branch; instead it recovers the map from the information saved for precise interrupts. This scheme has no a priori limits on the depth of speculation.

In addition, the unmapping rule must be modified. A physical register is now considered unmapped if it is not mapped in the current mapping table, or any of the saved mapping tables. Under the previous definition, a physical register would be considered unmapped (and therefore reclaimable) if its associated register name was the output register for some speculated instruction. This case would arise in the example below (assume branch is predicted not taken)

\begin{align*}
(1) & \quad r5 := r10/r2 \\
(2) & \quad r1 := r2 + r3 \\
& \quad \text{brgt0 } r5, r8 \\
(3) & \quad r1 := r3 \times r4
\end{align*}

Reclaiming the physical register storing the result of (2) would be correct as long as the branch was correctly predicted. However, if it was mispredicted, the value might be needed by some instruction on the other path. Therefore, it cannot be reclaimed as long as the branch is unresolved. The modified unmapping rule (in combination with the fact that saved maps are discarded as soon as they are resolved) ensures that physical registers are not reclaimed too soon, and that they can be reclaimed if necessary after the branch is resolved.

As long as branches are predicted correctly, the complete flag and the counters are left untouched. Resetting them at a mispredicted branch is closely tied to the way the pipeline is flushed of the speculative instructions. If, for instance, all mispredicted speculated instructions in the pipeline are run to completion, and then discarded, complete flags and counters can be handled unchanged.

An interesting case is when the processor waits for all instructions issued before a mispredicted branch to complete before starting recovery, and then flushes all instructions in the pipe. At that point, since all instructions prior to the branch have completed execution, all values must have been read, so all counters must be set to zero. Further, all instructions prior to the branch have completed, so that all complete flags can be set to true. The speculative instructions do not matter, since they are all being discarded.

\footnote{There is an equivalent, but more complex definition, that can be implemented more cheaply.}
<table>
<thead>
<tr>
<th>Insn-Fetch</th>
<th>Op-Fetch</th>
<th>Execute</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>brgt r3 r2</td>
<td>Incr p3, p2</td>
<td>brgt p3, p2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save Map</td>
<td>Read p3, p2</td>
<td></td>
</tr>
<tr>
<td>r2 := r1 + r3</td>
<td>Map r2 to p5</td>
<td>Decr p3, p2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Incr p1, p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p5 := p1 + p3</td>
<td>Map r2 to p6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unmap p5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Incr p5, p4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p6 := p5 - p4</td>
<td>Read p5, p4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Forward p5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pop Map</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unmap p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p5 := p1 + p3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: Correctly Predicted Branch
The examples in Fig. 7.3 and Fig. 7.4 illustrate the way our design handles speculation. The architecture is further extended by the addition of a 2-entry map save stack. However, the current map is always stored in this stack. Therefore, the processor, as shown, can speculate past at most 1 branch. Mispredicted branches are assumed to be issued only after all prior instructions have completed. Counters and complete flags are handled according to the above scheme.

When the branch instruction is issued, the processor saves the current map on the map save stack. The next instruction remaps \( r2 \) to \( p5 \). However, the previously mapped physical register \( p2 \) is still not consider unmapped, since it is mapped in the map save stack. The next instruction again remaps \( r2 \). \( p5 \) is considered unmapped, since it is no longer mapped by the current or any of the saved maps. Finally, the branch is resolved in the 4th cycle.

Fig. 7.3 shows the processor behavior when the branch was correctly predicted. The saved map is popped from the stack. \( p2 \) is now no longer mapped by any map, and its unmap flag is set to true. In fact, it can now be reclaimed.

Fig. 7.4 shows the processor behavior if the branch was mispredicted. All but the top saved map are flushed from the map save stack. This map is used to restore the mapping table. The physical registers not mentioned in this map are readied for reclamation by setting their unmap and complete flags, and zeroing their counters. Of course, the instructions currently in the pipeline are flushed.

### 7.5 Precise interrupts

Any one of several precise interrupt mechanisms [54] can be adapted to the proposed design. The major modification necessary is that when an interrupt occurs, not only must the register state reflect the precise state, but the mapping table must be restored to its value at the time the interrupting instruction was issued. Actually, it is not necessary for all the physical registers to be restored to their value at the time the interrupting instruction was issued; it is enough to restore only those registers that were actually mapped at that point.

For instance, if we wish to adopt a checkpoint-retry mechanism [26] then, at the checkpoint, we would need to save the current map, and the values of the physical registers mapped by the current map. On an interrupt, the checkpointed values would be used to reset the map and the relevant physical registers.

The history buffer mechanism is a particularly attractive candidate. A history buffer is a FIFO, to which every instruction is added when it is fetched. When the instruction writes to the register file, the value it overwrites is saved in with it. Instructions are removed from the top of the history buffer when they complete. Any exception caused by an instruction is reported only when it comes to the top of the stack. At that point, the register file values are restored by "roll back"—basically, starting from the bottom of the buffer, all the saved register values are written back to the register file. The straight-forward adaptation of a history buffer to our scheme will save with each instruction the old physical register mapped by its output register. On an interrupt, both the map and the register file are restored to their precise state by rolling back the history buffer.
<table>
<thead>
<tr>
<th>Insn-Fetch</th>
<th>Op-Fetch</th>
<th>Execute</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>brgt r3 r2</td>
<td>Incr p3, p2</td>
<td>Save Map</td>
<td></td>
</tr>
<tr>
<td>r2 := r1 + r3</td>
<td>Map r2 to p5</td>
<td>Incr p1, p3</td>
<td></td>
</tr>
<tr>
<td>brgt p3, p2</td>
<td>Read p3, p2</td>
<td>Decr p3, p2</td>
<td></td>
</tr>
<tr>
<td>p5 := p1 + p3</td>
<td>Map r2 to p6</td>
<td>Read p1, p3</td>
<td></td>
</tr>
<tr>
<td>brgt p3, p2</td>
<td>Unmap p5</td>
<td>Decr p1, p3</td>
<td></td>
</tr>
<tr>
<td>p6 := p5 - p4</td>
<td>Incr p5, p4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brgt p3, p2</td>
<td>Pop all but top Map</td>
<td>Complete p5,p6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unmap p5,p6</td>
<td>Clear Counters</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore Map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble</td>
<td>Bubble</td>
<td>Bubble</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.4: Mispredicted Branch
7.6 Summarizing the design

In this section, we shall bring together the different components that we have introduced above. The different pieces comprising the design are summarized below;

**Mapping Logic:** The mapping hardware is responsible for translating each source register specifier to the corresponding physical register. This component is discussed at length in Section 9.

**Map Modify Logic:** The map modify logic is responsible for updating the current map after each cycle. Briefly, the mapping for each architected register is updated as follows:

- On a reset or a context switch, the entry is set to some default value, such as the entry number. Thus, initially, architected register \( N \) maps to physical register \( N \).
- On a mispredicted branch, the map is updated from the map at the top of the map save stack.
- If the architected register is the same as the output register from one of the fetched instructions, then it is replaced by the free physical register allocated to that instruction.
- Otherwise it is set to its old value.

**Map Save Table:** The map save table, as indicated before, is a FIFO that saves the map values at each branch. On a branch, the current map is saved at the bottom of the table (if possible; otherwise fetching is stalled). On a mispredicted branch, the table is flushed, and the current map set to the top map in the table. On a correctly predicted branch, the top saved map is removed from the table.

**Free Physical Register Logic:** Freeing physical registers is possibly the most complex part of the design. It involves keeping track of when an instruction has completed, the number of outstanding uses, and whether it has been unmapped. Completion is easy; whenever a physical register is written, the register's completion flag is set. Keeping track of outstanding reads involves a multi-input up-down counter, which is incremented by the number of instructions that are fetched each cycle, and decremented by the instructions executed each cycle. Unmapping, as discussed above, is implemented by setting the unmap flag for a register when it is not present in the current or any of the saved tables. There exists a inexpensive implementation for detecting when a register is no longer in the tables, and can be unmapped.

**Precise Interrupt Logic:** A mechanism, such as the history buffer, is needed to ensure that interrupts are precise.
7.7 Optimization

The design described above is fairly general, and can be made even more general. For instance, it could be adapted to handle branch prediction resolution in any order. However, providing this generality requires extra hardware. In this section, we shall introduce certain constraints on the implementation. These constraints enable us to greatly simplify the implementation, by combining many of the different components listed above into one simple structure.

One major source of complexity is determining when physical registers can be reused. This happens when the register has been written to, there are no outstanding reads and the associated architected register has been remapped. These must all be true if all instructions up to and including the instruction that displaced the physical register have completed. Clearly, the instruction that was allocated the physical register must have completed, and therefore the register must have been written to. All the instructions that could access this register must precede the displacing instruction, and since they have completed, all reads of the register must have been accomplished. And, of course, the register must have been remapped. This criteria—all instructions up to and including the instruction that remaps the physical register must have completed—is fairly simple to implement.

The implementation can be built around a history buffer. An instruction is added to the bottom of the history buffer when it is fetched. An instruction is removed from the top of the history buffer after it completes. Thus, the history buffer contains instructions in program order. To support register freeing, when an instruction is added to a history buffer, the physical register it remapped is saved with it. When the instruction is removed from the top of the history buffer, this physical register is freed.

A mechanism that frees registers as described above can be used to simplify precise state recovery significantly. We only need to recover the values in those registers that were mapped at the time the excepting instruction was issued. In our scheme, a register can only be overwritten after it is freed, and it is freed only after all instructions that were issued while it was mapped complete. Thus, all values which are part of the precise state are present in physical registers, which have not been freed. Therefore, when an instruction traps, only the map has to be restored to what it was when the instruction was issued; none of the registers that were mapped at that time would have been freed and overwritten.

Interrupt recovery and misprediction recovery can be facilitated by maintaining an in-order map, i.e., the map used by the instruction at the top of the history. This can be implemented by saving, for each instruction in the history buffer, the physical register which the instruction's output register was mapped to before being remapped by the instruction. When an instruction is retired from the top of the history buffer, the in-order map is updated appropriately. If exceptions are reported only when the excepting instruction reaches the top of the history buffer, the map can be restored in a single cycle by copying the in-order map instead of rolling back the history buffer. Similarly, if mispredicted branches are handled only when the branch reaches the top of the history buffer, the map can be reloaded using the in-order map.

The in-order map also suggests another optimization: there is no need to save the physical register that was displaced in the history buffer to support freeing. Instead, the register to be freed can be selected from the inorder map by using the architected output register of the instruction at
the top of the history buffer.

To summarize: the history buffer has an entry for every instruction. The information contained in this entry includes the instruction, the physical register displaced from the current map by the instruction when it was issued, and a bit indicating whether the instruction has completed. There is an in-order map, which satisfies the invariant that it is the same as the current map used by the instruction at the top of the history buffer. The actions performed are the following.

- When an instruction is fetched, it is added to bottom of the history buffer. The physical register to which the instruction’s output register is remapped is also added (assuming, of course, that the instruction has an output register).

- When the instruction at the top of the history buffer completes, it is removed from the history buffer. The physical register in the in-order map that corresponds to the architected output register of the instruction is freed. It is replaced by the physical register saved with the instruction being removed.

- If the instruction at the top of the history buffer is a mispredicted branch, then all registers not in the in-order map are freed. The in-order map is copied to the current map.

- The behavior on an interrupt is identical to the behavior at a mispredicted branch.

Thus, it is possible to dispense with the counters, map save stack, register save stack etc., and implement physical register freeing and mispredict recovery economically using the history buffer mechanism described above. This mechanism also allows us to implement single-cycle precise state recovery. All this is accomplished at the cost of some generality.

### 7.8 Implementation

We have discussed the need for additional hardware, including logic to perform the mapping, modify the map, free registers, etc. We also assumed the existence of logic for branch prediction and an instruction dispatch mechanism\(^5\). However, lack of space prevents us from discussing the detailed implementation of all of this logic. Instead, we shall concentrate on the mechanism for actually performing the mapping, since this is the only logic which could potentially increase the machine cycle time.

In the past, renaming has been implemented in the instruction dispatch stage. This stage is considered to be one of the most critical stages in superscalar processors in determining cycle time [23,64]. We propose to implement our renaming scheme in the instruction fetch stage, as shown in Fig. 7.5. The critical paths are shown by the solid lines; paths shown by the dashed lines represent feedback paths, i.e., values used in the next cycle. As indicated in Fig 7.5, mapping takes place before set selection; this means that all sets go through the mapping logic, which must be replicated according to the associativity of the cache.

\(^5\)We have deliberately not specified whether instruction dispatch uses reservation stations or a centralized mechanism; our approach works with both.
Figure 7.5: Instruction Fetch Stage
The proposed scheme requires that each source register name be mapped using the mapping table to its associated physical register. If there are 32 architected registers, the mapping is implemented in the mapping logic as a 32:1 mux. Using a pass-transistor based multiplexor implementation in CMOS, this will probably involve 4 logic stages. The source register fields must control several multiplexors; this requires they be initially powered up to obtain a fanout estimated to be about 40.

As shown in the figure, the critical path is determined either by the tag match logic required for cache operations or by the mapping logic discussed previously. The path through the tag match logic is influenced by four factors, namely, the associativity of the cache, its size, the address size and the fan-out requirements of the tag compare. For the purposes of this discussion, we assume that the cache is not direct mapped and that the address size is 32 bits.

The tag match logic starts off by checking, depending on the cache size, possibly the high 18–24 bits of the address and the program counter for equality, which can be implemented in about 4 stages of logic stages. The result bits then serve as control for a number of multiplexors. If the implementation attempts to issue 4 instructions per cycle, this implies a fanout of about 128\(^6\). This requires a powering tree that will add 2–3 logic stages (assuming some of the powering is buried in the compare).

Putting all the requirements together, we observe that both paths require about the same number of logic stages: about 6–7. This suggests that, assuming CMOS technology and a set associative cache, the proposed scheme may not increase the machine cycle time.

Notice that only the source registers go through the mapping logic; specifically, the time at which the opcodes become available to the instruction decode stage does not change. This may allow us to overlap some of the mapping with the initial phases of the instruction decode in the decode stage, if necessary.

None of the other mechanisms required add to the cycle time. Updating the map to reflect any remapped registers is not time critical. It can be done in the next cycle, in parallel with the I-cache access. The mechanism to free physical registers is also not time-critical. All that matters is that there be enough free physical registers, not which ones, i.e. throughput matters, not latency. Adding an extra cycle to the time to free a physical register can be compensated by expanding the number of physical registers by the number required per cycle.

So far in the discussion, we have assumed that all instructions are 3-address instructions. There will be several instructions with no output register (e.g., branches), and some with only one input register. Superscalar issue adds a related problem—if some source register name is the same as the result register name of some instruction fetched at the same time, but prior in the program order, it must use the new physical register allocated to that register name, rather than the entry in the mapping table. It would seem that we would need to do some decoding in the instruction-fetch stage.

However, we can delay these decisions till the next cycle. All instructions are initially mapped as though they were independent 3-address instructions. In the next cycle, as part of the remap logic, the instructions are decoded to determine whether they actually had an output, and source and result registers are compared for identity. If the instruction did not have an output, or multiple

\(^6\)The fanout can be of this order, or worse, even if less instructions are issued per cycle, depending on the physical organization and read out of the cache arrays.
instructions had the same output register, the mapping is updated appropriately. Thus, decodes are not added to the critical path.

A similar fixup has to be performed in the instruction decode stage. Determining the appropriate register to use can be done in parallel with register access. If the physical register provided by mapping table is incorrect, the value read can be discarded. This results in no loss in performance. Either the instruction did not need that value, or the wrong register was accessed because some other instruction fetched in parallel had the same output register name. In the second case, the value needed is the one that will be provided by the other instruction. That instruction, obviously, cannot have completed, so the instruction with the incorrectly mapped register cannot be issued immediately, anyway. Thus, there is at least a full cycle in which to determine the correct register mapping.

### 7.9 Comparison

The performance gained by adding a new mechanism has to be balanced against the amount of hardware required to implement the mechanism, and that implementation's potential impact on the cycle-time of the processor. Both of these are extremely difficult to quantify without actually an actual implementation, and even with such an implementation it is sometimes difficult to isolate the impact caused solely by the new mechanism, or by its interactions with other processor elements. So, the comparison of our scheme with other schemes that implement register renaming, dynamic speculation and precise interrupts will be more qualitative than quantitative.

One critical path in register renaming is the path from the instruction fetch to execution. On a vanilla RISC processor, this path is implemented in two stages, Instruction-Fetch and Instruction-Decode (also called Operand-Fetch). Logically, it follows the following steps:

\[
\text{Fetch} \rightarrow \text{Decode} \rightarrow \text{ReadValues} \rightarrow \text{Execute}
\]

Adding renaming adds an extra step:

\[
\text{Fetch} \rightarrow \text{Decode} \rightarrow \text{Rename} \rightarrow \text{ReadValues} \rightarrow \text{Execute}
\]

If these steps are implemented serially, obviously either an extra stage will be required, as in [20, 49], or the cycle-time of the existing two stages will be increased. The only way to avoid this is to implement some of the steps in parallel. [29] discusses several ways of folding together register renaming with register read, all of which require content-addressable storage. The time to perform an associative lookup is obviously going to be greater than that of a register access. It may turn out that the associative access will end up increasing the cycle time.

Our approach chooses to fold renaming with instruction fetch. A preliminary evaluation suggests that the mapping mechanism used can execute in parallel with elements of the instruction fetch stage without impacting the critical path through that stage. The rest of the path should not change from that in a vanilla RISC processor. In particular, reading values involves a normal register access.
Associative lookup requires content-addressable storage to implement efficiently. This storage requires complex and expensive hardware. The designs proposed by [20] and [29] use associative lookup to access register values. Similarly, the implementation described in [39] uses content-addressable storage for register control. Our scheme, on the other hand, requires no such complexity.

Finally, our scheme, like [20] and [29], but unlike [39], enables us to recover precise register state in a single cycle. This in turn lets us implement single cycle interrupt recovery and speculation past an arbitrary number\footnote{Not entirely arbitrary; it is limited to the number of slots in our history buffer.} of branches.

7.10 Conclusion

In this paper, we have presented a proposal that is in line with the current thrust of processor design and implementation. It exploits both the regular nature of RISC instructions and the fast, pass-gate based, multiplexors available in CMOS technology to implement renaming efficiently in the instruction fetch stage. This early implementation of renaming has simplifying repercussions for the rest of the design.

The optimized implementation proposed here integrates speculation, renaming and precise interrupts. The synergy resulting from this integration results in a simple and potentially cheap implementation of the three mechanisms.

It seems fairly clear that the hardware costs of our proposal are fairly modest, especially in the context of the 3 million or more transistors being used to implement current microprocessors. The open question, therefore, is the following: can the proposal be implemented without increasing the cycle-time? We hope to add our mechanisms to an existing microprocessor, and demonstrate, at least via simulation, that the addition will not impact the cycle-time of the processor.
Chapter 8

Conclusions and Future Work

In this dissertation, we have examined issues related to the implementation and exploitation of static speculation. Here, we summarize our research and present the significance of this work. We also sketch out future directions of research in this area.

8.1 Conclusions and Contributions

In this dissertation, we showed that static speculation should be a feature of multiple-issue architectures with a high issue-width. This required addressing pragmatic concerns such as exception handling in the presence of speculative instructions and the implementation of speculative instructions, as well as performance related questions.

Past work in the area of static speculation glossed over the problem of supporting exceptions. The need for addressing exception handling was recognized, since it is key to implementing features like virtual memory; however, the solutions proposed were somewhat ad-hoc. We systematically examined the interaction between exception handling and static speculation. Our investigation showed that it is possible to produce code such that there is no difference in handling exceptions for non-speculative and speculative programs. We derived the compiler modifications necessary to make this possible.

Existing techniques for implementing speculative instructions are not uniformly applicable. They impose restrictions on the kinds of speculation expressible, especially when multi-path speculation is being performed. We introduced speculative tagging, a mechanism for implementing speculative instructions that was designed with both error-recovery and multi-path speculation in mind. Speculative tagging reduces the restrictions and cost of error-recovery. It is uniform in that it allows all types of instruction to be speculated, and allows all multi-path speculations to be expressed.

We developed a new algorithm, whole-DAG scheduling, for exploiting static speculation when scheduling acyclic code. Our approach differs from previous work in that it provides fine grain control over the amount of speculation desired. Also, it uses a significantly better instruction-selection heuristic than previous algorithms. A performance evaluation shows that our scheduling algorithm can achieve results that are within 10% of optimal.
8.2 Future Work

8.2.1 Non-Speculative Architectures

It is unlikely that speculative instructions will be added to current generation micro-processor families. Note, however, speculative instructions are needed primarily to support instructions that could cause exceptions. Clearly, there will exist instructions that will not cause exceptions even when speculated, and thus can be moved above branches even on architectures without speculative instructions. It would be interesting to investigate whether a compiler can detect enough such instructions to justify implementing "safe" speculation in a compiler for non-speculative architectures.

8.2.2 Register Allocation

Register allocation is an aspect of code-generation that we have not addressed in this dissertation. Speculative tagging makes it simple to spill registers, so a standard graph-coloring based post-pass register allocator [9] may be adequate. However, it may be possible to do better. One possibility is to implement an integrated scheduling and allocation algorithm, such as by adapting the integrated algorithm described in [19]. Another possibility is to use a post-pass allocation algorithm that substitutes register-to-register moves at basic block boundaries for spills, in a manner similar to the "chameleon intervals" of [22].

8.2.3 May-Dependence Speculation

We saw earlier that performance gets retarded when the compiler cannot determine whether or not two memory operations use the same address, i.e. when aliasing is present. This results in a may-dependence: the two instructions may or may not be dependent on each other. One way of obtaining performance even in the presence of may-dependences is to split the instruction into two, optimize one assuming the instructions are dependent and the other assuming they are independent, and then use speculation to appropriately schedule the instructions. This process is illustrated in Fig. 8.1.

Unfortunately, this approach leads to exponential code-expansion. Consider an instruction which is may-dependent on two other instructions. To circumvent both these dependencies requires 4 copies of the instruction, one for each assumption of dependence/independence. An alternative solution might be to allow hardware to perform this kind of speculation. However, this also apparently leads to hardware that is exponential in the number of may-dependences bypassed.

It would be interesting to investigate combined hardware/software techniques that can increase performance in the presence of may-dependences, without exponential code expansion.

8.2.4 Dynamic Multi-Path Speculation

In the near future, it may be practical for a processor to dynamically speculate down multiple paths. The number of paths the processor can follow will be restricted—possibly 2–4. The processor must
allocate these paths in a manner that optimizes expected performance. Work should be performed to discover the best strategies for allocating these paths; perhaps some of the work we have performed to discover optimal static speculation strategies will be applicable.
Appendix A

Proof of Speculative Equivalence

In this chapter, we shall present a formal proof that a non-speculative program that is similar to a speculative program is also speculatively equivalent to it, if the speculative program follows the constraints outlined in Chapter 3.

In Section A.1, we describe the formal model of execution on which we will base the proof. In Section A.2 through Section A.4, we shall define similarity, the compiler constraints, and speculative equivalence in terms of the model. Finally, in Section A.5, we shall give the formal proof.

A.1 The Model

In this section, we shall describe the components of our model. Their types are summarized in Fig. A.1. A more elaborate description follows, including the restrictions we place on the various atoms and functions, and their interaction.

A.1.1 Atoms

The atomic or primitive types used in the model are Value, Locn, Label, and Cmd. Value is the set of values stored in locations and manipulated by commands. Locn, the set of locations, is partitioned into two disjoint sets of speculative and non-speculative locations. Label is the set of program labels. It is, as indicated, totally ordered. We shall make use of this total ordering; in particular, we shall use $\text{succ}_{Label}$, the successor function implied by the total ordering. Cmd, the set of commands is also divided into speculative and non-speculative commands. We shall show how the behavior of a command is described later in this section.

A.1.2 Map

For each speculative location, there exists a non-speculative location. The functions, $M$, defines a (possibly many-to-one) mapping from speculative locations to non-speculative locations. This function is defined for all speculative locations; i.e.,
Atoms:
\[ u, v \in Value \quad \text{set of values} \]
\[ k, l \in \text{Locn} = \text{SpecLocn} \uplus \text{NonLocn} \quad \text{set of locations} \]
\[ \alpha, \beta \in \text{Label} \quad \text{totally ordered set of labels} \]
\[ c, d \in \text{Cmd} = \text{SpecCmd} \uplus \text{NonCmd} \quad \text{set of commands} \]

Primitive Objects:
\[ \mathcal{P}, Q \in \text{Program} : \text{Label} \rightarrow \text{Cmd} \]
\[ \sigma, \phi \in \text{State} : \text{NonLocn} \rightarrow (\text{Value} \mid \text{Locn}) \uplus \perp \]
\[ \text{SpecLocn} \rightarrow <(\text{Value} \mid \text{Locn}), \text{Label}> \uplus \perp \]

Functions:
\[ \mathcal{M} : \text{SpecLocn} \rightarrow \text{NonLocn} \]
\[ \text{Rd} : \text{Cmd} \times \text{State} \rightarrow \text{Locn}^* \]
\[ \text{Op} : \text{Cmd} \rightarrow (\text{Value} \mid \text{Locn})^* \rightarrow (\text{Value} \mid \text{Locn})^* \]
\[ \text{Swr} : \text{SpecCmd} \rightarrow (\text{Value} \mid \text{Locn})^* \rightarrow \text{SpecLocn}^* \]
\[ \text{Nwr} : \text{NonCmd} \rightarrow (\text{Value} \mid \text{Locn})^* \rightarrow \text{NonLocn}^* \]

Semantic Functions:
\[ \mathcal{E} : \text{Label} \times \text{Cmd} \times \text{State} \rightarrow \text{State} \]
\[ \mathcal{N} : \text{Label} \times \text{Cmd} \times \text{State} \rightarrow \text{Label} \]
\[ \mathcal{R} : \text{Program} \times \text{Number} \times <\text{State}, \text{Label}> \rightarrow <\text{State}, \text{Label}> \]

Auxiliary Functions:
\[ \text{MapContent} : (\text{Value} \mid \text{Locn}) \rightarrow (\text{Value} \mid \text{NonLocn}) \]
\[ \text{Content} : \text{State} \times \text{Locn} \rightarrow (\text{Value} \mid \text{Locn}) \]
\[ \text{LabelOf} : \text{State} \times \text{SpecLocn} \rightarrow \text{Label} \]
\[ \text{Pair} : \text{Program} \times \text{Number} \times <\text{State}, \text{Label}> \rightarrow \text{Number} \uplus \perp \]

Figure A.1: Atoms and Function Types
\[ \forall k' \in SpecLocn : \exists k \in NonLocn : k = \mathcal{M}(k') \]

The associated auxiliary function, \( \text{MapContent} \), reads a value or a location, and returns a value of the respective type. However, if the input is (the address of) a speculative location, it converts it to the corresponding non-speculative location, by \( \mathcal{M} \).

**Definition 1:**
\[
\text{MapContent}(u) = \begin{cases} 
  u & u \in \text{Value} \cup \text{NonLocn} \\
  \mathcal{M}(u) & u \in \text{SpecLocn}
\end{cases}
\]

### A.1.3 Program

A program is a mapping from labels to commands. Note that in our model, we the labels are a totally ordered set. We assume that there exists a distinguished label \( \alpha_0 \) in which the program is supposed to start executing.

### A.1.4 State

A state is a mapping from locations to values. However, we shall restrict ourselves to considering legitimate states, where a legitimate state is one that satisfies the following properties. First, we restrict the state so that only speculative locations to hold pointers to speculative locations.

**Property 1:**
\[ \forall k \in \text{NonLocn} : \sigma(k) \notin \text{SpecLocn} \]

Further, we want to ensure that at most one speculative location updates a non-speculative location at a commit. So, all legitimate states will satisfy the following property as well:

**Property 2:**
\[ \forall k', l' \in \text{SpecLocn} : (\sigma(k') =< u, \alpha > \land \sigma(l') =< v, \alpha > \land \mathcal{M}(k') = \mathcal{M}(l')) \]
\[ \implies k' = l' \]

We shall later show that any executing any command in a legitimate state produces a legitimate state.

We use some auxiliary functions to isolate the values associated with locations in a state.

**Content:** returns the value contained in a location. In particular, if the location is a speculative location, it extracts the value from the tuple associated with the location.

**Definition 2:**
\[
\text{Content}(\sigma, k) = \begin{cases} 
  u & k \in \text{NonLocn} \land \sigma(k) = u \\
  v & k \in \text{SpecLocn} \land \sigma(k) =< v, \alpha > \\
  \bot & \text{else}
\end{cases}
\]

**LabelOf:** returns the label associated with the value stored in a speculative location; i.e., the second element of the tuple.

**Definition 3:**
\[
\text{LabelOf}(\sigma, k') = \begin{cases} 
  \alpha & \sigma(k') =< v, \alpha > \\
  \bot & \text{else}
\end{cases}
\]
It will be necessary to compare the values in the non-speculative locations of two states for equality. If they are equal, we say that the non-speculative components of two states are equivalent, written $\sigma \equiv_{NonSpec} \phi$. Thus,

**Definition 4:**

$\sigma \equiv_{NonSpec} \phi \iff \forall k \in NonLocn : \sigma(k) = \phi(k)$

### A.1.5 Sequences

We shall, in several places, be using sequences of values and/or locations. A sequence will be indicated with an vector overbar. Thus:

$\overline{u} \in (Value | Locn)^*$

$\overline{k} \in Locn^*$

The $i$-th element of a sequence will be indicated by subscripting with an integer. Typically the subscripting variable will be implicitly quantified as ranging between 1 and the length of the sequences. Thus:

$\forall I : Prop(k_I) \equiv \forall I, 1 \leq I \leq |\overline{u}| : Prop(k_I)$

Through an abuse of notation, we assume that each of the functions, when applied to a state and a sequence of values of the appropriate type, returns a sequence of the same size containing the function applied to each element of the input. For example:

$Content(\sigma, \overline{k}) = \overline{I}$, where $\forall I : \overline{I}_I = Content(\sigma, k_I)$

### A.1.6 Command

We shall not differentiate between most commands, beyond distinguishing speculative and non-speculative commands. Instead, as shall be shown in the next section, we shall define their behavior in abstract terms. We shall identify certain properties that these behaviors must satisfy, and use those properties in our proofs.

However, two instructions are interesting in the way they modify state. The first is **commit** and the other **skip**. Also, two instructions, **if** and **goto** are unique in the way that they modify the control-flow through the program.

The execution of a statement is modeled using two functions. The first, $E$, models the effect executing a statement has on the program state. The second, $N$, models the effect executing the statement has on control-flow, i.e., the next statement to be executed.

#### Non-speculative Commands

The behavior of a general non-speculative command (i.e., not including commit or skip) is summarized by the equation given below, with appropriate definitions for $Rd$, $Op$ and $Nwr$:

**Definition 5:**

$E(\alpha, c \in NonCmd, \sigma) = \phi$, where

$\overline{k} = Rd(c, \sigma)$
\[
\bar{\nu} = \text{Content}(\sigma, \overline{k}) \\
\bar{\sigma} = \text{Op}(c, \bar{\nu}) \\
\overline{I} = \text{Nwr}(c, \bar{\nu}) \\
\phi(n) = \begin{cases} 
\overline{\sigma}_H & n = \overline{I}_H \\
\sigma(n) & \text{else}
\end{cases}
\]

In particular, the command reads the values in locations identified by \(Rd\). Using these values, the command produces output values as specified by \(Op\), and uses them to modify the locations specified in \(Nwr\).

The functions defining the behavior of a non-speculative command must satisfy the following properties:

- The command reads only non-speculative locations.

**Property 3:**
\[
\forall c \in \text{NonCmd}, \sigma \in \text{State} : k \in Rd(c, \sigma) \Rightarrow k \in \text{NonLocn}
\]

Since the command reads only non-speculative location, as a consequence of Prop. 1, the values read by the command are either values or (the addresses of) non-speculative locations.

**Property 4:**
\[
\forall c \in \text{NonCmd}, \sigma \in \text{State} : u \in \text{Content}(\sigma, Rd(c, \sigma)) \\
\Rightarrow k \in \text{NonLocn} \cup \text{Value}
\]

- The number of values produced and the number of locations modified by the command are identical.

**Property 5:**
\[
\forall c \in \text{NonCmd}, \sigma \in \text{State} : \bar{\nu} = \text{Content}(\sigma, Rd(c, \sigma)) \\
\Rightarrow |\text{Op}(c, \bar{\nu})| = |\text{Nwr}(c, \bar{\nu})|
\]

- The locations modified by the command are non-speculative. Note that this follows from the type of \(Nwr\).

**Property 6:**
\[
\forall c \in \text{NonCmd}, \bar{\nu} \in (\text{Value} \mid \text{Locn})^* : l \in \text{Nwr}(c, \bar{\nu}) \Rightarrow l \in \text{NonLocn}
\]

- We want to ensure that any state produced as a result of executing a non-speculative command is legitimate (assuming, of course, that the command was executed in a legitimate state). This requires that the following property hold:

**Property 7:**
\[
\forall c \in \text{NonCmd}, \bar{\nu} \in (\text{Value} \mid \text{NonLocn})^* : v \in \text{Op}(c, \bar{\nu}) \\
\Rightarrow v \in \text{NonLocn} \cup \text{Value}
\]

Additionally, commands must satisfy the property detailed below. It states that the \(I\)-th location read by a command depends only on the command and the values in the previous locations accessed. Note that this property must be satisfied by both speculative and non-speculative commands.
**Property 8:**
\[ \forall c \in Cmd, \sigma, \phi \in State : \]
let
\[ \vec{k} = Rd(c, \sigma) \]
\[ \vec{l} = Rd(c, \phi) \]
in
\[ |\vec{k}| = |\vec{l}| = 0 \]
\[ \lor (\vec{k}_1 = \vec{l}_1) \]
\[ \land (\forall I : \]
\[ (\forall H, 1 \leq H < I : \vec{k}_H = \vec{l}_H \land Content(\sigma, \vec{k}_H) = Content(\phi, \vec{l}_H)) \]
\[ \Rightarrow \vec{k}_I = \vec{l}_I) \]

**Speculative Command**

The specification of a speculative command is much like that of a non-speculative command, except that the speculative locations in which the values produced by the command are to be stored till committed need to be specified. This is modeled by \( Swr \). Thus,

**Definition 6:**
\[ \mathcal{E}(\alpha, c' \in SpecCmd, \sigma) = \sigma', \text{ where} \]
\[ \vec{k}' = Rd(c', \sigma) \]
\[ \vec{i}' = Content(\sigma, \vec{k}') \]
\[ \vec{n}' = Op(c', \vec{i}') \]
\[ \vec{m}' = Swr(c', \vec{n}') \]
\[ \sigma'(n) = \begin{cases} < \vec{n}'_H, \alpha > & n = \vec{n}'_H \\ \perp & n \in SpecLocn \land n \notin \vec{m}' \land LabelOf(\sigma, n) = \alpha \\ \sigma(n) & \text{else} \end{cases} \] (1)

Note (1) in particular: this ensures that the effects of any previous execution of the same speculative instruction are discarded. This is exactly what is done by safe-bits; in boosting, the discarding could occur earlier.

Certain restrictions imposed on the functions that model a speculative command include:

- The number of values produced and the number of locations modified by the command are identical.

**Property 9:**
\[ \forall c' \in SpecCmd, \sigma \in State : \vec{i}' = Content(\sigma, Rd(c', \sigma)) \]
\[ \Rightarrow |Op(c', \vec{i}')| = |Swr(c', \vec{i}')| \]

- All locations modified immediately by a speculative command are are speculative. Note that this follows from the type of \( Swr \).
Property 10:
\[ \forall c' \in \text{SpecCmd}, \bar{u}' \in (\text{Value} \mid \text{Locn})^* : m' \in \text{Swr}(c', \bar{u}') \Rightarrow m' \in \text{SpecLocn} \]

- We wish to ensure that the state produced as a result of executing an instruction satisfies Prop. 2. To do so, we restrict Swr as follows:

Property 11:
\[ \forall c' \in \text{SpecCmd}, \bar{u}' \in (\text{Value} \mid \text{Locn})^* : \forall m', n' \in \text{Swr}(c', \bar{u}') : \quad M(m') = M(n') \Rightarrow m' = n' \]

Commit and Skip

The commit command moves the values produced by a speculative instruction from speculative locations to non-speculative locations. Briefly,

Definition 7:
\[ \epsilon(\alpha, \text{commit} \beta, \sigma) = \sigma' \text{, where} \]
\[ \sigma'(k) = \begin{cases} \text{MapContent}(w) & m' \in \text{SpecLocn} \land k = M(m') \\ \sigma(k) & \land \sigma(m') = < w, \beta > \\ \text{else} & \end{cases} \]

The behavior of the commit function can be divided into two parts. One is to copy certain values stored in speculative locations to non-speculative locations. The other has to deal with the fact that there could be a pointer to some speculative location stored in one of speculative locations being committed. Any such pointers are updated to point to their non-speculative location.

skip is the usual, do-nothing, instruction. Thus,

Definition 8:
\[ \epsilon(\alpha, \text{skip}, \sigma) = \sigma \]

Control Flow

Most of the commands, speculative and non-speculative, have no effect on the control flow; the next statement is executed. The other two, if and goto are also as expected. goto changes control flow unconditionally, while if does it conditionally. Note that both statements are non-speculative. This implies that the location referred to in the if command, k, is a non-speculative location.

Also, we have used succ_{Label}, the successor function on Label, mentioned above. We assume the existence of a value true, which controls the behavior of the if command.

The effect of the various instructions on control flow is summarized below:

Definition 9:
\[ N(\alpha, \text{if}(k) \beta, \sigma) = \gamma \text{, where} \]
\[ \gamma = \begin{cases} \beta & \sigma(k) = \text{true} \\ \text{succ}_{Label}(\alpha) & \text{else} \end{cases} \]
Definition 10:
\[ \mathcal{N}(\alpha, \text{goto}(\beta), \sigma) = \beta \]

Definition 11:
\[ \mathcal{N}(\alpha, c, \sigma) = \text{succ}_{\text{Label}}(\alpha) \]

A.1.7 Program Execution

We are interested in a point by point comparison of the behavior of speculative and non-speculative programs; therefore, instead of trying to define the result of execution of a program, we shall define the result of executing \( I \) commands of the program. Thus:

Definition 12:
\[ \mathcal{R}(\mathcal{P}, 0, < \sigma_0, \alpha_0 >) = < \sigma_0, \alpha_0 > \]

Definition 13:
\[ \mathcal{R}(\mathcal{P}, I, < \sigma_0, \alpha_0 >) = \]

\[
\begin{align*}
\text{let} & \quad \langle \phi, \beta \rangle = \mathcal{R}(\mathcal{P}, I - 1, < \sigma, \alpha >) \\
& c = \mathcal{P}(\beta) \\
\text{in} & \quad \langle \varepsilon(\beta, c, \phi), \mathcal{N}(\beta, c, \phi) \rangle
\end{align*}
\]

It will often be convenient to be able to extract either member of a State-Labeltuple. The following auxiliary functions will be convenient:

Definition 14:
\[ \text{Label}(< \sigma, \alpha >) = \alpha \]

Definition 15:
\[ \text{State}(< \sigma, \alpha >) = \sigma \]

Legitimate States

We had earlier restricted the set of states we considered to those that are legitimate, i.e. satisfy Prop. 1 and Prop. 2. We have, therefore, to show that executing a command in a legitimate state results in a legitimate state.

We shall first show that it is not possible for the execution of a command in a state that satisfies Prop. 1 to result in a state that violates Prop. 1, i.e.

\[ \forall c, \sigma : \]

\[
\text{let} \quad \phi = \varepsilon(\alpha, c, \sigma)
\]
\[\forall k \in \text{NonLocn} : \sigma(k) \notin \text{SpecLocn} \rightarrow \forall l \in \text{NonLocn} : \phi(l) \notin \text{SpecLocn}\]

The only commands that modify locations in \text{NonLocn} are the non-speculative commands and \text{commit}. The non-speculative commands, from Prop. 7, cannot update the state by (the address of) a speculative location. \text{commit}, as can be seen from Defn. 7 updates locations by MapContent(m), which cannot be a speculative location. The unmodified locations, of course, continue to not contain speculative locations. Consequently, if any command is executed in a state satisfying Prop. 1, the resultant state satisfies Prop. 1.

We shall now show that it is not possible for the execution of a command in a state that satisfies Prop. 2 to result in a state that violates Prop. 2, i.e.

\[\forall c, \sigma :\]

\[\begin{align*}
\phi &= \mathcal{E}(\alpha, c, \sigma) \\
\begin{align*}
\forall k', l' \in \text{SpecLocn} : (\sigma(k') &=< u, \alpha > \land \sigma(l') &=< v, \alpha > \\
\land \mathcal{M}(k') &= \mathcal{M}(l') \Rightarrow k' = l') \\
\Rightarrow \forall k', l' \in \text{SpecLocn} : (\phi(k') &=< u, \alpha > \land \phi(l') &=< v, \alpha > \\
\land \mathcal{M}(k') &= \mathcal{M}(l') \Rightarrow k' = l')
\end{align*}
\]

The only commands that can modify speculative locations are \text{commit} and speculative-commands. \text{commit} sets certain speculative locations to \bot, clearly not causing a violation of Prop. 2. Executing a speculative command, by Prop. 11, also cannot cause Prop. 2 to be violated. Hence, if a command is executed in a state satisfying Prop. 2, the resulting state also satisfies Prop. 2.

Thus, if a command is executed in a legitimate state, the state produced is also legitimate.

Pairs

Whenever we encounter a commit, we will need to refer to the instruction that it is going to commit. More precisely, we need to refer to a specific instance of that instruction. This instance is defined for commit \(\beta\), if the command at \(\beta\) was executed and no intervening commit \(\beta\) has been executed (see (2)). Further, it is the closest such execution of the instruction at \(\beta\) (see (3)). This is illustrated by Fig. A.2. The \(J\)-th command executed is the command being committed by the commit instruction at \(I\). No instruction in between can have either label \(\alpha\) or \(\beta\).

**Definition 16:**

\[\text{Pair}(P', I, <\sigma_0, \alpha_0>) =\]

\[\begin{align*}
\text{let} \\
\alpha &= \text{Label}(\mathcal{R}(P', I, <\sigma_0, \alpha_0>)) \\
\text{in} \\
\text{if}(P'(\alpha) &= \text{commit} \beta \\
\land \exists J, 0 \leq J \leq I : \beta &= \text{Label}(\mathcal{R}(P', J, <\sigma_0, \alpha_0>)) \\
\land \forall H, J < H < I : \beta \neq \text{Label}(\mathcal{R}(P', H, <\sigma_0, \alpha_0>)))
\end{align*}\]
\[ J = Pair(P', I, <\sigma_0, \alpha_0>) \]

\[ \land \forall H, J < H < I : \]
\[ \text{commit } \beta \neq P'(\text{Label}(R(P', H, <\sigma_0, \alpha_0>))) \]

\[
\begin{cases}
\text{then} & J \\
\text{else} & \bot
\end{cases}
\]

A.2 Similarity

In this section, we shall define what we mean when we say that a non-speculative program is speculatively equivalent to a speculative program. First, we shall define what we mean when we say a non-speculative command \(c\) is similar to a speculative command \(c'\), written \(c \bowtie c'\).

**Definition 17:**
\[ c \bowtie d' : c \in \text{NonCmd}, d' \in \text{SpecCmd} \Rightarrow \]
\[ (\forall \sigma, \phi' \in \text{State} : \]
\[ \text{let} \]
\[ \vec{k} = Rd(c, \sigma) \quad \vec{k}' = Rd(d', \phi') \]
\[ \vec{u} = \text{Content}(\sigma, \vec{k}) \quad \vec{u}' = \text{Content}(\phi', \vec{k}') \]
\[ \vec{v} = Op(c, \vec{u}) \quad \vec{v}' = Op(d', \vec{u}') \]
\[ \vec{l} = Nwr(c, \vec{u}) \quad \vec{l}' = Swr(d', \vec{u}') \]
\[ \text{in} \]
MapContent(\textit{Content}(\phi', \vec{k}')) = \textit{Content}(\sigma, \mathcal{M}(\vec{k}'))
\Rightarrow
\begin{align*}
\vec{k} &= \mathcal{M}(\vec{k}') \land \\
\vec{u} &= \text{MapContent}(\vec{u}') \land \\
\vec{v} &= \text{MapContent}(\vec{v}') \land \\
\vec{I} &= \mathcal{M}(\vec{m}')
\end{align*}

We use the definition of \(\Rightarrow\) listed above to define a transformation. A non-speculative program \(\mathcal{P}\) is similar under transformation to a program \(\mathcal{P}'\), written \(\mathcal{P} \sim \mathcal{P}'\) if:

\textbf{Definition 18:}
\(\mathcal{P} \sim \mathcal{P}'\) \Rightarrow
\begin{align*}
\text{dom}(\mathcal{P}) &= \text{dom}(\mathcal{P}') \land \\
\forall c \in \mathcal{P} : c \in \text{NonCmd} \land \\
\forall \alpha \in \text{dom}(\mathcal{P}') : \\
&\quad \begin{cases} \\
&\text{if}(\mathcal{P}'(\alpha) \in \text{SpecCmd}) \text{ then } \mathcal{P}(\alpha) = \text{skip} \\
&\text{else if}(\mathcal{P}'(\alpha) = \text{commit } \beta) \text{ then } \mathcal{P}(\alpha) \Rightarrow \mathcal{P}'(\beta) \\
&\text{else } \mathcal{P}(\alpha) = \mathcal{P}'(\alpha)
\end{cases}
\end{align*}

Note that, if it is possible to find a non-speculative similar instruction for every speculative instruction, the \textbf{Defn. 18} allows us, for every speculative program, to construct an non-speculative program that is similar under transformation to it.

\section{A.3 Compiler Constraints}

The compiler defines some distinguished initial status \(<\sigma_0, \alpha_0>\) in which program execution should be initiated. The compiler produces code so that it satisfies the conditions listed below if execution is initiated by executing the instruction at label \(\alpha_0\) in initial state \(\sigma_0\). Note that the initial state is assumed to be legitimate, i.e., satisfy both \textbf{Prop. 1} and \textbf{Prop. 2}.

The compiler must ensure that, if a commit is executed, then there must have been an execution of the command being committed, without an intervening commit of that command; i.e., for every commit, its \textit{Pair} must exist.

\textbf{Property 12:}
\(\forall I : <\sigma', \alpha'> = \mathcal{R}(\mathcal{P}', I, <\sigma_0, \alpha_0>) \land \mathcal{P}'(\alpha') = \text{commit } \beta \Rightarrow \\
\exists J : J = \text{Pair}(\mathcal{P}', I, <\sigma_0, \alpha_0>)
\)

The compiler must ensure that, if a command is being committed, then the values that it used must be available. In particular, if the values were stored in non-speculative locations, they must be available in the same locations, while if they were stored in the speculative locations, they must have been previously committed to the appropriate non-speculative location. Further, if one of the values read was a pointer to a speculative location, then that value must have been updated to point to some non-speculative location.

\textbf{Property 13:}
\[ \forall I:\langle \sigma', \alpha' \rangle = R(P', I, \langle \sigma_0, \alpha_0 \rangle) \land P'(\alpha') = \text{commit } \beta \Rightarrow \]
\text{let}
\[ J = \text{Pair}(P', I, \langle \sigma_0, \alpha_0 \rangle) \]
\[ \langle \phi', \beta \rangle = R(P', J, \langle \sigma_0, \alpha_0 \rangle) \]
\text{in}
\[ \forall k' \in Rd(P(\beta), \phi') : \]
\[ \text{MapContent(Content}(\phi', k') \text{)} = \text{Content}(\sigma', M(k')) \]

Finally, the compiler must ensure that the appropriate actions occur at a commit. In particular, this implies that all values that were produced by the speculative command are available in the appropriate speculative locations, and will be copied to the locations determined when the command was executed. Note that the values may be pointers to speculative locations; Some of these will have been updated to point to the equivalent non-speculative location, the others will need to be updated to point to the equivalent non-speculative location at the commit.

Property 14:
\[ \forall I:\langle \sigma', \alpha' \rangle = R(P', I, \langle \sigma_0, \alpha_0 \rangle) \land P'(\alpha') = \text{commit } \beta \Rightarrow \]
\text{let}
\[ J = \text{Pair}(P', I, \langle \sigma_0, \alpha_0 \rangle) \]
\[ \langle \phi', \beta \rangle = R(P', J, \langle \sigma_0, \alpha_0 \rangle) \]
\[ d' = P'(\beta) \]
\[ \overline{k'} = \text{Rd}(d', \phi') \]
\[ \overline{\sigma}' = \text{Content}(\phi', \overline{k'}) \]
\[ \overline{\sigma}' = \text{Op}(d', \overline{\sigma}') \]
\[ \overline{m}' = \text{Swr}(d', \overline{\sigma}') \]
\text{in}
\[ \forall H: \sigma(m'_H) = < \overline{\sigma}'_H, \beta > \land \sigma(\overline{m}'_H) = < \text{MapContent}(\overline{\sigma}'_H), \beta > \]

A.4 Speculative Equivalence

We say that two programs \( P \) and \( Q \) are speculatively equivalent with respect to initial state \( < \sigma_0, \alpha_0 > \) written \( P \equiv_{<\sigma_0,\alpha_0>} Q \) when the following holds:

Definition 19:
\[ P \equiv_{<\sigma_0,\alpha_0>} Q \iff \]
\[ \forall I:\]
\text{let}
\[ < \sigma, \alpha >= R(P, I, < \sigma_0, \alpha_0 >) \]
\[ < \sigma', \alpha' >= R(P', I, < \sigma_0, \alpha_0 >) \]
\text{in}
\[ \alpha = \alpha' \land \sigma \equiv_{\text{NonSpec}} \sigma' \]
A.5 Proof

We have to show that given a speculative program $\mathcal{P}'$, and a non-speculative program $\mathcal{P}$ that is similar under transformation to it, i.e. $\mathcal{P} \sim \mathcal{P}'$, then, if all the compiler constraints are satisfied, the programs are speculatively equivalent, i.e. $\mathcal{P} \equiv_{<\sigma_0,\alpha_0>} Q$. Thus, we have to prove:

$$\forall I:
\begin{align*}
&\text{let} \\
&\langle \sigma, \alpha \rangle = \mathcal{R}(\mathcal{P}, I, <\sigma_0, \alpha_0>) \\
&\langle \sigma', \alpha' \rangle = \mathcal{R}(\mathcal{P}', I, <\sigma_0, \alpha_0>) \\
&\text{in} \\
&\alpha = \alpha' \land \sigma \equiv_{NonSpec} \sigma' \\
\end{align*}$$

We shall proceed by induction on the value of $I$.

Base: $I = 0$:

By Defn. 12

$$\langle \sigma, \alpha \rangle = \mathcal{R}(\mathcal{P}, 0, <\sigma_0, \alpha_0>) = <\sigma_0, \alpha_0>$$
$$\langle \sigma', \alpha' \rangle = \mathcal{R}(\mathcal{P}', 0, <\sigma_0, \alpha_0>) = <\sigma_0, \alpha_0>$$

Consequently,

$$\forall k \in NonLocn : \sigma(k) = \sigma_0(k) = \sigma'(k)$$
$$\land \alpha = \alpha_0 = \alpha'$$

Induction Step:

Assume true for $I$, i.e.

$$\text{let}$$

$$\langle \sigma_I, \alpha_I \rangle = \mathcal{R}(\mathcal{P}, I, <\sigma_0, \alpha_0>)$$
$$\langle \sigma'_I, \alpha'_I \rangle = \mathcal{R}(\mathcal{P}', I, <\sigma_0, \alpha_0>)$$

in

$$\sigma_I(k) \equiv_{NonSpec} \sigma'_I(k) \land \alpha_I = \alpha'_I$$

To prove for $I + 1$

$$\langle \sigma_{I+1}, \alpha_{I+1} \rangle = \mathcal{R}(\mathcal{P}, I + 1, <\sigma_0, \alpha_0>)$$
$$\langle \sigma'_{I+1}, \alpha'_{I+1} \rangle = \mathcal{R}(\mathcal{P}', I + 1, <\sigma_0, \alpha_0>)$$

By induction hypothesis

$$\langle \sigma_I, \alpha_I \rangle = \mathcal{R}(\mathcal{P}, I, <\sigma_0, \alpha_0>)$$
$$\langle \sigma'_I, \alpha'_I \rangle = \mathcal{R}(\mathcal{P}', I, <\sigma_0, \alpha_0>)$$

By definition of $\mathcal{R}$ (Defn. 13)

$$\sigma_{I+1} = \mathcal{E}(\alpha_I, \mathcal{P}(\alpha_I), \sigma_I)$$
$$\sigma'_{I+1} = \mathcal{E}(\alpha_I, \mathcal{P}'(\alpha_I), \sigma'_I)$$

and
\[ \alpha_{I+1} = \mathcal{N}(\alpha_I, P(\alpha_I), \sigma_I) \]
\[ \alpha'_{I+1} = \mathcal{N}(\alpha_I, P'(\alpha_I), \sigma'_I) \]

Thus, it is enough to prove
\[ \mathcal{E}(\alpha_I, P(\alpha_I), \sigma_I) \equiv_{\text{NonSpec}} \mathcal{E}(\alpha_I, P'(\alpha_I), \sigma'_I) \]
\[ \wedge \mathcal{N}(\alpha_I, P(\alpha_I), \sigma_I) = \mathcal{N}(\alpha_I, P'(\alpha_I), \sigma'_I) \]

We shall split this proof into two steps. First, we shall show that \( \sigma_{I+1} \equiv_{\text{NonSpec}} \sigma'_{I+1} \) holds. Then we shall show that \( \alpha_{I+1} = \alpha'_{I+1} \) holds

**Step I: \( \sigma_{I+1} \equiv_{\text{NonSpec}} \sigma'_{I+1} \)**

By cases, based on type of \( P'(\alpha_I) \)
Let
\[ c = P(\alpha_I) \]
\[ c' = P'(\alpha_I) \]

**Case I: \( c' \in \text{SpecCmd} \)**

By **Prop. 6**, the only locations modified by \( c' \) are speculative locations
\[ \forall k \in \text{NonLocn} : \sigma'_{I+1}(k) = \sigma'_I(k) \]

From definition of \( \sim \)
\[ c = \text{skip} \]

By definition of \( \text{skip} \):
\[ \forall k \in \text{Locn} : \sigma_{I+1}(k) = \sigma_I(k) \]

By the induction hypothesis:
\[ \forall k \in \text{NonLocn} : \sigma_I(k) = \sigma'_I(k) \]

Consequently
\[ \forall k \in \text{NonLocn} : \sigma_{I+1}(k) = \sigma_I(k) = \sigma'_I(k) = \sigma'_{I+1}(k) \]

and therefore
\[ \sigma_{I+1} \equiv_{\text{NonSpec}} \sigma'_{I+1} \]

**Case II: \( c' = \text{commit} \beta \)**

First, let us compute the value of \( \sigma'_{I+1} \).

By **Prop. 12**, \( Pair(P', I + 1, < \sigma_0, \alpha_0 >) \) exists
Let
\[ J = Pair(P', I + 1, < \sigma_0, \alpha_0 >) \]
\[ < \phi', \beta >= \mathcal{R}(P', J, < \sigma_0, \alpha_0 >) \]
\[ d' = P'(\beta) \] (4)

and
\[ \bar{k}' = Rd(d', \phi') \]
\[ \bar{\omega}' = \text{Content}(\phi', \bar{k}') \]
\[ \bar{\omega}' = \text{Op}(d', \bar{\omega}') \]
\[ \bar{\omega}' = \text{Swr}(d', \bar{\omega}') \]

From the definition of \( \text{commit} \), Defn. 7
\forall n : \sigma'_i+1(n) = \begin{cases} 
\text{MapContent}(w) & m' \in \text{SpecLocn} \land n = \mathcal{M}(m') \\
\sigma'_i(n) & \text{else}
\end{cases} \\
\wedge \sigma'_i(m') = \langle w, \beta \rangle 

By Prop. 14

\forall H : \sigma'_i(\vec{m}'_H) = \langle \vec{v}'_H, \beta \rangle \lor \sigma'_i(\vec{m}'_H) = \langle \text{MapContent}(\vec{v}'_H), \beta \rangle

Further, from the definition of speculative commands, Defn. 6, all locations with previous value of \langle w, \beta \rangle were set to \bot when \vec{d}' was executed. From the definition of Pair, Defn. 16, no intervening execution of a command with label \beta can have been executed. Therefore, these are the only locations s.t. \sigma'_i(m') = \langle w, \beta \rangle. Thus:

\forall H : n = \mathcal{M}(\vec{m}'_H) \Rightarrow
\sigma'_i+1(n) = \text{MapContent}(\text{Content}(\sigma'_i, \vec{m}'_H)) \\
\in \{ \text{MapContent}(\vec{v}'_H), \text{MapContent}(\text{MapContent}(\vec{v}'_H)) \}

and therefore

\forall H : n = \mathcal{M}(\vec{m}'_H) \Rightarrow \sigma'_i+1(n) = \text{MapContent}(\vec{v}'_H)

from which it follows that

\forall n : \sigma'_i+1(n) = \begin{cases} 
\text{MapContent}(\vec{v}'_H) & n = \mathcal{M}(\vec{m}'_H) \\
\sigma'_i(n) & \text{else}
\end{cases}

Now, let us compute the value of \sigma_i+1.

From definition of \sim \triangleright

\overrightarrow{c \Rightarrow \vec{d}'}

let

\vec{k} = Rd(\vec{c}, \sigma_i+1) \\
\vec{u} = \text{Content}(\sigma_i+1, \vec{k}) \\
\vec{v} = \text{Op}(\vec{c}, \vec{u}) \\
\vec{l} = Nwr(\vec{c}, \vec{u})

From Prop. 14

\forall H : \text{MapContent}(\text{Content}(\phi'_i, \vec{k}'_H)) = \text{Content}(\sigma'_i, \mathcal{M}(\vec{k}'_H))

where \phi'_i is the state at the execution of the instruction being committed, from (4).

Further, by the induction hypothesis:

\forall k \in \text{NonLocn} : \sigma_i(k) = \sigma'_i(k)

Consequently, since

\forall k' : \mathcal{M}(k') \in \text{NonLocn}

from Prop. 13, it follows

\forall H : \text{MapContent}(\text{Content}(\phi'_i, \vec{k}'_H)) = \text{Content}(\sigma_i, \mathcal{M}(\vec{k}'_H))

Now, by the definition of \sim \triangleright, Defn. 17:

\text{MapContent}(\text{Content}(\phi'_i, \vec{k}'_i)) = \text{Content}(\sigma_i, \mathcal{M}(\vec{k}'_i))

\Rightarrow

\vec{k}' = \mathcal{M}(\vec{k}'_i) \wedge \\
\vec{u}' = \text{MapContent}(\vec{u}'_i) \wedge \\
\vec{v}' = \text{MapContent}(\vec{v}'_i) \wedge \\
\vec{l}' = \mathcal{M}(\vec{m}'_i)
Consequently
\[
\bar{l}_H = M(\bar{m}'_H) \\
\bar{v}_H = MapContent(\bar{v}'_H)
\]

By the definition of non-speculative commands, Defn. 5
\[
\forall n : \sigma_{I+1}(n) = \begin{cases} 
\bar{v}_H = MapContent(\bar{v}'_H) & n = \bar{l}_H = M(\bar{m}'_H) \\
\sigma_I(n) & \text{else}
\end{cases}
\]

By the induction hypothesis:
\[
\forall k \in NonLocn : \sigma_I(k) = \sigma'_I(k)
\]

Consequently
\[
\forall n \in NonLocn : \sigma_{I+1}(n) = \begin{cases} 
\bar{v}_H = MapContent(\bar{v}'_H) & n = \bar{l}_H = M(\bar{m}'_H) \\
\sigma_I(n) = \sigma'_I(n) & \text{else}
\end{cases}
\]

which is the same as for \(\sigma'_{I+1}\).

Thus, we can conclude:
\[
\forall n \in NonLocn : \sigma_{I+1}(n) = \sigma'_I(n)
\]

and therefore
\[
\sigma_{I+1} \equiv _{NonSpec} \sigma'_{I+1}
\]

Case III: \(c' \in NonCmd\)

let
\[
\bar{k}' = Rd(c', \sigma'_I) \\
\bar{u}' = Content(\sigma'_I, \bar{k}') \\
\bar{v}' = Op(c', \bar{u}') \\
\bar{l}' = Nwr(c', \bar{u}')
\]

From Defn. 5
\[
\forall k : \sigma'_{I+1}(k) = \begin{cases} 
\bar{v}'_H & k = \bar{l}'_H \\
\sigma'_I(k) & \text{else}
\end{cases}
\]

By definition of \(\sim\)
\[
c = c'
\]

let
\[
\bar{k} = Rd(c, \sigma_I) \\
\bar{u} = Content(\sigma_I, \bar{k}) \\
\bar{v} = Op(c, \bar{u}) \\
\bar{l} = Nwr(c, \bar{u})
\]

By the induction hypothesis
\[
\forall k \in NonLocn : \sigma_I(k) = \sigma'_I(k)
\]

From this it is clear that both \(c\) and \(c'\) will read the same values if they read from the same non-speculative locations. By Prop. 3, they only read from non-speculative locations. Further, from Prop. 8, it can be shown that, since the two commands are identical, they will read exactly the same sequence of values. Therefore:
\[
\bar{k} = \bar{k}' \\
\bar{u} = \bar{u}' \\
\bar{v} = \bar{v}'
\]
\[ \tilde{I} = \tilde{I} \]

Thus
\[
\forall k \in \text{NonLocn} : \sigma_{I+1}(k) = \begin{cases} 
\tilde{v}_H = \tilde{v}'_H & k = \tilde{k}_H = \tilde{k}'_H \\
\sigma_I(k) = \sigma'_I(k) & \text{else}
\end{cases}
\]

Consequently
\[
\forall k \in \text{NonLocn} : \sigma_{I+1}(k) = \sigma'_I(k)
\]

and therefore
\[
\sigma_{I+1} \equiv_{\text{NonSpec}} \sigma'_I
\]

Thus, we have shown that, for all \( P(\alpha_I) \) and \( P'(\alpha_I) \)
\[
\sigma_{I+1} \equiv_{\text{NonSpec}} \sigma'_I + 1
\]

**Step II:** \( \alpha_{I+1} = \alpha'_I \)

As shown above, to prove \( \alpha_{I+1} = \alpha'_I \), given the induction hypothesis it is enough to show that
\[
N(\alpha_I, P(\alpha_I), \sigma_I) = N(\alpha_I, P'(\alpha_I), \sigma'_I)
\]

Again, we shall proceed by cases, this time distinguishing the if and goto commands.

**Case I:** \( P'(\alpha_I) = \text{goto}(\beta) \)

By the definition of \( \mathcal{N} \)
\[
\mathcal{N}(\alpha_I, \text{goto}(\beta), \sigma'_I) = \beta
\]

Since \( \text{goto}(\beta) \) is non-speculative, by the definition of \( \rightarrow \)
\[
P(\alpha_I) = \text{goto}(\beta)
\]

And
\[
\mathcal{N}(\alpha_I, \text{goto}(\beta), \sigma_I) = \beta
\]

Thus
\[
\mathcal{N}(\alpha_I, P'(\alpha_I), \sigma'_I) = \mathcal{N}(\alpha_I, P(\alpha_I), \sigma_I)
\]

**Case II:** \( P'(\alpha_I) = \text{if}(k)\beta \)

By the definition of \( \mathcal{N} \)
\[
\mathcal{N}(\alpha_{I+1}, \text{if}(k)\beta, \sigma'_I) = \begin{cases} 
\beta & \sigma_I'(k) = \text{true} \\
\text{succ}_{\text{Label}}(\alpha_{I+1}) & \text{else}
\end{cases}
\]

Since \( \text{if}(k)\beta \) is non-speculative, by the definition of \( \rightarrow \)
\[
P(\alpha_I) = \text{if}(k)\beta
\]

By the definition of \( \mathcal{N} \)
\[
\mathcal{N}(\alpha_{I+1}, \text{if}(k)\beta, \sigma_I) = \begin{cases} 
\beta & \sigma_I(k) = \text{true} \\
\text{succ}_{\text{Label}}(\alpha_{I+1}) & \text{else}
\end{cases}
\]

By the induction hypothesis
\[
\forall k \in \text{NonLocn} : \sigma_I(k) = \sigma'_I(k)
\]

Consequently, since \( k \in \text{NonLocn} \)
\[
\mathcal{N}(\alpha_{I+1}, \text{if}(k)\beta, \sigma'_I) = \begin{cases} 
\beta & \sigma'_I(k) = \sigma_I = \text{true} \\
\text{succ}_{\text{Label}}(\alpha_{I+1}) & \text{else}
\end{cases}
\]

And
\[
\mathcal{N}(\alpha_I, P'(\alpha_I), \sigma'_I) = \mathcal{N}(\alpha_I, P(\alpha_I), \sigma_I)
\]

**Case III:** \( P'(\alpha_I) = \text{some other instruction} \)

By the definition of \( \rightarrow \) if \( c' = P'(\alpha_I) \) is not a goto or a if, neither is \( c = P(\alpha_I) \). Therefore, both \( c \)
and $c'$ must execute the next instruction, i.e.

$$N(\alpha_I, P'(\alpha_I), \sigma'_I) = \text{succ}_{\text{Label}}(\alpha_I)$$

$$= N(\alpha_I, P(\alpha_I), \sigma_I)$$

Thus, we have shown:

$$N(\alpha_I, P(\alpha_I), \sigma_I) = N(\alpha_I, P'(\alpha_I), \sigma'_I)$$

and therefore

$$\alpha_{I+1} = \alpha'_{I+1}$$

**Conclusion**

It follows, by induction, that the claim is true for all $I$, i.e.

$$\forall I :$$

let

$$< \sigma, \alpha_{I+1} > = R(P, I, < \sigma_0, \alpha_0 >)$$

$$< \sigma', \alpha_{I+1}' > = R(P', I, < \sigma_0, \alpha_0 >)$$

in

$$\sigma \equiv_{\text{NonSpec}} \sigma' \wedge \alpha_{I+1} = \alpha_{I+1}'$$

Thus, we have shown that, if the compiler constraints are satisfied for programs executing with initial state $< \sigma_0, \alpha_0 >$ then,

$$P \Rightarrow P' \Rightarrow \equiv_{< \sigma_0, \alpha_0 >} Q$$
Appendix B

Implementation of Whole-DAG Scheduling

In Chapter 5, we had described our algorithm, without giving concrete details of how certain aspects of the algorithm were realized. This was because the implementation we used relied intimately on the way DFGs represented information. In this appendix, we shall provide those details. We assume that the reader is familiar with the details of dependence flow graphs (DFG), single-entry single-exit regions (SESE), and the program structure tree (PST).

Broadly, each function in the program is completely processed before going to the next one. For each function, each of its DAGs is isolated, scheduled and completely simulated before processing the next one. This process is outlined in Fig. B.1. Our description shall follow this order.

B.1 Preprocessing Functions

The first step is to pre-process the function so as to convert the program into a form which makes scheduling easier, and then to isolate the DAGs that will be scheduled and simulated.

```c
for each function in program {
    preprocess function
    for( each acceptable DAG in function ) {
        compute EARLY/LATE
        schedule DAG
        simulate scheduled DAG
    }
}
```

Figure B.1: Whole-DAG Scheduling and Simulation: Top-Level Structure
B.1.1 Splitting Control-Edges

The control structure of the program is modified so that there is no control-flow edge that is both a member of a fork and a join. To accomplish this, we add dummy basic blocks to the control-flow graph. This allows us to always be able to move code up past both sides of a join, without worrying about the possibility of introducing it on a path where it was not already present.

Note that all branches in the control-flow graph as presented to us are either unconditional, or are 2-way conditional branches. Consequently, the number of edges in the graph is at most twice the number of instructions; even if we split every edge in the graph, we would no more than quadruple its size.

B.1.2 Isolating DAGs

After the control flow graph has been altered, the maximal acyclic SESE regions in the graph are identified. An SESE is maximally acyclic if it is itself acyclic, and is not contained in another acyclic SESE region. Such regions are easily found by a top-down walk of the PST, as shown in Fig. B.2.

These acyclic SESEs are then examined for control structure. Basically, if they have a conditional branch (a switch in DFG parlance), then they are considered interesting.

However, this is not enough. Our simulation technique evaluates each path through the DAG individually. If the DAG contains too many control-flow paths, the simulation process will take too long. So, what we do is enumerate the number of paths, and reject the DAG if the number crosses 1000. Note that this is not a restriction on the scheduling algorithm; it is a constraint imposed on the selection process to control the time taken by the simulation process.

B.2 Computing EARLY and LATE

The scheduling heuristics use a metric called LATE. Determining it requires that we first compute a fairly similar metric called EARLY. In this section, we shall describe how a LATE and EARLY value is computed for each instruction.

Note that this the computation of LATE is done once before scheduling, by means of a single pass through the DFG. These numbers are never re-evaluated for an instruction unless the instruction is replicated or moved in the graph. In that case, updating the value of LATE for the instruction can be done by examining the instructions neighbors in the DFG; it is not necessary to walk the graph again.

B.2.1 Computing Early

The EARLY value of an instruction is the instruction’s distance from the beginning of the DAG. In the dependence-flow graph, it can be found by weighting all dependence out-edges with the latency of their source instruction, and then finding the longest such path in the graph to the instruction. The
find_sese_regions(PST_Node node, Queue DAG_Q)
{
    DAG * dag;
    if( node is acyclic && node is not empty &&
        ( node has no sequentially composed predecessor ||
        predecessor of node is cyclic ) ) {
        /* found the beginning of a sequentially composed chain */
        dag = NULL;
        add node to dag;
        while( sequentially composed successor exists ) {
            node = sequentially composed successor of node;
            add node to dag;
        }
        add dag to DAG_Q;
    }
    else {
        /* go look in the children of the node */
        for( child = child of node in PST ) {
            find_sese_regions(child, DAG_Q);
        }
    }
}

Figure B.2: Maximal Acyclic SESE
weight on an edge joining two instructions is also known as the distance between them. Similarly, the distance between two instructions is the sum of the weights on the longest path joining them.

This is illustrated by the example in Fig. B.3. START has an EARLY value of 0, and a latency of 0. Every other instruction has an EARLY value that is the maximum of the sum along the paths to the instruction. In particular, I depends only on START, so it has an EARLY of 0. I has latency 2, so any instruction that depends on it must have an EARLY of at least 2. In particular J depends only on I, and so has EARLY of 2. K depends on both I and J. The longest path to K is through I and J with total length 3. Thus K has an EARLY of 3. We can also say that the distance from I to K is 3.

Anti, Output and Flow

The rationale behind EARLY is that it represents the earliest point in the schedule that an instruction can be scheduled. Thus, by weighting each dependence edge with the latency of its source instruction, we are in effect saying that any instruction that depends on the source cannot be issued till the source instruction completes. This is clearly true for both output and flow dependencies. They are, therefore, weighted by the latency of the instruction.

However, if two instructions are joined by an anti dependence, then the dependent instruction is not waiting for the instruction to complete, just for it to be issued. Thus, it can be issued in the cycle after the source instruction is issued, and consequently its weight should be 1. These weights or distances are summarized in Fig. B.4.

Merges

In a DAG, an instruction can have different distances from START for different control-flow paths through the DAG. One must somehow combine these distances. One approach is to ignore the
DIST(START, I) = 0 \quad \forall \text{ insns I that are successors of START}
DIST(I, J) = \text{latency}(I) \quad \forall \text{ insns J that are flow dependent on I}
DIST(I, J) = \text{latency}(I) \quad \forall \text{ insns J that are output dependent on I}
DIST(I, J) = 1 \quad \forall \text{ insns J that are anti dependent on I}
DIST(S, J) = 0 \quad \forall \text{ insns J that depend on switch S}
DIST(I, M) = 0 \quad \forall \text{ merges M that depend on instruction I}
DIST(I, END) = \text{latency}(I) \quad \forall \text{ insns I on which END depends}

Figure B.4: DFG Edge Weights for EARLY

![DFG Edge Weights](image)

Figure B.5: EARLY using def-use chains

fact that the effect of control-flow and continue to take the maximum of all dependence-paths from START to the instruction. However, if an EARLY for an instruction is very large on a very low probability path, it will lead to bad choices when we use that value in our heuristic. Instead, we would like to use a value of EARLY which takes the probability into account. One possible way of computing a probability-weighted EARLY for an instruction is to compute the distance from START separately for each of the control-flow paths on which the instruction lies, and then take a weighted average of the values, where the weight is the probability of each path.

Unfortunately, with a def-use chain based dependence representation, there is no connection between the dependence and the control structure representation of the program. Consider Fig. B.5 Assume that I, J and K are each executed 50% of the time, and have an EARLY value of 0. From the dependence edges shown in Fig. B.5(a), we cannot determine the control-flow structure of the program. and therefore we cannot determine how we should average the EARLY values. Depending on the control structure, the average EARLY value should be 13 (Fig. B.5(b)) or 8 (Fig. B.5(c)).

In the DFG representation, however, dependence edges that cross basic block boundaries\(^1\) are

\(^1\)Actually, its more complicated than that; bypassing allows us to avoid considering dependence edges where there is no need for them. [27] contains the details.
Figure B.6: EARLY using DFG

The Data-Flow Equations

The definition of EARLY we posited in the last section is impractical. It could involve enumerating, for each instruction, all of the control flow paths in a DAG, separately computing the EARLY value for the instruction, and then averaging it. Since the number of control flow paths in a DAG is potentially exponential in the size of the DAG, we clearly cannot use this definition directly, and would like to come up with an approximation.

The approximations we make are very simple, and are an immediate consequence of the structure of the DFG. Basically:

- The EARLY value of a node\(^2\) other than a merge is found by first computing, for each of its predecessors, the sum of EARLY for the predecessor node and distance between the two nodes, and then using the maximum such value.

- For a merge node, we first compute the value for each of the control flow directions separately, i.e., for each control-flow direction being joined, we find the maximum EARLY plus distance of all edges entering the merge from that direction. We then compute the weighted average of these maximums, where the weight is the relative probability of each of the directions.

It is possible to compute the EARLY values for each instruction by solving the data-flow equations shown in Fig. B.7. We shall not discuss how these equations can be implemented efficiently; see [27] for details. In a DAG, solving these equations involves visiting each dependence-edge exactly once in a forward walk of the DAG. The order in which the edges are visited forms a topological sort of the edges; i.e., no edge is visited until all its predecessor edges have been visited, and the value of EARLY has been computed for all preceding nodes.

---

\(^2\)Remember, a node in a DFG may be an instruction, a switch or a merge node
EARLY(START) = 0
EARLY(J) = max(EARLY(N)+DIST(N,J))
\forall nodes N on which instruction J depends
EARLY(M) = \sum_i \text{prob}(B_i) \times (max(EARLY(N_i) + DIST(N_i,M)))
\forall basic-blocks B_i that are predecessors of merge M
\text{where } N_i \text{ are nodes in } B_i \text{ on which } M \text{ depends.}
EARLY(S) = max(EARLY(N)+DIST(N,S))
\forall nodes N on which switch S depends
EARLY(END) = max(EARLY(N)+DIST(N,END))
\forall nodes N on which END depends

Figure B.7: EARLY: Data Flow Equations

B.2.2 Computing LATE

LATE is the exact converse of EARLY; where EARLY of an instruction is the path-averaged distance from START, LATE of the instruction is its path-averaged distance to END. It will be computed analogously, by solving a data-flow equation. However, the solution will involve a backward walk through the DAG, in which every edge is processed once, in some reverse topological order. There are some subtleties in the computation of LATE that were not present in the computation of EARLY, which shall be detailed in this section.

EARLY and branches

LATE is a measure of the distance of an instruction to END along the dependence edges. Now, no instruction has a data-dependence from branches; thus, if we take only data-dependences into account, a branch is not on the critical path, and can be scheduled as close to END as desired. Correspondingly, any instructions that are used solely to compute the conditional for a branch can also be executed fairly close to END.

However, we have to account for the effect of control dependence. Consider the situation without speculation. If the execution of a branch gets delayed, then the execution of all instructions which are control-dependent on the branch will also get delayed. Therefore, branches and the instructions that compute the conditionals for those branches should be scheduled early. The question is, how soon before END should we execute a branch, so as to take into account the effects of control dependence?

Assume that the DAG could be scheduled so that all instructions would execute as soon as possible. Then, on the average, an instruction would be executed EARLY cycles after the start of the computation. Further, the distance between the execution of the instruction and END would be the difference between their EARLY values. This is the value we use for the LATE of a branch: the difference between the EARLY of END and the EARLY of the branch.
\[ \text{LATE(START)} = \max(\text{LATE(N)}+\text{DIST(N,START)}) \]
\[ \forall \text{ nodes N which depend on START} \]
\[ \text{LATE(BR)} = \text{EARLY(END)} - \text{EARLY(BR)} \]
\[ \forall \text{ branch instructions BR} \]
\[ \text{LATE(I)} = \max(\text{LATE(N)}+\text{DIST(I,N)}) \]
\[ \forall \text{ nodes N which depend on instruction I, I not a branch} \]
\[ \text{LATE(M)} = \max(\text{LATE(N)}+\text{DIST(M,N)}) \]
\[ \forall \text{ nodes which depend on merge M} \]
\[ \text{LATE(S)} = \sum_i \text{prob}(B_i)\left(\max(\text{LATE}(N_i) + \text{DIST}(N_i,S))\right) \]
\[ \forall \text{ basic-blocks } B_i \text{ that are successors of the fork} \]
\[ \text{where } N_i \text{ are nodes in } B_i \text{ which depend on switch } S \]
\[ \text{LATE(END)} = 0 \]

Figure B.8: LATE: Data Flow Equations

Switches

Just as we combined EARLYs values from different control flow paths at joins by using merge nodes, so, while computing LATE, we combine LATE values from different control flow paths at a fork by using switch nodes. This can be seen in the data-flow equations we use to compute LATE, shown in Fig. B.8. Note that it is the existence of both switch and merge nodes makes it possible to solve both of these problems in a symmetric fashion.

B.3 Scheduling DAGs

Most of the details of scheduling DAGs were covered in Chapter 5. In this section, we shall skimp on those parts, and focus on the areas not previously covered. In particular, we examine how we determine when an instruction is ready, and how we compute LATE for instructions that are moved or replicated.

B.3.1 Determining Readiness

An instruction will become ready only after all its predecessors have been scheduled. At this point, it is possible to compute the exact cycle at which the instruction will become ready. So, basically, every time we schedule an instruction, we examine all its data-dependent successors. If, for any successor, this instruction was the last unscheduled predecessor, we compute the READY value for that successor. The READY value of an instruction represents the earliest cycle at which the instruction can be scheduled. It can be found by adding the SCHED time (i.e. the cycle at which actually scheduled) of all predecessors to the distance between the predecessor and the instruction, and then taking the maximum such value. This process is sketched in Fig. B.9.
sched_node(PST_Node sched, int cyle)
/* sched is to be scheduled this cycle */
{
/* Book-keeping, code-motion and other actions */

SCHED(sched) = cycle;
for( node in descendants of sched ) {
    if( all predecessors of node have been scheduled ) {
        ready = 0;
        for( pred in predecessors of node ) {
            ready = max(ready, SCHED(pred)+DIST(pred, node))
        }
    READY(node) = ready;
    if( node is switch or node is merge ) {
        /* schedule node */
        SCHED(node) = ready;
        set_ready_successors(node, ready);
    } else {
        add node to appropriate ready queue.
    }
}
}

Figure B.9: Computing READY

Note that we handle switches and merges specially. If an instruction is ready, all other instructions that depend on it may become ready. These instructions may be in different basic blocks, in which case the there will be an intervening DFG node such as a switch or a merge. So, if scheduling the instruction makes the switch or merge node ready, we immediately start processing all successors of the node for readiness. We had mentioned in Chapter 5 that an instruction is not considered ready till all its predecessors on all control-paths were ready. As can be seen from Fig. B.9, we do not consider a merge node ready till all its predecessors on all control paths are ready; thus, any instruction that depends on the node will be examined only after the merge node is ready, and, consequently, the instructions predecessors on all paths are themselves ready.

B.3.2 Recomputing LATE

During the scheduling process, instructions are moved and/or replicated. As a consequence, their LATE values may need to be recomputed. Note that when an instruction is moved up through
a join, its LATE value remains the same. Consequently, the LATE value of any instructions (as opposed to merge nodes) that it depends on also remain untouched. Similarly, when an instruction is moved up through a branch, its LATE value remains the same. When an instruction is moved up through a branch, it is being speculated. Consequently, there are no additional dependences introduced, to instructions on the other side of the branch. Thus, its path(s) to end remain the same, and consequently its LATE value. Further, since the only time an instruction is moved through a switch is when it is being scheduled, all of its predecessors must already have been scheduled. There is no need to update their LATE values.

The only time an instruction’s LATE value needs to be updated is when an instruction is moved down past a branch, and replicated on both sides. As shown in Fig. B.10, instead of there being two paths to end, that need to be merged at the switch nodes, there are two instructions, each with one path. Updating the LATE values is fairly trivial; it merely requires examining the immediate successors of the node, and applying the equations given in Fig. B.8.

When moving multiple instructions below a branch, we first move those instructions that do not depend on any instruction that is to be moved. After moving those, and recomputing their LATE values, we then move those instructions that do not depend on any of the remaining instructions to be moved, and so on. This bottom-up order of moving instructions ensures that all LATE values will be correctly updated.

B.4 Best-3 and Tags

In this section, we shall briefly argue why the speculations performed using the best-3 strategy can be implemented using 3 tags, without requiring any spills. As we had mentioned in Chapter 6, there are two reasons why a best-N strategy might require more than N tags for some basic block:

Duplicate speculation

Live-through

We shall argue that neither of these cases can arise with the best-3 strategy in the current implementation.
Figure B.11: Duplicate Speculation: I

B.4.1 Duplicate Speculation

For duplicate speculation to arise, an instruction must be speculated past two branches. This is the situation shown in Fig. B.11. Note that $I_1^*$ will succeed if the both branches are true, while the $I_2^*$ will succeed if the first branch is false and the second is true. As was discussed in Chapter 6, we need two tags to represent the speculation of $I$; one to represent the true-true speculations and the other the true-false speculations.

On the basis of Fig. B.11, it may appear that the best-3 strategy can give rise to duplicate speculation; simply pick blocks A, B, and C. Notice, however, that the X to B edge is both a member of a fork and a join. During the preprocessing step, we would have split this edge into two, giving rise to the DAG shown in Fig. B.12. As shown in that figure, if blocks A, B, and C had been picked, and $I$ had been speculated, it would have resulted in $I_2^*$ being copied into D. Since D is not a block picked by the best-3 selection strategy for code motion into X, the $I_2^*$ can never be moved into A. A similar situation would arise if block D, B, and C were chosen; in that case, one of the copies of the $I$ would be moved into A, and then no further. Clearly, we need to pick all 4 blocks, A, B, C, and D, before duplicate speculation becomes possible.

B.4.2 Live-through

For a speculated instruction, $I^*$, to be live through a block X, there must be some earlier block A into which the instruction was moved from its original home block H. The speculation of $I^*$ must be unresolved through X. Thus, there must be a path from A through X to H. The blocks picked for speculation into X itself, however, must not include H. This implies that there must, for the best-N strategy, be N descendants of X that are connected to X by paths not including N. Further, the best-N strategy picks the best descendants of a block. If a best-N strategy for X does not pick H, then the best-N strategy for A will pick H only if there is an alternate path from A to H (not including X) on which H has a higher probability. The smallest situation in which all these conditions can be satisfied is shown in Fig. B.13.
Note, however, that it contains 2 edges that are members of both a fork and a join. Splitting them would yield the graph shown in Fig. B.14. Now, if A picks B, H and some other descendant, possibly X, the resulting speculation as implemented would not be as shown. In particular, the speculation would not be live through X. This is a consequence of our redundancy elimination algorithm. It halts as soon as it reaches a branch that does not lead to another node. Therefore, it will not eliminate $I^*_2$, and thereby make $I^*$ live through X.

Not that if we were using a best-4 strategy, and A picked B, H, X and C, then the elimination would occur, yielding the situation shown in Fig. B.15.
Figure B.14: Live-through: II

Figure B.15: Live-through: III
Bibliography


