HARDWARE/SOFTWARE MECHANISMS FOR INCREASING RESOURCE UTILIZATION ON VLIW/EPIC PROCESSORS

by

Mikhail Smelyanskiy

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

Doctoral Committee:

Professor Edward S. Davidson, Co-Chair
Assistant Professor Scott A. Mahlke, Co-Chair
Assistant Professor Steven K.Reinhardt
Assistant Professor Dennis Sylvester
Assistant Professor Gary S. Tyson

ABSTRACT

HARDWARE/SOFTWARE MECHANISMS FOR INCREASING RESOURCE UTILIZATION ON VLIW/EPIC PROCESSORS

by

Mikhail Smelyanskiy

Cochairs: Edward S. Davidson and Scott A. Mahlke

VLIW/EPIC (Very Large Instruction Word / Explicitly Parallel Instruction Computing) processors are increasingly used in signal processing, embedded and general-purpose applications. To achieve efficient instruction schedules in order to meet the high performance demands of these applications, these processors rely on an optimizing compiler that uses aggressive optimizations, such as predication and software pipelining, to expose and exploit instruction level parallelism. To capitalize fully on the parallelism offered by these optimizations requires increasing critical processor resources, such as function units, register and memory ports, and architected registers, which is costly in terms of cycle time, power and area. To this end, this dissertation proposes three novel schemes for achieving higher processor performance by means of more efficient utilization of the existing processor resources in the context of predication and software pipelining.

We developed deterministic predicate-aware scheduling (DPAS), which can combine operations with mutually-exclusive predicates to share the same resource in the same cycle. To support DPAS, the processor pipeline is adapted to read predicates

early and discard the operations guarded under False predicates. Mutual exclusivity guarantees that runtime conflicts will never occur. The overall effect of DPAS is to use the limited existing resources more efficiently, thereby increasing the performance of the applications studied by an average 10% when resource constraints are a bottleneck.

To increase resource utilization further, we developed a powerful generalization of DPAS, called probabilistic predicate-aware scheduling (PPAS), which can assign arbitrary predicated operations to share the same resource in the same cycle. Contrary to DPAS, PPAS can result in runtime conflicts, as it allows more than one predicate of a set of combined operations to be True in the same runtime cycle. Assignment is performed in a probabilistic manner using a combination of predicate profile information and predicate analysis aimed at maximizing the benefits of sharing in view of the expected degree of conflict. The processor pipeline is further modified to detect and recover from such conflicts. By allowing more flexibility in resource sharing than DPAS, PPAS achieved an average 19% performance gain for the resource-constrained instruction schedules.

Finally, to effectively deal with the architected register pressure and code size problems in software-pipelined loops, we have developed a hardware/software mechanism called Register Queues. By decoupling an existing register space into a small set of architected registers and a large set of physical registers, register queues enable efficient software-pipeline schedules for high operation latencies with almost no increase in either architected registers or code size.

© Mikhail Smelyanskiy 2004
All Rights Reserved

To my caring parents, Inna and Naum, and my brother Dima.

ACKNOWLEDGEMENTS

I would like to thank Edward Davidson for his help and guidance during all the years of my graduate study at the University of Michigan. His wit, wisdom, patience and sense of humor helped me through the various stages of my career as a graduate student. His originality and scientific intuition as well as a remarkable faculty for the "ideal line of expression" remain a great source of inspiration to me.

I would also like to thank Scott Mahlke for his invaluable help during my last two years in Ann Arbor. His experience and in-depth knowledge of compiler optimizations made our research and brainstorming meetings very motivating and fruitful.

I wish to thank my committee, Gary Tyson, Steve Reinhardt and Dennis Sylvester, for their help and advice throughout this dissertation. I also wish to thank Tryggve Fossum for his suggestion to investigate the probabilistic scheduling technique.

While I worked on my research, my fellow graduate students provided great research discussions as well as a social environment which enriched my studying life. Special thanks to Murali Annavaram, Mike Chu, Nate Clark, Shih-Hao Hung, Kevin Fan, Manjunath Kudlur, Hien-Hsin Lee, Ramu Pyreddy, Rajiv Ravindran, Vijayalakshmi Srinivasan, Stevan Vlaovic and Hongtao Zhong. I greatly appreciate the patience and diligence with which they attended and participated in the numerous practice talks that I had to give in the course of my graduate career. I would also like to thank Denise DuPrie, the ACAL Lab administrator, for her competence and

promptness in resolving many urgent administrative issues.

I also wish to thank my best friend George Dunlap. Our running together during cold, snowy winter months and hot, humid summer days in Michigan as well as conversing on an array of subjects, from Russian cuisine to theology, greatly helped me to recuperate from the stress and tiredness of long working days. My other good friends who I was fortunate to meet during my graduate years at the University of Michigan and who I would also like to mention and thank here are Hye Sung Chun, Chad Creighton, Stephany Hitztaler, Kristi Stewart, Nicole Maturen, Joshua McKinley, Everette Robertson, Laurel Sandor, Scott Thompson and Beth Weihe. I am very grateful to them for providing a wonderful intellectual and social environment.

Last, but by no means least, I would like to thank my close family. My parents, Inna and Naum, and my brother Dima, for their unflagging and unconditional support throughout this endeavor. Their belief in me during all these years helped me to overcome frustration and depression in a sometimes tense life.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

LIST OF TABLES x

CHAPTERS

1 INTRODUCTION 1
- 1.1 Research Contributions 6
- 1.2 Dissertation Overview 12

2 DETERMINISTIC PREDICATE-AWARE SCHEDULING 13
- 2.1 Introduction 13
- 2.2 Background and Motivation 17
  - 2.2.1 Example Code Segment 19
  - 2.2.2 Applying Predicate-aware Scheduling 21
  - 2.2.3 Characteristics of Improvable Regions 24
- 2.3 Deterministic Predicate-aware Architecture 27
- 2.4 Deterministic Predicate-aware Scheduling 30
  - 2.4.1 DGG Latency Extension 33
  - 2.4.2 Deterministic Predicate-aware Unified Scheduling Algorithm 36
  - 2.4.3 'Extend-all' DPALS 40
  - 2.4.4 'First-fit' DPALS 43
  - 2.4.5 Additional Extensions for DPAMS 46
- 2.5 Performance Evaluation 49
  - 2.5.1 Benchmarks and Processor Models 50
  - 2.5.2 Evaluation Results 53
- 2.6 Summary 67

3 PROBABILISTIC PREDICATE-AWARE SCHEDULING 68
- 3.1 Introduction 68

3.2 Motivation 70
3.2.1 Example of Probabilistic Predicate-aware Modulo Scheduling 70
3.2.2 Characteristics of Improvable Regions 75
3.3 Probabilistic Predicate-aware VLIW Processor Architecture . 78
3.4 Probabilistic Predicate-aware Scheduling 82
3.4.1 DGG Latency Extension 84
3.4.2 Computing Expected Delays Due to Conflicts 87
3.4.3 Probabilistic Predicate-aware List Scheduling Extensions 105
3.4.4 Probabilistic Predicate-aware Modulo Scheduling Extensions 120
3.5 Performance Evaluation 128
3.5.1 Benchmarks and Processor Models 129
3.5.2 Evaluation Results 130
3.6 Related Work 165
3.6.1 Hardware Support for Predication 165
3.6.2 Compiler Support for Predication 167
3.7 Summary 171

4 A STORAGE MECHANISM FOR EFFICIENT SOFTWARE PIPELINING 174
4.1 Introduction 174
4.2 Prior Work 178
4.3 Register Queues 183
4.3.1 SP Scheduling Using Register Queues 187
4.3.2 Managing Queue Overflow 193
4.3.3 Further Reducing Architected Register Pressure . . . 196
4.4 Performance Evaluation 199
4.4.1 Register Queues Study for Various Load Latencies . . 201
4.4.2 Software Pipelining Using MVE, RR and RQs 202
4.4.3 Scheduling Multiple-use Lifetimes for FIFO Queues With and Without Destructive Reads 208
4.4.4 Use of Register Queues in Predicate-aware Machines . 210
4.5 Summary 216

5 CONCLUSIONS AND FUTURE DIRECTIONS 218
5.1 Summary 218
5.2 Future Directions 224

BIBLIOGRAPHY 230

LIST OF FIGURES

Figure

2.1 Example code segment 20
2.2 Data dependence graph for code segment 21
2.3 LS schedule (a) versus DPALS schedule (b) 22
2.4 IMS kernel (a) versus DPAMS kernel schedule (b) 23
2.5 Characteristics of 'dpa-improvable' regions 25
2.6 Baseline vs. deterministic Predicate-aware machine models 28
2.7 Scheduling algorithm flowchart 31
2.8 DPAS latency extension procedure 35
2.9 Baseline scheduling functions 37
2.10 Predicate-aware reservation table 38
2.11 Predicate-aware **ResourceConflict**() function 39
2.12 Example of 'extend-all' DPALS 41
2.13 Example of 'first-fit' DPALS 44
2.14 PAMS scheduling of the example in Figure 2.1 48
2.15 'Extend-all' DPALS speedup over BALS for acyclic regions only . . . 56
2.16 'First-fit' DPALS speedup over BALS for acyclic regions only 59
2.17 DPAMS speedup over BAMS for cyclic regions only 62
2.18 Overall Speedup 66
3.1 Example of if-converted loop body and its data dependence graph . . 71
3.2 Three schedules for the example code segment 72
3.3 Characteristics of 'dpa-improvable' and 'ppa-improvable' Regions . . 75
3.4 Baseline verses deterministic and probabilistic predicate-aware pipeline organizations 79
3.5 Design alternatives to dispatch conflicting operations 80
3.6 Scheduling algorithm flowchart 83
3.7 PPAS latency extension procedure 85
3.8 Predicate Relationship Graph (PRG) 91
3.9 Algorithm to eliminate redundant conditions from an execution vector 96
3.10 Example of PRG reduced to a tree 98

3.11 Generic PPALS scheduling algorithm 106
3.12 Assembly code and data dependence graph to demonstrate PPALS . 111
3.13 Schedule Reservation Table for baseline LS 111
3.14 Schedule Reservation Table for 'extend-all' PPALS 112
3.15 Algorithm to choose operation with the least combining potential . . 116
3.16 Schedule Reservation Table for 'first-fit' PPALS 118
3.17 Main PPAMS scheduling routines 123
3.18 PPAMS Scheduling of the example in Figure 3.1 for II_{static}=6 and $II_{CflDelay}$=0.42 125
3.19 'Extend-all' PPALS speedup over BALS for acyclic regions only (CDRL=1) 134
3.20 'Extend-all' DPALS and PPALS over BALS speedup for acyclic regions only (CDRL=1, cmpp latency=3) 136
3.21 'First-fit' PPALS speedup over BALS for acyclic regions only (CDRL=1) 141
3.22 'First-fit' DPALS and PPALS speedup over BALS for acyclic regions only (CDRL=1 for PPALS) 142
3.23 PPAMS over BAMS speedup for cyclic regions only (left bar has cmpp latency=1, middle has 2, right has 3) 145
3.24 DPAMS and PPAMS speedups over BAMS for cyclic regions only (cmpp latency=3) 150
3.25 Overall speedup of PPAS over baseline for cmpplat=1, 2 and 3 156
3.26 Overall speedups of DPAS and PPAS over baseline (cmpplat = 3) . . 157
3.27 Speedup of PPAS over baseline that correspond to training and reference input sets (left bar corresponds to acyclic regions speedup, middle bar to cyclic, and right bar to overall) 159
3.28 Overall speedups of the 6-wide baseline (left bar) and the 4-wide PPAS with cmpplat=3 and CDRL=1 (right bar), relative to the 4-wide baseline 161
4.1 Software pipeline example. This sample program adds elements of a floating-point array and stores the sum in a scalar. Shown are multiple iterations of the loop with an initiation interval of 2 cycles ($II = 2$). . 178
4.2 Microarchitectural extensions to support RQs for a machine with 32 architected registers, n queues of length 4, and 256 physical registers. 183
4.3 Software pipeline schedule for the sample code (see Figure 1) using RQs 189
4.4 Software pipeline schedule with queue overflow (*fload* latency = 11). . 194
4.5 Software pipeline schedule with architected register pressure (*fadd* latency = 1). 197
4.6 Loop statistics 202
4.7 A study of RR, MVE and RQs schemes for machine model 2 (with limited resources). 203
4.8 A study of RR, MVE and RQs schemes for machine model 1 (with unlimited resources). 204
4.9 Histogram of the number of multi-instance variables in a loop. 206

4.10 Histogram of the number of instances of variables containing multiple live instances (averaged over all loops). 207
4.11 Average queue size as latency increases for machine models 1 (limited resources) and 2 (unlimited). 207
4.12 Number of uses for each instance of variables with multiple instances. 209
4.13 Histogram of the number of live-in values 211
4.14 Architected register (RQs vs. RR) requirements for DPAMS and PPAMS with cmpp latency 3 and CDRL=1 213
4.15 Architected register requirements (RQs vs. RR) for DPAMS and PPAMS with cmpp latencies 1, 3 and 5 (CDRL=1 for PPAMS) 215

LIST OF TABLES

Table

2.1 Description of a sample processor with a fetch/execute width of 3 operations . . . 19
2.2 Benchmark set (MediaBench [33]) . . . 50
2.3 Scheduling headroom estimates for the deterministic predicate-aware schedulers . . . 53
3.1 Scheduling headroom estimates for the probabilistic predicate-aware schedulers . . . 131
3.2 Schedule length achieved by BALS, 'extend-all' DPALS and 'extend-all' PPALS (CDRL=1) . . . 138
3.3 Schedule length achieved by BALS, 'first-fit' DPALS and 'first-fit' PPALS (CDRL=1) . . . 144
3.4 Schedule length achieved by BAMS, DPAMS and PPAMS for cmp-plat=3 and CDRL=1 . . . 151
3.5 Pipeline utilization statistics per predicated operation for baseline and probabilistic predicate-aware processors (using 'first-fit' PPALS and PPAMS) . . . 163

CHAPTER 1

INTRODUCTION

VLIW/EPIC (Very Large Instruction Word / Explicitly Parallel Instruction Computing) processors are increasingly used for signal processing, embedded and general-purpose applications. In order to meet the high performance requirements of these demanding applications, VLIW/EPIC processors rely on an optimizing compiler which tries to expose the inherent instruction level parallelism in an application and then capitalize on the resulting parallelism to achieve an efficient instruction schedule.

It is well known that there is generally insufficient instruction level parallelism within individual basic blocks, but higher levels of parallelism can be obtained by exploiting the increased instruction level parallelism provided among several successive basic blocks. In straight line code, for example, trace scheduling is used to merge several basic blocks along a frequent execution path into a single enlarged block [17, 24]. These blocks can be further enlarged by using predicated execution to merge several control paths [35]. Similarly, in loop code, software pipelining [22] extends the scope of compilation beyond one basic block by overlapping the execution of consecutive loop iterations [7, 8, 22, 32, 42, 43]. Predication can also be applied to loop code prior to software pipelining in order to merge blocks across conditional statements found

within a loop iteration [10, 41].

To take full advantage of the parallelism offered by these optimizations requires increased processor resources. For example, when predication merges several control paths together, the resource requirements (function units, register file ports, etc.) are summed over all merged paths. Increases in the number of function units in a VLIW/EPIC processor, in conjunction with growing wire delays (relative to logic delays) [2, 38, 39], makes the already critical single centralized structures, such as register files and bypass interconnect, an even more critical factor in determining the cycle time, area and power dissipation of the processor. For example, for N identical function units, the delay of the register file grows as $N^{3/2}$, the area as N^3, and the power dissipation as N^3 [47].

The register bypass, which is used to forward data from one function unit to another for a subsequent use, also becomes a more critical resource as the number of function units grows. Palacharla [39] showed that the delay of the bypass logic grows quadratically with increased function units. Bypass complexity also grows quadratically with the number of function units: if instructions are simultaneously issued to N identical 2-input function units with s pipe stages after execution to bypass from, then a fully-bypassed design would require $2 \times N^2 \times s$ separate bypass connections. In other words, in spite of the growing numbers of transistors available to architects, it is becoming increasingly difficult to design large centralized structures that help to exploit instruction level parallelism without compromising clock speed and increasing design complexity, and thus limiting the scalability of processors into the future.

Clustering is often proposed by research and industry projects [4, 5, 14, 18, 56] as a way to achieve higher processor performance by overcoming the bottlenecks

posed by centralized processor resources. In clustered microarchitecture key processor resources are distributed across multiple clusters, each of which contains a subset of the register files and function units, together with intra-cluster bypassing. As a result of decreasing the size and bandwidth requirements of the register files, the access times of these cycle-time critical structures are greatly reduced. The simplification of these structures also reduces their design complexity. The primary disadvantage of clustered microarchitectures is their reduced number of instructions per cycle (IPC) compared to a centralized design with the same total resources; their IPC is lowered due to the relatively slow intercluster communication between dependent instructions that are executed on different clusters. Although originally designed to overcome the scalability problems of centralized resources, clustered microarchitectures introduce scalability problems of their own: as more processors are added to the cluster or as more clusters are added, the longer it takes to communicate between more distant clusters.

This dissertation addresses the issue of how to improve processor performance in the context of predicated execution without increasing the processor's critical resources. Two predicate-aware scheduling schemes are proposed along with some changes to pipeline organization that achieve higher performance on predicated code by means of more efficiently utilizing existing processor resources.

First, a deterministic predicate-aware scheduling (DPAS) scheme is developed that assigns multiple mutually-exclusive predicated operations to share the same resource in the same cycle, thereby appearing to "oversubscribe" that resource in that cycle. To support DPAS, the processor pipeline is extended to read the predicates of these operations early, and discard (i.e. nullify) those operations that are guarded under False predicates. The mutual-exclusiveness property of a set of "combined" opera-

tions, which share the same resource in the same cycle, guarantees that at most one of their predicates will be True in a given cycle at runtime. Hence, at most one of these operations will actually require the resource during that cycle and runtime conflicts will never occur. The overall effect of DPAS is to use limited existing resources more efficiently, thereby increasing performance when resource constraints are a bottleneck.

Second, to further increase resource utilization, a powerful generalization of DPAS, called probabilistic predicate-aware scheduling (PPAS) is developed, which can assign arbitrary predicated operations to share the same resource in the same cycle. Contrary to DPAS, with PPAS more than one of these predicates may be True in a given runtime cycle, thus resulting in runtime conflicts. The processor pipeline must be modified to detect and recover from such conflicts. Operation scheduling is performed in a probabilistic manner using a combination of predicate profile information and predicate analysis aimed at maximizing the benefits of resource sharing in view of the expected degree of conflict. By allowing more flexibility in resource sharing than DPAS, PPAS gains additional performance on resource-constrained instruction schedules.

After predication has exposed sufficient instruction level parallelism, the compiler can then successfully hide operation latencies by efficiently scheduling the code while enforcing the resource constraints of the machine and the dependence constraints among the operations. However, schedules exhibiting high concurrency generally result in higher register requirements. Register pressure is further aggravated by the fact that as clock rates increase, pipelines become deeper and deeper, leading to increases in instruction latencies. Hiding longer latencies requires higher concurrency and leads to longer register lifetimes which is compounded with the higher concurrency and leads to more overlap between lifetimes, which results in increased register

pressure.

Such register pressure and its related problems are readily apparent when using software pipelining, which increases architected register pressure by overlapping a number of simultaneously live instances of loop variables from different iterations of the original loop body. To accommodate these loop variables, each of the simultaneously live instances needs its own register. Furthermore, each instance must be uniquely identified (or named) to differentiate among live instances from different overlapped iterations, and to match the definitions from the current iteration with their uses in future iterations. Two common schemes that support this form of register allocation and naming are modulo variable expansion (MVE) [32], and the rotating register file (RR) [44, 46]. As operation latencies increase and machines get wider, both schemes require large increases in the number of architected registers. In addition, MVE causes increases in the code size. Besides increasing the size and complexity of the register file, adding more architected registers to the instruction set architecture (ISA) increases the number of bits per instruction that are required for register encoding, which can also increase the code size. This increase leads to larger instruction footprints and more cache misses, and therefore decreases performance.

Several techniques have been proposed to deal with the architected register pressure problem in software pipelined loops. Register spilling is one such technique that has proved to be very effective [62]. Other algorithms try to achieve software pipelined schedules with the highest possible performance for a given number of registers [12, 23]. But the main problem still remains: for highly concurrent VLIW/EPIC processors, performance degradation due to register pressure is still significant and could be avoided if additional registers could be provided, preferably without significantly increasing the cycle time which would degrade the performance improvement

gained by these additional registers.

To effectively deal with the architected register pressure problem in software-pipelined loops, this dissertation proposes a hardware/software mechanism called Register Queues (RQs), which effectively separates the available physical register space into a small set of architected registers and a larger physical register space that is organized as a set of circular queues and accessed indirectly, where each entire queue is represented by just one architected register. Using RQs, the compiler can allocate distinct physical registers to store all the live values in a software pipelined loop while minimizing the pressure placed on architected registers. As operation latencies and processor concurrency increase, RQs can enable efficient schedules with almost no increase in either architected registers or code size.

1.1 Research Contributions

Deterministic Predicate-aware Scheduling (DPAS)

To extract instruction level parallelism more effectively in the presence of branches and reduce branch overhead, predicated (conditional) execution is often employed. With predicated execution, operations are augmented with an additional Boolean operand known as the guarding predicate. When the guarding predicate is True, the operation executes normally. Conversely, when it is False, the operation is nullified.

Though generally effective at dealing with branches, predicated execution increases the overall resource requirements. For branches that are if-converted in a code segment, the resources of the *then* and *else* clauses are conservatively added to determine the overall resource requirements for the resultant sequence of predicated operations, regardless of whether the corresponding predicate is True or False. As a

result, to avoid oversaturation of the processor resources [36], a compiler must apply if-conversion carefully, which precludes achieving the full potential of predication.

To overcome the problem of superfluous resource utilization by nullified operations, a technique referred to as deterministic predicate-aware scheduling (DPAS) [51] is proposed. The main inspiration behind DPAS is that an operation with a False predicate only requires those resources that it uses between fetching the operation and determining that its predicate is False (called must-use resources). All resources that occur later in the processor pipeline (called may-use resources) are superfluous when assigned to a nullified operation and need not be committed to that operation. The central idea of DPAS is to allow the assignment of several operations to a may-use resource in the same cycle. The processor pipeline extension for supporting DPAS reads predicates in an early stage (before may-use resources are committed) and immediately discards those scheduled operations that are guarded under a False predicate.

However, as the word "deterministic" implies, dynamic over-subscription of resources must not take place. Thus, the compiler must guarantee that no two operations that are assigned to the same resource at the same time will ever have their predicates both evaluate to True in the same runtime cycle. To this end, the compiler uses predicate analysis [27] to identify disjoint operations, i.e. pairs of operations whose predicates can never evaluate to True at the same time. A set of operations with disjoint predicates is allowed to reserve the same may-use resource at the same time, as the compiler guarantees that at most one of these operations will be simultaneously active at runtime.

The overall effect of DPAS is to use the processor resources more efficiently, thereby increasing performance when resource constraints are a bottleneck. This

dissertation develops and applies DPAS to both acyclic and cyclic code regions. To allow more resources to be shared by predicated operations, the predicate should be read as early as possible in the pipeline. However, early reading increases the latency of the predicate-defining operation (the operation that sets a predicate for future use by predicated operations). Our studies show that the cyclic regions are generally more resource-constrained than the acyclic regions of the MediaBench [33] applications studied in this dissertation. Therefore, increasing the latency of the predicate-defining operation causes minimal degradation degradation in the performance improvement derived from resource sharing in software pipelined loops. However, the acyclic regions are more latency bound than the cyclic regions. Hence, predicate-aware scheduling is less effective for acyclic regions than for cyclic regions. Increasing the latency of a predicate-defining operation increases the critical path length which, at some point as latencies become higher, becomes more constraining than resource limitations and then begins to reduce the benefits of resource sharing due to predicate-aware scheduling. This point generally occurs earlier for acyclic regions than for cyclic regions.

Our experimental studies show that DPAS achieves an average of 10% performance improvement over all predicated cyclic regions, and a more modest 4% over all predicated acyclic regions, with an overall average performance gain of 7%. This speedup assumes that whenever a predicated region (a region with at least one predicated operation) achieves the same or worse performance with DPAS than with the original baseline scheduler, the region is scheduled with the baseline scheduler and DPAS is said to achieve no (or 0%) speedup over that region.

Probabilistic Predicate-aware Scheduling (PPAS)

To further reduce the problem of superfluous resource utilization by nullified operations, this disseration proposes probabilistic predicate-aware scheduling (PPAS) [50]. The central idea of PPAS, as in DPAS, is to allow over-subscription of may-use resources wherein multiple operations are allowed to reserve the same resource at the same time. However, PPAS is a generalization of DPAS in that operations are allowed to share a may-use resource even when they are not disjoint, i.e. dynamic over-subscription of resources is allowed to take place even if two or more resource-sharing operations may have their predicates evaluate to True at runtime, which would result in a resource conflict. The processor pipeline is modified to detect and recover from such conflicts when they occur.

By allowing conflicts to occur, PPAS finds many more combinable operations than DPAS. PPAS tries to estimate and account for conflicts so that it can maximize the benefits of oversubscription. Predicates are probabilistically analyzed using a combination of predicate profile information and predicate analysis [27]. Profile information provides statistics on the expected number of times that a predicate will evaluate to True. Predicate analysis computes implication and disjointness relations among predicates to identify when two or more predicates are guaranteed to conflict, or guaranteed not to conflict. Probabilistic analysis, based on profile information, is used to identify profitable opportunities for resource oversubscription. The scheduler takes advantage of these opportunities when they lead to a tighter schedule without an undue expected conflict penalty. By allowing more flexibility in resource sharing than DPAS, PPAS achieves an average of 19% performance improvement over the predicated cyclic regions and 7% over the predicated acyclic regions, with an overall performance gain of 11%.

Register Queues (RQs)

To increase the instruction level parallelism of a cyclic region, high performance compilers use software pipelining [7, 8, 22, 32, 42, 43] to overlap consecutive iterations of the loop. Software pipelining generally results in higher performance, but increased architected register requirements due to the large number of simultaneously live instances of loop variables from different overlapped iterations of the original loop body, where each such instance requires a unique register. Furthermore, each instance must be uniquely identified (or named) to differentiate among live instances from different overlapped iterations and to match the definitions from the current iteration with their uses in future iterations. Two common schemes that support register allocation for software pipelined loops are modulo variable expansion (MVE) [32] and the rotating register file (RR) [44, 46]. Each of these schemes increases the architected register requirements of the software pipelined loops. In addition, MVE also increases the code size, sometimes quite drastically.

The RQs scheme [52] proposed in this dissertation enables a decoupling of the total register space for software pipelined loops into a small set of architected registers and a larger physical register space that is organized as a set of circular queues and accessed indirectly, where each entire queue is represented by just one architected register. Variables with multiple simultaneously live instances are assigned to a queue which holds all the live instances of the given variable. Hence, all these instances are represented by a single architected register, rather than using a separate architected register for each live instance as is done in conventional schemes. All other variables are assigned to single entry architected registers.

As a new instance of each multiple live instance variable is defined in each iteration, this instance is automatically written into the next position in the queue

without overwriting any previous instance that is still live. To access the correct live instance, the register queues are accessed indirectly. The indirect access is accomplished through an architected register by using a special *connect* instruction which connects this architected register to a specified position (offset) in the queue. All future read accesses to this architected register automatically read the value from this queue position. The specified position corresponds to the instance defined in the appropriate previous iteration; the regular access pattern guaranteed by software pipelining, together with the RQs implementation, ensures that the correct value is always found at that specified position.

By using register queues, the architected register requirements of a software pipelined loop are independent of the number of overlapped simultaneously live instances of the loop variables. Our experimental results show that the RQs method significantly reduces both the architected register requirements and the code size of software pipelined loops. Note however, that when applied to predicated code the RQs method does require additional support for disambiguation when several predicated producers supply the same user.

Our studies show that the RQs method can also be very effective when used in conjunction with DPAS and PPAS schemes. Application of these scheduling schemes to software pipelined loops increases the architected register requirements of the loops relative to the baseline software pipelining scheme; RQs can then be used to reduce those requirements. More specifically, our results show that for a given set of 122 loops extracted from the MediaBench applications, by using rotating registers with only 32 architected registers, 87% of all loops can be scheduled with DPAS or 72% with PPAS. However, 99% of these loops would require no more than 32 architected registers if they were scheduled by using the RQs scheme with either DPAS or PPAS.

1.2 Dissertation Overview

This dissertation is organized as follows. Deterministic predicate-aware scheduling is presented in Chapter 2, probabilistic predicate-aware scheduling is presented in Chapter 3, and the register queues technique is described in Chapter 4. Conclusions are presented in Chapter 5.

CHAPTER 2

DETERMINISTIC PREDICATE-AWARE SCHEDULING

2.1 Introduction

VLIW processors rely on an intelligent compiler for extracting, enhancing, and exposing sufficient instruction-level parallelism (ILP) to deliver high performance. To extract ILP more effectively in the presence of branches and reduce the branch overhead, predicated (conditional) execution is often employed. With predicated execution, operations are augmented with an additional Boolean operand known as a guarding predicate. When the guarding predicate is True, the operation executes normally. Conversely, when it is False, the operation is nullified. Predicated execution can be exploited by compilers that use if-conversion to convert branching code into straight-line segments of predicated operations [3, 40]. As a result, many branches and the difficulties associated with them can be eliminated.

Though generally effective at dealing with branches, predicated execution introduces a serious overhead of its own. Predicated execution trades off sequential execution of conditional operations for increased resource requirements. If-conversion

is additive with respect to resources across branches to which it is applied. For branches that are if-converted, the resources of the *then* and *else* clauses are added to determine the overall resource requirements for the resultant sequence of predicated operations. Intuitively this makes sense, since to remove a branch both clauses must be scheduled, with the appropriate one nullified at runtime. As a result, a compiler must apply if-conversion carefully to avoid oversaturation of the processor resources [36].

Compile-time assignment of resources (e.g., fetch slots, register ports, function units, memory ports) to predicated operations is traditionally handled in a conservative manner. The compiler assumes that any predicate may evaluate to True at runtime and accordingly ensures that all resources required by an operation are unconditionally available. However, this is not necessary. At runtime, an operation requires resources only when its predicate evaluates to True; an operation with a False predicate only requires a subset of its resources. In particular, only the resources used between fetching the operation and determining that its predicate is False are necessary. All later resources assigned to a nullified operation are superfluous.

For a predicated architecture, processor resources can be broken down into two categories: must-use and may-use. A must-use resource is required by an operation regardless of its runtime predicate value. Conversely, a may-use resource is only required for those instances of the operation where its predicate evaluates to True. The classification of resources into these two categories is based on the point in the processor pipeline where operations with False predicates are nullified. Resources before the nullification point are must-use; those after are may-use. A fundamental design tradeoff exists with respect to where the nullification point is placed within the processor pipeline: nullification later in the pipeline reduces the latency from

predicate computations to uses of those predicates, whereas nullification earlier in the pipeline minimizes the number of must-use resources.

To overcome the problem of superfluous resource utilization by nullified operations, we propose a technique referred to as deterministic predicate-aware scheduling (DPAS). The central idea of DPAS is to allow static over-subscription of may-use resources wherein multiple operations are allowed to reserve the same resource at the same time. However, as the word "deterministic" implies, dynamic over-subscription of resources must not take place. Thus, the compiler must guarantee that no two operations that are assigned to the same resource at the same time will ever have their predicates both evaluate to True in the same runtime cycle. In Chapter 3 probabilistic predicate-aware scheduling (PPAS) is introduced which relaxes this constraint and therefore needs to employ a recovery technique whenever the hopefully unlikely event of dynamic oversubscription occurs. DPAS and PPAS are collectively referred to as predicate-aware scheduling (PAS). The overall effect of DPAS is to increase the utilization of may-use resources, thereby increasing processor performance. A secondary benefit is that with relaxed resource constraints, more aggressive if-conversion can be applied to extract further benefit from branch elimination.

In order to accomplish predicate-aware scheduling, predicate analysis is employed in the resource reservation process. Specifically, relational properties among the predicates used in a program segment are derived [27, 36, 49]. The most important property for applicability of dpas is disjointness, wherein two predicates are disjoint if they can never evaluate to True at the same time. For instance in an *if-then-else* statement, the predicates controlling the *then* and *else* clauses are disjoint. Such predicate analysis is already used extensively in compilers to assist with dataflow analysis, optimization, and register allocation of predicated code [11, 20]. A set of operations with pairwise

disjoint predicates is allowed to reserve a common resource; the compiler's predicate analysis has guaranteed that in any dynamic instance of this set of operations, at most one of these operations will be active, i.e. at most one will have a True predicate.

One obvious alternative to DPAS is to simply build a wider processor with more resources. When the number of resources is sufficiently large, the problem of resource contention goes away. However, this solution may have a high cost; additional function units, register file ports, busses, bypass logic, etc. may be necessary. For either cost-sensitive or power-sensitive environments, this cost may be unacceptable. DPAS offers an approach to increase the utilization of existing processor resources and therefore increase application performance with a fixed, or more modest, set of resources.

For this chapter, we present the necessary compiler and architecture extensions to accommodate both deterministic predicate-aware acyclic scheduling and deterministic predicate-aware cyclic scheduling. In the next section, a brief background on scheduling is presented followed by a motivational example to illustrate the potential benefit of DPAS. Section 2.3 describes a deterministic predicate-aware processor to support DPAS. Section 2.4 contains a description of the compiler support used to accomplish deterministic predicate-aware acyclic scheduling and software pipelining. The effectiveness of DPAS is evaluated experimentally on a sample deterministic predicate-aware processor in Section 2.5. The final section, Section 2.6, summarizes and presents conclusions.

2.2 Background and Motivation

Code scheduling refers to the process of binding operations to time slots and resources for execution. In this section, we briefly describe two common scheduling techniques: list scheduling (LS) [1] to schedule acyclic code regions and iterative modulo scheduling (IMS) [41] to schedule innermost loop regions. Each technique is applied to a basic block as a simple illustration.

The goal of IMS is to find a valid schedule of minimum length for an acyclic code region. The minimum achievable schedule length is constrained by the maximum of two lower bounds. The resource-constrained lower bound is equal to the number of busy cycles required by the most heavily used resource during a single execution of the region. The latency-constrained lower bound is determined by the sum of the latencies along the longest path through the data dependence graph (i.e. the critical path) of the region.

IMS finds a valid schedule for an innermost loop that can be overlapped with itself multiple times using a constant interval (Initiation Interval (II)) between successive iterations. The goal of IMS is to find such a schedule with the minimum II. The II-cycle code region that achieves the maximum overlap between iterations is called the kernel. The kernel is preceded in the schedule by a prologue that gradually builds up to the maximum iteration overlap, and followed by an epilogue that tapers down.

The scheduler chooses its initial II to be the maximum of the two lower bounds which, for the purposes of the IMS, are calculated in the limit, i.e. by implicitly assuming that the performance will be dominated by repeated executions of the loop kernel. The resource-constrained lower bound $ResMII$ is equal to the number of cycles that the most heavily used resource is busy during a single iteration of the loop, or equivalently during the loop kernel. The recurrence-constrained lower bound

($RecMII$) is determined by the ceiling function of the maximum ratio $\lceil D(C)/P(C) \rceil$ among all recurrence (or "loop-carried dependence") cycles, C, in the dependence graph, where $D(C)$ is the sum of the operation latencies over all edges of the cycle C, and $P(C)$ is the sum of all loop-carried dependence distances (measured in iterations) over those edges.

As the number of machine resources increases, recurrence and latency constraints begin to dominate the schedule length for IMS and LS techniques, respectively. In general, LS is more latency-constrained than IMS, because IMS can look for independent operations across loop iteration boundaries, whereas LS is limited to operations within a single execution of a code region. Note that for loops, loop unrolling in conjunction with LS can be applied to approximate the benefits of IMS.

Both scheduling techniques schedule operations at particular cycles so that both data dependences and resource constraints are satisfied. To satisfy these scheduling constraints, LS and IMS use a data structure known as the schedule reservation table (SRT). The SRT records which specific operations (in an acyclic region or a single loop iteration) use each particular resource (column) at each time (row) [9, 41]. Scheduling an operation at a particular time is permitted only if its resource usage does not result in a resource conflict, i.e. it does not attempt to reserve any resource at a time when some other operation already reserved that same resource, and no latency constraints of prior operations on which the operation being scheduled depends are violated. In addition IMS uses a Modulo Reservation Table (MRT) (equivalent to taking the first II rows of the SRT and successively overlaying the next II rows until the entire SRT has been entered in the MRT) to facilitate tracking the modulo constraints. The modulo constraint states that two operations that use the same resource may not be scheduled an integer multiple of II cycles apart from one another, i.e. each cell of

Function Unit	Operations	Mnemonics	Lat.
ALU (A)	Add	**add**	1
	Subtract	**sub**	1
	Multiply	**mult**	3
	Or	|	1
	And	&	1
	Predicate Compare	**cmpp**	1,2,3
Memory (M)	Load	**load**	2
	Store	**store**	1
Branch (B)	Branch on condition	**if**	1

Table 2.1: Description of a sample processor with a fetch/execute width of 3 operations

the MRT can hold no more than one operation.

IMS is generally applied to innermost loops that contain only a single basic block. In processors that support predicated execution, if-conversion [3, 57] is applied to broaden the class of loops that can be modulo scheduled. If-conversion can also be used in conjunction with LS on acyclic regions to increase the effectiveness of the scheduler.

2.2.1 Example Code Segment

To illustrate the application of conventional LS and IMS along with the potential benefits of making each scheduler predicate-aware, we consider a simple code example and processor model. The example processor, which can fetch and execute up to three operations per cycle, has three fully pipelined function units, as detailed in Table 2.1. The mnemonics for the various operations, binding of operations to units, and the operation latencies are shown in the table. The tables list three latencies, 1,2 and 3, for the predicate-defining operation, cmpp, because, as discussed in Section 2.3, to support predicate-aware scheduling, the cmpp latency must be increased by at least one cycle; in this simple example, the cmpp latency is increased from 1 to 2 cycles.

```
1:   for ( i = 0; i < im_size; i++)
2:   {
3:        prod = q_im[i] * bin_size;
4:        if (q_im[i] ≥ 1)
5:             res[i] = prod − correction;
6:        else
7:             res[i] = prod + correction;
8:   }
```

(a) Source code

```
op1: t1 = load(i1, q_im) if p0;
op2: prod = mult(t1, tbs) if p0;
op3: p1,p2 = cmpp.lt.uu(t1, 1) if p0;
op4: t2 = sub(prod, tcor) if p1;
op5: t2 = add(prod, tcor) if p2;
op6: store(i1+ = 4, res, t2) if p0;
op7: if(i++ < im_size) goto op1 if p0;
```

(b) Assembly code after if-conversion (p0=True)

Figure 2.1: Example code segment

The motivational example in Chapter 3, which also assumes this machine model, increases the predicate defining operation latency to 3 cycles.

The example code segment is a slight modification of a loop extracted from the *unquantize_image()* function from the *epic* application in the MediaBench benchmark suite [33]. Figure 2.1(a) shows the C source for the example loop. Figure 2.1(b) shows the assembly code for the loop after if-conversion. For conciseness of the example we assume that the instruction set supports post-increment load and store operations. In this example, the *if-then-else* statement is replaced by the corresponding predicate-defining operation (p1,p2 = cmpp.lt.uu$(t1, 1)$ **if** $p0$). Predicate $p1$ is set to True and $p2$ is set to False when the if condition $(t1 < 1)$ evaluates to True; whereas condition False sets $p1$ False and $p2$ True. The detailed semantics of cmpp operations are described in [28].

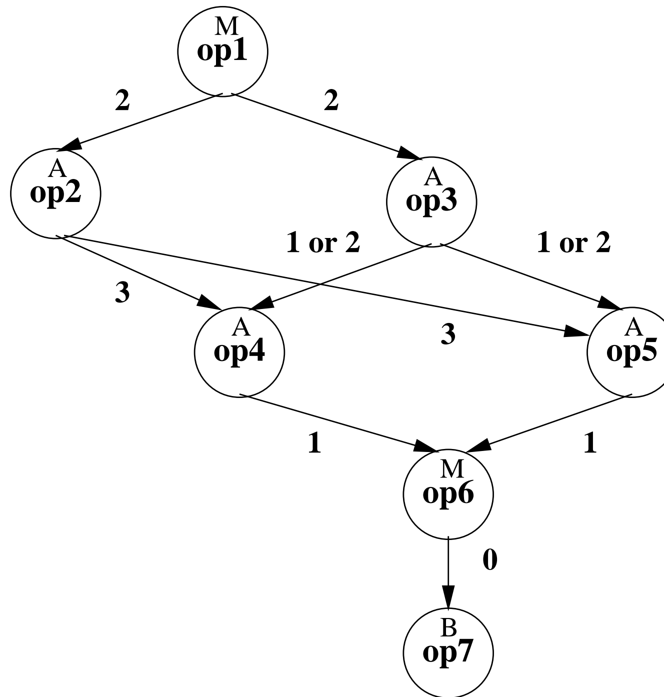


Figure 2.2: Data dependence graph for code segment

The data dependence graph of the if-converted loop segment is presented in Figure 2.2. Each node is annotated with the type of the operation (A=ALU, M=memory, B=branch). Each edge is marked with the latency of that edge. Note that the edges in the graph are all flow dependences with the exception of the edge from op6 to op7 which is a control dependence.

2.2.2 Applying Predicate-aware Scheduling

As stated above, the reservation table enforces resource constraints for both LS and IMS. That is, operations that use the same resource cannot be scheduled in the same cycle. Predicate-aware scheduling relaxes this constraint by allowing operations guarded by disjoint predicates (referred to hereon simply as disjoint operations) to reserve (or share) the same resource in the same clock cycle. The if-converted code

Time	A	M	B
0		op1	
1			
2	op2		
3	op3		
4			
5	op4		
6	op5		
7		op6	op7

(a) SchedLen = 8

Time	A		M	B
0			op1	
1				
2	op2			
3	op3			
4				
5	op4	op5		
6			op6	op7

(b) SchedLen = 7

Figure 2.3: LS schedule (a) versus DPALS schedule (b)

shown in Figure 2.1(b) is used to illustrate conventional LS and its counterpart deterministic predicate-aware list scheduling (DPALS), along with conventional IMS and its counterpart deterministic predicate-aware modulo scheduling (DPAMS).

2.2.2.1 LS versus DPALS

The application of LS to the example results in the 8 cycle schedule presented in Figure 2.3(a). This schedule is optimal for this machine model. *op4* is scheduled at cycle 5 which is the earliest time at which it can be scheduled. The earliest schedule time for *op5* is also cycle 5, but due to resource conflict with *op4*, it gets scheduled at the next cycle. Hence, the earliest schedule time for *op6*, which depends on both *op4* and *op5*, is cycle 7. Note that both *op4* and *op5* are executed conditionally, but reserve the ALU unconditionally. In fact, during any particular iteration of the loop at runtime, only one of these operations is executed; the other is nullified. As a result, we effectively waste either cycle 5 or cycle 6 for each iteration of the loop because the ALU is not utilized during the cycle in which it is assigned to a nullified operation.

With DPALS, a 7 cycle schedule can be achieved, as shown in Figure 2.3(b). Operations *op4* and *op5* (from the *then* and *else* paths, respectively) can now both

Time	A	M	B
0	op5	op1	
1	op4	op6	
2	op2		
3	op3		op7

(a) $II = 4$

Time	A		M	B
0	op3		op1	
1	op4	op5		
2	op2		op6	op7

(b) $II = 3$

Figure 2.4: IMS kernel (a) versus DPAMS kernel schedule (b)

be scheduled at their earliest schedule time; both operations may reserve the ALU in cycle 5 because they are provably disjoint. In the SRT of Figure 2.3(b), each resource conceptually provides two schedule slots at each time. This allows up to two disjoint operations to occupy the same resource at the same time. Two slots is not a restriction of this technique. Rather, for this example, there are only 2 control paths; thus we know that at most two operations can be mutually disjoint.

The overall result of the DPALS schedule is that the ALU resource is always utilized in cycle 5 and the achieved schedule length is 7 cycles, a 14% speedup over the 8 cycle LS schedule shown in Figure 2.3(a). Note that for this basic block, a schedule of length 7 is optimal for any machine configuration. Since the latency-constrained lower bound (critical path length in Figure 2.2) is 7 cycles.

2.2.2.2 IMS versus DPAMS

The application of IMS to this same loop example results in the II=4 schedule presented in the MRT shown in Figure 2.4(a). Since each of the four ALU operations (*op2, op3, op4, op5*) must reserve the ALU resource in a different cycle to avoid conflict, $ResMII$=4 and this schedule is optimal. Note that $RecMII$ is set to 1 by default since there are no recurrence cycles.

With DPAMS, an II=3 schedule can be achieved as shown in Figure 2.4(b).

Again, this improvement is achieved by enabling the provably disjoint operations, *op4* and *op5*, to reserve the ALU in the same cycle. Overall, DPAMS results in a 33% speedup over IMS for this example. Note that for this particular loop, no compiler strategy can do better than $II = 3$ because there is only one ALU and each control path has three operations that require it.

This example shows that by allowing disjoint operations to reserve the same resource in the same time-slot, the resource requirement for a code segment can be reduced. For code that is resource constrained, this relaxation results in a tighter schedule, and hence a performance improvement. Of course if resources are not a limiting factor, the benefit of predicate-aware scheduling is lessened. If the example processor had two ALUs instead of one, LS would achieve an optimal list schedule of 7 cycles for this example, and IMS would achieve an optimum modulo schedule, with an II of 3 cycles. However, the two ALUs would be poorly utilized by this example.

2.2.3 Characteristics of Improvable Regions

The previous section showed an example that can derive benefit from deterministic predicate-aware scheduling. The central issue for further investigation is whether application regions, both acyclic and cyclic, generally have properties that benefit from DPAS. For details of the experimental evaluations see Section 2.5.

Two related metrics are: the fraction of runtime spent in regions with disjoint operations, and the potential to combine the disjoint operations in those regions. Figure 2.5 addresses these two metrics. Each stacked bar shows, for one benchmark application or the overall Average, the dynamic operation breakdown in terms of the code region they belong to.

In general there are three kinds of regions. First are acyclic regions with at least

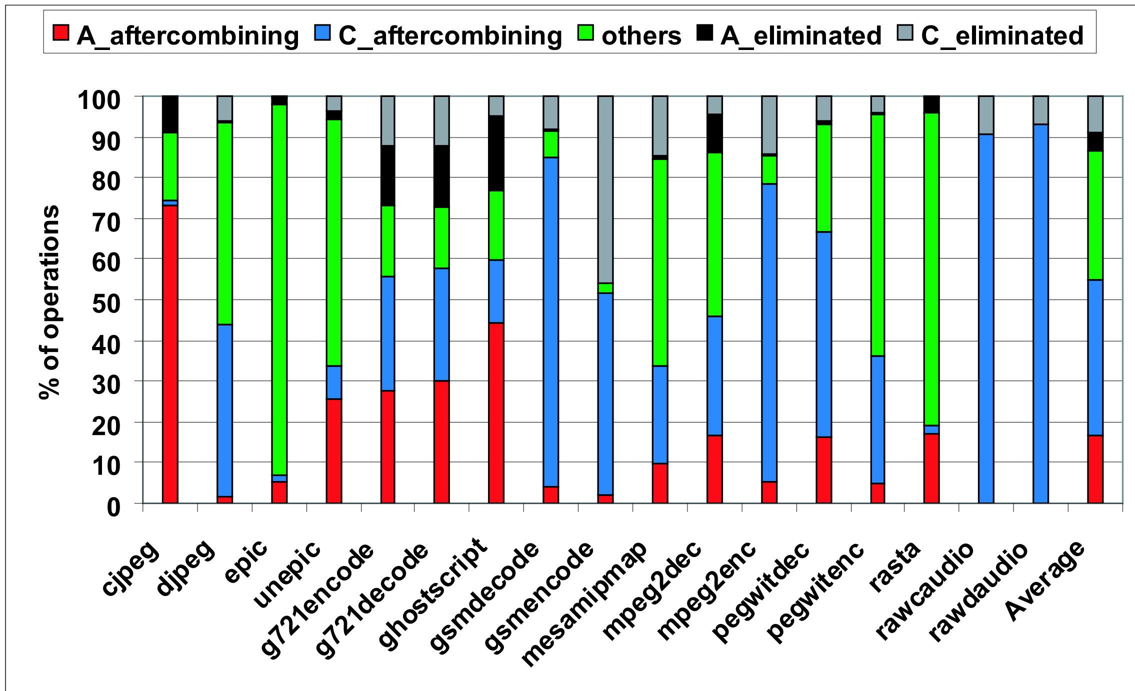


Figure 2.5: Characteristics of 'dpa-improvable' regions

one pair of disjoint operations and hence some opportunity for combining, which we call acyclic dpa-improvable regions (A). Second are cyclic regions with at least one pair of disjoint operations, called cyclic dpa-improvable regions (C). Note that in our case cyclic regions are only single-basic block inner-most loops with no early exits that can be scheduled with our cyclic scheduler. Third are *other* regions with no disjoint operations and hence no opportunity for combining.

The A and C regions are each further broken down into 2 distinct sub-regions where each sub-region is represented by a sub-bar in Figure 2.5. The A_aftercombining sub-bar contains the percent of all the dynamic operations that are in A regions and remain after optimistic combining (an operation can represent a group of dynamic operations and is counted as one operation, see below).

Optimistic combining produces groups of disjoint operations (regardless of their

type and latency) in the region. If an operation is disjoint from any group already formed (that is, it is disjoint from every operation in that group), it joins the group and is not counted; in other words, it is eliminated as the result of combining. Conversely, if an operation is not disjoint from any group already formed, it forms a new group and is counted once. Thus during optimistic combining each group of combined operations counts as one operation.

Similarly, the C_aftercombining sub-bar shows the percent of all the dynamic operations that are in C regions and remain after optimistic combining. A_eliminated and C_eliminated show the percent of all dynamic operations that are from A and C regions, respectively, and are eliminated from the code as the result of combining. Note that the number of dynamic operations in a region after combining is the total number of dynamic operations in that region before combining minus the number of operations that were eliminated as the result of combining. Finally, the fifth (*others*) sub-bar shows the remaining operations, i.e. the percent of all dynamic operations that are not in a dpa-improvable region.

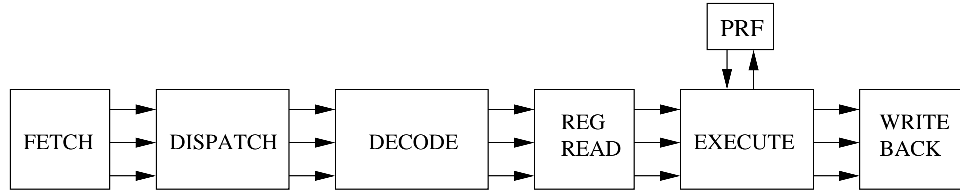
By adding together the heights of the A_aftercombining and A_eliminated sub-bars, we obtain the percent of all dynamic operations that lie in acyclic dpa-improvable regions, an average of 21%. Similarly, by adding together the heights of the C_aftercombining and C_eliminated sub-bars, we obtain the percent of all dynamic operations that lie in cyclic dpa-improvable regions, an average of 47%; thus nearly half of the operations over all these benchmarks are from dpa-improvable loops. In fact rawcaudio and rawdaudio spend almost all of their execution time in dpa-improvable loops. The sum of all 4 sub-bars (68%) is the percent of the original dynamic operations (before combining) that lie in either an acyclic or cyclic dpa-improvable region.

The overall potential benefit of DPAS depends on the frequency of the improvable regions, on the percentage of disjoint operations within these regions, and on the percentage of those disjoint operations that are eliminated by optimistic combining. From Figure 2.5 we see that on average 4.5% of all dynamic operations are eliminated operations from acyclic regions and 9% of all operations are eliminated operations from cyclic regions, resulting in the elimination of a total of 13.5% of all dynamic operations. Note that this does not imply an upper bound on the speedup obtained with predicate-aware scheduling; the actual performance benefits can be higher or lower as illustrated by the prior example. In that example, there are two disjoint operations, one of which is eliminated by optimistic combining, which is 14% of the original 7 dynamic operations. We have seen that DPALS achieves a 14% speedup over LS, but DPAMS achieves a 33% speedup over IMS.

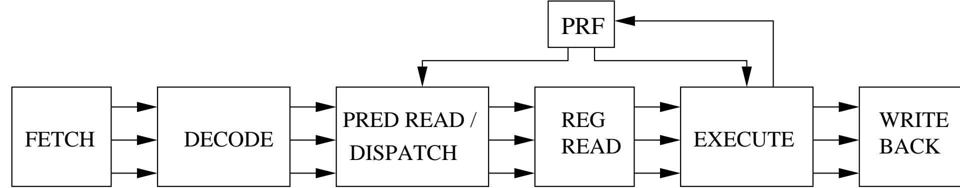
2.3 Deterministic Predicate-aware Architecture

All resources in the generic predicate-aware architecture can be divided into two categories: may-use and must-use. Each resource used after the value of the guarding predicate becomes known can be may-use, i.e. it can be reserved by several disjoint operations at the same time. All the remaining resources are must-use and can only be reserved by a single operation at a given time. Therefore, the earlier the predicates are read, the more resources can be may-use, which can lead to shorter schedules. On the other hand, accessing the predicate register file earlier in the processor pipeline increases the latency of the predicate defining operation. This can be problematic if many of the predicate defining operations lie on the critical path of the application.

The datapath pipeline of the baseline machine is shown in Figure 2.6(a). This



(a) Baseline Machine



(b) Deterministic Predicate-aware Machine

Figure 2.6: Baseline vs. deterministic Predicate-aware machine models

processor pipeline organization is similar to the TI 'C6x architecture [56], except that its unified register read and execution stage is separated here into two stages. The baseline processor pipeline has 6 stages: fetch, dispatch, decode, register read, execute and write back. The predicate register file is read (and written) only during the execution stage. Thus, resources in the execute stage and the preceding stages are must-use. Only the resources in the write-back stage are may-use.

In order to make the baseline pipeline predicate-aware, four issues must be addressed. First, nullification should be performed earlier in the pipeline to make more may-use resources available. Second, the disjoint operations should be easily identifiable. Third, the cmpp latency should be kept as small as possible. Fourth, the pipeline complexity should not be increased so substantially that it compromises the cycle time. To this end, we make two main changes in the baseline pipeline to make it predicate-aware, as shown in Figure 2.6(b).

Change 1: The first change is to move the predicate register file (PRF) read to

the dispatch stage to allow predicates to be read early in the pipeline. This allows nullification to occur at the end of the dispatch stage. As a result, all the resources in subsequent stages (general / floating-point register ports, function units, etc.) are may-use. During the dispatch stage, the PRF is accessed early in the cycle to read the predicates for all the operations. Then, the dispatch logic nullifies those operations guarded under False and assigns the rest of the operations to their corresponding function units. Note that compiler's task is to assign only provably disjoint operations to share the same resource, which ensures that no runtime resource conflict can ever occur.

Change 2: In the baseline processor, the latency of the cmpp operation is 1 cycle. The first change above, however, increases it to 4 cycles. This can penalize the performance of the predicated code, especially in the regions in which predicated operations lie on the critical path. To mitigate the impact of increased cmpp latency, our second change is to reverse the order of the decode and dispatch stages, thereby delaying the predicate read by one stage, which reduces the cmpp latency to 3 cycles. However, since decode now occurs before dispatch, the complexity of the decode logic is increased somewhat. In the worst case, FW (fetch width) general purpose decoders, one per operation in the instruction word, are required. In the alternative, where decode follows dispatch, it might be possible to use more specialized and hence less expensive decoders.

An interesting consequence of our design is that it is possible to selectively increase the cmpp latency, so as to restrict the increase in critical path length. Only some operations will see an increased cmpp latency; all other operations will see a cmpp latency of 1, which means that they will execute unconditionally on their resources. To allow both kinds of operations to read their predicates in the same cycle, the 1-bit

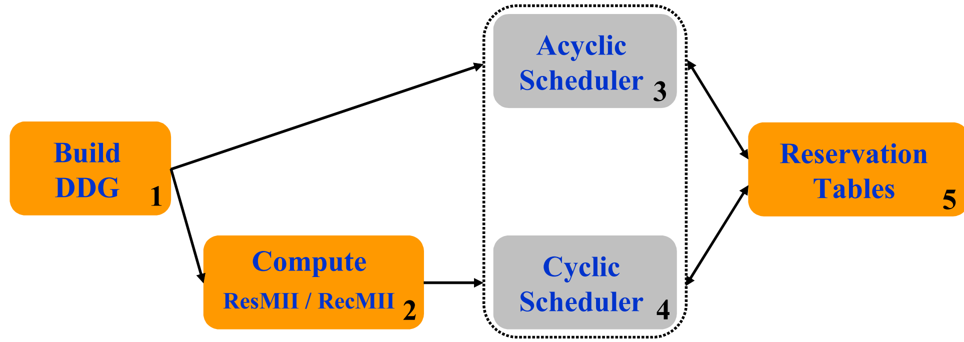
wide PRF in Figure 2.6(b) can be simultaneously accessed from the execute stage, as well as the predicate read and dispatch stage. Thus, twice as many PRF read ports are required, but they are only 1-bit wide. If this poses a design problem, a shadow PRF could be added for access by one of these stages.

When the operation's predicate is first accessed in the predicate read and dispatch stage, it may not be available because it has not been produced yet by the operation's corresponding cmpp. In this case, the operation reserves its resources unconditionally and proceeds forward to the register read stage and then to the execute stage where its predicate is accessed the second time and is guaranteed to be available, just as in the baseline processor. Again, the scheduler must ensure that regardless of where the operation's predicate is accessed no resource conflict will ever occur.

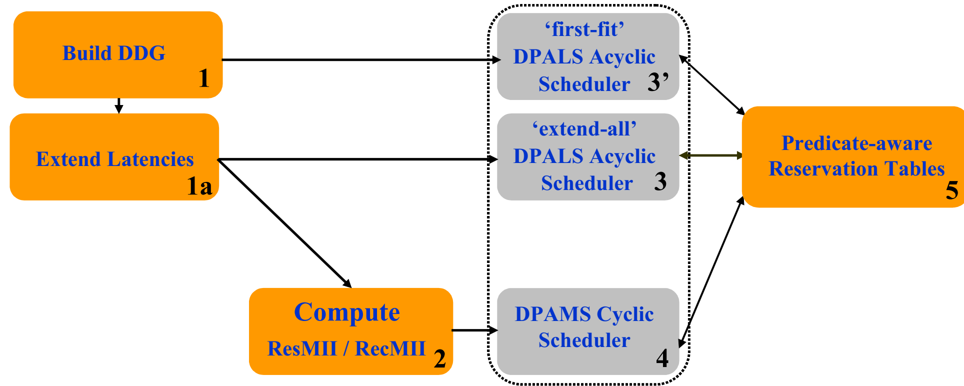
Finally, it is possible to have more pipeline stages between predicate read / dispatch and execute as the result of increased processor frequency; register read or comparator logic may then be split into multiple stages, and cmpp latency would increase correspondingly. As the results in Section 2.5 indicate, higher cmpp latency will degrade the performance of DPALS because cmpp operations are often on the critical paths of acyclic regions. However, modest cmpp latency increases will not have a significant impact on the performance of DPAMS because cmpp operations are rarely on the critical paths of cyclic regions.

2.4 Deterministic Predicate-aware Scheduling

In this section, the details of the acyclic and cyclic deterministic predicate-aware scheduling algorithms are presented. Deterministic predicate-aware list scheduling (DPALS) and deterministic predicate-aware modulo scheduling (DPAMS) are exten-



(a) Conventional Scheduler



(b) Deterministic Predicate-Aware Scheduler

Figure 2.7: Scheduling algorithm flowchart

sions of conventional LS and IMS, respectively. As discussed in the previous section, both techniques aim to decrease schedule length by relaxing resource constraints, specifically by allowing a set of disjoint operations to reserve the same resource in the same cycle.

Figure 2.7(a) shows the five main scheduling steps for conventional list and modulo scheduling algorithms. First, the data dependence graph is constructed in which edges between dependent operations are marked with the corresponding latencies (Step 1). Next, in the case of modulo scheduling, the resource and recurrence based lower bounds on II are computed to provide a starting value of II for the main scheduler

(Step 2). The main scheduler schedules each operation in accord with the resource and latency constraints, until the entire region is scheduled with either the list scheduler for acyclic regions (Step 3), or the modulo scheduler for cyclic regions (Step 4). Each scheduler uses a resource reservation table to avoid resource conflicts (Step 5).

Our deterministic predicate-aware scheduling technique extends the conventional scheduler (discussed in Section 2.4.2) as shown in Figure 2.7(b). Each extension is discussed in the following sections. Section 2.4.1 describes the data dependence graph latency extension step (Step 1a) which follows DDG construction (Step 1) and selectively extends the cmpp latency for every operation that can potentially be combined with some other disjoint operation. We have designed and implemented three scheduling algorithms: two alternative DPALS algorithms (Steps 3 and 3'), and one DPAMS algorithm (Step 4).

The first algorithm, called 'extend-all' DPALS (Step 3), is described in Section 2.4.3 and is used to schedule acyclic regions. 'Extend-all' DPALS requires the latency extension step (Step 1a), as shown by the arrow from Step 1a to Step 3 in Figure 2.7(b), so that when an operation with extended cmpp latency is scheduled, it is always far enough from its cmpp producer to be able to read its predicate early in the pipeline (during the predicate read and dispatch stage), and hence conditionally reserve and share its resource with another disjoint operation in the region. Note that an operation in our scheduling examples explicitly reserves either an ALU, a Memory Port, or a Branch Unit as its execution unit. We refer to this unit briefly as "its resource". Other may-use resources (such as register ports) reservations are implicit; reservations of the fetch width must-use resource are also generally implicit, but are occasionally shown explicitly.

The second algorithm, called 'first-fit' DPALS (Step 3'), is described in Sec-

tion 2.4.4 and may alternatively be used to schedule acyclic regions. Unlike 'extend-all' DPALS, 'first-fit' DPALS does not require the latency extension step. Instead, an operation can be scheduled at the earliest time that satisfies its resource and dependence constraints, regardless of the distance from cmpp that defines its predicate. However, an operation cannot share its resource with any other operation if it is scheduled so close to its cmpp producer that its predicate is not available during the early read stage; in this case the operation must reserve its resource unconditionally. Each predicated operation has an associated cmpp operation that produces its predicate; we refer to that cmpp operation as "its cmpp producer" or sometimes as "its cmpp" for short. Furthermore we often refer to the latency from its cmpp to the operation simply as the operation's "cmpp latency."

DPAMS (Step 4) is used to schedule cyclic regions, as described in Section 2.4.1. It is similar to 'extend-all' DPALS in that it also requires the latency extension step. Note that in addition DPAMS requires the $ResMII$ and $RecMII$ bounds to be computed (in a deterministic predicate-aware manner), as indicated by the arrow in Figure 2.7(b) from Step 2 to Step 4.

Both DPALS and DPAMS share much of the same underlying scheduling infrastructure, which employs a predicate-aware reservation table module (Step 5), as described in Section 2.4.2.

2.4.1 DGG Latency Extension

As we said in Section 2.3, it is possible to selectively increase cmpp latency of the predicated operations in the region as to restrict the growth of the critical path length. 'Extend-all' DPALS and DPAMS perform this latency extension immediately after the DDG construction step.

The cmpp latency of a given predicated operation is the latency that places a latency constraint on the schedule distance between the operation's predicate-defining cmpp and the predicated operation itself. As we said in Section 2.3, each predicated operation can be selectively scheduled with one of the two cmpp latencies: *shortlatency*, in which case it must reserve its resource unconditionally and cannot share it with other disjoint operations since its predicate is not available until the end of the execute stage, or *extendelatency*, in which case it can conditionally reserve and share its resource with other disjoint operations since its predicate is available early in the pipeline during the predicate read and dispatch stage.

The DDG latency extension step optimistically extends the cmpp latency of a given operation to *extendedlatency* cycles if it can potentially be combined with some other disjoint operation, i.e. only if it might result in some performance improvement. If the operation cannot be combined with any other operation in the region, its cmpp latency is set to *shortlatency*. Note that even when an operation's latency is extended, during actual scheduling the combining may or may not take place. Moreover, even if such combining occurs during scheduling, it is not guaranteed to result in a schedule length reduction.

The procedure **DPAS-DDGLatency-Adjust** performs the latency extension, as shown in Figure 2.8. It takes four parameters: *ddg* - the data dependence graph of the region, *regiontype* - a region type flag which indicates whether the region is acyclic or cyclic, *shortlatency* - a short latency of the cmpp operation, and finally *extendedlatency* - an extended longer latency of the cmpp operation.

The procedure consists of two phases. The first phase (lines 1-8) sets the *mustres* bit field of every predicated operation. The *mustrest* bit is set to 0 if the operation can be combined with some other operation, potentially resulting in some performance

```
DPAS-DDGLatency-Exend(ddg, regiontype, shortlatency, extendedlatency)
1    /* phase 1: mark mustres bit of each operation */
2    for each predicated operation node, op, in ddg do
3      op.mustres = 1;
4      independenceSet = all operations in ddg independent from op1
5      if(op is disjoint from at least one operation from independenceSet then
6            op.mustres = 0
7      end if
8    end for
9    /* phase 2: adjust cmpp latencies */
10   for each cmpp operation outgoing edge, cmpp_outedge) do
11     op = cmpp_outedge.destination;
12     if op.mustres == 1 then
13        setlatency(cmpp_outedge, shortlatency)
14     else
15        setlatency(cmpp_outedge, extendedlatency)
16     end if
17   end for
```

Figure 2.8: DPAS latency extension procedure

improvement; otherwise the *mustres* bit is set to 1. More specifically, at the beginning of the first phase the *mustres* bit of every operation is conservatively initialized to 1 (line 3), assuming that there exists no combining opportunity for this operation. The condition in line 5 tests if the operation is disjoint from at least one other operation in its independence set, *independeceSet*, which consists of all operations in ddg that are independent from this operation. If an operation is disjoint from at least one operation in its *independeceSet*, the operation's *mustres* bit is set to 0 (line 6), as these two operations might get combined during scheduling and thus potentially result in some performance improvement.

In the second phase (lines 10-16), the cmpp latency of those operations with a *mustres* bit of 1 is set to *shortlatency* (line 13): these operations will reserve their resources unconditionally. All the other operations with the *mustres* bit of 0 have their cmpp latency extended to *extendedlatency* cycles (line 15): those operations

will reserve their resources conditionally.

2.4.2 Deterministic Predicate-aware Unified Scheduling Algorithm

As shown in Figure 2.7, list and modulo scheduling algorithms share much of the same underlying scheduling infrastructure, in particular the Step 5 of the scheduling algorithm. Thus, we begin this section with a unified discussion of both algorithms, referred to as unified scheduling or simply scheduling. The term reservation table (RT) is used in a generic sense to represent either a schedule reservation table (SRT) for a list scheduling algorithm or a modulo reservation table (MRT) for a modulo scheduling algorithm.

2.4.2.1 Baseline Unified Scheduling Algorithm

The heart of a typical instruction scheduling algorithm employs two important functions (embodied here in **FindTimeSlot** and **ResourceConflict**, see below) to identify a conflict-free time for each operation to be scheduled. The central data structure used to identify resource conflicts is the RT. The general realization of an RT (similar to Figure 2.3(a)) is a two-dimensional matrix in which columns correspond to resources and rows correspond to schedule slots.

In our implementation, the scheduler selects an operation from the pool of unscheduled operations and calls the **FindTimeSlot** function (see pseudo code shown in Figure 2.9(a)). This function scans forward from $MinTime$ to $MaxTime$ looking for the first conflict free slot in RT to schedule the operation. $MinTime$ is the earliest start time that the operation can have as constrained by its already scheduled predecessors. $MaxTime$ (which is usually a very large number) is the latest time at

```
FindTimeSlot(Operation, MinTime, MaxTime) {
      /* Successively try each time in the range */
      for ( CurrTime = MinTime; CurrTime ≤ MaxTime;
            CurrTime++) {
            while ( there are remaining resource alternatives) do {
                  resource_alt = next resource alternative for Operation
                  if ( ResourceConflict(resource_alt, Operation,
                        CurrTime) == False )
                        return CurrTime;
            }
            ...
}
```

(a) **FindTimeSlot**() function

```
ResourceConflict(resource_alt, Operation, CurrTime) {
      while (there are remaining resources in resource_alt) do {
            resource = next resource from resource_alt;
            if ( IS_EMPTY(ReservationTable [CurrTime][resource])
                  == False)
                  return True;
      }
      return False;
}
```

(b) **ResourceConflict**() function

Figure 2.9: Baseline scheduling functions

which the scheduler will try to schedule the operation before giving up. Frequently, an operation may execute on any one of several function units; in this case, the operation is said to have multiple resource alternatives. All resource alternatives are tried inside the *while* loop, and for each alternative, the function *ResourceConflict* is called.

The **ResourceConflict** function, shown in Figure 2.9(b), checks if the operation can be scheduled without conflict on *resource_alt* at time *CurrTime*. Each *resource_alt* is a set of resources that the operation needs during execution. For example, one such alternative may contain a set of two resources: a function unit (a real resource) and fetch width resource (a virtual resource) which is used to ensure that the fetch width constraints are not violated (an example using the fetch width resource is given in Section 2.4.5.1). If for example, there are 2 ALUs and the operation may use either one, that operation has two resource alternatives. For each resource alternative the corresponding entries in the *ReservationTable* must be checked. If there is a conflict

Time	Res 1 may		Res 2 may				Res m must	
0	op1 op2	pred_expr01			...	...	op3	true
1					...	...	op4	true
...	...	...	...	...	...	...	...	...
n			op5	pred_exprn2	...	...		

Figure 2.10: Predicate-aware reservation table

at any of those entries, then the operation cannot be scheduled on this resource alternative at this time; otherwise, it can. Scheduling an operation is accomplished at this level by reserving the entries in the RT that correspond to the first conflict-free set of entries that is found.

2.4.2.2 Predicate-aware Extensions

Predicate-aware scheduling is accomplished by using the Predicate Query System (PQS) [27] to determine the disjointness of two operations based on their predicates. The PQS analyzes operations to determine relations between predicate values. These relations (or facts) are stored as boolean expressions which can be efficiently manipulated. For a set of predicates, the boolean expression essentially represents the disjunction of all the paths on which these predicates evaluate to True. For example, the predicate expression that represents *p0* from Figure 2.1 is True, since the predicate evaluates to True on all the paths. To check if a predicate is disjoint from another predicate, the corresponding predicate expressions are ANDed. If the result is False, the predicates are disjoint, meaning that regardless of the execution path at most one of the predicates will be True at any given time. Otherwise, the predicates are not disjoint.

Predicate-aware scheduling uses a predicate-aware RT, as shown in Figure 2.10. Each entry in the predicate aware RT has two fields: a list of disjoint operations which have already reserved the entry, and a predicate expression (*pred_expr*), which

```
ResourceConflict(resource_alt, Operation, CurrTime) {
    pred = get_predicate (Operation);
    while (there are remaining resources in resource_alt) do {
        resource = next resource from resource_alt;
        rt_entry = ReservationTable[CurrTime][resource]
        if ( IS_DISJOINT (rt_entry.pred_expr, pred, PQS) == False)
            return True;
    }
    return False;
}
```

Figure 2.11: Predicate-aware **ResourceConflict**() function

represents the union of the predicates of those operations.

In predicate-aware scheduling, **FindTimeSlot** calls the predicate-aware **ResourceConflict** function (see Figure 2.11) which does the following. First, the operation's guarding predicate, *pred*, is obtained. For each entry, *ReservationTable*[*CurrTime*][*resource*] of the predicate-aware RT, a call is made to the **IS_DISJOINT** function. This function takes three arguments: operation's guarding predicate, *pred*, the predicate expression for this entry in the RT, *pred_expr*, and the PQS module, PQS, which is used to test the disjointness by performing the conjunction of the first two arguments, *pred* and *pred_expr*. If the conjunction is False, the **ResourceConflict** returns True; the operation is disjoint from every other operation in the list, and therefore it can also reserve the resource *resource* at time *CurrTime*. Otherwise the value returned is False and the operation is not disjoint from one or more of the operations currently in the list. Therefore, this operation cannot reserve this resource at $CurrTime$.

If there are no resource conflicts, the operation is placed into the operation list of the corresponding entry in the RT. The operation's predicate is then ORed into the current *pred_expr* in that RT entry to reflect the new condition under which the resource is busy.

As we said in Section 2.3, the predicate-aware scheduler divides machine resources

into two categories: may-use and must-use. May-use resources can be reserved in the same cycle by disjoint operations. Must-use resources can only be reserved by one particular operation in a given cycle, as on the baseline machine. The categorization rule is that every resource that is after the predicate nullification point in the pipeline is may-use. May-use resources can be reserved by disjoint operations because the operations whose predicates evaluate to False are discarded before those resources are used. Conversely, resources before the nullification point are must-use, and only one operation can reserve them at any time as these resources are used regardless of the operation's predicate value.

For our implementation, we add a pseudo must-use resource called the *fetch width* (or FW) resource. This resource limits the maximum number of operations that can be fetched in a given clock cycle. Since the fetch stage is a must-use resource, even operations that will be nullified must be fetched. Note that in general the fetch width can differ from the execution width, which is the maximum number of operations that can be simultaneously issued to function units in a given clock cycle. However, in our experiments these widths are the same.

2.4.3 'Extend-all' DPALS

Now that we have described the unified scheduling infrastructure that all three deterministic predicate-aware scheduling algorithms share, in this and the next two sections we describe the main differences for each of the three techniques. We start with the 'extend-all' DPALS algorithm which calls **DPAS-DDGLatency-Adjust** to extend the cmpp latency for every operation that could potentially be combined with another.

To illustrate 'extend-all' DPALS scheduling, DPALS is applied to the example

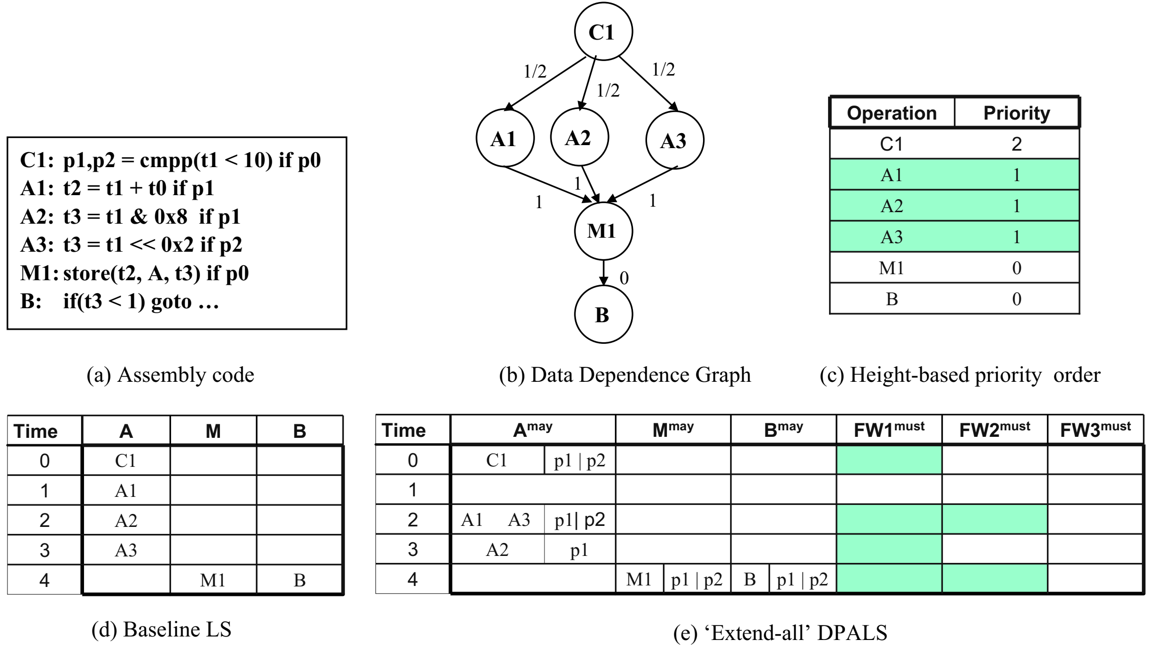


Figure 2.12: Example of 'extend-all' DPALS

in Figure 2.12. Figure 2.12(a) shows a predicated basic block. Figure 2.12(b) shows the corresponding data dependence graph. Each outgoing edge from a cmpp node is marked with the *shortlatency* and *extendedlatency* of the operation. In order to simplify this example we assume that *shortlatency* is 1 cycle and *extendedlatency* is 2 cycles. One of these latencies will be chosen for each of these edges during the latency extension phase. Note that in this case every predicated operation can be combined with at least one other predicated operation, hence every edge's latency is extended to 2 cycles. Finally, since scheduling algorithms schedule operations in decreasing priority order, Figure 2.12(c) shows the commonly used height-based priority (HBP) order of operations from highest to lowest. The HBP of an operation is computed as the length of the longest path from the operation to a node with

no immediate successors. Note that in our example the HBP priority order of the operations is the same before and after the cmpp latency extension step. Therefore, we only show each operation's priority before latency extension. Operations $A1$, $A2$ and $A3$ are highlighted because they have the same priority. We assume the deterministic machine model in Table 2.1.

The application of the baseline list scheduler to the example results in the schedule length of 5 cycles as presented in the SRT shown in Figure 2.12(d). Note that operations $A1$ through $A3$ are executed conditionally but reserve the ALU unconditionally.

When the 'extend-all' DPALS scheduler is applied to this example, as Figure 2.12(e) shows, it places $C1$ at Time 0 of SRT reserving the resource A, and the predicate expression is updated to $p1|p2$ since this operation occurs on both control paths. Here and in the other scheduling examples one FW resource is reserved (indicated by shading in Figure 2.12(e)) for each scheduled operation. Placing $C1$ frees up $A1$, $A2$ and $A3$ and makes them *ready* for scheduling. All three of them have the same priority and any of the three operations can be combined with at least one of the others: $A1$ can combine with $A3$, $A2$ can also combine with $A3$, and $A3$ can combine with both $A1$ and $A2$. The scheduler picks $A1$ and schedules it at its earliest schedule time which is Time 2 in the SRT. The entry's predicate expression is updated with $p1$ ($A1$'s guarding predicate) to reflect the condition under which the operation will reserve the resource. Next, $A2$ is scheduled in Time 3 also updating the predicate-expression to $p1$. Next, operation $A3$ is chosen by the scheduler and is placed in Time 2 together with its disjoint operation $A1$. $A3$ also reserves the A resource and updates the predicate expression to $p1|p2$, which means the resource is now reserved on both control paths: either $A3$ will utilize the resource on the False path ($p1$ is

False, $p2$ is True), or $A3$ will utilize the resource on the True path ($p1$ is True, $p2$ is False). Finally, $M1$ and B are placed in Time 4, resulting in 5 cycle schedule.

As can be seen, we have not gained any speedup over the 5-cycle baseline schedule despite the fact that $A1$ and $A3$ share the same resource. The failure to achieve any speedup is due to the fact that extending cmpp latency increases the critical path length by 1 cycle. This removes the 1-cycle reduction in the schedule length gained from the resource sharing by $A1$ and $A3$ at Time 2. If cmpp latency were extended to 3 cycles instead of 2, 'extend-all' DPALS would only achieve a 6 cycle schedule - a one cycle loss in performance relative to baseline schedule.

2.4.4 'First-fit' DPALS

As the previous example demonstrates, and as is experimentally shown in Section 2.5, extending cmpp latency often increases the length of critical path in the acyclic code regions and degrades the performance of DPALS. To address this problem, our second deterministic predicate-aware list scheduling algorithm, 'first-fit' DPALS, does not extend cmpp latency prior to scheduling, but takes advantage of combining opportunities as they arise and extends the latency only where needed.

'First-fit' DPALS, as opposed to 'extend-all' DPALS, schedules an operation at the first available cycle in which the resource required by this operation is available. If that cycle is less than $extendedlatency$ cycles away from its already scheduled cmpp producer, its predicate cannot be read early and therefore that operation must reserve its resource unconditionally at this cycle, and therefore cannot share the resource with other disjoint operations. In the compiler an unconditional reservation is accomplished by setting the predicate-expression of the corresponding SRT entry to be the union of all the control paths in the region; as a result, no other operation

<table>
<tr><th>Time</th><th colspan="2">A^{may}</th><th colspan="2">M^{may}</th><th colspan="2">B^{may}</th><th>FW1^{must}</th><th>FW2^{must}</th><th>FW3^{must}</th></tr>
<tr><td>0</td><td>C1</td><td>p1|p2</td><td colspan="2"></td><td colspan="2"></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td>A1</td><td>p1 | p2</td><td colspan="2"></td><td colspan="2"></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>A2 A3</td><td>p1|p2</td><td colspan="2"></td><td colspan="2"></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td colspan="2"></td><td>M1</td><td>p1|p2</td><td>B</td><td>p1|p2</td><td></td><td></td><td></td></tr>
</table>

Figure 2.13: Example of 'first-fit' DPALS

will be allowed to share its resource in this entry. If the first cycle where the resource is available is at least *extendedlatency* cycles away from its cmpp, the operation's predicate can be read early, and, therefore, the operation can conditionally reserve and share its resource with other disjoint operations.

The other difference between the 'extend-all' DPALS and 'first-fit' DPALS is which operation the scheduler chooses from *hpr_oplist_ready*, the list of the highest priority ready operations. As shown in the example in Section 2.4.3, an operation becomes ready when all of its predecessors have been scheduled. As we have seen, for 'extend-all' DPALS, the choice of the highest priority ready operation is unimportant from the resource sharing point of view, since each operation in the *hpr_oplist_ready* whose latency has been extended will be able to share its resource regardless of when it is scheduled. 'First-fit' DPALS, on the other hand, chooses from *hpr_oplist_ready* the ready operation with the *least combining potential*. An operation in the *hpr_oplist_ready* list is said to have the least combining potential if it can be combined with the fewest other operations in *hpr_oplist_ready*. Such an operation is chosen first, because if this operation gets scheduled unconditionally (that is fewer than *extendedlatency* cycles after its cmpp), it gives the remaining ready operations (particularly those with higher combining potential) a chance to be scheduled conditionally at a later time where they might take advantage of their higher potential for resource sharing.

We now show in Figure 2.13 how to apply the 'first-fit' DPALS scheduler to the example in Figure 2.12(a). As in 'extend-all' DPALS, it begins by placing $C1$ at Time 0 and updating the corresponding predicate-expression. The *hpr_oplist_ready* list currently contains three operations, $A1$, $A2$ and $A3$. $A1$ has a combining potential of 1 since it can only combine with $A3$, not with $A2$. By the same token, $A2$ also has a combining potential of 1. $A3$, on the other hand, can combine with either $A1$ or $A2$, and hence its combining potential is 2. Since $A1$ and $A2$ have the smallest combining potential, $A1$ is randomly chosen and is scheduled at Time 1. Since $A1$ is then only one cycle ($< extendedlatency = 2$ cycles) away from C1, the predicate-expression of the corresponding SRT entry is set to the union of all paths, i.e., $p1|p2$, to reflect the fact that this entry reserves its resource unconditionally and no sharing is allowed. The *hpr_oplist_ready* list now contains $A2$ and $A3$ each of which has a combining potential of 1, since each can combine with the other. $A2$ is chosen and is scheduled at Time 2, and the entry's predicate expression is set to $p1$. $A3$, the remaining operation in *hpr_oplist_ready*, is then also scheduled at Time 2 and shares the ALU with $A2$; the predicate expression is set to the union of their predicates, $p1|p2$. Note that although $A3$'s earliest scheduling time is Time 1 and $A3$ is disjoint from $A1$ (scheduled at Time 1), $A3$ cannot be scheduled together with $A1$, since $A1$ has reserved the resource unconditionally. Finally, $M1$ and B are scheduled at Time 3, resulting in a 4 cycle overall schedule - a 1 cycle improvement over baseline LS and 'extend-all' DPALS schedules in Figure 2.12(c) and Figure 2.12(d), respectively.

Note that if 'first-fit' DPALS had chosen $A3$ first instead of $A1$, $A3$ would have reserved the ALU unconditionally at Time 1. $A1$ and $A2$ then would have been scheduled at Time 2 and Time 3, respectively, forcing $M1$ and B to be scheduled at Time 4, resulting in a longer 5-cycle schedule for 'first-fit' DPALS. Thus choosing

one of the less-combinable $A1$ or $A2$ operations for scheduling before choosing the more-combinable $A3$ operation is critical to the success of 'first-fit' DPALS in finding a short schedule.

2.4.5 Additional Extensions for DPAMS

As in 'extend-all' DPALS, the deterministic predicate-aware modulo scheduler (DPAMS) also calls **DPAS-DDGLatency-Adjust**. Note that there is no 'first-fit' DPALS counterpart for DPAMS. As we will show in Section 2.5, modulo scheduled loops tend to be resource bound rather than latency bound, and extending the cmpp latency has a very small impact on the overall schedule length for most loops. Consequently, it is generally preferable to provide the maximum possible combining opportunity even if cmpp latencies are sometimes extended unnecessarily.

An additional predicate-aware extension for supporting DPAMS is to compute $ResMII_{dpams}$ in a predicate-aware manner. The baseline modulo scheduler, or IMS, computes a resource-constrained lower bound for each resource by adding up the number of cycles that each operation uses that type of resource and dividing that total by the number of resources of that type. The resource with the highest bound determines the overall $ResMII_{bams}$ for IMS.

For DPAMS, the deterministic predicate-aware resource-constrained lower bound ($ResMII_{dpams}$) is computed by a similar calculation, except that (as in MRT) a predicate expression is maintained for each resource usage that can now be shared by several operations. Whenever the current operation being considered can be disjointly combined with a previously combined set of operations, it is combined. When the new operation joins the set, the usage count of that resource is not incremented, but the predicate expression for that usage is updated to reflect that the current

operation now shares the usage of this resource with those previous operations. If the current operation is not disjoint from any such set of previous operations, a new usage is created for this resource, the usage's predicate expression is updated with current operation's predicate, and the resource usage count is incremented to account for this new usage. Again, the resource with the highest usage determines the overall $ResMII_{dpams}$ for DPAMS.

2.4.5.1 Example of Applying DPAMS

To illustrate predicate-aware scheduling, DPAMS is applied to the example in Section 2.2.1 (see Figure 2.1). The machine model in Table 2.1 is assumed with a cmpp latency of 2 cycles and a fetch width equal to the number of function units (3). DPAMS searches for a schedule with $II = 3$.

As each operation is scheduled at some time slot, the appropriate resource is marked at that time slot in both the SRT and MRT. In addition, the predicate expression of the corresponding SRT entry is updated.

Figure 2.14(a) and Figure 2.14(b) show the partial (up to *op4*) SRT and the final MRT after the DPAMS algorithm is applied to this loop. *op1* is scheduled at Time 0 of the SRT and the MRT (there is no MRT conflict) reserving resource M, and the predicate expression is updated to $p1|p2$ since this operation occurs on both control paths. Arithmetic operation *op2* is also on both control paths and is data dependent on two cycle *op1*. Therefore, the A resource is reserved at Time 2 in both the SRT and the MRT, with its predicate expression set to $p1|p2$ in the corresponding SRT entry. *op3*'s earliest scheduling time is Time 2, but it has a resource A conflict with the currently scheduled *op2* and is, therefore, scheduled at the next Time, 3, of the SRT (Time 0 of MRT), also with predicate expression $p1|p2$.

Time	A^{may}		M^{may}		B^{may}		FW1 (must)	FW2 (must)	FW3 (must)
0			op1	p1 | p2					
1									
2	op2	p1 | p2							
3	op3	p1 | p2							
4									
5									
6									
7	op4	p1							
8									

(a) Partial schedule reservation table

Time	A^{may}		M^{may}	B^{may}	FW1 (must)	FW2 (must)	FW3 (must)
0	op3		op1				
1	op4	op5					
2	op2		op6	op7			

(b) Final modulo reservation table

Figure 2.14: PAMS scheduling of the example in Figure 2.1

The earliest scheduling time for *op4* is Time 5 (since it is dependent on the two-cycle predicate defining operation *op3*), but *op2* uses resource A at Time 2, which causes resource conflicts with Time 5 since both map to Time 2 of the MRT (i.e. 2 and 5 are congruent modulo the II of the MRT, which is 3). By the same token, *op4* cannot be scheduled at Time 6 because of the conflict with *op3* currently scheduled at Time 3. So, *op4* gets scheduled at Time 7 of the SRT (Time 1 of the MRT) which has no conflicts. It reserves resource A and the predicate expression is set to $p1$ (*op4*'s guarding predicate) to reflect the condition under which the operation will reserve the resource.

The rest of the schedule is not shown in Figure 2.14(a) but is shown in Figure 2.14(b). The earliest scheduling time for *op5* is also at Time 5. But as with *op4*, it cannot be scheduled until Time 7. It gets scheduled at Time 7 of the SRT (Time 1 of the MRT), the same time as its disjoint operation, *op4*. *op5* also reserves the A resource and updates the predicate expression to $p1|p2$, to indicate that the

resource is now reserved on both control paths: either *op4* will utilize the resource on the False path ($p1$ is True), or *op5* will utilize the resource on the True path ($p1$ is False, $p2$ is True). Next, operation *op6* is scheduled at Time 8 of the SRT (Time 2 of the MRT). Finally, as is customary in IMS (and, hence, also in DPAMS), the region ending branch is replaced with a special control operation (called *brtop* in [45]) that is scheduled somewhere within the first II rows of the SRT. *op7* at Time 2 of the MRT in Figure 2.14(b) shows this operation. The result is a successful modulo schedule with II=3.

2.5 Performance Evaluation

We use an existing VLIW compiler technology, Trimaran [55], to evaluate the effectiveness of our technique. This compiler system is capable of performing if-conversion with hyperblock formation [36], scalar and modulo scheduling, and predicate analysis, among other back-end optimizations. We implemented the deterministic predicate-aware reservation table within the resource management module of ELCOR (Trimaran's back-end compiler); our deterministic predicate-aware resource manager uses Predicate Query System [27] to analyze predicated code and construct the relationships among the predicates, in particular the disjointness relationship. In addition, we implemented two deterministic predicate-aware list scheduling algorithms ('extend-all' DPALS and 'first-fit' DPALS) and a deterministic predicate-aware modulo scheduling algorithm (DPAMS) within ELCOR's list and modulo scheduling modules, respectively.

Benchmark	Description
cjpeg	JPEG image compression
djpeg	JPEG image decompression
epic	Efficient Pyramid Image coder
unepic	Efficient Pyramid Image decoder
g721encode	G.721 voice encoder
g721decode	CCITT G.721 voice decoder
ghostview	PostScript Interpreter & 38 & 107 & 91
gsmdecode	GSM 06.10 full-rate speech decoder
gsmencode	GSM 06.10 full-rate speech encoder
mesa	3-D graphics library
mpeg2enc	MPEG2 video encoder
mpeg2dec	MPEG2 video decoder
pegwitenc	public key encryption
pegwitenc	public key decryption
rasta	program for the rasta-plp processing
rawcaudio	Adaptive Difference Pulse Code Modulation compression
rawdaudio	Adaptive Difference Pulse Code Modulation decompression

Table 2.2: Benchmark set (MediaBench [33])

2.5.1 Benchmarks and Processor Models

All evaluations presented in this dissertation use the set of seventeen MediaBench [33] applications benchmarks shown in Table 2.2. Note there is a total of 22 MediaBench applications. We have not performed the experiments on the other 5 applications because the Trimaran compiler was not able to compile them correctly due to an error in one of Trimaran's modules that is difficult to fix at this time. The MediaBench suite was chosen because of its control-intensive nature, as well as the fact that these media applications spend the majority of their execution time in loops [19].

We use the notation (F,E,I,FP,M,B,C) to represent the processor in this study. **F** is fetch width, **E** - execution width, **I** - number integer units, **FP** - number of floating-point units, **M** - number of memory units, **B** - number of branch units, and **C** - latency of the predicate defining operation (cmpp). We use two base processors in our study: (4,4,2,1,1,1,1) and (6,6,4,2,1,1,1) called $P_{base}(4)$ and $P_{base}(6)$, respectively.

In addition, we assume 64 scalar and 64 rotating registers in our experiments and operation latencies that match the Itanium processor.

Each baseline processor $P_{base}(i)$ is compared with three corresponding deterministic predicate-aware processors $P_{dpas}(i, 1)$, $P_{dpas}(i, 2)$ and $P_{dpas}(i, 3)$ with the same number of resources as the corresponding baseline processor, but with a cmpp latency of 1, 2 and 3, respectively. Note that when we refer to predicate-aware processor *cmpp latency* in our experiments, we always imply an extended cmpp latency of *extendedlatency* cycles, that is the scheduling distance required between an operation and its cmpp to guarantee that the predicate value is available for read in the predicate read and dispatch stage; the short cmpp latency, *shortlatency*, which can only guarantee the availability of the predicate value in the execute stage, is always assumed to be one cycle in our experiments. Although maintaining a cmpp latency of one cycle in a deterministic predicate-aware architecture is almost impossible (it would effectively require predicate read and dispatch, register read and execute to happen in a single cycle, severely compromising the clock speed), the results when compared with those for cmpp latencies of 2 and 3 indicate the performance degradation that is specifically due to the increased cmpp latency.

We evaluate all of our applications, applying deterministic predicate-aware scheduling optimizations to every region in the entire code. The various measurements (such as schedule length, resource- and latency-constrained lower bounds, etc.) are reported as a weighted average, weighted by the execution frequency of each individual region in the benchmark. Clearly, DPAS can only benefit if-converted dpa-improvable regions of code (regions that contain at least two disjoint operations); DPAS will be ineffective for other code regions due to their lack of disjoint operations. The dpa-improvable regions can come from a large variety of the original source code control constructs; our

compiler can aggressively predicate nested *if-then* and *if-then-else* statements as well as *case selection* statements. We also assume that non dpa-improvable regions will execute with the deterministic predicate-aware support turned off and will be scheduled using the baseline list and modulo scheduling algorithms, abbreviated as BALS and BAMS in this section. In addition, whenever the DPAS schedule for a dpa-improvable region results in lower performance than the corresponding baseline schedule for that region, the baseline scheduler is used instead, so as to avoid degradation in performance. Therefore in our experiments the deterministic predicate-aware schemes, by incorporating baseline scheduling within them, will never produce a schedule that is worse than the baseline scheme. Note that for loops, a schedule with a lower II is considered better because it executes iterations at a faster rate; however, a negative speedup may occasionally occur relative to the baseline schedule for loops with a short trip count and a long start-up time in its modulo schedule, as explained in Section 2.5.2.4.

The reported DPAS processor speedup over the baseline processor for various regions is measured as the number of dynamic cycles that the baseline processor spends in these regions divided by the number of dynamic cycles that the DPAS processor spends in these regions.

The remaining sections show the results for 'extend-all' and 'first-fit' DPALS, for DPAMS, as well as for the overall speedup achieved by deterministic predicate-aware scheduling for each benchmark, and over the entire suite.

Benchmark	SL_{bals}	CPL1	CPL2	CPL3	$ResMSL_{bals}$	$ResMSL_{dpals}$	II_{bams}	RecMII1	RecMII2	RecMII3	$ResMII_{bams}$	$ResMII_{dpams}$
cjpeg	67.42	56.13	61.35	66.13	60.51	54.26	10.00	1.00	1.00	1.00	10.00	7.00
djpeg	20.13	14.30	15.15	16.22	16.70	15.20	58.64	1.00	1.00	1.00	58.64	53.17
epic	7.06	5.47	6.24	6.35	5.20	4.05	23.01	1.00	1.00	1.00	23.01	18.84
unepic	20.72	18.99	19.17	20.72	8.15	7.04	6.00	1.00	1.00	1.00	6.00	5.00
g721encode	33.62	22.87	25.12	27.29	28.58	20.66	30.00	1.00	1.00	1.00	30.00	21.00
g721decode	29.53	20.03	22.03	24.03	25.06	18.01	30.00	1.00	1.00	1.00	30.00	21.00
ghostscript	17.77	12.54	13.98	15.47	12.44	9.76	44.10	7.88	7.88	7.88	43.13	33.35
gsmdecode	37.07	33.07	34.33	34.53	35.00	31.20	27.92	7.93	9.68	11.44	27.73	24.95
gsmencode	40.84	33.53	35.26	38.00	38.47	34.18	76.40	7.93	8.47	9.02	75.93	47.69
mesamipmap	51.90	37.63	39.63	41.62	37.82	34.82	22.00	1.00	1.00	1.00	22.00	16.33
mpeg2dec	45.68	32.11	33.85	36.83	39.68	28.23	28.35	1.00	1.00	1.00	28.35	25.85
mpeg2enc	44.04	37.71	35.19	35.50	26.47	23.83	20.01	2.98	3.98	4.97	20.01	17.02
pegwitdec	21.68	20.23	21.21	22.29	13.05	12.00	20.67	1.97	1.97	1.97	20.67	18.70
pegwitenc	22.49	19.90	20.90	22.25	16.49	15.43	22.91	1.00	1.00	1.00	22.91	20.20
rasta	19.14	17.10	18.14	19.33	11.44	10.19	8.00	1.96	1.96	1.96	8.00	7.00
rawcaudio	25.80	20.00	20.99	21.99	18.97	12.99	26.00	20.00	25.00	30.00	24.00	22.00
rawdaudio	12.91	10.99	11.99	12.99	11.97	7.99	20.00	6.00	8.00	10.00	20.00	18.00
Average	30.46	24.27	25.56	27.15	23.88	19.99	27.88	3.86	4.47	5.07	27.67	22.18

(a) Scheduling Headroom for P(4)

Benchmark	SL_{bals}	CPL1	CPL2	CPL3	$ResMSL_{bals}$	$ResMSL_{dpals}$	II_{bams}	RecMII1	RecMII2	RecMII3	$ResMII_{bams}$	$ResMII_{dpams}$
cjpeg	54.98	56.13	61.35	66.13	30.44	29.09	5.00	1.00	1.00	1.00	5.00	4.00
djpeg	14.78	14.30	15.15	16.22	8.74	8.52	29.32	1.00	1.00	1.00	29.32	26.84
epic	6.55	5.47	6.24	6.35	3.19	2.49	11.50	1.00	1.00	1.00	11.50	11.17
unepic	20.43	18.99	19.17	20.72	4.08	4.07	4.00	1.00	1.00	1.00	4.00	4.00
g721encode	26.45	22.87	25.12	27.29	14.58	12.25	15.00	1.00	1.00	1.00	15.00	12.00
g721decode	23.16	20.03	22.03	24.03	12.85	10.67	15.00	1.00	1.00	1.00	15.00	12.00
ghostscript	14.44	12.54	13.98	15.47	6.43	6.22	22.55	7.88	7.88	7.88	21.58	19.63
gsmdecode	35.67	33.07	34.33	34.53	17.93	16.07	14.75	7.93	9.68	11.44	13.87	12.94
gsmencode	33.09	33.53	35.26	38.00	19.35	17.55	39.40	7.93	8.47	9.02	38.42	30.91
mesamipmap	41.56	37.63	39.63	41.62	19.11	18.15	12.33	1.00	1.00	1.00	12.33	10.67
mpeg2dec	34.48	32.11	33.85	36.83	20.10	18.32	14.19	1.00	1.00	1.00	14.19	13.04
mpeg2enc	48.94	37.71	35.19	35.50	13.70	12.28	11.00	2.98	3.98	4.97	10.00	9.01
pegwitdec	20.97	20.23	21.21	22.29	6.54	6.54	10.84	1.97	1.97	1.97	10.84	9.84
pegwitenc	21.09	19.90	20.90	22.25	8.47	7.83	11.56	1.00	1.00	1.00	11.56	10.58
rasta	18.06	17.10	18.14	19.33	6.52	6.20	4.00	1.96	1.96	1.96	4.00	4.00
rawcaudio	20.86	20.00	20.99	21.99	9.99	8.99	20.00	20.00	25.00	30.00	12.00	11.00
rawdaudio	11.91	10.99	11.99	12.99	5.99	5.98	10.00	6.00	8.00	10.00	10.00	9.00
Average	26.32	24.27	25.56	27.15	12.24	11.25	14.73	3.86	4.47	5.07	14.04	12.39

(b) Scheduling Headroom for P(6)

Table 2.3: Scheduling headroom estimates for the deterministic predicate-aware schedulers

2.5.2 Evaluation Results

2.5.2.1 Scheduling Headroom for DPAS

The goal of the deterministic predicate-aware scheduler is to reduce the length of resource constrained baseline schedules on a baseline machine of a given width. The deterministic predicate-aware scheduler takes the advantage of the gap between the upper-bound defined by the length of the baseline schedule and the lower bound which is the maximum of the resource-constrained and latency-constrained schedule lengths of the deterministic predicate-aware processor. For cyclic regions the resource-constrained schedule length is equal to $ResMII_{dpams}$ (Section 2.4.5). For acyclic re-

gions, the resource-constrained schedule length ($ResMII_{dpals}$) is computed in exactly the same way as for cyclic regions, ignoring all data dependencies in the original data dependence graph. The latency-constrained schedule length is the length of the critical path. For cyclic regions the critical path length is equal to $RecMII_{dpams}$. For acyclic regions the critical path length is the length of the longest path from the ddg node with no immediate predecessors to a ddg node with no immediate successors. The gap between these lower bounds and the upper bound (the length of the baseline schedule) constitutes the headroom for the deterministic predicate-aware scheduler.

Table 2.3(a) shows an estimate of the deterministic predicate-aware scheduler headroom on acyclic (columns 2-7) and cyclic (columns 8-13) dpa-improvable regions for the 4-wide deterministic predicate-aware machine. Table 2.3(b) shows similar data for the 6-wide machine. The data is presented for each benchmark shown in column 1. The last tow of each table shows the average over entire suite. Column 2 shows the length of the baseline acyclic schedule. Columns 3-5 show the critical path length for cmpp latencies 1, 2 and 3, respectively. Columns 6 and 7 show the resource-constrained schedule length for baseline processor, and the deterministic predicate-aware processor, respectively. Since the resource-constrained schedule ignores all data dependencies, cmpp latency has no effect here. Columns 8-13 show similar data for the cyclic regions.

For the dpa-improvable acyclic regions, we see from Table 2.3 that on average the critical path length for cmpp latencies of 1, 2 and 3 cycles, respectively is 20.32%(7.79%), 16.09% (2.89%) and 10.87% (-3.15%) shorter than the schedule length on the baseline 4(6)-wide machine (with a cmpp latency of 1 cycle). As the latency of the cmpp operation increases, the latency-constrained lower bound more closely approaches the length of the achieved baseline schedule, and eventually ex-

ceeds it. We can thus infer that cmpp operations are often on the critical path of the acyclic region, and that increases in cmpp latency will significantly reduce the headroom for DPALS in acyclic regions.

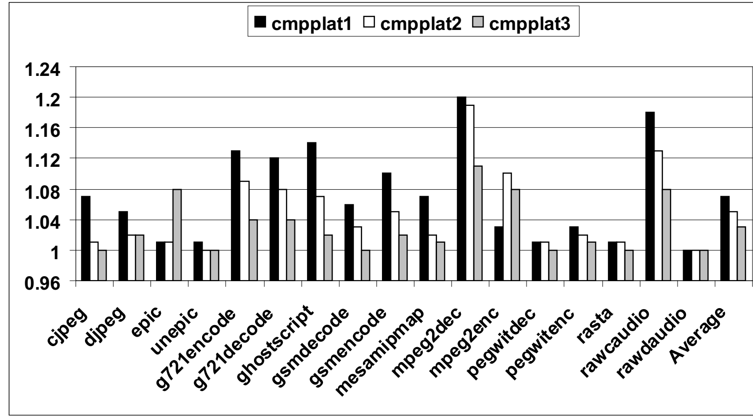
On the other hand, cmpp latency is not a limiting performance factor in cyclic regions as seen from the fact that $ResMII_{dpams}$ is much larger than $RecMII1$, $RecMII2$, and $RecMII3$ for both 4 and 6-wide machines. The only exceptions are the rawcaudio and rawdaudio benchmarks for which $RecMII$ in most cases approaches or exceeds $ResMII_{dpams}$. These two benchmarks have a dominating loop with a recurrence cycle that passes through all the cmpp operations.

2.5.2.2 'Extend-all' DPALS

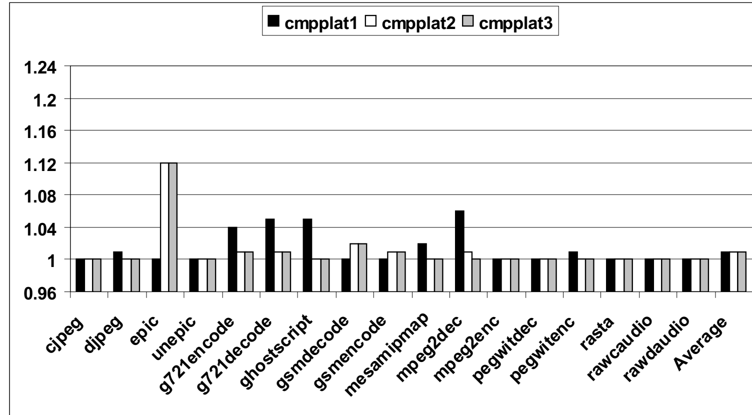
Figure 2.15(a) shows the speedup relative to the 4-wide baseline processor that is achieved by 'extend-all' DPALS on the acyclic regions only by the three 4-wide deterministic predicate-aware processors with cmpp latencies of 1, 2 and 3 cycles. Figure 3.19(b) shows similar data for the 6-wide processors.

As stated earlier, the schedule length for acyclic regions tends to be constrained by the latencies of the operations, with cmpp operations generally being on the critical path. Therefore, the performance of DPALS decreases with increased cmpp latency for both processor models, resulting in a very small speedup of 5% and 3% on two 4-wide machines with cmpp latency of 2 and 3 cycles, respectively, and 1% speedup for the 6-wide machines with the same latencies.

For epic we see that the 4-wide deterministic predicate-aware machine with cmpp latencies of 1 and 2 actually performs worse than the 4-wide deterministic predicate-aware machine with cmpp latency 3. This behavior is counterintuitive, since the increase in cmpp latency, should cause an increase in the critical path length, which



(a) Speedup of $P_{dpas}(4{,}1/2/3)$ over $P_{base}(4)$



(b) Speedup of $P_{dpas}(6{,}1/2/3)$ over $P_{base}(6)$

Figure 2.15: 'Extend-all' DPALS speedup over BALS for acyclic regions only

in turn should cause an increase in the schedule length. This anomaly is due to the side-effects of how the list scheduler works in general, and the way ELCOR schedules regions in particular. The ELCOR list scheduler consists of three main phases: pre-pass scheduling, register allocation and post-pass scheduling. During pre-pass scheduling a unique register drawn from an infinite number of virtual registers is allocated to each operation's destination operand within the region. Therefore, the data dependence graph constructed from this region is least constrained: the only

dependences that result in an edge between the operations are flow dependences, all the other dependences (such as anti and output- dependences) are eliminated by the virtue of the unique destination identifiers. In addition, if an operation is followed by a subroutine call operation, no edge is drawn between this operation and the call operation during the pre-pass scheduling stage, since the callee will use different virtual registers than the caller; hence, the callee does not need to save any of the caller's registers. The register allocation which is done after the pre-pass scheduling phase, tries to allocate the limited number of physical registers for a given region in order to minimize spilling. After this register allocation, post-pass scheduling is done. When the post-pass scheduler builds the data dependence graph, it must take the physical registers into account. Because physical registers are limited and hence are often reused, there may now be anti-, output-, and other dependence edges in the graph. In addition, if a long latency operation is followed by a subroutine call operation, and the operation's source operands happen to be caller saved (as a result of register allocation), new flow edges will be constructed between the long latency operation and the store operations that save these operands in memory and, most importantly, between the store operations and the subroutine call operation to assure that the source operand registers will be saved correctly across the call.

In the case of epic, there are a number of cases in one of the time-critical regions where a long latency divide operation, which lies on a ddg path with several cmpp operations, is followed by a subroutine call operation. For the 4-wide machine with longer cmpp latencies of 2 or 3, the pre-pass scheduler places the divide operation after the subroutine call operation, so that no edges are drawn between them during post-pass scheduling. For the 4-wide machine with the shorter cmpp latency of 1, it so happens that the pre-pass scheduler places the divide operation (with its source

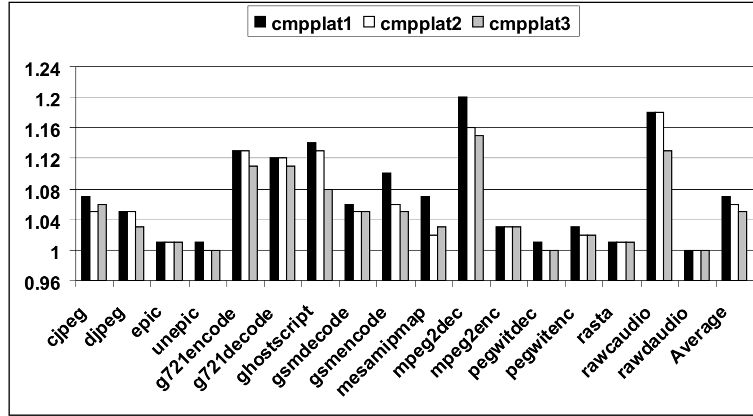
operands allocated to the caller saved registers by the register allocator) before the subroutine call operation, which causes long latency edges to be drawn between these operations and the register-saving stores during post-pass scheduling. A number of such long latency edges makes the schedule latency-bound on the 4-wide machine with cmpp latency of 1, and thus causes its increase in schedule length over the 4-wide machines with longer cmpp latencies of 2 and 3. Since this anomaly occurs in several other instances in this and in the next chapter, we will refer to it as the *long latency operation anomaly*.

We have seen that as we go to a wider machine, while keeping the latency of the deterministic predicate-aware machine fixed, the latency-constrained lower bound remains the same. However, the length of the baseline schedule (and the resource-constrained upper bound) decreases, approaching (and even finally falling below) the latency-constrained lower bound as resources (FW and/or others) are added. This reduction of headroom for the deterministic predicate-aware scheduler explains the degradation in the speedup achieved by 'extend-all' DPALS as we go from a 4-wide to the corresponding 6-wide baseline machine.

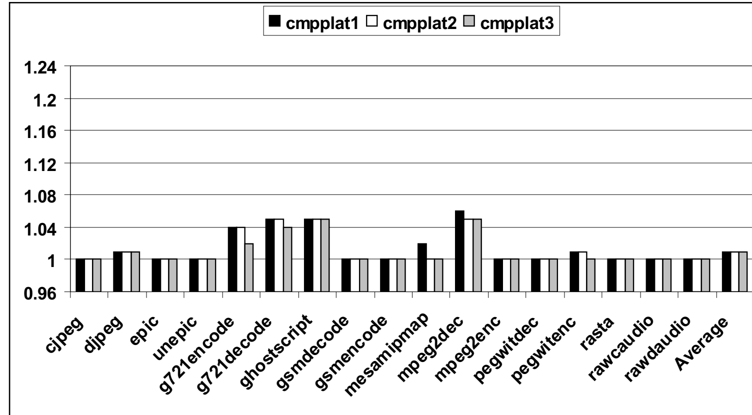
2.5.2.3 'First-fit' DPALS

In this section we study the performance of 'first-fit' DPALS which, unlike 'extend-all' DPALS, does not extend the operation's cmpp latency, but instead schedules the operation in the earliest time slot that satisfies data and resource constraints. DPALS simply allows the scheduled operation to combine whenever the opportunity arises and tries to increase those opportunities by scheduling less combinable operations first.

As we said in Section 2.4.4 , if an operation is scheduled at a cycle that is less than



(a) Speedup of $P_{dpas}(4,1/2/3)$ over $P_{base}(4)$



(b) Speedup of $P_{dpas}(6,1/2/3)$ over $P_{base}(6)$

Figure 2.16: 'First-fit' DPALS speedup over BALS for acyclic regions only

extendedlatency cycles away from its cmpp, this operation must reserve its resource unconditionally and cannot share the resource with other disjoint operations. On the other hand, if the operation is scheduled at a cycle that is at least *extendedlatency* cycles away from its cmpp, the operation can conditionally reserve and share its resource with other disjoint operations.

Figure 2.16(a) shows speedup over the 4-wide baseline processor that is achieved on the acyclic regions by 'first-fit' DPALS on the three 4-wide deterministic predicate-

aware processors with cmpp latencies of 1, 2 and 3 cycles.

By comparing with Figure 2.15 we see that 'first-fit' DPALS achieves 1% and 2% more speedup than 'extend-all' DPALS for the 4-wide machine and cmpp latencies of 2 and 3 cycles, respectively. This improvement comes from the fact that 'first-fit' DPALS does not extend the cmpp latency of every operation, which allows the latency-constrained lower bound for cmpp latencies of 2 and 3 cycles to remain the same as for cmpp latency of 1 cycle. The bound may be very optimistic, but it will never grow beyond the baseline schedule length, as happens with 'extend-all' DPALS as cmpp latency grows to 2 and then to 3 cycles.

We also see that for the 6-wide machines with cmpp latencies of 2 and 3, 'first-fit' DPALS achieves no performance improvement over 'extend-all' DPALS. As indicated by the results in Section 2.5.2.1, for the 6-wide machine the baseline schedule length is very close to the critical path length for almost all acyclic regions, which leaves a very small headroom for both DPALS schedulers to improve upon. However, for some individual benchmarks, such as g721encode, g721decode, ghostscript and mpeg2dec, 'first-fit' DPALS does achieve a noticeably better performance than 'extend-all' DPALS. For gsmdecode and gsmencode, 'first-fit' actually performs worse than 'extend-all' DPALS due to a long latency operation anomaly (see Section 2.5.2.2).

These results indicate that 'first-fit' DPALS should always be used instead of 'extend-all' DPALS, provided that a long latency operation anomaly can be prevented. One way to prevent this anomaly from happening is to modify the register allocator so that the source operands of long latency operations, such as divide, are callee saved, rather then caller saved as in the current version of our compiler. In this case, the caller will not issue the store operation to save the long latency operation's source registers. This will eliminate the long latency dependence chain between this

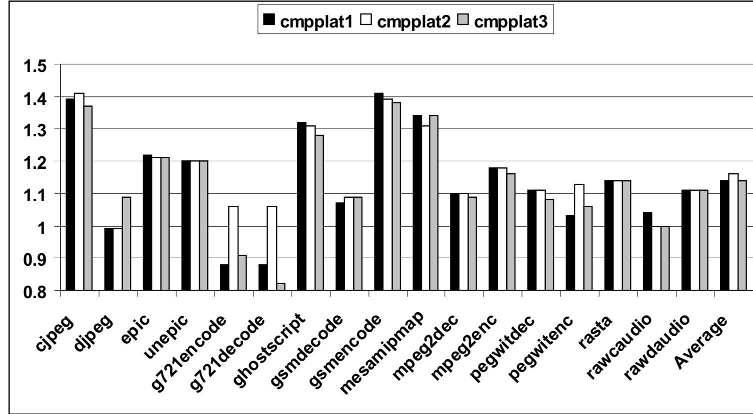
operation and the register-saving stores, and between the stores and the subroutine call operation, thus giving the scheduler freedom to move the long latency operation across the subroutine call where its latency may more effectively be masked. However, this is not done in our current implementation.

2.5.2.4 DPAMS Speedup

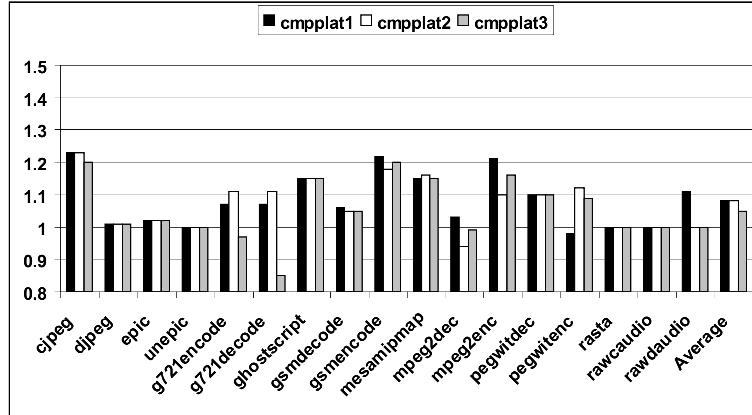
In this section we present the performance results of DPAMS. In order to explain the results, we first need to explain how the epilogue affects the performance of the modulo scheduled loops.

The epilogue stage count of a modulo scheduled loop is defined as 1 less than the ceiling of the loop's single iteration schedule length divided by the achieved II. The runtime of a modulo scheduled loop with the trip count, n, and epilogue stage count, esc, is equal to $(n + esc) \times II$ for a given II. If n is small, a large esc can significantly lower the performance of the modulo scheduled loop. As a result, it is possible for a low trip count loop with a larger II and shorter epilogue to outperform the same loop with a smaller II and longer epilogue. For example, a 5 iteration loop with esc of 4 and II of 8 will take a total of 72 cycles ($(5 + 4) \times 8) = 72$) to complete, whereas the same loop with esc of 1 and II of 10 will take 60 cycles ($(5 + 1) \times 10) = 60$) to complete: a 20% speedup over the same loop with a shorter II, but longer epilogue.

If the scheduler fails to find a valid schedule for a given II due to resource or latency constraints of the machine, it increases the II and tries again. Increase in II, generally leads to shorter epilogue size, since it enables each operation to have more scheduling slots within a given II, which may lead to a shorter single iteration schedule. Therefore, although smaller II is generally the primary performance objective, a small trip-count loop, which has higher II due to the resource or latency



(a) Speedup of P_{dpas}(4,1/2/3) over P_{base}(4)



(b) Speedup of P_{dpas}(6,1/2/3) over P_{base}(6)

Figure 2.17: DPAMS speedup over BAMS for cyclic regions only

constraints of the machine, but a shorter epilogue, may result in better performance than when the same loop is scheduled on the less constrained machine with smaller II and longer epilogue. We refer to this anomaly as *the low trip count* problem.

Figure 2.17(a) shows the cyclic regions only speedup relative to the 4-wide baseline processor that is achieved by DPAMS on the three 4-wide deterministic predicate-aware processors with cmpp latencies 1, 2 and 3 cycles. Figure 2.17(b) shows similar data for the 6-wide processors.

In the case of cyclic regions, the DPAMS lower-bound is determined by the resource-constrained schedule length of the deterministic predicate-aware processor ($ResMII_{dpams}$, see Table 2.3). The latency-constrained schedule length ($RecMII$) is not a limiting factor for either the 4-wide or 6-wide machine models (for given operation latencies, $RecMII$ is the same for both 4- and 6-wide machines, since $RecMII$ only depends on operation latencies and does not depend on other resource constraints of the machine). As Table 2.3 shows $RecMII$, even for a cmpp latency of 3, is much smaller than $ResMII_{dpams}$ for all of the applications except rawcaudio and rawdaudio. Therefore, as Figure 2.17(a) and (b) show, DPAMS achieves substantial speedups for all cmpp latencies (14%, 16% and 14% for the 4-wide machines with cmpp latency of 1, 2 and 3 respectively, and 8%, 8% and 5% for the corresponding 6-wide machines). The speedup is more modest for the 6-wide machines since the 6-wide machines have more resources than the 4-wide machines and thus resource sharing is less helpful and less is done.

We also see that the DPAMS speedup for an individual application varies with cmpp latency, although DPAMS is less sensitive than DPALS to cmpp latency. For some of the benchmarks, DPAMS performance decreases as cmpp latency increases. For example, for ghostscript performance decreases for 4-wide machines as cmpp latency increases from 1 to 2 and from 2 to 3 cycles. For such benchmarks the performance degradation for increased cmpp latencies occurs because higher cmpp latency increases the length of the loop epilogue (although the II remains the same), which leads to longer total execution time. The drop in performance due to increased cmpp latency is particularly visible for loops with low trip count. For example, for g721encode and g721decode the speedup drops below 1.0 for a cmpp latency of 3 cycles. Note that we do report this loss in performance here, instead of assuming

that these loops will be scheduled with the baseline modulo scheduler and reporting a speedup of 0%. The reason for this is as follows. In all these cases DPAMS still satisfies its primary objective of achieving a smaller II than the baseline scheduler. However, DPAMS makes no attempt to control the size of loop epilogue. DPAS only employs baseline scheduling when it fails to improve II; it may thus result in negative speedup when the epilogue size is significantly increased over the baseline, and the trip count is small.

On the other hand, for some benchmarks the performance increases as cmpp latency increases. For example, for djpeg on the 4-wide machine the speedup increases when the cmpp latency increases from 2 to 3. For g721encode the speedup increases as the cmpp latency increases from 1 to 2 for the 6-wide machine. These increases are due to the fact the DPAS machine with the higher cmpp latency sometimes simply fails to find a schedule for the same II as the machine with the lower cmpp latency. Hence, the II is increased which results in a shorter epilogue, which in turn can lead to better performance in loops with a low trip count. In some cases, as cmpp latency increases, DPAMS failure to find a valid schedule for a given II happens due to rotating register limitations; the increase in cmpp latency results in an increase in the length of the single iteration schedule, which causes an increase in the number of rotating registers required for the modulo scheduled loop. When there are insufficient rotating registers to support a schedule with the current II, the current II is increased. Finally in other cases, as cmpp latency increases, DPAMS failure to find a valid schedule for a given II happens due to an increase in the recurrence constrained lower bound, $RecMII_{dpams}$. The presence of recurrence cycles degrades the achieved performance since the modulo scheduler becomes more constrained in terms of where it can place operations in order to satisfy the loop carried dependence.

This can often result in the scheduler's failure to find a valid schedule for a given II, in which case the II must be increased.

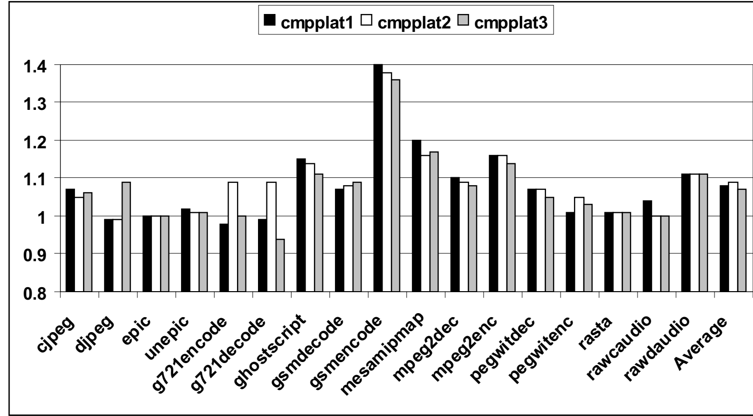
The performance increase for a cmpp latency of 2 compared with a cmpp latency of 1 is substantial in several cases (such as g721encode and g721decode); these few cases result in the small average performance gain of $P_{dpas}(4,2)$ over $P_{dpas}(4,1)$.

Finally, for some applications, such as rasta, the performance remains the same for all three cmpp latencies. This happens because the dominant loops are long trip count loops with a small $RecMII$ for all three cmpp latencies. Therefore $RecMII$ is not a limiting factor, and the size of the epilogue does not impact the overall performance since the loops spend most of their execution time in the steady-state kernel.

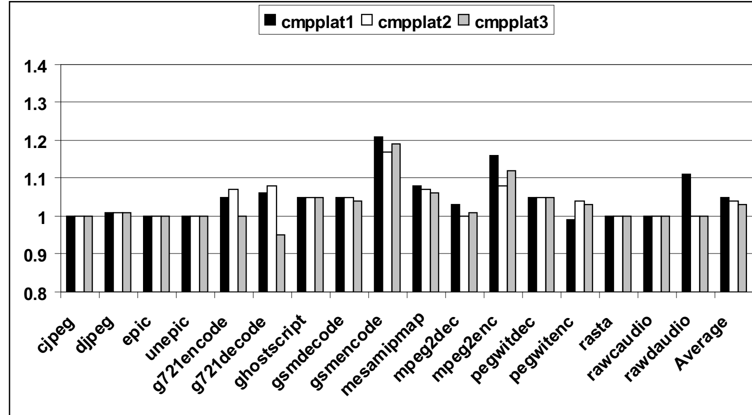
2.5.2.5 Overall Results

The full application results presented in this subsection assume that the acyclic regions are scheduled with 'first-fit' DPALS (rather than 'extend-all' DPALS). Figure 2.18 shows the overall speedup due to deterministic predicate-aware scheduling. Figure 2.18(a) shows the speedup of each of the three deterministic predicate-aware processors, $P_{dpas}(4,1)$, $P_{dpas}(4,2)$, $P_{dpas}(4,3)$, over the corresponding baseline processor, $P_{base}(4)$, for each application. Figure 2.18(b) shows similar data for the 6-wide processors.

The average speedups achieved over all applications are 8%, 9% and 7% for the 4-wide machines, and 5%, 4% and 3% for the 6-wide machines with cmpp latencies of 1, 2 and 3 cycles, respectively. Note that overall, the 4-wide deterministic predicate-aware machine with cmpp latency 2 ($P_{dpas}(4,2)$) outperforms the same machine with cmpp latency 1 ($P_{dpas}(4,1)$. This happens due to the fact that, as explained in



(a) Speedup of P_{dpas}(4,1/2/3) over P_{base}(4)



(b) Speedup of P_{dpas}(6,1/2/3) over P_{base}(6)

Figure 2.18: Overall Speedup

Section 2.5.2.4, $P_{dpas}(4, 2)$ achieves better performance than $P_{dpas}(4, 1)$ for the cyclic regions. Also note that in some cases, the speedup is less than 1.0, such as in case of the g721decode benchmark on the 4-wide DPAS machine with a cmpp latency of 3 cycles. Such a loss in performance relative to the baseline processor, is caused by a loss of performance in the benchmark's cyclic regions that correspondingly occurs with increased cmpp latency, as also explained in Section 2.5.2.4.

For the 6-wide machine with cmpp latencies of 2 and 3, most of the performance

improvement comes from DPAMS, since as Figure 2.16(b) shows, DPALS achieves an average improvement of only 1% for all 6-wide machines. Of course, the speedup achieved by these 6-wide machines on the entire application is smaller than the speedup achieved on dpa-improvable cyclic regions alone, since as Figure 2.5 shows, these regions constitute on average only 38% of the total baseline execution time over all these application benchmarks.

2.6 Summary

We have proposed and evaluated new deterministic predicate-aware scheduling techniques which can achieve better schedules on both acyclic and cyclic predicated code regions by reducing the resource wastage in VLIW/EPIC processors that occurs with predicated execution. These techniques enable the compiler to schedule operations on the same processor resource in the same cycle as long as two conditions hold: (i) the compiler can prove that the operations are guarded by disjoint predicates, and (ii) the processor nullifies all operations guarded under False before they use any may-use resource.

These predicate-aware modulo and scalar schedulers have been implemented and evaluated on the Mediabench suite of applications. The overall results show an average performance gain of 7% and 3% for 4-issue and 6-issue VLIW/EPIC processors, respectively. These gains are primarily due to loops where resources, and not dependences, often limit performance. For loops, predicate-aware scheduling achieves an average gain of 14% and 5% for these same processors.

CHAPTER 3

PROBABILISTIC PREDICATE-AWARE SCHEDULING

3.1 Introduction

In the previous chapter, we presented a deterministic predicate-aware scheduling technique (DPAS), wherein the compiler schedules disjoint predicated operations to conditionally oversubscribe the same resource. The hardware reads operations' predicates and discards the ones guarded under False conditions. Disjointness guarantees that runtime conflicts will never occur. Although, DPAS is effective for many regions, its application is still limited only to regions that contain at least one if-then-else clause. However, as will be shown in Section 3.2.2, there are many regions (both acyclic and cyclic) that contain many predicated operations, not necessarily disjoint, whose predicates are False a large fraction of the time during program execution. Allowing two or more of these operations to oversubscribe the same resource will result in better utilization of this resource and reduced schedule length, but will also result in a runtime conflict, whenever the predicates of more than one of these operations evaluate to True at the same time during program execution.

To overcome the problem of superfluous resource utilization by nullified operations, we propose probabilistic predicate-aware scheduling (PPAS). The central idea of PPAS, as in DPAS, is to allow over-subscription of may-use resources wherein multiple operations are allowed to reserve the same resource at the same time. However, PPAS is a generalization of DPAS, as it allows for dynamic over-subscription of resources to take place even when two or more resource-sharing operations may have their predicates evaluate to True at the same time during program execution, resulting in a resource conflict. By allowing some conflicts to occur, PPAS finds many more combinable operations than DPAS. Thus PPAS significantly increases the utilization of may-use resources and leads to improved processor performance. A secondary benefit of PPAS is that with resource constraints much lessened, more aggressive if-conversion can be applied to extract further benefit from branch elimination.

To deal with the problem of dynamic resource over-subscription, our proposed scheduling technique tries to estimate and account for conflicts while maximizing the benefits of over-subscription. Predicates are probabilistically analyzed using a combination of predicate profile information and predicate analysis [27]. Predicate profile information provides statistics on the expected number of times that a predicate will evaluate to True. Predicate analysis computes superset/subset and disjointness relations among predicates to identify when two or more predicates are guaranteed to conflict, or guaranteed not to. Probabilistic analysis is used to identify profitable opportunities for resource oversubscription. The scheduler takes advantage of these opportunities when they lead to a tighter schedule.

In this chapter, we present the necessary hardware and software extensions to support both acyclic and cyclic PPAS. In the next section, a motivational example is presented to illustrate the potential benefit of probabilistic predicate-aware scheduling

by applying it to a modulo scheduled loop. Section 3.3 describes a probabilistic predicate-aware processor with conflict detection and recovery. Section 3.4 describes the compiler support used to accomplish PPAS. The effectiveness of PPAS is evaluated experimentally in Section 3.5. The final two sections describe related work and present conclusions.

3.2 Motivation

3.2.1 Example of Probabilistic Predicate-aware Modulo Scheduling

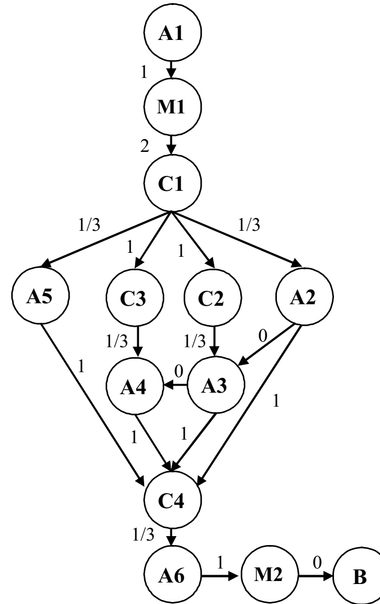
As in Chapter 2, we illustrate the application of probabilistic predicate-aware modulo scheduling (PPAMS) by applying it to a simple modulo scheduled loop. We use the same processor model as in Table 2.1, except that to support predicate-aware scheduling in this example we increase the cmpp latency to 3 cycles.

Figure 3.1(b) shows the assembly code of the if-converted loop body whose source code is shown in Figure 3.1(a). The predicate-defining operation *cmpp*, which replaces the branch in the original control statement, sets the predicate(s). For example $C1$ sets two disjoint predicates, $p1$ and its complement $p2$, replacing a branch in the original if-then-else statement. Operations which were on either the *then* or *else* paths are now guarded under the corresponding predicate. For example, $A2$ is guarded under $p1$ and $A5$ is guarded under $p2$. Figure 3.1(c) shows the data dependence graph for the code in Figure 3.1(b). Each node shows the corresponding operation, and its outbound arcs are labeled with the operation's latency. Note that the edges in the graph are all data dependences with the exception of the edge from $M2$ to B which is a control dependence. Figure 3.1(d) lists each predicate used by the code along

```
for(i=0; i < N; i++)
{
  lv = A[i];
  if ( lv < 20) {
   a=0;

   if ( lv > 10) {
     a = lv+2;
   }
   if ( lv == 14) {
     a = lv-4;
   }
  }
  else {
   a = lv | 0x7;
  }
  if ( a == 0) {
   t = a+lv;
   A[i] = t;
  }
}
```

(a) Original C source code



(c) Data Dependence Graph

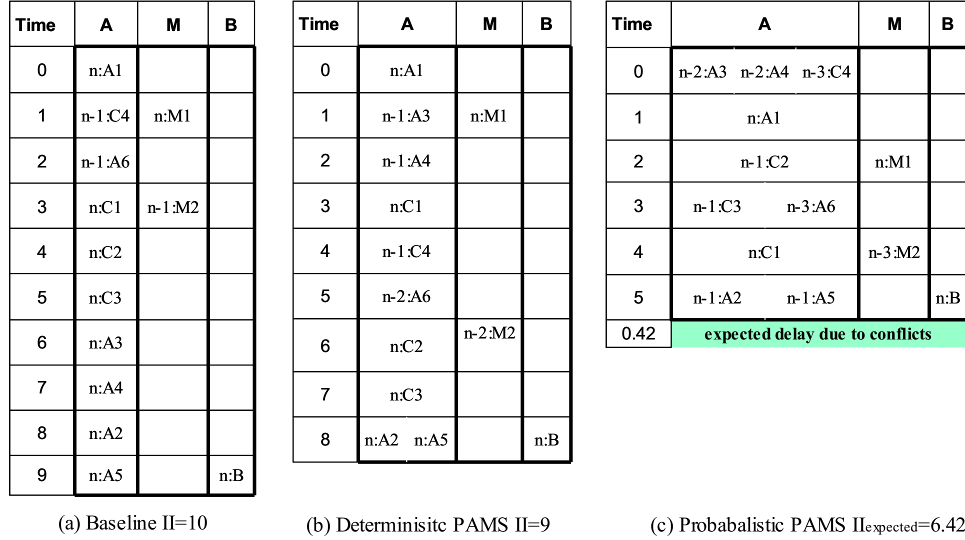
```
A1:  i = i + 4 if p0
M1:  lv = load(i, A) if p0
C1:  p1,p2 = cmpp(lv < 20) if p0
A2:  a = 0 if p1
C2:  p3 = cmpp(lv > 10) if p1
A3:  a = lv + 2 if p3
C3:  p4 = cmpp(i == 14) if p1
A4:  a = lv – 4 if p4
A5:  a = lv | 0x7 if p2
C4:  p5 = cmpp(a==0) if p0
A6:  t = a + lv if p5
M2:  store(i, A, t) if p5
B:   if(i < 400) goto A1 if p0
```

(b) Assembly code

Predicates	Freq.	Operation	Freq.	HBPr
p1:	0.2	A1:	1.0	10
p2:	0.8	M1:	1.0	9
p3:	0.09	C1:	1.0	7
p4:	0.01	C2:	1.0	6
p5:	0.11	C3:	1.0	6
		A2:	0.2	5
		A5 :	0.8	5
		A3:	0.09	5
		A4:	0.01	5
		C4:	1.0	4
		A6	0.11	1
		M2:	0.11	0
		B:	1.0	0

(d) Activation and execution frequencies

Figure 3.1: Example of if-converted loop body and its data dependence graph



Time	A	M	B
0	n:A1		
1	n-1:C4	n:M1	
2	n-1:A6		
3	n:C1	n-1:M2	
4	n:C2		
5	n:C3		
6	n:A3		
7	n:A4		
8	n:A2		
9	n:A5		n:B

(a) Baseline II=10

Time	A	M	B
0	n:A1		
1	n-1:A3	n:M1	
2	n-1:A4		
3	n:C1		
4	n-1:C4		
5	n-2:A6		
6	n:C2	n-2:M2	
7	n:C3		
8	n:A2 n:A5		n:B

(b) Determinisitc PAMS II=9

Time	A	M	B
0	n-2:A3 n-2:A4 n-3:C4		
1	n:A1		
2	n-1:C2	n:M1	
3	n-1:C3 n-3:A6		
4	n:C1	n-3:M2	
5	n-1:A2 n-1:A5		n:B
0.42	expected delay due to conflicts		

(c) Probabalistic PAMS $II_{expected}$=6.42

Figure 3.2: Three schedules for the example code segment

with its activation frequency, which is the fraction of all (profiled) loop iterations in which the predicate evaluates to True. It also shows the execution frequency of each operation which is equal to the activation frequency of its guarding predicate. Note that we use unconditional cmpp operations in this example. An unconditional cmpp operation *always* writes a value to its destination predicate. In this case, the predicate input acts as an input operand rather than as a guarding predicate, and the cmpp operation is never nullified. The value written to the destination predicate register is simply the conjunction of the input predicate and the compare result. Thus, the execution frequency of an unconditional cmpp operation is always 1.0. For example, $C2$ has an activation frequency of 1.0 even though the activation frequency of $p1$ is only 0.2. For more details about unconditional cmpp operations see [28].

The application of the conventional iterative modulo scheduler to this example results in the II=10 schedule presented in the MRT shown in Figure 3.2(a). The notation used for each operation is *iteration:opname*, where *iteration* is the iteration to which the operation belongs relative to the most recently initiated itera-

tion, which is called the current (n^{th}) iteration. Since each of ten ALU operations ($A1, A2, A3, A4, A5, A6, C1, C2, C3, C4$) must reserve the ALU resource at a different cycle to avoid conflict, $ResMII$=10 and this schedule is optimal. Note that $RecMII$=1 by default as there is no loop-carried dependence in this example.

Note that predicated operations $A2, A3, A4, A5$ and $A6$ are executed conditionally but reserve the ALU unconditionally in the cycles in which they are scheduled. Consequently an ALU cycle is wasted every time the guarding predicate of its assigned operation is False. We can increase ALU utilization and hence reduce the schedule length if we combine two or more of these predicated operations to share the ALU in the same cycle.

Deterministic Predicate-Aware Modulo Scheduling (DPAMS), presented in the previous chapter, guarantees that no conflicts will occur by combining only provably disjoint operations to share the same resource. A DPAMS schedule is shown in Figure 3.2(b). Disjoint operations $A2$ and $A5$ are scheduled in the same slot of the MRT. As a result, the ALU is always utilized in cycle 5, and the achieved schedule length is 9 cycles, an 11% performance improvement over the 10 cycle baseline schedule in Figure 3.2(a). Note that the ALU is still underutilized in cycles 1, 2 and 5 by operations $A3$, $A4$ and $A6$, but none of these operations can be combined by DPAMS because no pair of them is disjoint.

Probabilistic Predicate-Aware Modulo Scheduling (PPAMS) achieves a 6 cycle *static* schedule by combining operations $A3$ and $A4$ from the same iteration and $C4$ from the previous iteration to share the ALU at Time 0 as shown in Figure 3.2(c). In addition, $C3$ is also combined with $A6$ from two iterations earlier at Time 3. Note that none of these new combinations is disjoint. Because $C4$ always requires the ALU resource, each time at least one of operations $A3$ and $A4$ executes there will be at

least one conflict, and therefore a recovery delay due to conflict. Operations $C3$ and $A6$ will also conflict and delay the execution whenever $A6$ executes. If $A3$, $A4$ and $A6$ always execute, there will be at least 2 cycles of delay at Time 0 and 1 cycle of delay at Time 3 of the MRT in Figure 3.2(c), resulting in no improvement over DPAMS. But as shown in detail in Section 3.4.2.1 and Section 3.4.4.3, we expect an average of 0.42 cycles of penalty (as entered in the last row of MRT in Figure 3.2(c)). PPAMS thus achieves an *expected* schedule length ($II_{expected}$) of 6.42 cycles: a 40% performance improvement over DPAMS, and 56% over the baseline.

Note that even though any number of operations can simultaneously reserve each function unit, the processor can only fetch a maximum of three operations per cycle. Thus, the scheduler must also ensure that the fetch width (FW) constraint is not violated, namely that no more than three operations are scheduled in any MRT row. Note that for Figure 3.2(a) and (b), $FW = 2$ is sufficient, whereas to achieve the 6 cycle schedule in Figure 3.2(c), the processor must have $FW \geq \left\lceil \frac{\text{13 operations}}{\text{6 cycles schedule}} \right\rceil = 3$.

This simple example shows that allowing predicated operations to reserve the same resource in the same time-slot reduces the resource requirements and static schedule length for a predicated code segment, but will cause conflicts at runtime whenever two or more operations that are combined together have their predicates evaluate to True in the same cycle. In general, the goal of probabilistic predicate-aware scheduling (both acyclic and cyclic) is to decrease the *expected* schedule length by maximizing the degree of useful overlap (which reduces the *static* component of the expected schedule length), subject to controlling the *expected delay due to conflict* component of expected schedule length so that the benefit of reducing the static component is not undone.

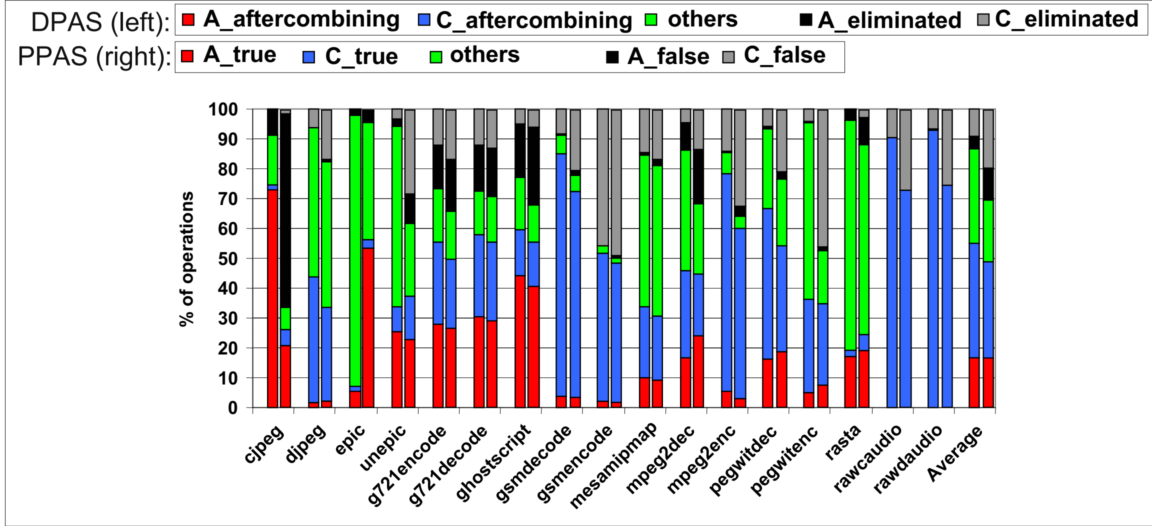


Figure 3.3: Characteristics of 'dpa-improvable' and 'ppa-improvable' Regions

3.2.2 Characteristics of Improvable Regions

The previous section showed that an isolated example can derive a significant benefit from probabilistic predicate-aware modulo scheduling. The central issue for further investigation is whether the regions, both acyclic and cyclic, in applications generally have properties that benefit from PPAS. For details of the experimental evaluations, see Section 3.5.

Two related metrics are: the fraction of time spent in the regions with predicated operations and the number of predicated operations in these regions that are nullified at runtime.

Figure 3.3 addresses these two metrics. There are two bars for each application or the overall Average. The left bar is the same as in Figure 2.5 (see Section 2.2) and is shown for comparison. The right bar shows the dynamic operation breakdown in terms of the type of code region they belong to and whether their predicates evaluate to True or False. As in Section 2.2 there are three kinds of regions. First are acyclic regions with at least one predicated operation and hence some opportunity

for combining; these we call acyclic ppa-improvable regions (A). Second are cyclic regions with at least one predicated operation, called cyclic ppa-improvable regions (C). As in Section 2.2.3, all these cyclic regions must be single-basic block inner-most loops, with no early exits, that can be scheduled with our cyclic scheduler. Third are *others* regions with no predicated operations and hence no opportunity for probabilistic predicate-aware combining.

The A and C regions are each further broken down into 2 distinct sub-regions where each sub-region is represented by a sub-bar in Figure 3.3, along with a fifth sub-bar representing *others* regions. Each of the five sub-bars indicates the percentage of all dynamic operations that it represents. The A_true sub-bar represents the dynamic operations from acyclic ppa-improvable regions whose predicates evaluate to True during program execution. Similarly the C_true sub-bar represents the dynamic operations from cyclic ppa-improvable regions whose predicates evaluate to True during program execution. A_false and C_false, respectively, represent the dynamic operations from A and C ppa-improvable regions and whose predicates evaluate to False.

By adding together the heights of the A_true and A_false sub-bars, we obtain the percentage of all dynamic operations that lie in acyclic ppa-improvable regions, an average of 27%. Similarly, by adding together the heights of C_true and C_false sub-bars, we obtain the percentage of all dynamic operations that lie in cyclic ppa-improvable regions, an average of 52%; over half of the operations over these benchmarks are from ppa-improvable loops. In fact rawcaudio and rawdaudio spend almost all of their execution time in ppa-improvable loops. The sum of all 4 sub-bars (72%) is the percentage of the original dynamic operations that lie in either an acyclic or cyclic, ppa-improvable region. Note that the *others* sub-bar represents the remaining 28%

of all the original dynamic operations, i.e. the dynamic operations before combining, summed over all benchmarks.

The overall benefit of PPAS depends not only on the frequency of the improvable regions, but also on the percentage of dynamic operations whose predicates evaluate to False during program execution. From Figure 3.3 we see that on average 30% of all dynamic operations (A_false (10%) + C_false (20%)) have their predicates evaluate to False and hence get nullified at runtime. This means that without predicate-aware scheduling, 30% of the time that a function unit is reserved, it does not do useful work.

It is interesting to compare the left and right bars. The 30% percent of all operations eliminated due to False predicates, the sum of A_false and C_false sub-bars of the right bar, is twice the percentage of operations eliminated as the result of deterministic combining, namely 14.5% which is the sum of A_eliminated and C_eliminated sub-bars of the left bar. The additional 15.5% of all operations that have False predicates represent nullified predicated operations for which there were no corresponding disjoint operations that the deterministic scheduler could combine them with.

By combining several predicated operations to share the same function unit in the same cycle, PPAS increases the utilization of that function unit. If the group of combined predicated operations is counted as one operation, then by combining several operations, PPAS effectively eliminates some of the nullified operations from the dynamic operation stream. However, note that the fact that 30% of all operations are nullified does not imply an upper bound on performance with probabilistic predicate-aware scheduling; as is the case with deterministic predicate-aware scheduling, the actual performance benefits can be higher or lower. Nevertheless, this 30% does indicate that PPAS has substantial headroom over what DPAS has achieved.

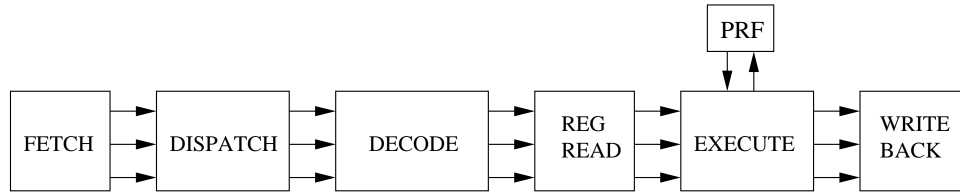
This headroom motivated the further development and analysis of PPAS as presented in the remainder of this chapter.

3.3 Probabilistic Predicate-aware VLIW Processor Architecture

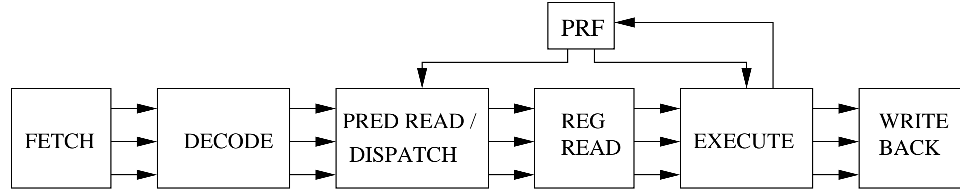
In this section we present the main extensions to the deterministic predicate-aware architecture described in Section 2.3, for supporting PPAS. Our baseline architecture is shown in Figure 3.4(a) and the deterministic predicate-aware architecture in Figure 3.4(b). The main change necessary to ensure the correct execution of the probabilistic predicate-aware schedule, as shown in Figure 3.4(c), is the addition of a *Resource Conflict Detection and Recovery Unit*. During the predicate read and dispatch stage, if more than one of the operations scheduled to execute on a particular function unit has its predicate set to True, the conflict is detected and the *conflict* signal is set. As a result the *stall* signal is sent to the fetch and decode stages, and a recovery process is executed.

A recovery process must address two important issues: first is how to dispatch the conflicting operations during recovery and second is when to start dispatching these operations.

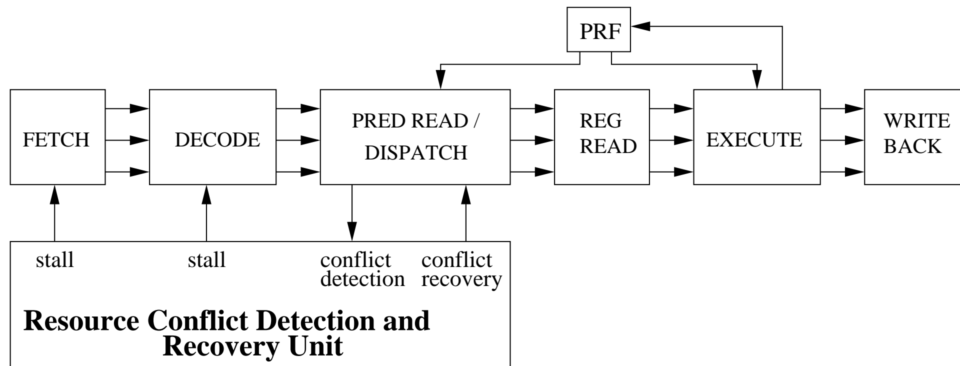
To address the **first** issue, we note that there are two alternative schemes to dispatch operations to recover from such a conflict. In the first scheme, those operations in the dispatch stage that have True predicates are dispatched in parallel to the function units that they were originally assigned to, and one such operation is dispatched to each function unit in each cycle until there are no such operations left for that function unit. This process continues until the entire instruction word is dispatched. This



(a) Baseline machine



(b) Deterministic predicate-aware machine



(c) Probabilistic predicate-aware machine

Figure 3.4: Baseline verses deterministic and probabilistic predicate-aware pipeline organizations

	A1 if 1	A2 if 1	A3 if 1	A4 if 0	A5 if 1
0	A1			A5	
1	A2				
2	A3				
	ALU1			**ALU2**	

(a) one operation per *assigned* FU

	A1 if 1	A2 if 1	A3 if 1	A4 if 0	A5 if 1
0	A1			A5	
1	A2			A3	
	ALU1			**ALU2**	

(b) one operation per any FU (not evaluated)

Figure 3.5: Design alternatives to dispatch conflicting operations

scheme results in a simpler design, but a longer stall time than the second scheme. In the second alternative scheme, an operation may be dispatched to any available function unit that can serve it (rather than only to the function unit it was originally assigned to, as in the first scheme). This scheme will reduce the stall time, but its design is somewhat more complex. In our experiments we use the former, simpler recovery scheme.

Figure 3.5 demonstrates these two recovery schemes with a simple example. The bottom row of each table indicates that there are two function units, ALU1 and ALU2. The top row of each table shows 5 operations together with the values of their guarding predicates, and indicates that 3 operations with True predicates are assigned to ALU1 (a 3-way conflict) and 2 operations are assigned to ALU2 with no conflict because one operation has a False predicate. The first recovery scheme, shown in Figure 3.5(a) takes a total of three cycles to dispatch all operations, as it can simultaneously dispatch operations $A1$ and $A5$ to function units ALU1 and ALU2 in cycle 0, and $A2$ and $A3$ to ALU1 in cycles 1 and 2. The second scheme takes only 2 cycles to complete as it can reassign $A3$ to ALU2, as shown in Figure 3.5(b).

The **second** issue is whether the recovery itself may begin immediately, in the same cycle in which a conflict is detected, or requires additional time to initiate. To model the effect on performance, a machine parameter called the conflict detection

and recovery unit latency ($CDRL$) is used in the experiments. We investigate $CDRL$ values of 0 cycles, wherein the first conflicting operation is dispatched in the conflict detection cycle itself, and 1 cycle, wherein it is dispatched one cycle later.

To address the issue of delay and complexity of the conflict detection and recovery unit, we note that there already are reasons why a dispatcher may be unable to dispatch an operation or group of operations in a particular cycle, e.g. the input buffer of a required execution unit may be full. The conflict described above is just one more simple reason, and the 'stall' or interlock mechanism it needs to invoke is no more complex than the mechanisms that already exist in today's microprocessors for this purpose. The stall signal is produced by simple combinational logic; as soon as the predicates are read early in the cycle, this logic generates the stall signal if more than one predicate that has reserved the same resource has a True value. This 'stall' signal could simply be added as an additional trigger to any of those existing mechanisms to effect a stall or fetch/decode restart, as appropriate.

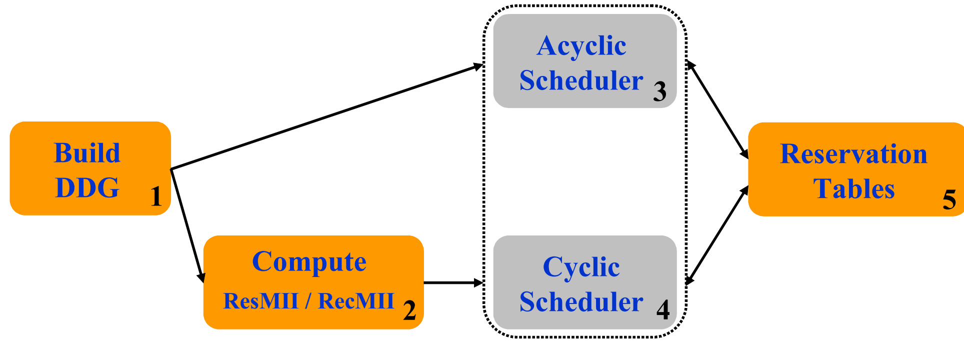
As with DPAS (see Section 2.3), it is possible to selectively (rather than always) increase the cmpp latency of the operations scheduled to execute on the probabilistic predicate-aware machine, so as to restrict the increases in the critical path lengths. Only some operations will see an increased cmpp latency; these operations will execute conditionally and require their may-use resources only when their predicates are True. All other operations will see a cmpp latency of 1, which means that they will always execute unconditionally on their resources, i.e. their execution frequency is increased to 1.0.

3.4 Probabilistic Predicate-aware Scheduling

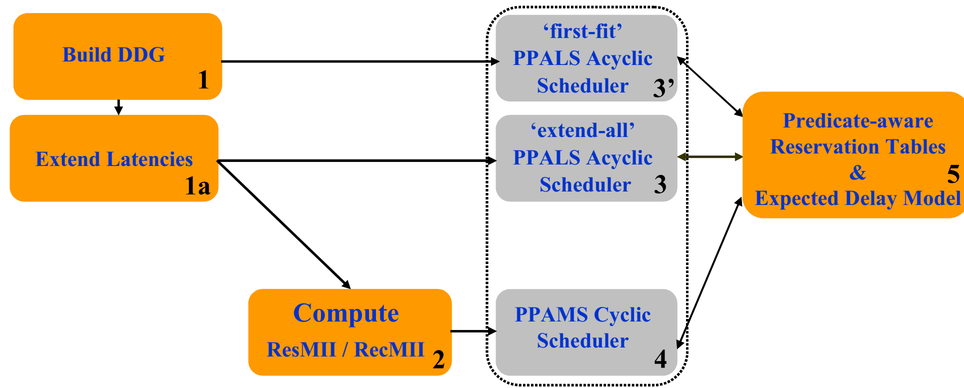
In this section, we present the details of two probabilistic predicate-aware scheduling (PPAS) algorithms: probabilistic predicate-aware list scheduling (PPALS) and probabilistic predicate-aware modulo scheduling (PPAMS). Both algorithms are the extensions of the corresponding conventional list and modulo scheduling algorithms. They decrease the expected *dynamic* schedule length by relaxing resource constraints. They achieve this by allowing arbitrary operations to reserve the same resource in the same cycle, while estimating and accounting for the resulting expected delay due to conflicts.

Figure 3.6(a) shows the five main scheduling steps for conventional list and modulo scheduling algorithms. These steps are described in Section 2.4. Our probabilistic predicate-aware scheduling technique extends the conventional scheduler as shown in Figure 3.6(b). Each extension is discussed in the following sections. Section 3.4.1 describes the data dependence graph latency extension step (Step 1a) which follows DDG construction (Step 1) and selectively extends the cmpp latency for every predicated operation that can potentially be combined with some other predicated operation in a way that might lead to some performance improvement. Section 3.4.2 describes the probabilistic expected delay model that is integrated into the resource reservation module to compute the expected delay due to conflicts (Step 5). We have designed and implemented three scheduling algorithm: two alternative PPALS algorithms (Steps 3 and 3'), and one PPAMS algorithm (Step 4). Both PPALS algorithms and PPAMS use the delay computation model to derive better acyclic and cyclic schedules as described in Section 3.4.3 and Section 3.4.4, respectively.

The first algorithm, called 'extend-all' PPALS (Step 3), is described in Section 3.4.3.1 and is used to schedule acyclic regions. 'Extend-all' PPALS performs



(a) Conventional Scheduler



(b) Probabilistic Predicate-Aware Scheduler

Figure 3.6: Scheduling algorithm flowchart

the latency extension step, so that when an operation with extended cmpp latency is scheduled, it is always far enough from its cmpp to be able to conditionally reserve its resource and thus share it with other predicated operations in its region.

The second algorithm, called 'first-fit' PPALS (Step 3') may alternatively be used to schedule acyclic regions and is described in 3.4.3.2. Unlike 'extend-all' PPALS, 'first-fit' PPALS does not require the latency extension step. Instead, an operation can be scheduled at the earliest time that satisfies its resource and dependence constraints regardless of the distance from its cmpp producer. However, if the operation is scheduled so close to its cmpp producer that its predicate is not available during

early read, the operation must reserve its resource unconditionally; this will increase the operation's execution frequency to 1.0 and, hence, can result in higher delay due to conflicts when combining with other operations.

PPAMS (Step 4), which is used to schedule cyclic regions, uses the same delay computation model as PPALS in conjunction with its backtracking to derive a cyclic schedule with the objective of minimizing II, as described in Section 3.4.4. Similar to 'extend-all' PPALS, PPAMS performs the latency extension step, as shown by the arrows in Figure 3.6(b). In addition PPAMS requires the $ResMII$ and $RecMII$ bounds to be computed (in the probabilistic predicate-aware manner) as also shown by the arrows in Figure 3.6(b).

3.4.1 DGG Latency Extension

As we said in Section 3.3, it is possible to selectively increase cmpp latency for the predicated operations so as to restrict the growth of the critical path length in a region. 'Extend-all' PPALS and PPAMS perform this latency extension immediately after the DDG construction step.

PPAS optimistically extends the cmpp latency of a given predicated operation if (i) it can potentially be combined with some other predicated operation, and (ii) such combining may result in some performance improvement. As in DPAS (see Section 2.4.1), the actual performance improvement may vary depending on the number of function units of the machine and the latencies of the operations within the region.

The procedure **PPAS-DDGLatency-Adjust** performs the latency extension as shown in Figure 3.7. It takes four input parameters: *ddg* - the data dependence graph of the region, *regiontype* - a region type flag, which indicates whether the region is acyclic or cyclic, *shortlatency* - a short latency of the cmpp operation, and finally an

```
PPAS-DDGLatency-Extend (ddg,  regiontype, shortlatency, extendedlatency)
1     /* phase 1:  mark 'mustres' bit of each operation */
2     for each predicated operation node, op1, in ddg  do
3       op1.mustres = 0
4       if op1 is unconditional cmpp then
5         op1.mustres = 1
6       end if
7       if  regiontype is acyclic then
8         independenceSet = all operations in ddg independent from op1
9         if independenceSet is empty then
10           op1.mustres = 1
11        else
12           if   there exists other operation, op2, in independenceSet  s.t. delay due to conflict
13                that results from combining op1 and op2 is less than 1.0
14                op1.mustres = 0
15           end if
16        end if
17    end if
18    /* phase 2: adjust cmpp latencies */
19    for each cmpp operation outgoing edge, cmpp_outedge do
20      op1 = cmpp_outedge.destination;
21      if op1.mustres == 1 then
22        setlatency(cmpp_outedge, shortlatency)
23      else
24        setlatency(cmpp_outedge, extendedlatency)
25      end if
26    end for
```

Figure 3.7: PPAS latency extension procedure

extendedlatency - an extended longer latency of the cmpp operation.

The procedure consists of two phases. The first phase (lines 1-17) sets the *mustres* bit field of each predicated operation to 0 if it can be combined with some other operation potentially resulting in some performance improvement. Otherwise, the *mustres* bit is set to 1. More specifically, at the beginning of the first phase the *mustres* of each predicated operation, $op1$, is optimistically initialized to 0 (line 3). However, if $op1$ is an unconditional cmpp (line 4), its *mustres* bit is then set to 1, because in this case the operation always executes and thus requires its resource unconditionally. The first phase is complete at this point if the region is cyclic.

Lines 8-16 are for acyclic regions only. Line 8 computes the independence set, *independenceSet*, of all operations in ddg which are independent of $op1$. Since two dependent operations must be scheduled at different times in an acyclic region, an operation can be combined only with operations from its independence set. Thus if the *independenceSet* is empty (line 9), $op1$ cannot be combined with any other operation in this region. In this case we can safely set $op1$'s *mustres* bit to 1, since extending its cmpp latency cannot result in any performance improvement, but may in fact worsen the performance by increasing the critical path length. If the *independenceSet* is not empty, the test in lines 12-13 is executed. The test checks if there exists some other predicated operation, $op2$, from the *independenceSet*, such that the expected delay due to conflict that would result from combining $op1$ and $op2$ is less than 1.0. This test is based on the main idea of PPALS (described in Section 3.4.3) that allows two operations to be combined only if the overall conflict delay does not exceed 1 cycle. If the tested condition is True, the $op1$'s *mustres* bit is set to 0, since this operation has at least one combining opportunity that may improve performance.

If the region is cyclic, $op1$'s mustres bit remains 0 (line 3). Indeed, for cyclic regions, we optimistically assume that $op1$ can always be combined with another predicated operation either from the same iteration (if there are some operations that are independent of $op1$) or from a different iteration. In the latter case, the latency extension procedure optimistically ignores loop-carried dependences and assumes that operations across different iterations are always independent. Also note that for cyclic regions, unlike for acyclic regions, we do not require that the operations can be combined only if the overall delay due to conflicts does not exceed 1 cycle. As discussed in Section 3.4.4 and experimentally shown in Section 3.5.2.4, PPAMS can sometimes achieve higher performance by combining several operations even when

the conflict delay is greater than 1 cycle.

In the second phase (lines 18-26), the cmpp latency of those operations with $mustres = 1$ is set to *shortlatency* (line 22): these operations will reserve their resources unconditionally. All other operations have $mustres = 0$ and their cmpp latency is extended to *extendedlatency* cycles (line 24): those operations will reserve their resources conditionally.

3.4.2 Computing Expected Delays Due to Conflicts

A key feature of PPAS is its method of estimating the expected delay due to conflicts when two or more predicated operations share the same resource. We use the example code in Figure 3.1 and the machine model in Table 2.1 to demonstrate this technique.

We define an *execution vector* for a group of predicates as an assignment of a particular boolean value to each of these predicates. In general, the overall expected delay, ED_{cfl}, for a group of predicated operations is the sum of $Ed_{cfl}(ev)$ over all legal execution vectors, ev, that result in conflicts, where $Ed_{cfl}(ev)$ is the expected delay due to execution vector ev.

$$ED_{cfl}(\text{group of predicated operations}) = \sum_{\forall \text{ legal conflicting execution vectors, } ev} Ed_{clf}(ev)$$

An execution vector is illegal if its assignment of boolean values will never occur. For example, execution vector $(p1 = T, p2 = T)$, where $p1$ and $p2$ are disjoint predicates, is illegal. For a particular legal execution vector, ev, that results in a conflict,

the conflict delay, $Ed_{cfl}(ev)$, is the product of the number of extra delay cycles, nextradelaycycles(ev), due to that execution vector times the probability of occurrence, $P(ev)$, of that execution vector.

$$ED_{cfl}(\text{group of predicated operations}) =$$
$$\sum_{\forall \text{ legal conflicting execution vectors, } ev} Ed_{clf}(ev) =$$
$$\sum_{\forall \text{ legal conflicting execution vectors, } ev} \text{nextradelaycycles}(ev) \times P(ev)$$

Note that $\text{nextradelaycycles}(ev) = (\text{CDRL} + \text{ndispatchcycles}(ev) - 1)$, where CDRL is conflict detection recovery latency (either 0 or 1), ndispatchcycles(ev) is the total number of cycles required to dispatch all conflicting operations in ev (namely, the maximum number of True operations in ev that are assigned to any one function unit), and the -1 accounts for the fact that when there are no conflicts, it takes exactly one cycle to dispatch all the operations in ev. Hence,

$$ED_{cfl}(\text{group of predicated operations}) =$$
$$\sum_{\forall \text{ legal conflicting execution vectors, } ev} (\text{CDRL} + \text{ndispatchcycles}(ev) - 1) \times P(ev)$$

For example, assuming a conflict detection and recovery latency of 1 cycle (CDRL=1), the delay expected when operations $A3$, $A4$ and $A6$, guarded by the predicates $p3$, $p4$ and $p5$, respectively, are scheduled at the same time on a single ALU is computed as follows:

$$
\begin{aligned}
&ED_{cfl}(A3\ ?\ p3,\ \ A4\ ?\ p4,\ \ A6\ ?\ p5) \\
&\qquad = (1+2-1)\times P(p3=T, p4=T, p5=F) \\
&\qquad + (1+2-1)\times P(p3=T, p4=F, p5=T) \\
&\qquad + (1+2-1)\times P(p3=F, p4=T, p5=T) \\
&\qquad + (1+3-1)\times P(p3=T, p4=T, p5=T) \\
&\qquad = 2\times P(p3=T, p4=T, p5=F) + 2\times P(p3=T, p4=F, p5=T) \\
&\qquad + 2\times P(p3=F, p4=T, p5=T) + 3\times P(p3=T, p4=T, p5=T)
\end{aligned}
$$

The first three terms on the right side compute the expected delay for all possible execution vectors that cause exactly one conflict. For example, when the execution vector $(p3 = T, p4 = T, p5 = F)$ occurs, it will cause $A3$ and $A4$ to conflict over the ALU, resulting in 2 extra cycles of delay: 1 cycle (CDRL) to detect the conflict, plus 2 more cycles to dispatch the two conflicting ALU operations, minus 1 cycle to account for the fact that when there are no conflicts, it takes exactly one cycle to dispatch all operations. Similarly, to recover from conflicts in $(p3 = T, p4 = T, p5 = T)$ will take 3 extra cycles.

3.4.2.1 Example of Computing the Probability of an Execution Vector

To compute the probability of a given execution vector, we introduce a Predicate Relationship Graph (**PRG**), which is similar in concept to the partition graph in [27] and the predicate hierarchy graph in [36]. A PRG represents the relationship between the predicates in the predicated block of code. Each node in the graph corresponds to a predicate and is labeled with the activation frequency of this predicate, as obtained

from a profile run. There are two kinds of edges which connect the nodes of the PRG: implication edges (I-edges) and disjointness edges (D-edges). There is an I-edge from predicate pi to pj if pj implies pi, i.e. if whenever pj is True, pi is also True; this means that in the original non-predicated code, an operation guarded by pj lies on every control path that passes through an operation guarded by pi. A D-edge indicates that the two predicates it connects are disjoint, i.e. whenever one of these two predicates evaluates to True during execution, the other will evaluate to False; this means that there is no control path that contains both pi and pj.

One important assumption that we make in this approach is the *independence* assumption. This assumption states that any two predicates that are not connected by an edge are deemed to be independent, i.e. the probability that one of them evaluates to True (or False) is the same, regardless of the value of the other predicate. Note that no two predicates defined in different loop iterations are ever connected by an edge since predicates are always local to the loop iteration in which they are defined and are never carried into the successive loop iterations; that is, a predicate is never defined in one iteration and used in a later iteration. Thus any two predicates defined in different loop iterations are always independent. There are two main motivations behind the independence assumption. First, as shown in 3.4.2.2, it allows us to derive an exact closed form expression for $P(ev)$, which is the probability of occurrence of a given execution vector, ev. Second, as the experimental results in Section 3.5 show, this assumption is fairly accurate in that the error resulting from applying this assumption to both PPALS and PPAMS is very small.

The PRG for the example code of Figure 3.1 is shown in Figure 3.8. The root of a PRG is predicate $p0$ which always evaluates to True during the execution of this code, hence it has a frequency of 1.0. Obviously, $p1$ implies $p0$, as do $p2$ and $p5$. $p3$ implies

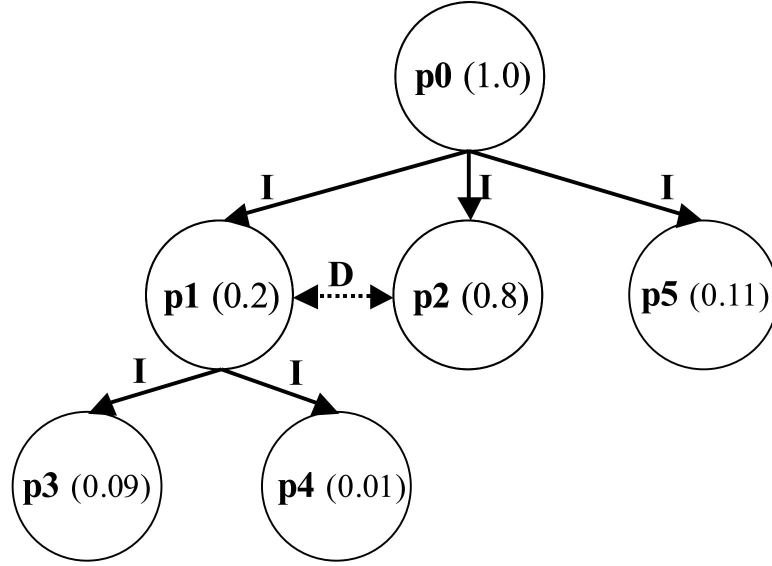


Figure 3.8: Predicate Relationship Graph (PRG)

$p1$ since the cmpp operation C2 may only set $p3$ if its guarding predicate, $p1$, is True (and its condition, $lv > 10$, is also True); likewise, $p4$ implies $p1$. Since implication is a transitive relation, we avoid cluttering the graph with redundant I-edges, e.g. we do not need to include an I-edge from $p0$ to $p3$ since there is an I-edge from $p0$ to $p1$ and one from $p1$ to $p3$. There is a D-edge between $p1$ and $p2$ since these two predicates (as defined in C1) are disjoint. Furthermore, since $p3$ and $p4$ imply $p1$, they must also be disjoint from $p2$. Therefore the corresponding D-edges between $p3$ and $p2$ as well as $p4$ and $p2$ are not shown, as they can be inferred from the graph.

We use the PRG, the independence assumption above, and techniques from elementary probability to compute the probability of occurrence of an execution vector. In this section we demonstrate this computation for a few interesting cases, the general derivation is given in the next section.

Case I. Suppose $A1$, $A3$, and $A6$ of the example are scheduled in the same row of an MRT. One probability we would need to compute is $P(p0 = T, p3 = F, p5 = T)$ for the case that $A1$ and $A6$ execute, but $A3$ does not. Furthermore, suppose that $A1$ and

$A3$ are scheduled at the same time in the SRT, but $A6$ is scheduled at a different time in the SRT, i.e. the $A1$ and $A3$ in the MRT row are from the same loop iteration, but the $A6$ in that MRT row is from a different iteration. Since predicates from different iterations are not disjoint nor can they imply one another, the independence assumption applies between them. Hence,

$$P(p0 = T, p3 = F, p5 = T) = P(p0 = T, p3 = F) \times P(p5 = T)$$

To compute $P(p0 = T, p3 = F)$, we notice from the PRG that $p0$ evaluates to True and $p3$ evaluates to False, when one of two events occurs: (i) $p0$ evaluates to True, $p1$ evaluates to True and $p3$ evaluates to False, or (ii) $p0$ evaluates to True and $p1$ evaluates to False (in which case $p3$ is guaranteed to be False by the implication relationship). Hence,

$$P(p0 = T, p3 = F) = P(p0 = T, p1 = T, p3 = F) + P(p0 = T, p1 = F)$$

We use the definition of conditional probability to compute each of the terms. For the second term,

$$P(p0 = T, p1 = F) = P(p1 = F | p0 = T) \times P(p0 = T)$$

For the first term,

$$
\begin{aligned}
P(p0 = T, &p1 = T, p3 = F) \\
&= P(p3 = F \mid p0 = T, p1 = T) \times P(p0 = T, p1 = T) \\
&= P(p3 = F \mid p1 = T) \times P(p1 = T \mid p0 = T) \times P(p0 = T)
\end{aligned}
$$

$P(pi = T)$ is the frequency of times that pi evaluates to True. $P(pi = T|pj = T)$ is the conditional edge transition probability that pi evaluates to True given that pj evaluates to True. When pi implies pj, $P(pi = T|pj = T)$ is equal to $\frac{P(pi=T)}{P(pj=T)}$. Furthermore, $P(pi = F|pj = T) = 1 - \frac{P(pi=T))}{P(pj=T)}$. Expanding each term, we obtain

$$
\begin{aligned}
P(p0 = T, &p3 = F, p5 = T) = P(p0 = T, p3 = F) \times P(p5 = T) \\
&= [P(p3 = F|p1 = T) \times P(p1 = T|p0 = T) \times P(p0 = T) \\
&+ P(p1 = F|p0 = T) \times P(p0 = T)] \times P(p5 = T) \\
&= [\frac{P(p1 = T) - P(p3 = T)}{P(p1 = T)} \times \frac{P(p1 = T)}{P(p0 = T)} \times P(p0 = T) \\
&+ \frac{P(p0 = T) - P(p1 = T)}{P(p0 = T)} \times P(p0 = T)] \times P(p5 = T)
\end{aligned}
$$

Hence, by plugging in the values, we obtain

$$
\begin{aligned}
P(p0 = T, &p3 = F, p5 = T) \\
&= [\frac{(0.2 - 0.09)}{0.2} \times \frac{0.2}{1.0} \times 1.0 + \frac{(1 - 0.2)}{1.0} \times 1.0] \times [0.11] \\
&= (0.11 + 0.8) \times 0.11 = 0.10
\end{aligned}
$$

Case II. Next, suppose $A3$ and $A4$ both execute, but $A6$ does not and the probability $P(p2 = F, p3 = T, p4 = T)$ is desired. Suppose all three operations are

scheduled at the same time in both the MRT and the SRT, i.e. these combined operations all come from the same loop iteration. First we notice that since $p2$ is disjoint from both $p3$ and $p4$, the condition $p2 = F$ is redundant and can be dropped from the execution vector, that is

$$P(p2 = F, p3 = T, p4 = T) = P(p3 = T, p4 = T)$$

Further, from PRG we see that both $p3$ and $p4$ imply $p1$ which, in turn, implies $p0$. Hence,

$$\begin{aligned} P(p3 = T, p4 = T) &= P(p0 = T, p1 = T, p3 = T, p4 = T) \\ &= P(p4 = T, p3 = T \mid p1 = T, p0 = T) \times P(p0 = T, p1 = T) \\ &= P(p4 = T, p3 = T \mid p1 = T) \times P(p1 = T \mid p0 = T) \times P(p0 = T) \end{aligned}$$

From the independence assumption, it follows that

$$P(p4 = T, p3 = T \mid p1 = T) = P(p4 = T \mid p1 = T) \times P(p3 = T \mid p1 = T)$$

Thus, finally,

$$P(p3 = T, p4 = T) = \frac{0.01}{0.2} \times \frac{0.09}{0.2} \times \frac{0.2}{1.0} \times 1.0 = 0.0045$$

Case III. Certain execution vectors are illegal since they will never occur, and hence have 0 probability. For example, $P(p1 = T, p2 = T) = 0$ since $p1$ is disjoint from $p2$ as indicated by the D-edge in the PRG. As another example, it is impossible for both $A2$ and $A6$ to execute and $M2$ not to execute if all three operations are from the same iteration; this is due to the fact that $A6$ and $M2$ are guarded under the same predicate $p5$, and clearly $P(p1 = T, p5 = T, p5 = F) = 0$.

3.4.2.2 General Formula to Compute the Probability of an Execution Vector

In general, the probability of an execution vector ev, given the PRG, can be expressed in terms of the PRG edge transition probabilities.

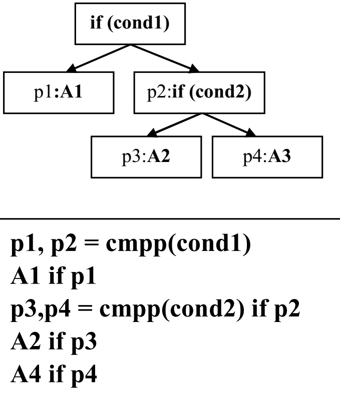
We start out by calling **Eliminate-Redundant-Conditions**, shown in Figure 3.9(a), to eliminate the redundant conditions from the execution vector, ev, and verify that the remaining conditions are legal. This routine iterates over all pairs of execution vector predicates found in ev.

If each predicate implies the other (line 3), there are two cases to consider. In the first case, the predicates have the same boolean values (line 4), in which case one of them is eliminated because if it is True then the other will always be True as well, and likewise for False. In the second case, the predicates have opposite boolean values (line 6), in this case the execution vector is illegal and will never occur; consequently, the False value is returned and the execution vector is not considered.

If the second predicate implies the first (line 10), but not vice versa, there are also two cases to be considered. In the first case both predicates are False (line 11), and therefore the second predicate is eliminated, since if the first predicate is False, the second predicate must be False (i.e., $P(\text{second predicate} = \text{False} \mid$

```
Eliminate-Redundant-Conditions(ev)
1    for each element, ev1, of ev do
2        for each element, ev2, which is not ev1, of ev do
3            if ev1 and ev2 predicates imply each other then
4                if ev1 and  ev2 have the same boolean values then
5                    eliminate ev1
6                else
7                    return False /* illegal vector */
8                end if
9            end if
10           if ev2 predicate implies ev1 predicate then
11               if ev1 has boolean value F and ev2 has boolean value F then
12                   eliminate  ev1
13               end if
14               if ev1 has boolean value F and ev2 has boolean value T then
15                   return False /* illegal vector */
16               end if
17           end if
18           if ev1 and ev2 predicates are disjoint then
19               if ev1 has boolean value T and ev2 has boolean value F then
20                   eliminate ev2
21               end if
22               if ev1 has boolean value F and ev2 has boolean value T then
23                   eliminate ev1
24               end if
25               if ev1 has boolean value T and ev2 has boolean value T then
26                   return False /* illegal vector */
27               end if
28           end if
29       end for
30   end for
31   if  ev contains complete disjoint set of predicates p1, p2, …, pn
32       in ev, s.t. corresponding ev1, ev2, …, evn have boolean value F then
33       return False
34   end if
35   return True and ev
```

(a) Algorithm pseudo-code



(b) Example of a **complete** set of disjoint predicates

Figure 3.9: Algorithm to eliminate redundant conditions from an execution vector

first predicate = False) = 1). In the second case, which occurs when the first predicate is False, but the second is True (line 14), the execution vector is illegal (i.e., $P(ev) = 0$), since it violates the implication relationship, the algorithm returns False and the execution vector is not considered.

If both predicates are disjoint (line 18), there are three cases. In the first case, the first predicate is True and the second predicate is False (line 19), the second predicate can be eliminated since if the first is True, the second must be False (i.e.,

P(second predicate = False | first predicate = True) = 1), and hence the second will not impact the probability of the execution vector. The second case (line 22) is similar, but with the first predicate False and the second one True. The third case occurs when both disjoint predicates have the value True (line 25): this is illegal, hence the False value is returned and the execution vector is not considered.

Note that it is possible for two disjoint predicates to both have value False in the execution vector; this case is special and is handled by the code in lines 31-34 of the **Eliminate-Redundant-Conditions** algorithm. To explain this condition, we define a set of disjoint predicates $p1$, $p2$, ..., pn to be *complete* if one and only one of them must evaluate to True at runtime. An example of a complete set of disjoint predicates is given in Figure 3.9(b), which shows a control flow graph and the corresponding predicated code. Predicates $p3$ and $p4$ are disjoint but not complete disjoint, since both of them can legally evaluate to False at runtime, namely whenever $p2$ evaluates to False. On the other hand, $p1$, $p3$ and $p4$ form a complete set of disjoint predicates, since one and only one of these predicates must evaluate to True on each control path. Hence the *if* condition in line 31 tests if there exists a complete disjoint set of predicates in ev, where every predicate in the set is assigned value False in ev. If this condition occurs, the execution vector is illegal, since by the definition of a complete set of disjoint predicates, exactly one of these predicates must evaluate to True in a legal execution vector. Lines 25 and 26 ensure that no more than one predicate in the set is assigned True; lines 31 through 33 ensure that not all of the predicates in the set are assigned False.

Note that when ev satisfies none of the cases that **Eliminate-Redundant-Conditions** explicitly checks for, the original ev is returned with the value True at line 35. Similarly when no illegal case is detected, the non-redundant ev is returned

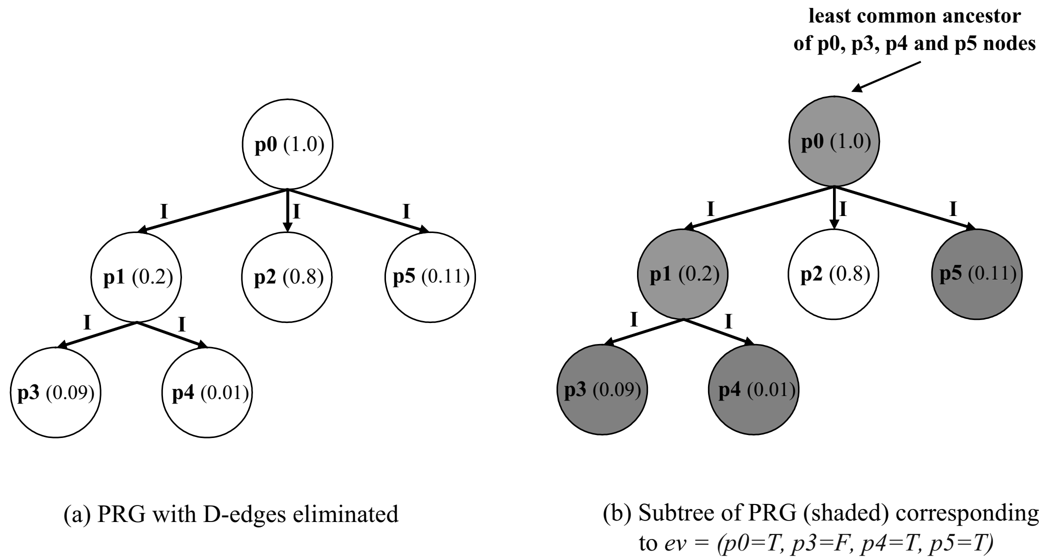


Figure 3.10: Example of PRG reduced to a tree

with the value True at line 35.

Assume now that ev is a legal execution vector, with all the redundant conditions eliminated, as found after **Eliminate-Redundant-Conditions** is run and returns a value of True. Initially, we assume that no two predicates in ev are disjoint (we later remove this assumption), i.e. assume there is no D-edge between any pair of nodes in the corresponding PRG. In this case the PRG reduces to a tree because no node in the PRG will have more than one immediate predecessor: due to the way that programs are written, a predicate, p, can never imply two other predicates, $p1$ and $p2$, if the two implied predicates are independent. If, however, $p1$ and $p2$ are dependent, then one implies the other, say $p2$ implies $p1$. But in this case, the implication edge from $p2$ to p is redundant (by transitivity) and is not shown in the PRG. Figure 3.10(a) shows the PRG from Figure 3.8 which is reduced to a tree after its D-edge is eliminated.

An execution vector, ev, must correspond to some subtree $S0$ in the PRG. The root of the subtree corresponds to the least common ancestor of all the ev predicates. Figure 3.10(b) highlights the subtree $S0$ of PRG which corresponds to the execution

vector $ev = (p0 = T, p3 = F, p4 = T, p5 = T)$. The nodes of $S0$ are shaded. The least common ancestor of all the predicates in ev is the $p0$ node. Note that it is possible that a least common ancestor is a predicate that is not explicitly listed in ev. For example, the least common ancestor of $ev = (p3 = F, p4 = T, p5 = T)$ is also $p0$.

We can write ev in the following form: $ev = (p0 = T, [G1], ..., [Gn])$, where $p0$ corresponds to the root of $S0$, and Gi $(i \geq 1)$ is the group of conditions in ev whose predicates belong to the ith subtree, Si, of the root of $S0$. For example, $ev = (p0 = T, p3 = F, p4 = T, p5 = T)$ is written as $ev = (p0 = T, [p3 = F, p4 = T], [p5 = T]) = (p0 = T, [G1], [G3])$. The group $G1 = [p3 = F, p4 = T]$ is the group of conditions whose predicates $p3$ and $p4$ belong to the leftmost subtree $S1$ of $S0$. $G3 = [p5 = T]$ is the group that has a single condition whose predicate $p5$ belongs to the rightmost (in this case the third) subtree $S3$ of $S0$. Note that the least common ancestor's predicate always has the value True in an execution vector, ev, that results in conflicts; otherwise, by implication, the execution vector would contain only False conditions and there would be no conflicts.

Suppose, as in this example, the group $G1$ does not contain the predicate $p1$ which is at the root of the first subtree $S1$. If the group $G1$ contains at least one True predicate (of the form $p = T$), we can safely extend $G1$ by adding a new condition: $p1 = T$. The extended execution vector is equivalent to the old execution vector, and has the same probability of occurrence, since if p executes, then by implication $p1$ must also execute, as $p1$ is the root of the subtree and is implied by every node in the subtree. Hence, the extended execution vector will have the following form: $(p0 = T, [p1 = T,\ G1], ..., [Gn])$. In this example, the group $G1 = [p3 = F, p4 = T]$ would thus be extended with the condition $p1 = T$ to become $G1 = [p1 = T, p3 = F, p4 = T]$ since whenever $p4$ is True, $p1$, which is the root of the first subtree of S0,

must also be True. As $G3$ already includes its subtree root, the extended execution vector in this example is $ev = (p0 = T, [p1 = T, p3 = F, p4 = T], [p5 = T])$.

On the other hand, suppose the group $G1$ contains no True predicates, only False ones. In this case, we represent the execution vector $ev = (p0 = T, [G1], [G2], ..., [Gn])$ as the union of two non-overlapping execution vectors, $ev1 = (p0 = T, [p1 = F], [G2], ..., [Gn])$ and $ev2 = (p0 = T, [p1 = T, G1], [G2], ..., [Gn])$. This representation is valid because all of group $G1$'s predicates are False in one of two cases: (i) either the root predicate $p1$, which is implied by each of $G1$'s predicates, is False, in which case all $G1$'s predicates must be False, or (ii) $p1$ is True, but the predicates in $G1$ are False. Note that both $ev1$ and $ev2$ contain the predicate $p1$ of the root of the first subtree $G1$. In addition, since these two vectors are non-overlapping, $P(ev) = P(ev1) + P(ev2)$. Following through with the example, consider a new execution vector with $p4 = F$, namely, $ev' = (p0 = T, [p3 = F, p4 = F], [p5 = T])$. At this step ev' is represented as the union of $ev1' = (p0 = T, [p1 = F], [p5 = T])$ and $ev2' = (p0 = T, [p1 = T, p3 = F, p4 = F], [p5 = T])$.

The above two steps, which extend the execution vector, and when appropriate represent it as a union of two non-overlapping execution vectors, are applied iteratively to the remaining groups of ev as necessary until the given ev is transformed into the union of a number of execution vectors in *normal* form. A normal form execution vector includes $p0 = T$ as its first element, followed by a number of groups where each group is associated with a subtree whose root predicate, pi, is connected by an implication edge to $p0$ in the PRG. Furthermore, each group must contain a value assignment for the root predicate pi of its corresponding subtree; if that assignment is $pi = F$ then that group contains no other predicate assignments. We represent a normal form execution vector with the notation $ev = (p0 = T, [p1 = B1, G1], [p2 =$

$B2, G2], ...[pn = Bn, Gn])$, where Bn is a boolean value, True or False. Note that if $pi = F$, the corresponding group Gi is empty.

We now show how to compute the probability of an execution vector ev that is written in normal form. Since we assumed that there are no D-edges between any pair of nodes in PRG, the predicates corresponding to the subtree roots are independent. Hence, by the definitions of elementary conditional probability and our independence assumption:

$$\begin{aligned} P(ev) &= P(p0 = T, [p1 = B1, G1], [p2 = B2, G2], ...[pn = Bn, Gn]) \\ &= P([p1 = B1, G1], [p2 = B2, G1], ...[pn = Bn, Gn] \mid p0 = T) \times P(p0 = T) \\ &= P([p1 = B1, G1] \mid p0 = T) \times \ldots \times P([pn = Bn, Gn] \mid p0 = T) \times P(p0 = T) \end{aligned} \tag{3.1}$$

$P([pi = F, Gi] \mid p0 = T)$ can be easily computed. As we said, if pi is False, Gi is empty, and hence $P(pi = F \mid p0 = T) = 1 - \frac{P(pi=T)}{P(p0=T)}$. Otherwise, we have $P([pi = T, Gi] \mid p0 = T)$ which is computed as follows:

$$\begin{aligned} P([pi = T, Gi] \mid p0 = T) &= \frac{P([pi = T, Gi], p0 = T)}{P(p0 = T)} \\ &= \frac{P([Gi] \mid pi = T, p0 = T) \times P(pi = T, p0 = T)}{P(p0 = T)} \\ &= \frac{P([Gi] \mid pi = T) \times P(pi = T \mid p0 = T) \times P(p0 = T)}{P(p0 = T)} \\ &= \frac{P(pi = T, [Gi])}{P(pi = T)} \times P(pi = T \mid p0 = T) \end{aligned} \tag{3.2}$$

By looking at Equation 3.1 and Equation 3.2, we see that the probability $P(ev)$ of the execution vector ev corresponding to the subtree in PRG rooted at $p0$ can

be expressed in terms of the probabilities of the sub-vectors of ev rooted at the corresponding subtrees of $p0$, as well as probabilities, $P(pi = T \mid p0 = T)$, of transition from root $p0$ to the root of each subtree, pi.

The only remaining terms that are not trivial to evaluate are the $P(pi = T, [Gi])$ terms. But each $(pi = T, [Gi])$ is just an ev on a tree that has fewer levels than the ev with which we began. Hence its probability can be evaluated by finite induction on the levels of the tree, using this same procedure at each level.

Finally, we dismiss the assumption that no pair of PRG nodes is disjoint. Note that the only nodes of interest are those whose predicates appear in an ev being evaluated. Since ev is evaluated recursively level by level, we can without loss of generality concern ourselves only with the set of pi's associated with the roots of the subtrees of $p0$, and in fact only with those pi's that appear in the ev being evaluated. Initially, let us suppose that there is only a single set of pairwise disjoint predicates, $p1$, $p2$, ..., pk, corresponding to the roots of some of the subtrees $S1$, $S2$, ..., Sk of $p0$. That is there is a D-edge between each pair of these root nodes, and no D-edges connected to any other subtree root node. Note that all of the predicates in this set of disjoint predicates must have the value False in the execution vector, i.e., $ev = (p1 = F, p2 = F, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots)$, since if one of them had value True, the others would be redundant and would therefore have been eliminated from ev by the **Eliminate-Redundant-Conditions** procedure.

The execution sub-vector of ev with predicates $p1$, $p2$, ..., pk omitted can be represented as a union of execution vectors with all the legal boolean assignments to $p1$, $p2$, ..., pk. That is,

$$
\begin{aligned}
&([pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad = (p1 = F, p2 = F, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad \cup (p1 = T, p2 = F, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad \cup (p1 = F, p2 = T, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad \cup \ldots \cup (p1 = F, p2 = F, \ldots, pk = T, [pk+1 = Bk+1,\ Gk+1], \ldots)
\end{aligned}
$$

The expression written on the first line of the right hand side is the *ev* of interest. Note that each valid boolean assignment to $p1, p2, ..., pk$ can contain at most one True value since the $p1$, $p2$, ..., pk are all mutually disjoint. Thus the execution vectors of the right hand side of the equation above represent all possible legal assignments to $p1, p2, ..., pk$.

Since the *ev* terms on the right hand side are pairwise non-overlapping, the probability of the right hand side can be found by evaluating the probability of each *ev* separately and then simply adding those probabilities together. Using this fact, and rearranging terms so that the *ev* of interest is on the left, we have

$$
\begin{aligned}
&P(p1 = F, p2 = F, \ldots, pk = F, [pk+1 = Bk+1,\ Gi], \ldots) \\
&\quad = P([pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad - P(p1 = T, p2 = F, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad - P(p1 = F, p2 = T, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots) \\
&\quad - \ldots - P(p1 = F, p2 = F, \ldots, pk = T, [pk+1 = Bk+1,\ Gk+1], \ldots)
\end{aligned}
$$

The first execution vector on the right hand side contains no disjoint predicates.

Each remaining execution vector contains one and only one disjoint predicate assigned value True and the remaining disjoint predicates are assigned value False. The **Eliminate-Redundant-Conditions** procedure then eliminates the False disjoint predicates from each of these execution vectors which changes the previous equation into the following form:

$$
\begin{aligned}
P(p1 = F, p2 = F, \ldots, pk = F, [pk+1 = Bk+1,\ Gk+1], \ldots)& \\
= P([pk+1 = Bk+1,\ Gk+1], \ldots)& \\
- P(p1 = T, [pk+1 = Bk+1,\ Gk+1], \ldots)& \\
- P(p2 = T, [pk+1 = Bk+1,\ Gk+1], \ldots)& \\
- \ldots - P(pk = T, [pk+1 = Bk+1,\ Gk+1], \ldots)&
\end{aligned}
$$

Note that none of the execution vectors on the right hand side contains more than one predicate of the set of mutually disjoint predicates associated with the roots of the subtrees. Hence we can use Equation 3.1 and Equation 3.2, together with the repetition of this procedure to eliminate mutually disjoint sets of nodes at lower levels. If at some level, several distinct sets of mutually disjoint predicates correspond to the subtrees at that level, the above procedure is applied to each set individually until at most one predicate from each set remains in any ev to be evaluated, and that predicate is assigned value True.

The above general evaluation procedure is quite complex. Two transformations of the execution vector that we described, (i) transforming ev into normal form and (ii) eliminating disjoint predicates from ev, can take time that is exponential in the size of an execution vector, in the worst case. However, in practice the benchmark

evaluations we carried out were quite straightforward and rarely required these transformations to be performed. In addition, the number of predicates considered for such evaluation hardly ever exceeded 2 or sometimes 3, with the conflict recovery scheme and the limited fetch width that we use in our probabilistic predicate-aware architecture (see Section 3.3). Remember that with this scheme the conflicting operations assigned to different function units can be dispatched into these function units in parallel, one per unit. Obviously, in this case, the execution vector need only contain the predicates of the operations that may conflict as the result of sharing the same unit; the predicates of the operations that do not share a unit with other operations need not to be included in the execution vector since they will never result in a conflict.

3.4.3 Probabilistic Predicate-aware List Scheduling Extensions

Our probabilistic predicate-aware list scheduling (PPALS) algorithm uses the expected delay computation technique to select profitable combining opportunities during operation scheduling with an overall goal of reducing the schedule length relative to the baseline algorithm. We have proposed and evaluated two PPALS scheduling algorithms. The first algorithm, called 'extend-all' PPALS, extends the cmpp latency from every cmpp operation to each of its consumers to create the maximum combining opportunity for every predicated operation. The second algorithm, called 'first-fit' PPALS, does not extend the cmpp latency, but instead allows the operation to be placed in the first available scheduling slot that satisfies its resource and data constraints. If it is scheduled less than *extendedlatency* cycles after its cmpp, the scheduled operation must reserve its resource unconditionally. As we said earlier,

```
PPALS-MainScheduler(oplist, ddg)
1  For every op in oplist, use ddg to initialize MinTime and compute priorities
2  ready_oplist = all operations in oplist with no incoming edges
3  CurrSchedLen = 0;
4  while ready_oplist is not empty do
5      hpr_ready_oplist =      highest priority operations in reay_oplist
6      if 'extend-all' PPALS then
7          curr_op = hpr_ready_oplist.pop()
8      else  if 'best-fit' PPALS then
9          curr_op = LeastCombiningPotential(hpr_ready_oplist )
10     end if
11     compute MinTime for curr_op
12     minCurrSLIncrease = 1  /* minimum current schedule length increase */
13     for t = MinTime;  t <= MaxTime; t++ do
14         for each resource alternative, resource_alt, on which curr_op can be scheduled do
15             if resource_alt is available at time t then
16                 if t <=  schedule time of the latest scheduled operation so far then
17                     CurrSLIncrease = compute delay due to conflict incurred from scheduling
18                                          curr_op in time t on resource_alt
19                 else          /* increases delay due to conflict component of current expected  SL */
20                     CurrSLIncrease = 1       /* increases static component of current expected SL */
21                 end if
22                 if CurrSLIncrease <= minCurrSLIncrease then
23                     minCurrSchedIncrease = CurrSLIncrease
24                     t_min = t
25                     minresource_alt = resource_alt
26                 end if
27             end if
28         end for
29     end for
30     Schedule curr_op in time t_min on minresource_alt
31     Update ready_oplist with the ready successors of curr_op;
32 end while
```

Figure 3.11: Generic PPALS scheduling algorithm

this increases the operation's execution frequency to 1.0 and, hence, combining this operation with other operations may result in higher conflict delay than when the operation is scheduled at least *extendedlatency* cycles after its cmpp and reserves it resource conditionally. These two algorithms are described below in Section 3.4.3.1 and Section 3.4.3.2, respectively.

Figure 3.11 shows the generic PPALS algorithm, **PPALS-MainScheduler**, which is common to both 'extend-all' PPALS and 'first-fit' PPALS. The main idea of

the algorithm is to schedule each operation at the time which minimizes the increase in the current expected partial schedule length (due to already scheduled operations). This increase may in part be due simply to an increase in the current static schedule length: if a new operation gets scheduled 1 or more cycles later than the latest scheduled operation so far, the static component of the current expected schedule length is increased. The increase may also be due in part to an additional conflict delay that results from combining the operation with other operations that are already scheduled. In this case the expected conflict delay component of the current expected schedule length gets increased.

The underlying principle of PPALS is to avoid combining an operation if the increase in the expected delay due to conflicts that it causes exceeds 1. If the operation cannot be successfully scheduled between cycle 0 and the *CurrentStaticScheduleLength* - 1 (which is the time of the latest scheduled operation so far) without violating this principle, the scheduler then places the operation one cycle after the latest scheduled operation (i.e., at cycle *CurrentStaticScheduleLength*), thus increasing the current static schedule length by 1 cycle. Since the overall goal is to reduce the expected schedule length, it is a better scheduling decision to increase the static component of the expected schedule length by only one cycle, than to increase the expected conflict delay component of the expected schedule length by more than one cycle; it results in less total increase at this step, and it provides more combining opportunity for the operations to be scheduled next.

The algorithm takes the list of all operations, *oplist*, and the corresponding data dependence graph, *ddg*, and constructs the valid schedule. As in the baseline scheduling algorithm, the **PPALS-MainScheduler** starts by computing each operation's earliest start time (*EarlyCycle*) and its scheduling priority using *ddg* (line 2). At

the beginning, the *ready_oplist*, which at every scheduling step contains the list of currently unscheduled operations whose predecessors have been already scheduled, is initialized with the operations from *oplist* that have no incoming edges (line 2). The main while loop (lines 4-32), iterates over the elements of *ready_oplist*, choosing and scheduling one operation from the list per iteration until the *ready_oplist* becomes empty, at which point the final schedule is produced and the routine **PPALS-MainScheduler** returns.

When an operation becomes scheduled, its outgoing edges are removed. This may free up some other operations, which are then put into the ready list. As in the baseline scheduler, the *hpr_ready_oplist* (line 5) is the list of highest priority ready operations in *ready_oplist*. From *hpr_ready_oplist* we choose one operation to be scheduled (lines 6-10). This choice depends on whether 'extend-all' or 'first-fit' PPALS is used and is discussed in the corresponding subsections.

Before scheduling the operation, $MinTime$ is computed. $MinTime$ is the earliest cycle at which the operation can be scheduled as constrained by the currently scheduled operations that the operation being scheduled is dependent upon. We also set the $minCurrSLIncrease$ variable to 1. This variable is used in the body of the for loop (lines 13-29), described below, to choose the scheduling time for the operation that corresponds to the minimum increase in the current expected schedule length; this initial 1 is used as an upper bound, as the algorithm avoids combining the operation if the conflict delay increase that results from the combination exceeds $minCurrSLIncrease$.

The most important part of the algorithm is the innermost *for* loop (lines 13-29) which tries to find a valid time slot for the operation to be scheduled. Recall that the baseline scheduler iterates over consecutive cycles, from $MinTime$ to $MaxTime$, and

multiple resource alternatives, until it finds the first cycle and a resource alternative with no resource constraints, at which point the operation is scheduled. $MaxTime = max(CurrentStaticScheduleLength, MinTime)$ is the first cycle at which the operation can be guaranteed to be schedulable. Unlike the baseline scheduler, which schedules the operation in the first cycle where there are no resource constraints, PPALS iterates over *every* cycle, from $MinTime$ to $MaxTime$ (lines 13-29).

For each time, t, PPALS tries to schedule the operation on every alternative resource, $resource_alt$ (lines 14-28). If the operation cannot be scheduled at this time (line 15), and if this time is less than or equal to the schedule time of the latest scheduled operation (*CurrentStaticScheduleLength - 1*) (line 16), the added delay due to conflicts that would be incurred by scheduling the operation on this resource alternative at this time is computed (lines 17-18) and added to $CurrSLIncrease$. Note that if this slot is empty, there is no resource sharing, and hence the added delay due to conflicts is obviously 0. On the other hand, if time t is greater than the schedule time of the latest scheduled operation (line 19), the $CurrSLIncrease$ is set to 1 (line 20) to reflect the fact that scheduling the operation at this time will increase the current static schedule length, *CurrentStaticScheduleLength*, and hence also the current expected schedule length by at least one cycle. Note that there is only one case in which t is two or more cycles larger than the schedule time of the latest scheduled operation. In this case, the operation being scheduled is dependent on a scheduled operation whose latency is not satisfied until two or more cycles past latest scheduled operation; therefore this operation will never share its resource and will be scheduled in the first (and only) iteration of its scheduling loop (line 13-29).

If $CurrSLIncrease$ is no greater than $minCurrSLIncrease$ (line 22), which is the minimum $CurrSLIncrease$ seen so far, $minCurrSLIncrease$ is updated with

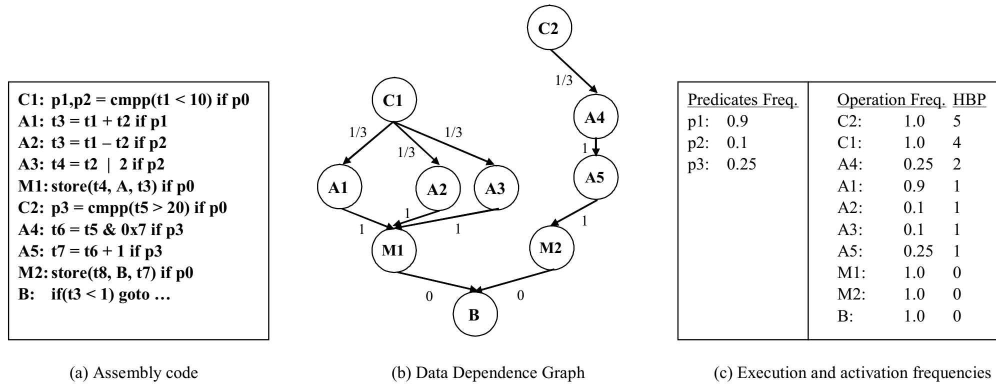
$CurrSLIncrease$, and t_{min} and $minresource_alt$ are updated with t and $resource_alt$, respectively. When the for loop (lines 13-29) terminates, t_{min} and $minresource_alt$ contain the time and resource that result in the minimum increase in the current expected schedule length for the current operation being scheduled. As a result, the operation, *curr_op*, gets scheduled on resource $minresource_alt$ at time t (line 30), the *ready_list* is updated with the freed successors of *curr_op*, and the scheduling process repeats until the *ready_list* becomes empty.

3.4.3.1 'Extend-all' PPALS

'Extend-all' PPALS calls **PPAS-DDGLatency-Adjust** to extend the cmpp latency for every cmpp operation whenever it might potentially improve the performance (see Section 3.4.1).

Specific to the 'extend-all' algorithm is how it chooses which operation from *hpr_ready_oplist* to schedule next. Choosing the best operation from *hpr_ready_oplist*, so as to guarantee the shortest final schedule, requires backtracking and may result in exponential complexity. By noticing that any predicated operation in *hpr_ready_oplist*, whose cmpp latency has been extended, may potentially result in improved performance when combined with another such operation from the list, 'extend-all' PPALS takes a simple heuristic approach and, as line 7 of Figure 3.11 shows, chooses the first ready operation from *hpr_ready_oplist* to schedule.

We now demonstrate the working of 'extend-all' PPALS using an example shown in Figure 3.12. Figure 3.12(a) shows the predicated basic block. Figure 3.12(b) shows the corresponding data dependence graph. Each edge is marked with the original short latency, *shortlatency*, of 1 cycle. The latency extension step also marks the incoming cmpp edge of each arithmetic operation $A1$ through $A5$ with the extended



(a) Assembly code (b) Data Dependence Graph (c) Execution and activation frequencies

Figure 3.12: Assembly code and data dependence graph to demonstrate PPALS

Time	A	M	B
0	C2		
1	C1		
2	A4?p3		
3	A1?p1		
4	A2?p2		
5	A3?p2		
6	A5?p3	M1	
7		M2	B

Figure 3.13: Schedule Reservation Table for baseline LS

latency, *extendedlatency*, of 3 cycles; all five of these operations are marked because each can be combined with at least one other arithmetic operation so that the resulting delay due to conflicts does not exceed 1. As above, we assume the machine model in Table 2.1 with cmpp latency of 3 and CDRL=1. Finally, Figure 3.12(c) shows activation frequencies of the predicates as well as execution frequencies of the operations. Note further that it shows the height-based priority (HBP) of each operation. Note that operations are listed in the decreasing order of their HBP. In addition, in our example the HBP priority order of the operations is the same before and after cmpp latency extension step. Therefore, we only show each operation's priority before the latency extension.

The application of the baseline list scheduler to the example results in a schedule

	Scheduling A4		Scheduling A1		Scheduling A2		Scheduling A3		Scheduling A5	
	hpr_ready_oplist		**hpr_ready_oplist**		**hpr_ready_oplist**		**hpr_ready_oplist**		**hpr_ready_oplist**	
	A4?p3		A1?p1 A2?p2 A3?p2 A5?p3		A2?p2 A3?p2 A5?p3		A3?p2 A5?p3		A5?p3	
Time	**Sched**	**CSLI**	**Sched**	**CSLI**	**Sched**	**CSLI**	**Sched**	**CSLI**	**Sched**	**CSLI**
0	C2		C2		C2		C2		C2	
1	C1		C1		C1		C1		C1	
2										
3	**A4?p3**	**1.0**	A4?p3		A4?p3		A4?p3		A4?p3	
4			**A1?p1**	**1.0**	A1?p1 **A2?p2**	**0.0**	A1?p1 A2?p2 **A3?p2**	**0.2**	A1?p1 A2?p2 A3?p2	
5					A2?p2	1.0	A3?p2	1.0	**A5?p3**	**1.0**
CSL	4		5		5		5.2		6.2	

(a) Scheduling record of A4, A1, A2, A3 and A5 (CSLI – current schedule length increase)

Time	A	M	B	CflDel
0	C2			0
1	C1			0
2				0
3	A4?p3			0
4	A1?p1 A2?p2 A3?p2			0.2
5	A5?p3	M1		0
6		M2	B	0
0.2	expected delay due to conflicts			

(b) Complete schedule

Figure 3.14: Schedule Reservation Table for 'extend-all' PPALS

length of 8 cycles, as presented in the SRT shown in Figure 3.13. Note that operations $A1$ through $A5$ are executed conditionally but reserve the ALU unconditionally. In fact, each of these operations only executes a fraction of the time, i.e. only when its predicate is True. In addition, among operations $A1$, $A2$ and $A3$, either $A1$ is executed or $A2$ and $A3$ are both executed, because $p1$ and $p2$ form a complete set of disjoint predicates. As a result, we effectively waste either cycle 3 or both cycles 4 and 5 whenever this code segment is executed.

With 'extend-all' PPALS, an expected schedule length of 7.2 cycles can be achieved as shown in Figure 3.14(b). Figure 3.14(a) shows how this schedule is derived. Each column corresponds to scheduling a single operation ($A1$ through $A5$). In the first column operations $C2$ and $C1$ have already been scheduled in cycles 0 and 1, respectively. The top of each column shows the list of highest priority ready operations

(hpr_oplist_ready) among which 'extend-all' PPALS picks the first ready operation to schedule. The schedule also shows the $CurrSLIncrease$ (abbreviated as $CSLI$ in the figure), which for a given Time t represents an increase in the current partial schedule length due to scheduling the current operation at this time (see description of **PPALS-MainScheduler** in Figure 3.11). The last row of the table shows the current partial schedule length (CSL) after each operation is scheduled.

Initially, $A4$ is the only operation in the hpr_oplist_ready and it gets placed at Time 3 since it is dependent on the three cycle cmpp operation $C2$, which is scheduled at Time 0. The corresponding $CurrSLIncrease$ is 1 in this case, since scheduling $A4$ at Time 3 increases the current schedule length by 2 cycles (see Line 20 of Figure 3.11 and corresponding explanation in the text). Scheduling $A4$ frees $A5$. As the second column shows, there are now 4 ready operations with the same highest priority (HBP=1). The scheduler picks the first operation $A1$ and places it at Time 4 - the first cycle which satisfies its data dependence with $C1$, and $CurrSLIncrease$ is set to 1.

Next, $A2$ is picked. It can be scheduled in one of two cycles: Time 4 to share the ALU resource with $A1$, resulting in $CurrSLIncrease = 0$, since $A1$ and $A2$ are disjoint and there is no conflict delay, or Time 5 resulting in $CurrSLIncrease = 1$, since the current schedule length is increased by 1 cycle. Both choices are shown in the Scheduling $A2$ column in Figure 3.14(a); the lower cost choice is highlighted. $A2$ gets scheduled at Time 4 since that has the lower $CurrSLIncrease$.

$A3$ is picked next and it can go either at Time 4 or time 5. Scheduling $A3$ at Time 4 to share the ALU resource with $A1$ and $A2$ will increase the schedule length by 0.2 cycles due to conflict delay (2 recovery cycles $\times$ $P(p2 = T) = 2 \times 0.1 = 0.2$). This delay results from the fact that $A1$ and $A2$ are disjoint, hence their joint execution

frequency is 1, and the $A3$ execution frequency is 0.1.

Finally, the only remaining operation with a height-based priority of 1 is $A5$. The earliest time it can be scheduled is time 4 since it is data dependent on $A4$ with a 1 cycle latency. However $A5$ cannot be scheduled at Time 4 since there are 3 operations already scheduled at this time and our machine has a fetch width of 3 operations. Therefore $A5$ gets scheduled at Time 5, which increases the current expected schedule length by 1 cycle.

Finally, operation $M1$ gets scheduled at Time 5, and operations $M2$ and B get scheduled at Time 6, resulting in the final schedule shown in Figure 3.14(b) which has an expected length of 7.2 cycles. The last column shows the delay due to conflicts for each schedule time slot.

3.4.3.2 'First-fit' PPALS

As shown in our experiments (see Section 3.5.2.1), extending the cmpp latency leads to critical path increases in acyclic code regions which degrades the performance of 'extend-all' PPALS. Consequently, we propose a second, alternative algorithm, called 'first-fit' PPALS which does not extend cmpp latency prior to scheduling, but only takes advantage of combining opportunities as they arise.

When 'first-fit' PPALS chooses an operation from *hpr_oplist_ready*, as in 'extend-all' PPALS, it also tries every alternative in the range from $MinTime$ to $MaxTime$ to find the cycle (and resource) that causes the smallest increase in the current schedule length, $CurrSLIncrease$. If an operation is scheduled less than $extendedlatency$ after its cmpp producer, this operation must reserve its resource unconditionally at this cycle, since its predicate cannot be read early (during the predicate read and dispatch stage, see Section 3.3). In this case, this operation's execution frequency

will be increased to 1.0, since it always requires its resource. If the operation is scheduled at least *extendedlatency* cycles after its cmpp, the operation's predicate can be read early, and therefore it can reserve its resource conditionally, just as in 'extend-all' PPALS.

Of course, if the operation reserves its resource unconditionally, it will incur a higher conflict delay penalty when combined with another predicated operation than if it reserves its resource conditionally. Since the scheduler chooses the cycle with minimum $CurrSLIncrease$, operations will only be scheduled unconditionally if this will lead to a minimum increase in the current schedule length. Note that an operation that unconditionally reserves its resource can legally (if not beneficially) be combined with other operations. However, even thought it is legal to combine for two operations that unconditionally reserve the same resource, they never will be combined since it is never beneficial to do so. Combining two such operations would result in a conflict delay of at least 1 cycle, and thus will be ruled out by our **PPALS-MainScheduler** algorithm because the algorithm will later consider scheduling the operation one cycle later than the end of the current schedule and will choose that schedule slot (or one with even lower cost) over this one.

Whereas 'extend-all' PPALS chooses the first ready operation from *hpr_oplist_ready* to schedule, 'first-fit' PPALS needs to make a more intelligent decision and calls the **LeastCombiningPotential** function to choose the operation with the *least combining potential*. An operation in *hpr_oplist_ready* is said to have the least combining potential if it would result in the highest conflict delay when combined with the other operations in *hpr_oplist_ready*.

The scheduler chooses an operation with the least combining potential first because scheduling this operation unconditionally gives the remaining ready operations

```
LeastCombiningPotential(hpr_ready_oplist)
1    for each operation, op, in hpr_ready_oplist do
2        if (op is cmpp operation) then
3            return op
4        end if
5    end for
6    mincombiningpotential = +inf
7    for each operation, op1, in hpr_ready_oplist do
8        combiningpotential = 0
9        for each operation, op2, which is not op1, in hpr_ready_oplist do
10           if(op1 is NOT disjoint from op2) then
11               combiningpotential -= CflDelay from combining op1 and op2
12           else
13               combiningpotential +=1
14           end if
15       end for
16       if combiningpotential < mincombiningpotential  then
17           mincombiningpotential = combiningpotential
18           op_mincombiningpotential = op1
19       end if
20   end for
21   return op_mincombiningpotential
```

Figure 3.15: Algorithm to choose operation with the least combining potential

in *hpr_oplist_ready*, those with higher combining potential, a chance to be scheduled conditionally at a later time where they may be far enough from their cmpp to achieve a small conflict delay when combined. Note that if an operation with a higher combining potential is scheduled earlier, it is more likely to be scheduled unconditionally, thus losing its opportunity to reserve its resource conditionally at a later time whenever it might combine with a smaller conflict delay. Symmetrically, if an operation with a lower combining potential is scheduled later, it is more likely to be scheduled conditionally, and thus could share its resource with other operations; however, because this operation's combining potential is low, it is more likely to be unable to take good advantage of that opportunity to combine due to the lack of an available low conflict partner to combine with.

The **LeastCombiningPotential** function is shown in Figure 3.15. It takes the

hpr_oplist_ready list and returns an operation with the least combining potential. The first *for* loop (lines 1-5) iterates over all the operations in *hpr_oplist_ready* and returns the first cmpp operation found, if any. This cmpp operation is given the highest scheduling priority for two reasons: (i) it must always reserve its resource unconditionally, (ii) scheduling it early gives more opportunity to its dependent operations to be scheduled *extendedlatency* cycles later so that they may reserve their resources conditionally.

If there are no cmpp operations in *hpr_oplist_ready*, i.e. the ready cmpp operations with the current highest priority have already been scheduled, the second loop executes (lines 7-15). The combining potential of each operation, *op*1, is computed by cumulatively adding to *combiningpotential* the combining potential of combining *op*1 with each other operation, *op*2, in the list (lines 9-15). If *op*1 and *op*2 are not disjoint (line 10), their combining potential is equal to the negative value of the delay due to conflict when both are combined (line 11). In this way, a higher conflict delay results in a smaller overall *combiningpotential*. If *op*1 and *op*2 are disjoint (line 12), we set their combining potential to 1 (line 13). In this way, operations with more disjoint partners will tend to be delayed more, which is desirable since disjoint operations result in zero delay due to conflicts when combined if they are delayed long enough to be *extendedlatency* cycles away from their cmpp operations. The last *if* condition (line 16) chooses the operation, *op_mincombiningpotential*, that has minimum combining potential, *mincombiningpotential*, so far (lines 17-18). When the *for* loop exits, *op_mincombiningpotential*, the operation with the smallest combining potential in *hpr_oplist_ready* is returned (line 21).

We now demonstrate the working of 'first-fit' PPALS using the same example shown in Figure 3.12. The scheduling record of all five predicated operations is

	Scheduling A4		Scheduling A5		Scheduling A3		Scheduling A1		Scheduling A2	
	hpr_ready	LCP	hpr_ready	LCP	hpr_ready	LCP	hpr_ready_oplist	LCP	hpr_ready_oplist	LCP
	A4?p3	—	A1?p1	1.55	A1?p1	2	A1?p1	1	A2?p2	—
			A2?p2	0.75	A2?p2	0.8	A2?p2	1		
			A3?p2	0.75	A3?p2	0.8				
			A5?p3	-0.55						
Time	**Sched**	**CSLI**	**Sched**	**CSLI**	**Sched**	**CSLI**	**Sched**	**CSLI**	**Sched**	**CSLI**
0	C2		C2		C2		C2		C2	
1	C1 A4?p3	2.0	C1		C1		C1		C1	
2	**A4?p3**	**1.0**	A4?p3		A4?p3 A3?p2	2.0	A4?p3		A4?p3	
3			**A5?p3**	**1.0**	A5?p3 **A3?p2**	**0.5**	A5?p3 A3?p2 A1?p1	3.5	A5?p3 A3?p2 A2?p2	3.5
4					A3?p2	1.0	**A1?p1**	**1.0**	A1?p1 **A2?p2**	**0.0**
5									A2 ? p2	1.0
CSL	3		4		4.5		5.5		5.5	

(a) Scheduling record of A4, A5, A3, A1 and A2 (CSLI – current schedule length increase)

Time	A	M	B	CflDel
0	C2			0
1	C1			0
2	A4?p3			0
3	A5?p3 A3?p2			0.5
4	A1?p1 A2?p2	M2		0
5		M1	B	0
0.5	expected delay due to conflicts			

(b) Complete schedule

Figure 3.16: Schedule Reservation Table for 'first-fit' PPALS

shown in Figure 3.16(a). Here again we assume the initial partial schedule in which operations $C2$ and $C1$ are already scheduled at Time 0 and 1, respectively. The list of ready highest priority operations (*hpr_oplist_ready*) along with the combining potential of each operation is shown at the top of each column. Among these, 'first-fit' PPALS picks the operation with the least combining potential and schedules it.

Initially, $A4$ is the only operation in the *hpr_oplist_ready*. $A4$'s earliest scheduling time is Time 1, since it is dependent on $C2$ which is scheduled at Time 0. Scheduling $A4$ in this cycle to share the ALU with $C1$ would increase the current schedule length by 2 cycles, because it results in 2 cycles of delay due to conflicts since both $C1$ and $A4$ must reserve the ALU unconditionally and will therefore always execute. On the other hand, scheduling $A4$ at Time 2 will increase the current schedule length by only

1 cycle. Hence, $A4$ gets scheduled at Time 2.

Scheduling $A4$ frees $A5$. As the second column shows, there are now 4 operations with the same highest priority of 1. Also shown is the combining potential for each operation. For example, the combining potential of $A1$ is equal to $1 + 1 - 2 \times P(p1 = T, p3 = T) = 1 + 1 - 2 \times 0.9 \times 0.25 = 1.55$. The first and second terms , which represent combining $A1$ with $A2$ and $A3$ respectively, are both 1 due to the fact that $A1$ is disjoint from both $A2$ and $A3$. The third term is the negative of the expected conflict delay due to nondisjointly combining $A1$ and $A5$. Among the operations in *hpr_oplist_ready* $A5$ has the least combining potential (-0.55), which is calculated as $-2 \times P(p1 = T, p3 = T) - 2 \times P(p2 = T, p3 = T) - 2 \times P(p2 = T, p3 = T) = -2 * 0.25 * 0.9 - 2 * 0.25 * 0.1 - 2 * 0.25 * 0.1 = -0.55$, where the first, second and third terms are the conflict delays due to combining $A5$ with $A1$, $A2$ and $A3$, respectively. That $A5$ has the least combining potential makes sense intuitively: $A5$ is not disjoint from any other operation, whereas the remaining operations in *hpr_oplist_ready*, $A1$, $A2$ and $A3$, are each disjoint from at least one other operation in the list. Thus $A5$ is chosen to be scheduled next and is placed at Time 3, which is the earliest cycle in which it can be placed.

In the next scheduling step, both $A2$ and $A3$ have the least combining potential of 0.8 as shown in the third column of Figure 3.16(a). Each of them can combine with $A1$ without conflict delay since they are disjoint from $A1$ and they can be combined with each other with a conflict delay penalty $2 \times P(p1 = T) = 0.2$. $A1$, on the other hand, is disjoint from both $A2$ and $A3$, hence its combining potential is 2. $A3$ is chosen and Times 2, 3 and 4 are tried. Scheduling $A3$ at Time 3 together with $A5$ gives the smallest conflict delay, namely, $2 \times P(p2 = T) = 0.5$. Note that $A3$ reserves the ALU unconditionally, since it is only 2 cycles away from $C1$, whereas $A5$ reserves

the ALU conditionally since it is 3 cycles away from $C2$.

In the next scheduling step, $A1$ and $A2$ have the same combining potential since they are disjoint from one another. $A1$ is chosen and scheduled at Time 4 which has lower combining potential than Time 3.

Then $A2$, is scheduled at Time 4, resulting in 0 conflict delay since it is disjoint from $A1$, which is also scheduled in this cycle, and both are 3 cycles away from $C1$ so that they can reserve the ALU conditionally.

The final schedule with an expected length of 6.5 cycles is shown in Figure 3.16(b). The last column shows the delay due to conflicts for each schedule timeslot. Note that this 'first-fit' schedule is 0.7 cycles shorter than the corresponding 'extend-all' schedule in Figure 3.14(b). This reduction in length is due to the fact that instead of having to schedule $A4$ at Time 3, as 'extend-all' does in order to satisfy the 3 cycle extended latency of $C2$, 'first-fit' places $A4$ at Time 2. Although $A4$ then reserves the ALU unconditionally in this cycle, the slot is not wasted as it was with the 'extend-all' scheduler.

3.4.4 Probabilistic Predicate-aware Modulo Scheduling Extensions

Probabilistic predicate-aware modulo scheduling (PPAMS), like 'extend-all' PPALS, calls **PPAS-DDGLatency-Adjust** which extends the cmpp latency for each operation when it may improve the performance. Note there is no 'first-fit' counterpart for PPAMS. Since, as we will show in Section 3.5, modulo scheduled loops are resource rather than latency bound, and extending cmpp latency has only a small impact on the overall schedule length for most loops.

Our probabilistic predicate-aware modulo scheduling algorithm also uses the ex-

pected delay computation technique to try to find the smallest expected initiation interval ($II_{expected}$) for which valid schedule can be found. $II_{expected} = II_{static} + II_{CflDelay}$, where II_{static} is the static initiation interval when the delay due to conflicts is ignored, and $II_{CflDelay}$ is the expected delay due to conflicts. The following describes our main changes to baseline iterative modulo scheduling (IMS) in order to support PPAMS.

3.4.4.1 Computing ResMII

As described in Section 2.4.5, the baseline modulo scheduler (IMS), computes the resource-constrained lower bound ($ResMII_{bams}$) by unconditionally adding the number of times that each operation reserves a particular type of resource to that resource's usage count, regardless of the operation's guarding predicate. The cumulative usage count for the most heavily used resource determines $ResMII_{bams}$ for IMS. For our example in Figure 3.1, $ResMII_{bams} = 10$, since there are 10 ALU operations and only a single ALU unit.

For PPAMS, a similar calculation is done, except that (as in DPAMS) an operation is allowed to reserve a particular may-use resource conditionally and use it during only a fraction of those reservations, based on its execution frequency. Each time an operation reserves a resource, this resource's usage count is only incremented by operation's execution frequency. Thus, for example in Figure 3.1, the probabilistic predicate-aware resource-constrained lower bound $ResMII_{ppams} = \frac{1.0(A1)+1.0(C1)+0.2(A2)+1.0(C2)+0.09(A3)+1.0(C3)+0.01(A4)+0.8(A5)+1.0(C4)+0.11(A6)}{1\text{ALU}} = 6.21$, which is the sum of the execution frequencies over all ALU operations since the single ALU is the most heavily used resource. The corresponding operation is shown in parentheses for each term in the numerator. Note that un-predicated operations have

a frequency of 1.0. Finally, since the must-use resources are reserved regardless of guarding predicate values, their usage count is computed as in the baseline case. The lower bound obtained is an optimistic estimate of the actual schedule length since such ideal combining may not be possible and the bound does not estimate the additional conflict delay that may occur. Note that for the same example, DPAMS resource-constrained lower bound $ResMII_{dpams} = 9$; the DPAMS bound is higher since only disjoint combinations are allowed.

3.4.4.2 Main Scheduler

The main PPAMS driver (shown in Figure 3.17(a) uses a binary search method to find the smallest value of $II_{expected}$ for which a valid schedule can be found. The lower bound, II_{low}, is computed as the maximum of $ResMII_{ppams}$ and $RecMII_{ppams}$. $RecMII_{ppams}$ is the recurrence-constrained lower bound computed from the ddg after latency extension step. The upper bound, II_{high}, is computed as the maximum of the $ResMII_{bams}$ and $RecMII_{bams}$. Note that $RecMII_{ppams}$ is never less than $RecMII_{bams}$, because latency extension may increase the length of the critical path, but never decreases it. The *while* loop in Figure 3.17(a) calls the **ppams_FindSchedule** routine which tries to find a valid schedule for II_{middle} - a halfway point between II_{low} and II_{high}. If a schedule is found, II_{high} is reduced to II_{middle}, otherwise II_{low} is increased to II_{middle}. These steps are repeated until the difference between II_{high} and II_{low} drops below some small threshold value ($smalldelta$), at which time the $II_{expected}$ value is set equal to the current value of II_{high}.

ppams_FindSchedule (Figure 3.17(b)) tries to find a valid schedule with II no greater than $II_{expected}$ (the sum of II_{static} and $II_{CflDelay}$). In fact, it is possible for several valid schedules of length $II_{expected}$ to exist for several different values of II_{static}

```
ppams-Main()
1   II_low = max(ResMII_ppams, RecMII_ppams)
2   II_high = max(ResMII_bams, RecMII_bams)
3   while II_high – II_low > smalldelta do
4       II_middle = (II_high + II_low) / 2
5       if ppams-FindSchedule(II_middle) == true do
6           II_high = II_middle
7       else
8           II_low = II_middle
9       end if
10  end while
11  return II_high
```

(a)

```
PPAMS-FindSchedule(II_expected)
1   for II_static = ceiling(max(WidthResMII, RecMII_ppams));
2           II_static < floor(II_expected) + 1; II_static++)    do
3       II_CflDelay = II_expected - II_static
4       if PPAMS-IterativeScheduler(II_static, II_CflDelay)==True then
5           return True
6       end if
7   end for
8   return False
```

(b)

Figure 3.17: Main PPAMS scheduling routines

and $II_{CflDelay}$ as long as both add up to $II_{expected}$. Therefore we vary II_{static} in the loop starting from the ceiling of the maximum of the width resource- and the recurrence-constrained lower bound up to the **floor** of $II_{expected}$. The width resource-constrained lower bound, *WidthResMII*, is defined as $\frac{\text{number of operations in the loop body}}{\text{machine fetch width } (FW)}$ and is a hard limit on II_{static} for a machine of the given fetch width (FW). Thus II_{static} can never be less than $WidthResMII$, and since II_{static} must be an integer, it cannot be less than **ceiling**(WidthResMII). II_{static} is also constrained by the recurrence-constrained lower bound, $RecMII_{ppams}$. Note that contrary to baseline modulo scheduling, with PPAMS, II_{static} can be less than $ResMII$. In addition, since we are only interested in schedules with $II \leq II_{expected}$, the II_{static} term of any schedule of interest cannot exceed $\mathbf{floor}(II_{expected})$.

For each value of II_{static} the corresponding value of $II_{CflDelay}$ is computed (as the difference between $II_{expected}$ and II_{static}) and the **ppams_IterativeScheduler** is called with both values passed as parameters. The **ppams_IterativeScheduler** returns True if a valid schedule is found, in which case **ppams_FindSchedule** also returns the value True. Otherwise, the next combination of II_{static} and $II_{CflDelay}$ is tried, until either the **ppams_IterativeScheduler** returns True or the **for** loop terminates. If the loop terminates, no valid schedule has been found for a given value of $II_{expected}$, and **ppams_FindSchedule** returns False to the **ppams_Main driver**.

ppams_IterativeScheduler(II_{static}, $II_{CflDelay}$) is a slightly modified version of the baseline iterative modulo scheduler algorithm [41] which iteratively schedules and unschedules the operations of the loop until either a valid schedule is found or the maximum number of allowed scheduling steps (a runtime budget) is exceeded. In PPAMS, the scheduler tries to place an operation, *op*, in a row, *r*, of the II_{static} rows of the MRT that is chosen so that the increase in total estimated conflict delay

Schedule Reservation Table

Time	A	M	CflDelay
0	A1		
1		M1	
2			
3	C1		
4	C2		
5	C3		
6			
7	A2?p1 A5?p2		
8	A3?p3 **A4?p4**		0.0018
9			0.02
10			0.02
11			0.02
12			0.02
13			0.02
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			

Modulo Reservation Table

Time	A	M	CflDelay
0	n:A1		0
1	n-1:A2 n-1:A5	n:M1	0
2	n-1:A3 n-1:A4		0.0018
3	n:C1		0
4	n:C2		0
5	n:C3		0
0.0018	**expected delay due to conflicts**		

(a) Scheduling A4

Schedule Reservation Table

Time	A	M	CflDelay
0	A1		
1		M1	
2			
3	C1		
4	C2		
5	C3		
6			
7	A2?p1 A5?p2		
8	A3?p3 A4?p4		
9			2.0
10			2.0
11			2.0
12			2.0
13			2.0
14	**C4**		0.1991
15			
16			
17			
18			
19			
20			
21			
22			
23			

Modulo Reservation Table

Time	A	M	CflDelay
0	n:A1		0
1	n-1:A2 n-1:A5	n:M1	0
2	n-1:A3 n-1:A4 **n-2:C4**		0.1991
3	n:C1		0
4	n:C2		0
5	n:C3		0
0.1991	**expected delay due to conflicts**		

(b) Scheduling C4

Schedule Reservation Table

Time	A	M	CflDelay
0	A1		
1		M1	
2			
3	C1		
4	C2		
5	C3		
6			
7	A2?p1 A5?p2		
8	A3?p3 A4?p4		
9			
10			
11			
12			
13			
14	C4		
15			
16			
17	**A6**		0.22
18		M2	0.22
19			0.22
20			invalid
21			0.22
22			0.22
23			0.22

Modulo Reservation Table

Time	A	M	CflDelay
0	n:A1	n-3:M2	0
1	n-1:A2 n-1:A5	n:M1	0
2	n-1:A3 n-1:A4 **n-2:C4**		0.1991
3	n:C1		0
4	n:C2		0
5	n:C3 n-2:A6		0.22
0.4191	**expected delay due to conflicts**		

(c) Scheduling A6

Figure 3.18: PPAMS Scheduling of the example in Figure 3.1 for II_{static}=6 and $II_{CflDelay}$=0.42

(i.e. the summation of the estimated conflict delays over all rows of the MRT) is minimized. If an operation cannot be placed without exceeding the delay constraint $II_{CflDelay}$, a backtrack step is executed in which some operations are chosen to be unscheduled (to be tried again later) so as to allow the current *op* to be scheduled without violating the delay constraint.

3.4.4.3 Example Application of PPAMS

To illustrate probabilistic predicate-aware scheduling, PPAMS is applied to the example in Figure 3.1 with the machine model in Table 2.1, a cmpp latency of 3 cycles, and CDRL of 1 cycle. The operations are scheduled in decreasing order of their

height-base priority (HBP) shown in Figure 3.1(d). **PPAS-DDGLatency-Adjust**, called prior to scheduling, sets the latencies of all outgoing cmpp edges, other than the $(C1, C2)$ and $(C1, C3)$ edges, to 3 cycles. Recall that cmpp operations always execute regardless of their distance to their cmpp producer. Therefore extending their cmpp latency will not provide any additional performance benefit, and the latency of an edge from 8ne cmpp operation to another is always set to the short latency, 1.

We show the details of a single call to the **ppams_IterativeScheduler** function. Suppose the goal is to find a valid schedule with II_{static} ==6 that also satisfies the delay constraint $II_{CflDelay} \leq 0.42$ cycles. As each operation is scheduled at some time slot, the appropriate resource is marked at that time slot in both the SRT and the MRT. In addition, the current conflict delay values in the $CflDelay$ column are updated: the overall expected delay due to conflicts, $II_{CflDelay}$, which is the summation of $CflDelay$ entries over all rows is shown in the last row of MRT. The sum of II_{static} and the overall $II_{CflDelay}$ is equal to the overall expected schedule length. We use the probability computation result from Section 3.4.2.2 and the PRG in Figure 3.8 to compute the value of $CflDelay$ at each step. We omit the B-resource column to save space: in traditional modulo scheduling a single branch operation $B1$ is always scheduled first within the first II rows of the SRT and never causes conflicts.

Figure 3.18(a) shows the partial SRT and MRT with operations $A1$, $M1$, $C1$, $C2$, $C3$, $A2$ and $A3$ and $A5$ already scheduled, and operation $A4$ is being scheduled. Note that the partial schedule (before $A4$ is scheduled) has no conflicts since each ALU operation occupies a separate entry in the MRT, except for $A2$ and $A5$ which are disjoint. The earliest schedule time for $A4$ (in the SRT) is Time 8 since $A4$ is dependent on the three-cycle $C3$ operation scheduled at Time 5. If $A4$ is scheduled at Time 8, it will share the ALU with $A3$ from the same iter-

ation and will result in 0.0018 cycles of expected delay due to conflict, namely 2 recovery cycles $\times P(p3 = T, p4 = T) = 2 \times 0.01 \times 0.09 = 0.0018$, as shown in the rightmost column. Scheduling $A4$ at any of Times 9 through 12 of the SRT would result in 0.02 cycles of delay (2 recovery cycles $\times P(p4 = T) = 2 \times 0.01 = 0.02$), since the operations scheduled at each of these times in the MRT always execute (execution frequency = 1.0). Scheduling $A4$ at Time 13 will also result in 0.02 cycles of delay since $A2$ and $A5$ are a complete set of disjoint operations (they are guarded under complementary predicates). The scheduler places $A4$ at Time 8 of the SRT (Time 2 of the MRT) which causes the smallest increase in the overall conflict delay for the given partial schedule.

Figure 3.18(b) shows the scheduling of $C4$. Scheduling it at any of Times 9 through 13 of the SRT will cause a 2 cycle delay since C4 itself always executes and so do the operations that $C4$ conflicts with at each of these times. Scheduling $C4$ at time 14 will make it share the ALU with $A3$ and $A4$ from the next iteration resulting in an estimated 0.1991 cycles of delay due to conflict, replacing the previous 0.0018 cycles delay at this row of MRT. This causes the smallest conflict delay increase for the given partial schedule, and hence $C4$ is scheduled at Time 14 of the SRT (Time 2 of the MRT). The right column of the second row of MRT is also updated to reflect the new overall conflict delay for this partial schedule.

Figure 3.18(c) shows the scheduling of $A6$. Its earliest scheduling time is time 17 in the SRT, since it is dependent on the 3 cycle $C4$. This produces a conflict with $C3$ at Time 5 of the MRT which causes a conflict delay of 0.22 cycles since $A6$ has execution frequency 0.11 and $C3$ has execution frequency 1.0. Scheduling $A6$ at any of Times 18 through 23, except Time 20, would cause the same conflict delay of 0.22 cycles. $A6$ cannot be scheduled at Time 20, because that would exceed the

fetch width resource constraint: there are already three operations $A3$, $A4$ and $C4$, scheduled at Time 2 of the MRT. Hence $A6$ is scheduled at Time 17 of the SRT (Time 5 of the MRT), the earliest SRT time with the minimum added conflict delay. At this point the overall delay is increased to 0.4191 cycles (0.1991 cycles from Time 2 of the MRT plus 0.22 cycles from Time 5 of MRT).

Finally the scheduler places the last operation $M2$ at Time 18 of the SRT (Time 0 of the MRT), and places B at Time 5 (not shown) in both the SRT and the MRT, which results in a valid schedule that satisfies both the resource and delay constraints.

3.5 Performance Evaluation

To study the effectiveness of PPAS, we use the same evaluation infrastructure as in Section 2.5. Specifically, we perform our evaluations using the **Trimaran** compiler on the set of seventeen MediaBench applications. We implemented probabilistic predicate-aware reservation table and the expected delay model within the resource management module of ELCOR (Trimaran's back-end compiler); our probabilistic predicate-aware resource manager uses the Predicate Query System module to analyze predicated code and construct predicate relationship graphs to compute the expected delay due to conflicts as described in Section 3.4.2. In addition, we implemented two probabilistic predicate-aware list scheduling algorithms ('extend-all' PPALS and 'first-fit' PPALS) and a probabilistic predicate-aware modulo scheduling algorithm (PPAMS) within ELCOR's list and modulo scheduling modules, respectively.

3.5.1 Benchmarks and Processor Models

We use the extended notation (F,I,FP,M,B,C, CDRL) to represent the processor in this study. As in Section 2.5, **F** is the fetch width, **I** the number integer units, **FP** the number of floating-point units, **M** the number of memory units, **B** the number of branch units, and **C** the latency of the predicate defining operation (cmpp). Here we add the seventh parameter, **CDRL**, which is the conflict detection and recovery latency defined in Section 3.3. We use two baseline processors in our study: (4,2,1,1,1,1,*undefined*) and (6,4,2,1,1,1, *undefined*) called $P_{base}(4)$ and $P_{base}(6)$, respectively. As in Section 2.5 we assume 64 scalar and 64 rotating registers in our experiments and operation latencies that match the Itanium processor.

Each baseline processor $P_{base}(i)$ with a cmpp latency of 1 cycle is compared with the six corresponding probabilistic predicate-aware processors $P_{ppas}(i, 1/2/3, 0/1)$, (i.e. $P_{ppas}(i, 1, 0)$, $P_{ppas}(i, 2, 0)$, $P_{ppas}(i, 3, 0)$, $P_{ppas}(i, 1, 1)$, $P_{ppas}(i, 2, 1)$ and $P_{ppas}(i, 3, 1)$) with the same number of resources as the baseline processor, but cmpp latency of 1, 2, and 3 cycles, as well as conflict detection latencies of 0 and 1 cycles.

For comparison, we also report the performance of three corresponding deterministic predicate-aware processors $P_{dpas}(i, 1)$, $P_{dpas}(i, 2)$ and $P_{dpas}(i, 3)$ with a cmpp latency of 1, 2 and 3 cycles, respectively. As described above in Chapter 2, in contrast to PPAS, DPAS avoids conflicts altogether by conservatively combining only provably disjoint operations.

We evaluated all of our applications, applying deterministic and probabilistic predicate-aware scheduling optimizations to every region in the entire code. The various measurements (such as schedule length, resource- and latency-constrained lower bounds, etc.) are reported as the same weighted average as in Chapter 2, where again each individual region is weighted by its execution frequency within its bench-

mark. Clearly, PPAS can only benefit ppa-improvable if-converted regions of code, which are the regions that contain at least one *if-then* clause; PPAS will be ineffective at improving other code regions. All the remaining regions are scheduled using the baseline list or modulo scheduling algorithms, abbreviated as BALS and BAMS, respectively. In addition, when a ppa-improvable region turns out to have a schedule that is longer than the corresponding baseline schedule, the baseline schedule is used instead so as to avoid degradation in performance. Therefore, in our experiments the probabilistic predicate-aware schemes will always produce schedules that are at least as good as the baseline scheme, where the metric of goodness for predicate-aware schedules is their expected schedule length; in particular, where a smaller $II_{expected}$ is considered better in cyclic regions (regardless of epilogue size).

The reported PPAS processor speedup over baseline processor for various regions is measured as the dynamic cycles that the baseline processor spends in these regions divided by the dynamic cycles that the PPAS processor spends in these regions.

The following sections show the results for 'extend-all' and 'first-fit' PPALS, and PPAMS, as well as the overall speedup achieved by probabilistic predicate-aware scheduling for each benchmark, and over the entire suite.

3.5.2 Evaluation Results

3.5.2.1 Scheduling Headroom for PPAS

The goal of the probabilistic predicate-aware scheduler is to reduce the length of the resource constrained baseline schedule on the baseline machine of fixed width. PPAS takes advantage of the gap between the upper-bound defined by the length of the baseline schedule and the lower bound which is the maximum of the resource-

Benchmark	SL_{bals}	CPL1	CPL2	CPL3	$ResMSL_{bals}$	$ResMSL_{ppals}$	II_{bams}	RecMII1	RecMII2	RecMII3	$ResMII_{bams}$	$ResMII_{ppams}$
cjpeg	46.11	37.20	41.51	46.00	40.80	30.55	6.49	1.31	1.62	1.93	6.49	4.69
djpeg	14.03	10.41	12.13	13.33	11.72	10.49	58.64	1.00	1.00	1.00	58.64	42.69
epic	13.55	11.57	12.54	13.58	5.67	5.51	21.41	2.16	2.54	2.93	21.02	16.38
unepic	20.11	18.43	20.52	22.70	7.75	6.06	13.27	1.00	1.00	1.00	13.27	10.52
g721encode	29.96	20.46	22.68	24.96	25.43	18.00	30.00	1.00	1.00	1.00	30.00	20.24
g721decode	29.53	19.89	22.30	24.37	25.13	17.87	30.00	1.00	1.00	1.00	30.00	20.26
ghostscript	15.14	10.81	12.30	13.86	11.10	8.67	44.10	7.88	7.88	7.88	43.13	31.35
gsmdecode	23.52	19.98	20.76	21.20	21.68	15.24	27.92	7.93	9.68	11.44	27.73	21.46
gsmencode	30.73	23.54	25.67	28.38	28.41	18.66	74.85	7.87	8.40	8.93	74.39	45.61
mesamipmap	50.49	36.56	39.42	43.20	36.91	33.03	22.00	1.00	1.00	1.00	22.00	15.92
mpeg2dec	16.44	11.95	13.48	14.95	14.42	11.54	28.35	1.00	1.00	1.00	28.35	18.70
mpeg2enc	32.38	27.52	28.14	27.31	20.26	15.20	20.27	2.97	3.95	4.93	20.27	12.36
pegwitdec	18.72	17.12	18.83	20.53	11.92	11.12	20.67	1.97	1.97	1.97	20.67	12.46
pegwitenc	16.57	13.87	15.61	16.80	12.67	10.88	19.33	1.61	1.61	1.61	19.33	12.67
rasta	18.71	14.04	15.64	17.29	12.66	11.25	6.81	3.02	3.58	4.14	6.80	4.85
rawcaudio	17.46	13.54	12.55	13.06	12.03	9.52	26.00	20.00	25.00	30.00	24.00	17.13
rawdaudio	12.91	10.99	11.99	12.99	11.97	11.95	20.00	6.00	8.00	10.00	20.00	14.21
Average	23.90	18.70	20.36	22.03	18.27	14.44	27.65	4.04	4.72	5.40	27.42	18.91

(a) Scheduling Headroom for P(4)

Benchmark	SL_{bals}	CPL1	CPL2	CPL3	$ResMSL_{bals}$	$ResMSL_{ppals}$	II_{bams}	RecMII1	RecMII2	RecMII3	$ResMII_{bams}$	$ResMII_{ppams}$
cjpeg	38.06	37.20	41.51	46.00	20.60	20.48	3.40	1.31	1.62	1.93	3.24	2.64
djpeg	11.47	10.41	12.13	13.33	6.25	6.10	29.32	1.00	1.00	1.00	29.32	25.18
epic	12.61	11.57	12.54	13.58	3.52	3.48	11.09	2.16	2.54	2.93	10.70	8.71
unepic	19.76	18.43	20.52	22.70	4.04	3.90	7.34	1.00	1.00	1.00	7.34	6.57
g721encode	23.60	20.46	22.68	24.96	13.00	11.07	15.00	1.00	1.00	1.00	15.00	11.14
g721decode	23.16	19.89	22.30	24.37	12.92	10.81	15.00	1.00	1.00	1.00	15.00	11.15
ghostscript	12.42	10.81	12.30	13.86	5.83	5.64	22.55	7.88	7.88	7.88	21.58	18.95
gsmdecode	22.04	19.98	20.76	21.20	11.29	9.24	14.75	7.93	9.68	11.44	13.87	10.76
gsmencode	25.10	23.54	25.67	28.38	14.40	11.60	38.66	7.87	8.40	8.93	37.65	29.32
mesamipmap	40.44	36.56	39.42	43.20	18.64	16.80	12.33	1.00	1.00	1.00	12.33	10.39
mpeg2dec	13.37	11.95	13.48	14.95	7.35	7.06	14.19	1.00	1.00	1.00	14.19	12.10
mpeg2enc	34.45	27.52	28.14	27.31	10.79	9.91	11.12	2.97	3.95	4.93	10.14	7.33
pegwitdec	18.07	17.12	18.83	20.53	6.13	6.11	10.84	1.97	1.97	1.97	10.84	7.73
pegwitenc	14.92	13.87	15.61	16.80	6.63	6.08	10.01	1.61	1.61	1.61	10.01	8.10
rasta	15.71	14.04	15.64	17.29	7.13	6.94	3.57	3.02	3.58	4.14	3.56	3.06
rawcaudio	12.98	13.54	12.55	13.06	6.52	6.52	20.00	20.00	25.00	30.00	12.00	8.83
rawdaudio	11.91	10.99	11.99	12.99	5.99	5.99	10.00	6.00	8.00	10.00	10.00	7.33
Average	20.59	18.70	20.36	22.03	9.47	8.69	14.66	4.04	4.72	5.40	13.93	11.14

(b) Scheduling Headroom for P(6)

Table 3.1: Scheduling headroom estimates for the probabilistic predicate-aware schedulers

constrained and latency-constrained schedule lengths of the probabilistic predicate-aware processor. The resource-constrained schedule length is computed ignoring all data dependencies. The latency-constrained schedule length is the length of critical path. The gap between these lower and upper bounds constitutes the headroom for PPAS.

Table 3.1(a) shows an estimate of the probabilistic predicate-aware scheduler headroom averaged over all the acyclic (columns 2-7) and cyclic (columns 8-13) ppa-improvable regions of each benchmark and over all benchmarks for a 4-wide probabilistic predicate-aware machine. Table 3.1(b) shows the corresponding data for

a 6-wide machine. The last row of each table shows the average over all benchmarks. The data is presented for each benchmark shown in column 1. Column 2 shows the length of the baseline acyclic schedule. Columns 3-5 show the critical path length for cmpp latencies 1, 2 and 3, respectively. Columns 6 and 7 show the resource-constrained schedule length for the baseline processor, and the probabilistic predicate-aware processor, respectively. Since the resource-constrained schedule ignores all data dependencies, cmpp latency has no effect here. Columns 8-13 show the corresponding data for the cyclic regions; the resource-constrained schedule length is defined by $ResMII$, and latency-constrained schedule length is defined by $RecMII$ for cyclic regions. For the ppa-improvable acyclic regions we see from Table 3.1 that on average the critical path length for cmpp latencies of 1, 2 and 3 cycles, respectively, is 21.26%(9.18%), 14.81% (1.12%) and 7.82% (-6.99%) shorter than the schedule length on the baseline 4(6)-wide machine (with a cmpp latency of 1 cycle).

Thus, as the latency of the cmpp operation increases, the latency-constrained lower bound closely approaches the length of the baseline schedule and eventually exceeds it. As we go to a 6-wide machine, the situation becomes even worse since a wider machine also decreases the length of the baseline scheduler reducing the gap between the baseline and the bounds even more. For example, for a 6-wide machine with cmpp latency 3, the latency-constrained lower bound is larger than the length of the baseline schedule in all but the one case of mpeg2enc. It is apparent therefore that, cmpp operations in acyclic regions often lie on the critical path, leaving little headroom for PPALS. On the other hand, cmpp latency is not limiting in cyclic regions as seen by the fact that $ResMII_{ppams}$ is generally much larger than *RecMII1*, *RecMII2*, and even *RecMII3* for both 4 and 6-wide machines. The only exceptions are the rawcaudio and rawdaudio benchmarks for which $RecMII$ comes close to and

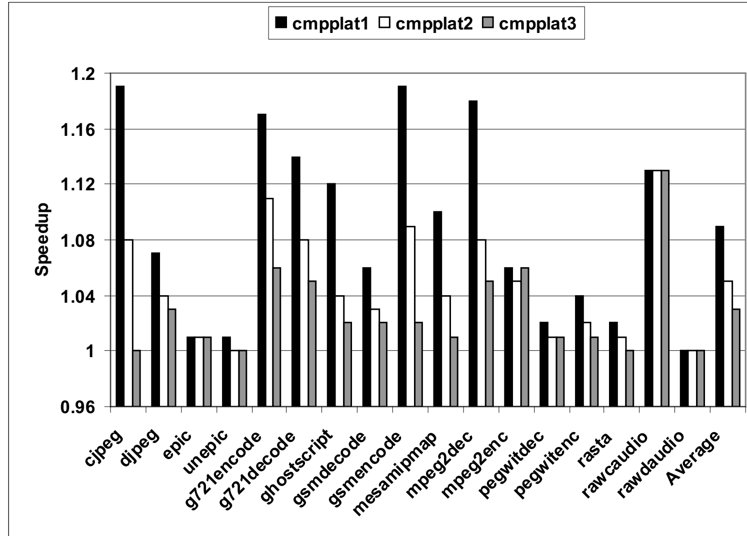
sometimes exceeds the $ResMII_{ppams}$. These two benchmarks each have a dominating loop with a recurrence cycle passing through all the cmpp operations.

We see that for mpeg2enc the baseline schedule length increases from 32.38 on the 4-wide machine to 34.45 for 6-wide. This increase is caused by a long latency operation anomaly, which is described in Section 2.5.2.2. In the case of mpeg2enc, there exists a number of cases in one of the time-critical regions where a long latency divide operation is followed by a subroutine call operation. For 4-wide machine, the pre-pass scheduler places the divide operation after the subroutine call operation, so that no edges are drawn between them during post-pass scheduling. For 6-wide machine, it so happens, that the pre-pass scheduler places the divide operation before the subroutine call operation, which causes a long latency edge to be drawn between these operations during post-pass scheduling. A number of such long latency edges makes the schedule latency-bound on the 6-wide machine, and thus causes its increase in schedule length over the 4-wide machine.

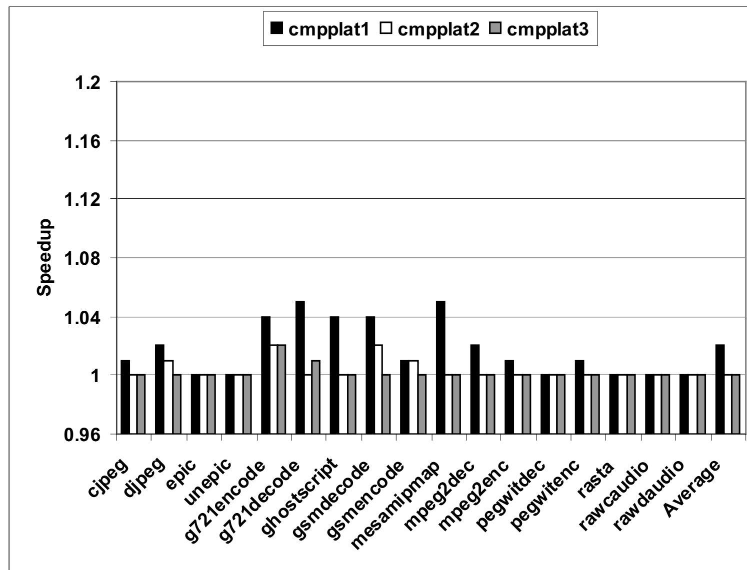
3.5.2.2 'Extend-all' PPALS

Figure 3.19(a) shows the speedup over the 4-wide baseline processor that is achieved on acyclic regions by 'extend-all' PPALS on the three 4-wide probabilistic predicate-aware processors with cmpp latencies of 1, 2 and 3 cycles. Figure 3.19(b) shows the corresponding data for 6-wide processors. In both cases the conflict detection and recovery latency (CDRL) is 1 cycle. As explained below, the results for CDRL=0 cycles are virtually identical, and hence we do not report them.

As stated above, the schedule length of acyclic regions is constrained by the latencies of the operations, with the cmpp operation generally being on the critical path. Therefore, the performance of PPALS decreases with increased cmpp latency for both



(a) Speedup of $P_{ppas}(4,1/2/3,1)$ over $P_{base}(4)$



(b) Speedup of $P_{ppas}(6,1/2/3,1)$ over $P_{base}(6)$

Figure 3.19: 'Extend-all' PPALS speedup over BALS for acyclic regions only (CDRL=1)

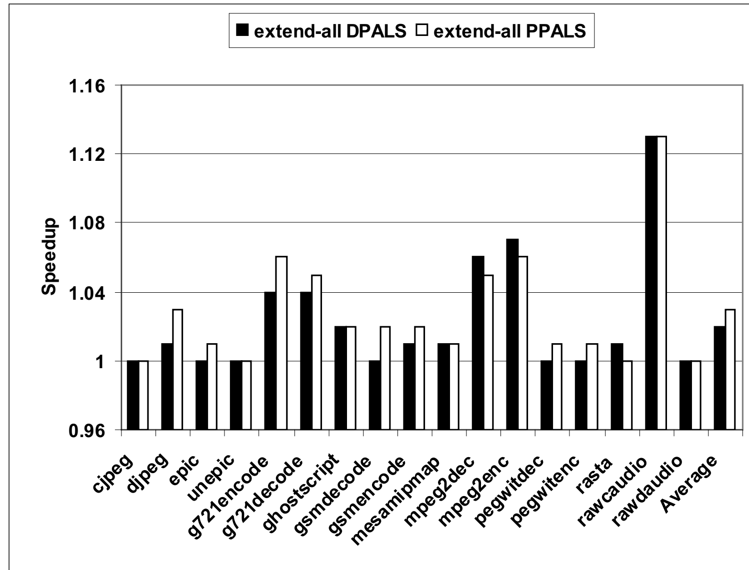
processor models, resulting in very small speedup (5% and 3% average speedup, respectively) for cmpp latencies of 2 and 3 cycles for 4-wide machines, and no speedup for 6-wide machines with these same latencies.

For mpeg2enc we see that the 4-wide probabilistic predicate-aware machine performs worse with cmpp latency 2 than with cmpp latency 3. This is caused, once again, by a long latency operation anomaly.

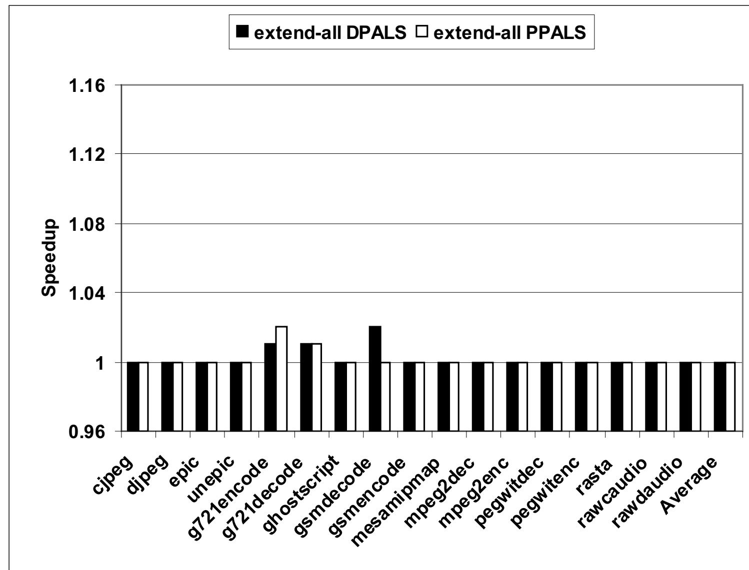
Recall that as we go to a wider machine while keeping the latency of the predicate-aware machine fixed, the latency-constrained lower bound remains the same; however, the resource-constrained lower bound decreases, approaching (and even finally falling below) the latency-constrained lower bound. Furthermore as resources are added, the length of the baseline schedule, which suffers no increase in cmpp latency, also decreases and it too would eventually become purely latency bound and fall below the cmpp latency dependent latency-constrained lower bound. Resource increases thus reduce the headroom for PPALS, which explains the degradation in PPALS speedup as we go from a 4-wide to a 6-wide baseline machine for fixed cmpp latency.

To see how much extra performance PPALS can gain over DPALS by enabling more aggressive combining, we next compare 'extend-all' PPALS with 'extend-all' DPALS. The left bar of Figure 3.20(a) shows the speedup over the 4-wide machine that is achieved by 'extend-all' DPALS on a 4-wide deterministic predicate-aware machine with cmpp latency 3. The right bar of Figure 3.20(a) shows the speedup over the 4-wide baseline machine that is achieved by 'extend-all' PPALS on a 4-wide probabilistic predicate-aware machine with cmpp latency 3 and a CDRL of 1 cycle. Figure 3.20(b) shows the corresponding data for the 6-wide machines.

We see that for a 4-wide machine, 'extend-all' PPALS achieves slightly better performance than 'extend-all' DPALS for 8 applications, and slightly worse performance



(a) Speedup of $P_{dpas}(4,3)$ and $P_{ppas}(4,3,1)$ over $P_{base}(4)$



(b) Speedup of $P_{dpas}(6,3)$ and $P_{ppas}(6,3,1)$ over $P_{base}(6)$

Figure 3.20: 'Extend-all' DPALS and PPALS over BALS speedup for acyclic regions only (CDRL=1, cmpp latency=3)

for 3 applications (mpeg2dec, mpeg2enc and rasta) resulting in an average speedup of 1% over DPALS. The reason that PPALS offers such a modest gain over DPALS is that, as Table 3.1 showed, for a cmpp latency of 3 there is very little scheduling headroom for the probabilistic predicate-aware scheduler to explore. The reason for PPALS performing worse in some of the cases than DPALS is that the latency adjustment phase described in Section 2.4.1 and Section 3.4.1 will only extend the latency of a cmpp operation to those consumers that can potentially be combined with some other operations in the region. Since PPALS allows more flexible combining than DPALS, it will extend the latency on more edges than DPALS. This causes regions scheduled with PPALS to have longer critical paths than the same regions scheduled with DPALS. In the case of these three benchmarks, the extra combining benefit offered by PPALS does not compensate for this critical path increase.

Table 3.2(a) presents the schedule length achieved by 'extend-all' PPALS on the three 4-wide machines with cmpp latencies of 1, 2 and 3 cycles, respectively, and a CDRL of 1 cycle. Table 3.2(b) shows the corresponding data for the three 6-wide machines. Column 2 shows the baseline schedule length. Columns 3-5 show the schedule lengths achieved by DPALS and PPALS for cmpp latency 1: column 3 shows the schedule length for DPALS, column 4 shows the static schedule length achieved by PPALS ignoring the compiler estimated conflict delay due to combining, and column 5 shows the compiler estimated expected schedule length achieved by PPALS including the expected delay due to conflicts. The difference between column 5 and column 4 is the expected delay due to conflict. Column 6 shows the actual dynamic schedule length achieved during program execution and column 7 shows the relative error between the achieved dynamic schedule length and compiler estimated schedule length. Note that for the last ('average') row, we show the average of the

		cmpplat1					cmpplat2					cmpplat3				
Benchmark	SL_{bals}	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err
cjpeg	46.11	44.01	39.09	39.20	39.21	0.03	45.71	42.46	42.60	42.60	0.00	46.11	46.10	46.10	46.10	0.00
djpeg	14.03	13.69	13.25	13.28	13.25	-0.23	13.88	13.47	13.51	13.49	-0.15	13.87	13.59	13.62	13.60	-0.15
epic	13.55	13.54	13.44	13.45	13.45	0.00	13.54	13.43	13.44	13.44	0.00	13.50	13.44	13.45	13.45	0.00
unepic	20.11	20.05	19.63	19.83	19.83	0.00	19.98	20.11	20.11	20.11	0.00	20.11	20.11	20.11	20.11	0.00
g721encode	29.96	26.60	25.89	25.89	26.03	0.54	27.68	26.89	26.89	27.01	0.44	28.75	28.25	28.25	28.25	0.00
g721decode	29.53	26.38	26.06	26.16	26.31	0.57	27.33	26.92	27.03	27.17	0.52	28.33	27.99	28.09	28.10	0.04
ghostscript	15.14	13.68	13.53	13.56	13.58	0.15	14.32	14.19	14.20	14.21	0.07	14.90	14.83	14.85	14.85	0.00
gsmdecode	23.52	22.61	22.17	22.17	22.18	0.05	23.03	22.79	22.79	22.79	0.00	23.52	22.97	22.97	22.97	0.00
gsmencode	30.73	29.28	25.52	25.52	25.52	0.00	29.48	28.10	28.45	28.45	0.00	30.36	30.05	30.05	30.05	0.00
mesamipmap	50.49	49.27	46.18	46.18	46.98	1.70	49.27	47.11	47.11	47.91	1.67	50.22	49.88	49.88	49.89	0.02
mpeg2dec	16.44	15.22	14.13	14.18	14.19	0.07	15.08	15.20	15.22	15.22	0.00	15.56	15.68	15.70	15.71	0.06
mpeg2enc	32.38	31.58	27.87	28.03	28.07	0.14	29.91	30.71	30.85	30.87	0.06	30.29	30.28	30.38	30.40	0.07
pegwitdec	18.72	18.68	18.35	18.35	18.35	0.00	18.64	18.46	18.46	18.46	0.00	18.67	18.50	18.50	18.50	0.00
pegwitenc	16.57	16.41	16.00	16.00	16.00	0.00	16.41	16.31	16.31	16.31	0.00	16.50	16.44	16.44	16.44	0.00
rasta	18.71	18.61	18.41	18.42	18.42	0.00	18.64	18.58	18.58	18.58	0.00	18.61	18.68	18.68	18.68	0.00
rawcaudio	17.46	15.47	15.47	15.47	15.47	0.00	14.97	14.97	14.97	14.97	0.00	15.47	15.47	15.47	15.47	0.00
rawdaudio	12.91	12.91	12.90	12.90	12.90	0.00	12.91	12.90	12.90	12.90	0.00	12.91	12.91	12.91	12.91	0.00
Average	23.90	22.82	21.64	21.68	21.75	0.20	22.99	22.51	22.55	22.62	0.17	23.39	23.24	23.26	23.26	0.02

(a) 4-wide machine

		cmpplat1					cmpplat2					cmpplat3				
Benchmark	SL_{bals}	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err
cjpeg	38.06	38.06	37.60	37.60	37.60	0.00	38.06	38.06	38.06	38.06	0.00	38.06	38.06	38.06	38.06	0.00
djpeg	11.47	11.41	11.29	11.31	11.29	-0.18	11.42	11.40	11.40	11.40	0.00	11.46	11.43	11.43	11.43	0.00
epic	12.61	12.61	12.59	12.59	12.59	0.00	12.55	12.55	12.55	12.55	0.00	12.55	12.57	12.57	12.57	0.00
unepic	19.76	19.76	19.76	19.76	19.76	0.00	19.76	19.76	19.76	19.76	0.00	19.76	19.76	19.76	19.76	0.00
g721encode	23.60	22.82	22.68	22.68	22.68	0.00	23.46	23.25	23.25	23.25	0.00	23.39	23.25	23.25	23.25	0.00
g721decode	23.16	22.09	21.96	22.02	22.04	0.09	22.96	23.03	23.09	23.09	0.00	22.96	22.89	22.96	22.95	-0.04
ghostscript	12.42	11.91	11.87	11.88	11.96	0.67	12.42	12.41	12.41	12.42	0.08	12.41	12.41	12.41	12.42	0.08
gsmdecode	22.04	22.04	21.17	21.17	21.17	0.00	21.62	21.62	21.62	21.62	0.00	21.62	22.04	22.04	22.04	0.00
gsmencode	25.10	25.08	24.82	24.82	24.82	0.00	24.98	24.95	24.95	24.95	0.00	24.98	25.09	25.09	25.09	0.00
mesamipmap	40.44	39.47	38.48	38.48	38.51	0.08	40.41	40.41	40.41	40.42	0.02	40.44	40.44	40.44	40.44	0.00
mpeg2dec	13.37	13.03	12.96	12.96	13.08	0.92	13.31	13.31	13.31	13.31	0.00	13.36	13.36	13.36	13.36	0.00
mpeg2enc	34.45	34.35	34.26	34.28	34.26	-0.06	34.45	34.45	34.45	34.45	0.00	34.45	34.45	34.45	34.45	0.00
pegwitdec	18.07	18.07	18.04	18.04	18.04	0.00	18.07	18.07	18.07	18.07	0.00	18.07	18.07	18.07	18.07	0.00
pegwitenc	14.92	14.84	14.83	14.83	14.83	0.00	14.92	14.91	14.91	14.91	0.00	14.92	14.91	14.91	14.91	0.00
rasta	15.71	15.68	15.60	15.68	15.66	-0.13	15.71	15.71	15.71	15.71	0.00	15.71	15.71	15.71	15.71	0.00
rawcaudio	12.98	12.98	12.98	12.98	12.98	0.00	12.98	12.98	12.98	12.98	0.00	12.98	12.98	12.98	12.98	0.00
rawdaudio	11.91	11.91	11.91	11.91	11.91	0.00	11.91	11.91	11.91	11.91	0.00	11.91	11.91	11.91	11.91	0.00
Average	20.59	20.36	20.16	20.18	20.19	0.12	20.53	20.52	20.52	20.52	0.01	20.53	20.55	20.55	20.55	0.01

(b) 6-wide machine

Table 3.2: Schedule length achieved by BALS, 'extend-all' DPALS and 'extend-all' PPALS (CDRL=1)

absolute values of the relative errors, so as to avoid mutual cancellation of the negative and positive errors. Columns 8-12 and 13-17 show the corresponding statistics for cmpp latency 2 and 3, respectively.

Perhaps the most interesting observation about the PPALS data presented in this table is that (i) the compiler estimated conflict is never more than 0.18% of the static schedule time and (ii) that the absolute difference between the compiler estimated time (including conflict delay) and the actual execution time is never more than 0.32% of the compiler estimated time, for any application on any of the six machines. This indicates that the compiler estimated time is very accurate. Furthermore the PPALS schedule conflicts that do occur have almost negligible effect over all the acyclic

regions. There are a couple of reasons for this behavior. First, is the restrictive nature of the combining algorithm, which only allows operations to be combined if the estimated penalty due to conflict for that timeslot does not exceed 1.0. Second, is that relatively few operations get combined, since there is a very limited range of time slots where a given operation can be scheduled without increasing the overall schedule length (this is different for the cyclic scheduler which, as we have seen, allows much more scheduling mobility for an operation). Third, is that many operations which are scheduled closely to one another because of similar priorities are disjoint and hence cause no penalty due to conflict when combined. Fourth, many of the predicated operations have zero execution frequency which in the actual benchmark runs, which results in zero penalty when they are combined with other operations. Finally, when we increase cmpp latency or go to a wider machine, the number of regions that benefit from PPALS decreases, and that causes the error to decrease even further.

In this and the following section, the PPALS results are only reported for a CDRL of 1 cycle. The results shown are almost identical to the results for CDRL of 0 cycles, which is due to the very small number of conflicts that occur, due to the five aforementioned reasons.

3.5.2.3 'First-fit' PPALS

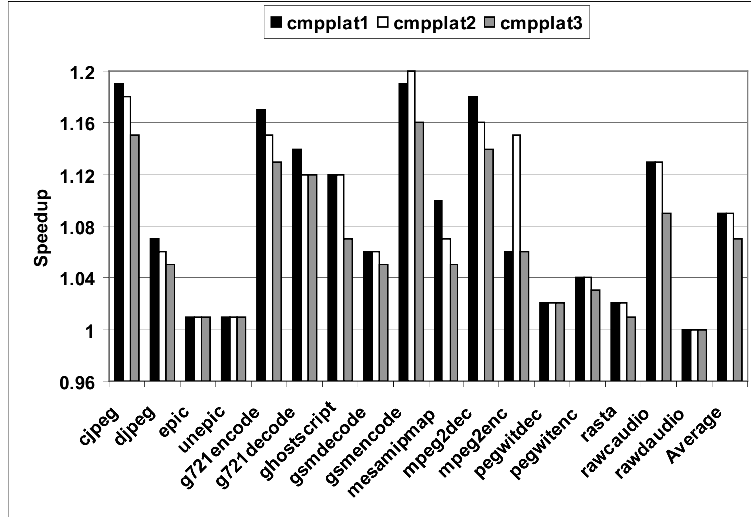
In this section we study the performance of 'first-fit' PPALS which, unlike 'extend-all' PPALS, does not extend cmpp latency prior to scheduling, but only takes advantage of combining opportunities as they arise.

As we said in Section 3.4.3.2 , if an operation is scheduled at a cycle that is less than *extendedlatency* cycles away from its already scheduled cmpp producer, this operation must reserve its resource unconditionally, and its execution frequency

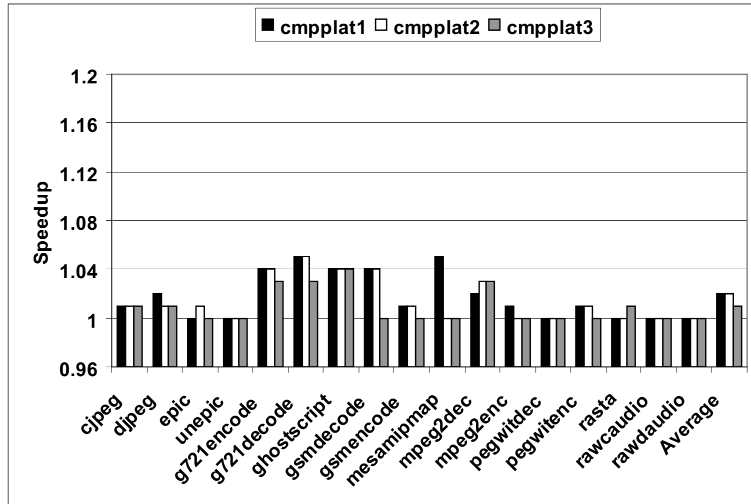
in such a schedule will be assigned as 1. On the other hand, if the operation is scheduled at a cycle that is at least *extendedlatency* cycles away from its scheduled cmpp producer, the operation's predicate can be read early, and it can reserve its resource conditionally. Obviously the delay due to combining with this operation will be larger in this case than in the second case.

Figure 3.21(a) shows, for acyclic regions only, the speedup over the 4-wide baseline processor that is achieved by 'first-fit' PPALS on the three 4-wide probabilistic predicate-aware processors with cmpp latencies of 1, 2 and 3 cycles. Figure 3.21(b) shows the corresponding data for the 6-wide processors. By comparing with Figure 3.19 we see that for cmpp latencies of 2 and 3 on the 4-wide machine, 'first-fit' PPALS achieves 4% more speedup over the baseline than 'extend-all' PPALS does. Whereas for the 6-wide machine, 'first-fit' PPALS gets only 2% and 1% more speedup than 'extend-all' PPALS for cmpp latencies 2 and 3, respectively. This improvement comes from the fact that 'first-fit' PPALS does not extend the cmpp latencies prior to scheduling, which makes the latency-constrained lower bound for the cases with extended cmpp latencies of 2 and 3 be the same as for a cmpp latency of 1. This bound may be very optimistic, but it will never grow beyond the baseline schedule length, as happens with 'extend-all' PPALS as cmpp latency increases to 2 and 3 cycles. Furthermore the priority scheduling and the 'first-fit' strategy of 'first-fit' PPALS limits the growth of the critical path in its schedules.

In the next set of results we compare the performance of 'first-fit' PPALS with the performance of 'first-fit' DPALS and 'extend-all' PPALS for 4- and 6-wide machines with a cmpp latency of 3. The left bar of Figure 3.22(a) shows the speedup over a 4-wide baseline machine that is achieved by 'extend-all' PPALS on a 4-wide probabilistic predicate-aware machine. The middle bar shows the speedup achieved by 'first-fit'

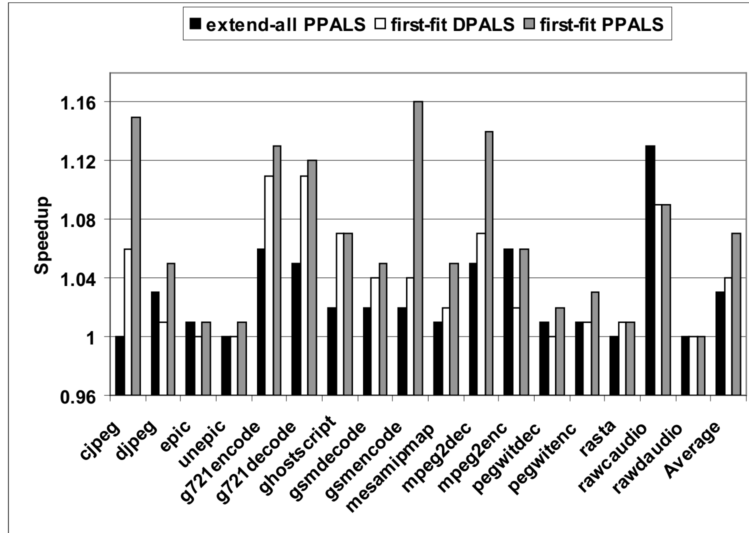


(a) Speedup of $P_{ppas}(4,1/2/3,1)$ over $P_{base}(4)$

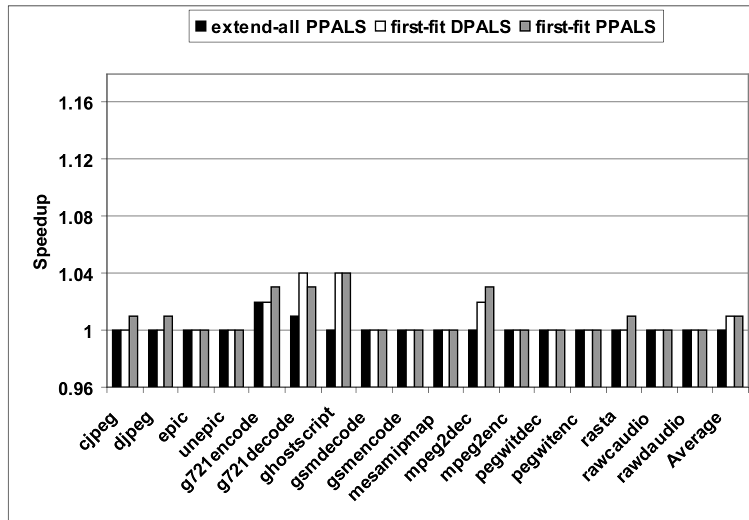


(b) Speedup of $P_{ppas}(6,1/2/3,1)$ over $P_{base}(6)$

Figure 3.21: 'First-fit' PPALS speedup over BALS for acyclic regions only (CDRL=1)



(a) Speedup of $P_{dpas}(4,3)$ and $P_{ppas}(4,3,1)$ over $P_{base}(4)$



(b) Speedup of $P_{dpas}(6,3)$ and $P_{ppas}(6,3,1)$ over $P_{base}(6)$

Figure 3.22: 'First-fit' DPALS and PPALS speedup over BALS for acyclic regions only (CDRL=1 for PPALS)

DPALS on a 4-wide deterministic predicate-aware machine. The right bar shows the speedup achieved by 'first-fit' PPALS on a 4-wide probabilistic predicate-aware machine with a CDRL of 1. Figure 3.22(b) shows the corresponding data for the 6-wide machines. As expected, 'first-fit' PPALS outperforms both 'first-fit' DPALS and 'extend-all' PPALS in all but one case of rawcaudio.

In rawcaudio, 'extend-all' PPALS performs better than the other two schemes. The situation here is similar to the mpeg2enc long latency operation anomaly discussed earlier, where a long latency operation scheduled before a subroutine call (as happens in the case of 'first-fit' DPALS and PPALS) causes a dependence edge between these two operations to produce a longer schedule than when this long latency operation is scheduled after a subroutine call operation (as happens in the case of 'extend-all' PPALS) producing no such dependence edge.

We also notice that 'first-fit' DPALS outperforms 'extend-all' PPALS on average since the ability of 'extend-all' PPALS to combine more operations than 'first-fit' DPALS generally does not compensate for its increased schedule length due to the increased critical path length caused by its extended cmpp latency.

Also note that despite having very little scheduling headroom for the 6-wide machine, on 7 applications 'first-fit' PPALS does achieves some speedup, from 1% for rasta, cjpeg and djpeg to 4% for ghostscript. By contrast, 'extend-all' PPALS achieves a speedup for only two applications: 1% for g721decode and 2% for g721encode.

Table 3.3 augments Table 3.2 with schedule length data for 'first-fit' DPALS and 'first-fit' PPALS with cmpp latencies of 1, 2 and 3 cycles. The baseline (BALS) schedule length is repeated here for comparison.

		cmppcombth1					cmppcombth2					cmppcombth3				
Benchmark	SL_{bals}	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err
cjpeg	46.11	44.01	39.09	39.20	39.21	0.03	44.01	39.09	39.20	39.21	0.03	43.64	39.97	40.06	40.08	0.05
djpeg	14.03	13.69	13.25	13.28	13.25	-0.23	13.69	13.25	13.28	13.25	-0.23	13.86	13.37	13.40	13.37	-0.22
epic	13.55	13.54	13.44	13.45	13.45	0.00	13.54	13.44	13.45	13.45	0.00	13.54	13.44	13.45	13.45	0.00
unepic	20.11	20.05	19.63	19.83	19.83	0.00	20.05	19.63	19.83	19.83	0.00	20.10	19.68	19.87	19.87	0.00
g721encode	29.96	26.60	25.89	25.89	26.03	0.54	26.60	25.89	25.89	26.03	0.54	27.03	26.25	26.25	26.47	0.83
g721decode	29.53	26.38	26.06	26.16	26.31	0.57	26.38	26.06	26.16	26.31	0.57	26.65	26.05	26.16	26.30	0.53
ghostscript	15.14	13.68	13.53	13.56	13.58	0.15	13.68	13.53	13.56	13.58	0.15	14.11	14.00	14.03	14.11	0.57
gsmdecode	23.52	22.61	22.17	22.17	22.18	0.05	22.61	22.17	22.17	22.18	0.05	22.64	22.31	22.31	22.32	0.04
gsmencode	30.73	29.28	25.52	25.52	25.52	0.00	29.28	25.52	25.52	25.52	0.00	29.66	26.28	26.55	26.55	0.00
mesamipmap	50.49	49.27	46.18	46.18	46.98	1.70	49.27	46.18	46.18	46.98	1.70	49.26	48.02	48.02	48.02	0.00
mpeg2dec	16.44	15.22	14.13	14.18	14.19	0.07	15.22	14.13	14.18	14.19	0.07	15.33	14.33	14.38	14.39	0.07
mpeg2enc	32.38	31.58	27.87	28.03	28.07	0.14	31.58	27.87	28.03	28.07	0.14	31.60	30.33	30.43	30.48	0.16
pegwitdec	18.72	18.68	18.35	18.35	18.35	0.00	18.68	18.35	18.35	18.35	0.00	18.68	18.39	18.39	18.39	0.00
pegwitenc	16.57	16.41	16.00	16.00	16.00	0.00	16.41	16.00	16.00	16.00	0.00	16.41	16.15	16.15	16.15	0.00
rasta	18.71	18.61	18.41	18.42	18.42	0.00	18.61	18.41	18.42	18.42	0.00	18.61	18.52	18.52	18.52	0.00
rawcaudio	17.46	15.47	15.47	15.47	15.47	0.00	15.47	15.47	15.47	15.47	0.00	15.97	15.97	15.97	15.97	0.00
rawdaudio	12.91	12.91	12.90	12.90	12.90	0.00	12.91	12.90	12.90	12.90	0.00	12.91	12.90	12.90	12.90	0.00
Average	23.90	22.82	21.64	21.68	21.75	0.20	22.82	21.64	21.68	21.75	0.20	22.94	22.12	22.17	22.20	0.15

(a) 4-wide machine

		cmppcombth1					cmppcombth2					cmppcombth3				
Benchmark	SL_{bals}	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err	SL_{dpals}	SL_{ppals_s}	SL_{ppals_c}	SL_{ppals_d}	%err
cjpeg	38.06	38.06	37.60	37.60	37.60	0.00	38.06	37.60	37.60	37.60	0.00	38.06	37.60	37.60	37.60	0.00
djpeg	11.47	11.41	11.29	11.31	11.29	-0.18	11.41	11.32	11.34	11.32	-0.18	11.41	11.35	11.36	11.35	-0.09
epic	12.61	12.61	12.59	12.59	12.59	0.00	12.61	12.53	12.53	12.53	0.00	12.61	12.59	12.59	12.59	0.00
unepic	19.76	19.76	19.76	19.76	19.76	0.00	19.76	19.76	19.76	19.76	0.00	19.76	19.76	19.76	19.76	0.00
g721encode	23.60	22.82	22.68	22.68	22.68	0.00	22.75	22.68	22.68	22.68	0.00	23.10	22.68	22.68	22.82	0.61
g721decode	23.16	22.09	21.96	22.02	22.04	0.09	22.03	22.03	22.09	22.09	0.00	22.36	22.36	22.43	22.44	0.04
ghostscript	12.42	11.91	11.87	11.88	11.96	0.67	11.92	11.87	11.88	11.95	0.59	11.93	11.89	11.90	11.98	0.67
gsmdecode	22.04	22.04	21.17	21.17	21.17	0.00	22.04	21.20	21.20	21.20	0.00	22.04	22.04	22.04	22.04	0.00
gsmencode	25.10	25.08	24.82	24.82	24.82	0.00	25.08	24.82	24.82	24.82	0.00	25.08	25.06	25.06	25.06	0.00
mesamipmap	40.44	39.47	38.48	38.48	38.51	0.08	40.40	40.35	40.35	40.37	0.05	40.40	40.36	40.36	40.36	0.00
mpeg2dec	13.37	13.03	12.96	12.96	13.08	0.92	13.03	13.00	13.00	13.00	0.00	13.07	12.97	12.97	12.97	0.00
mpeg2enc	34.45	34.35	34.26	34.28	34.26	-0.06	34.35	34.35	34.35	34.37	0.06	34.40	34.35	34.37	34.39	0.06
pegwitdec	18.07	18.07	18.04	18.04	18.04	0.00	18.06	18.03	18.03	18.03	0.00	18.06	18.03	18.03	18.03	0.00
pegwitenc	14.92	14.84	14.83	14.83	14.83	0.00	14.84	14.84	14.84	14.84	0.00	14.92	14.91	14.91	14.91	0.00
rasta	15.71	15.68	15.60	15.68	15.66	-0.13	15.71	15.60	15.66	15.63	-0.19	15.71	15.60	15.65	15.60	-0.32
rawcaudio	12.98	12.98	12.98	12.98	12.98	0.00	12.98	12.98	12.98	12.98	0.00	12.98	12.98	12.98	12.98	0.00
rawdaudio	11.91	11.91	11.91	11.91	11.91	0.00	11.91	11.91	11.91	11.91	0.00	11.91	11.91	11.91	11.91	0.00
Average	20.59	20.36	20.16	20.18	20.19	0.12	20.41	20.29	20.30	20.30	0.06	20.46	20.38	20.39	20.40	0.11

(b) 6-wide machine

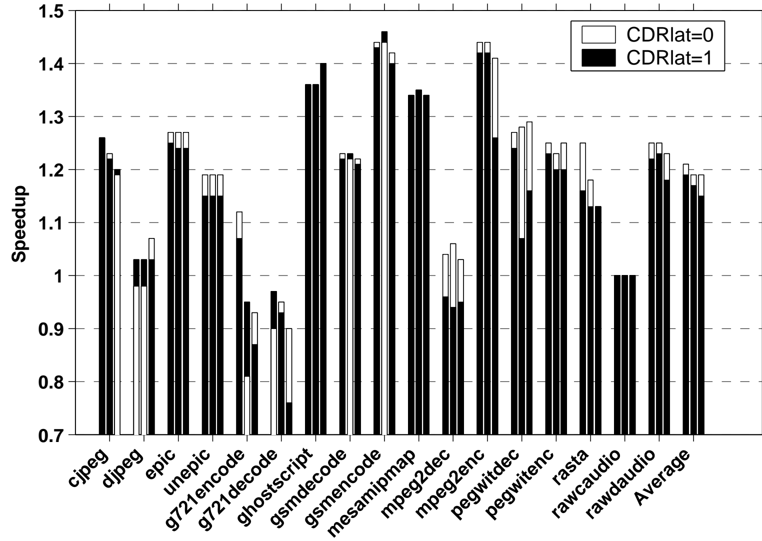
Table 3.3: Schedule length achieved by BALS, 'first-fit' DPALS and 'first-fit' PPALS (CDRL=1)

3.5.2.4 PPAMS

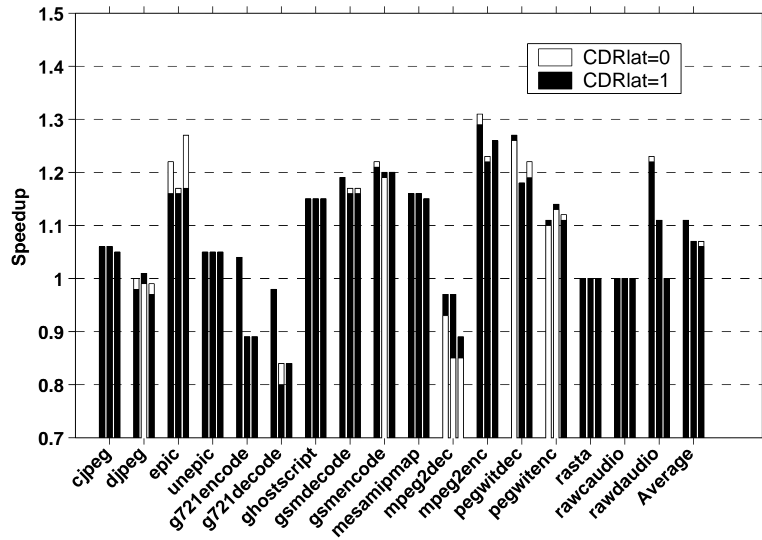
In this section we present the performance results of PPAMS. As was said in Section 3.4.4, PPAMS calls **PPAS-DDGLatency-Adjust** which extends the cmpp latency to each operation whenever that extension may improve the performance.

Speedup on Cyclic Regions

Figure 3.23(a) shows the performance improvement achieved by PPAMS over BAMS on cyclic regions only, for a 4-wide machine. The left, middle and right bars correspond to cmpp latencies of 1, 2, and 3 cycles, respectively. In addition each



(a) Speedup of P_{ppas}(4,1/2/3,0/1) over P_{base}(4)



(b) Speedup of P_{ppas}(6,1/2/3,0/1) over P_{base}(6)

Figure 3.23: PPAMS over BAMS speedup for cyclic regions only (left bar has cmpp latency=1, middle has 2, right has 3)

bar contains two overlaid sub-bars: black which shows speedup with CDRL = 0, and white which shows speedup with CDRL = 1. The sub-bar which corresponds to larger speedup is always plotted first and is then overlaid with the sub-bar which corresponds to the smaller speedup. This way only the higher part of the larger sub-bar, which shows the additional performance gain over the shorter sub-bar will be visible, and the shorter sub-bar will always be visible, unless both bars are the same height. For example in Figure 3.23(a), the third bar of djpeg (corresponding to cmpp latency 3) shows that PPAMS achieves a speedup of 7% with CDRL = 0 and 3% for CDRL = 1, with the white portion corresponding to the 4% difference between the two. If both sub-bars are the same height, the sub-bar corresponding to CDRL =1 is always overlaid on the sub-bar corresponding to CDRL = 0, so that only the black color is visible, as is the case, for example, with first cjpeg bar (corresponding to cmpp latency 1).

For cyclic regions, as we have seen from Table 3.1, the PPAMS lower-bound is determined by the resource-constrained schedule length ($ResMII_{ppams}$) of the probabilistic predicate-aware processor. recurrence-constrained lower bound ($RecMII$) is not a limiting factor for either 4-wide or 6-wide machine models. As Table 3.1 shows, $RecMII$, even for a cmpp latency of 3, is much smaller than $ResMII_{ppams}$. Being resource limited offers a substantial headroom for PPAMS improvement over BAMS for all three cmpp latencies. This headroom, computed as $100 \times \frac{(II_{bams} - ResMII_{ppams})}{II_{bams}}$, is 32% for the 4-wide machine, and 24% for the 6-wide machine. This headroom is much larger than for PPALS, especially for the higher cmpp latencies, which is why PPAMS achieves a substantially higher speedup than PPALS for all the probabilistic predicate-aware machines.

The actual performance of PPAMS does vary with cmpp latency. For some of

the benchmarks, the performance decreases as cmpp latency increases. For example, for cjpeg performance decreases for both 4- and 6-wide machines as cmpp latency increases from 1 to 2 and from 2 to 3 cycles. For such benchmarks the performance degradation for increased cmpp latencies occurs because higher cmpp latency increases the length of the loop epilogue (although the II remains the same), which leads to longer total execution time. This effect is particularly visible for loops with small trip count.

On the other hand, for some of the benchmarks, the performance anomalously increases as cmpp latency increases. For example, for djpeg the speedup increases as cmpp latency increases from 2 to 3 for the 4-wide machine with CDRL = 0. In this case, the PPAS machine with the higher latency fails to find a schedule for the same II as the machine with the lower cmpp latency. Hence, the II is increased, resulting in a smaller *esc*, and incidentally thereby achieving better performance when these loops have a low trip count. For djpeg the PPAMS failure to find a valid schedule for a given II happens due to rotating register limitations; as cmpp latency increases, the length of the single iteration schedule increases, and consequently so does the number of required rotating registers. When there are insufficient rotating registers to support a schedule with the current II, the current II is increased.

Finally, for some applications, such as rasta, the performance remains the same for all three cmpp latencies. This happens because the dominant loops are long trip count loops with a small $RecMII$ of 1 for all three cmpp latencies. Therefore $RecMII$ is not a limiting factor, and the size of the epilogue does not impact the overall performance since the loops spend the large majority of their execution time in the steady-state.

As Figure 3.23 shows, the performance of PPAMS also varies with CDRL. In

general, we expect PPAMS to have higher performance with CDRL = 0, which allows a smaller delay due to conflicts and therefore more operation combining than with CDRL = 1. This is true for most of the applications, but there are some exceptions. For example, as Figure 3.23(a) shows, cjpeg achieves 2% less speedup for CDRL=0 than for CDRL=1 on a 4-wide machine with a cmpp latency of 3. This is caused by the low trip count problem: the probabilistic predicate-aware machine with CDRL=0 achieves smaller II, but consequently higher esc than the corresponding machine with CDRL=1, which adversely affects the overall performance with CDRL=0 on the loops with low trip counts.

We also see that for some of the applications, such as mesamipmap, the achieved speedup is the same for both values of CDRL. This happens because these loops contain a large number of well balanced *if-then-else* statements and thus achieve most of their performance gain due to combining a large number of disjoint operations with zero delay due to conflict, thereby eliminating the effect of CDRL.

Some of the applications, such as g721decode on a 4-wide probabilistic predicate-aware machine and mpeg2dec on a 6-wide probabilistic predicate-aware machine, lose performance with PPAMS. As Table 3.4 shows below, they do achieve better II with PPAMS than with BAMS, but due to their low trip count and long epilogue, they deliver less overall performance with PPAMS than with BAMS. As in Chapter 2 we report this loss in performance. Some applications, such as rawcaudio on a 4-wide machine, and both rawcaudio and rawdaudio on a 6-wide machine, do not achieve any speedup. As Table 3.1 shows, these applications are $RecMII$ limited for these machines.

By examining Figure 3.23(a) and (b), we see that in most cases a 4-wide probabilistic predicate-aware processor loses more performance due to higher CDRL than

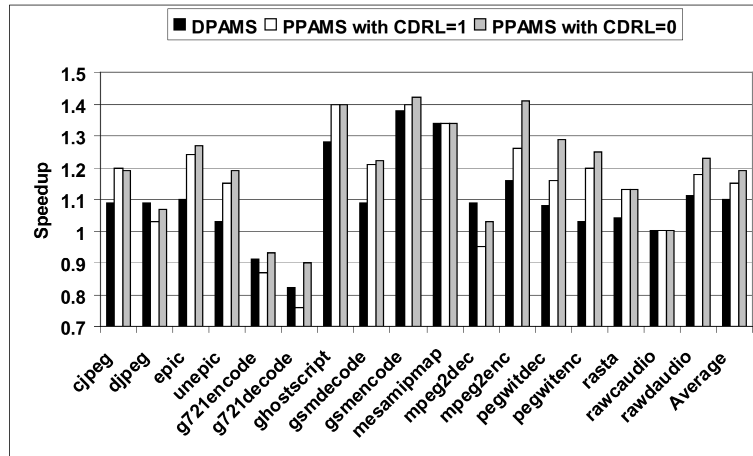
a 6-wide probabilistic predicate-aware processor. On average, a 4-wide PPAMS processor with a conflict detection latency of 0 cycles performs 5% better, and a 6-wide PPAMS processor performs 1% better, than the corresponding PPAMS processor with a conflict detection latency of 1 cycle. Higher conflict detection latency has more impact on the 4-wide PPAMS processor than on the 6-wide PPAMS processor because the 6-wide machine has more resources than the 4-wide machine, and thus has less resource sharing and fewer conflicts.

Comparison with DPAMS

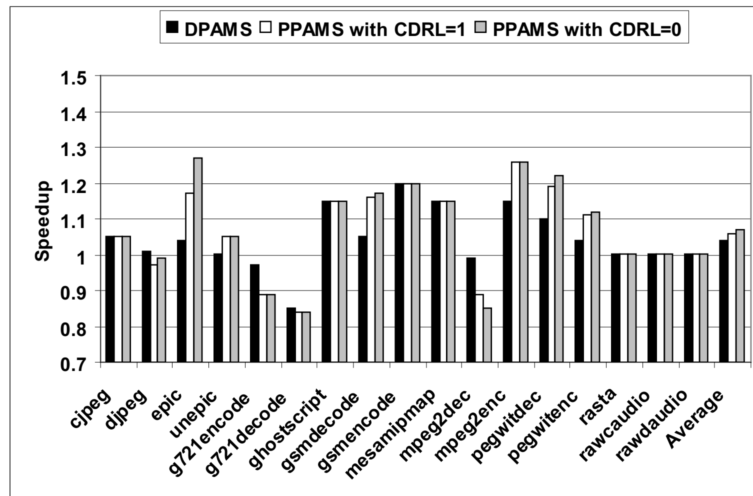
Figure 3.24 shows the speedup achieved by DPAMS and PPAMS schemes on the cyclic regions only, for cmpp latency 3. Figure 3.24(a) shows the speedup over the baseline machine that is achieved by the 4-wide DPAMS machine (P_{dpas}(4,3)) in the left bar, the speedup achieved by P_{ppas}(4,3,1) in the middle bar, and the speedup achieved by P_{ppas}(4,3,0) in the right bar. Figure 3.24(b) shows the corresponding data for the 6-wide machine.

On average the 4-wide PPAMS processor with a conflict detection latency of 0 cycles performs 8% better than the 4-wide DPAMS machine and 19% better than the corresponding baseline machine; these speedups are 3% and 7%, respectively for the corresponding 6-wide machines. Again we see that the performance gain of PPAMS on a 6-wide machine is diminished due to the fact the 6-wide machine has more resources than the 4-wide machine and thus benefits less from resource sharing. As CDRL increases to 1, the performance gain of PPAMS over DPAMS decreases becoming 5% for a 4-wide machine and only 2% for a 6-wide machine.

We see that for some benchmarks DPAMS performs better than PPAMS for both CDRL=0 and CDRL=1. This occurs, for example, for djpeg and mpeg2dec on both



(a) Speedup of $P_{dpas}(4,3)$, $P_{ppas}(4,3,1)$ and $P_{ppas}(4,3,0)$ over $P_{base}(4)$



(b) Speedup of $P_{dpas}(6,3)$, $P_{ppas}(6,3,1)$ and $P_{ppas}(6,3,0)$ over $P_{base}(6)$

Figure 3.24: DPAMS and PPAMS speedups over BAMS for cyclic regions only (cmpp latency=3)

Benchmark	BRec	BRes	BII	Besc	Brr	DRec	DRes	DII	Desc	Drr	PRec	PRes	PIIs	PIIc	PIId	%error	Pesc	Prr
cjpeg	1.31	6.49	6.49	2.00	10.74	1.93	5.89	6.35	2.44	13.57	1.93	4.69	5.71	5.71	5.72	0.18	3.74	14.91
djpeg	1.00	58.64	58.64	1.00	28.38	1.00	53.17	53.66	1.01	29.89	1.00	42.69	46.71	48.66	49.07	0.84	2.50	49.25
epic	2.16	21.02	21.41	1.77	12.78	2.93	19.10	19.48	2.00	13.86	2.93	16.38	16.48	16.94	16.89	0.30	5.00	14.77
unepic	1.00	13.27	13.27	1.85	19.31	1.00	12.84	12.84	2.70	21.70	1.00	10.52	10.74	10.91	11.42	4.67	3.14	27.23
g721encode	1.00	30.00	30.00	1.00	12.00	1.00	21.00	23.50	3.00	21.50	1.00	20.24	21.50	21.53	21.51	0.09	4.00	22.50
g721decode	1.00	30.00	30.00	1.00	12.00	1.00	21.00	23.00	4.00	24.00	1.00	20.26	22.00	22.01	22.00	0.05	5.00	30.00
ghostscript	7.88	43.13	44.10	1.02	20.85	7.20	33.35	34.33	2.02	23.79	7.88	31.35	31.41	31.42	31.42	0.00	1.06	26.82
gsmdecode	7.93	27.73	27.92	1.10	13.86	10.40	24.95	25.73	1.96	18.13	11.44	21.46	22.49	23.08	23.06	0.09	5.10	35.56
gsmencode	7.87	74.39	74.85	1.00	16.56	7.25	46.76	54.87	1.92	17.68	8.93	45.61	53.36	53.40	53.40	0.00	4.31	21.13
mesamipmap	1.00	22.00	22.00	1.67	20.67	1.00	16.33	16.33	2.67	27.00	1.00	15.92	16.33	16.33	16.33	0.00	2.00	22.00
mpeg2dec	1.00	28.35	28.35	1.51	17.11	1.00	25.85	25.85	1.85	15.96	1.00	18.70	20.96	23.21	23.78	2.46	2.96	31.10
mpeg2enc	2.97	20.27	20.27	1.00	8.21	4.93	17.31	17.31	1.98	10.19	4.93	12.36	14.32	14.58	14.56	0.14	9.86	44.81
pegwitdec	1.97	20.67	20.67	0.05	13.97	1.45	18.70	18.70	1.05	14.03	1.97	12.46	12.86	15.58	15.95	2.37	4.95	29.59
pegwitenc	1.61	19.33	19.33	0.96	13.36	1.57	18.26	18.26	1.61	16.18	1.61	12.67	13.39	13.79	14.57	5.66	4.94	26.81
rasta	3.02	6.80	6.81	1.25	7.42	4.14	6.53	6.68	1.84	7.86	4.14	4.85	5.99	6.02	5.99	0.50	3.39	12.79
rawcaudio	20.00	24.00	26.00	1.00	12.00	30.00	22.00	32.00	3.00	20.00	30.00	17.13	30.00	30.00	30.00	0.00	1.00	17.00
rawdaudio	6.00	20.00	20.00	1.00	13.00	10.00	18.00	18.00	3.00	18.00	10.00	14.21	17.00	17.00	17.00	0.00	2.00	15.00
Average	4.04	27.42	27.65	1.19	14.84	5.16	22.41	23.94	2.24	18.43	5.40	18.91	21.25	21.78	21.92	1.02	3.82	25.96

(a) 4-wide machine

Benchmark	BRec	BRes	BII	Besc	Brr	DRec	DRes	DII	Desc	Drr	PRec	PRes	PIIs	PIIc	PIId	%error	Pesc	Prr
cjpeg	1.31	3.24	3.40	3.49	14.77	1.93	3.05	3.82	4.58	18.74	1.93	2.64	3.82	3.82	3.82	0.00	4.58	18.74
djpeg	1.00	29.32	29.32	1.01	33.87	1.00	26.84	27.34	1.51	46.79	1.00	25.18	25.85	26.76	27.11	1.31	2.01	44.31
epic	2.16	10.70	11.09	2.77	15.40	2.93	10.55	10.55	4.00	16.78	2.93	8.71	9.39	9.60	9.39	2.19	6.31	16.01
unepic	1.00	7.34	7.34	2.29	23.07	1.00	7.34	7.34	2.72	23.84	1.00	6.57	6.93	6.95	6.93	0.29	3.14	25.08
g721encode	1.00	15.00	15.00	2.00	16.50	1.00	12.00	13.00	3.50	23.50	1.00	11.14	12.50	12.50	12.50	0.00	4.50	25.50
g721decode	1.00	15.00	15.00	2.00	16.50	1.00	12.00	13.00	4.50	26.00	1.00	11.15	12.50	12.56	12.51	0.40	5.00	28.50
ghostscript	7.88	21.58	22.55	1.04	21.84	7.20	19.63	19.65	1.06	20.95	7.88	18.95	19.64	19.64	19.64	0.00	1.07	22.01
gsmdecode	7.93	13.87	14.75	2.76	20.96	10.40	12.94	14.30	4.52	26.93	11.44	10.76	12.64	12.86	12.83	0.23	7.80	46.89
gsmencode	7.87	37.65	38.66	2.01	18.22	7.25	30.36	32.92	3.14	37.16	8.93	29.32	32.88	32.89	32.89	0.00	3.82	30.90
mesamipmap	1.00	12.33	12.33	2.33	23.33	1.00	10.67	10.67	4.33	34.00	1.00	10.39	10.67	10.67	10.67	0.00	3.67	28.67
mpeg2dec	1.00	14.19	14.19	2.88	22.75	1.00	13.04	13.04	3.75	29.86	1.00	12.10	12.82	12.89	12.92	0.23	4.38	36.14
mpeg2enc	2.97	10.14	11.12	2.00	13.32	4.93	9.15	9.15	6.91	17.28	4.93	7.33	8.16	8.17	8.16	0.12	8.92	27.33
pegwitdec	1.97	10.84	10.84	1.05	15.03	1.45	9.84	9.84	1.22	19.30	1.97	7.73	7.89	8.09	8.11	0.25	6.03	29.86
pegwitenc	1.61	10.01	10.01	2.59	23.94	1.57	9.62	9.62	3.30	22.20	1.61	8.10	8.41	8.48	8.61	1.53	5.08	28.79
rasta	3.02	3.56	3.57	3.26	11.63	4.14	3.56	4.69	2.88	10.99	4.14	3.06	4.69	4.69	4.69	0.00	2.88	10.71
rawcaudio	20.00	12.00	20.00	1.00	13.00	30.00	11.00	30.00	1.00	12.00	30.00	8.83	30.00	30.00	30.00	0.00	1.00	12.00
rawdaudio	6.00	10.00	10.00	2.00	16.00	10.00	9.00	11.00	3.00	17.00	10.00	7.33	10.00	10.00	10.00	0.00	3.00	15.00
Average	4.04	13.93	14.66	2.15	18.83	5.16	12.39	14.11	3.29	23.73	5.40	11.14	13.46	13.56	13.57	0.39	4.30	26.26

(b) 6-wide machine

Table 3.4: Schedule length achieved by BAMS, DPAMS and PPAMS for cmpplat=3 and CDRL=1

the 4- and 6-wide machines. As Table 3.4 below shows, PPAMS does achieve smaller II, but has a longer epilogue than DPAMS in all these cases which, due to the low trip count of the loops, results in PPAMS getting lower performance than DPAMS.

Other Scheduling Measurements of PPAMS

Table 3.4(a) shows various scheduling measurements for $P_{base}(4)$, $P_{dpas}(4, 3)$ and $P_{ppas}(4,3, 1)$ machines. These measurements were collected over all ppa-improvable regions. Columns 2, 7 and 12 show the $RecMII$ for the three schedulers. As expected, for some applications (epic, gsmdecode, gsmencode, mpeg2enc, rasta, rawcaudio and rawdaudio) $RecMII$ increases for both DPAMS and PPAMS. Also note that $RecM$ is slightly smaller for DPAMS than for PPAMS. This increase is due to the latency

extension step which extends latency on a larger number of edges for PPAMS than for DPAMS since PPAMS has more operations that can benefit from combining than DPAMS. Therefore, whenever some of the edges whose latencies were extended by PPAMS but not by DPAMS are on the critical path, PPAMS has a higher $RecMII$ than DPAMS.

Columns 3,8 and 13 show the value of $ResMII$ for the three schedulers, respectively. As explained in Section 2.4.5, the baseline scheduler increments the resource usage by one regardless of the operation's guarding predicate and its execution frequency. DPAMS increments the resource usage count by one per group of disjoint operations. Many ppa-improvable loops have a large number of *if-then-else* statements; DPAMS reduces their $ResMII$ on average by 22% relative to the baseline. PPAMS further decreases their $ResMII$ by an average of 18.5% with respect to DPAMS by incrementing the resource usage count only by the operation's (fractional) execution frequency. Note that for some of the benchmarks, such as gsmencode and mesamipmap, both DPAMS and PPAMS result in similar $ResMII$. This is due to the fact that these applications contain a number of dominating loops that consist primarily of a large number of well balanced *if-then-else* statements that provide ample disjoint operations for combining.

Columns 4, 9, and 14 show the achieved II for the three schedulers. For PPAMS column 14 shows the static II which does not account for the delay due to conflicts. PPAMS reduces the II by 12.6% with respect to DPAMS, since it allows more flexible operation combining than DPAMS. As the example in Section 3.2 demonstrated, PPAMS can combine both disjoint and non-disjoint operations from the same or different loop iterations, whereas DPAMS can only combine disjoint operations from the same loop iteration.

This flexibility in combining does result in an additional delay due to conflict ($II_{CflDelay}$). Column 15 shows the compiler estimate of the expected initiation interval ($II_{expected}$) which does account for $II_{CflDelay}$. Based on the formula $II_{expected} = II_{static} + II_{CflDelay}$, a given $II_{expected}$ can be achieved by allowing more sharing with tighter operation scheduling, which decreases II_{static} but increases $II_{CflDelay}$, or by allowing less sharing which increases II_{static} but decreases $II_{CflDelay}$. For example, by comparing columns 14 and 15 we can see that for pegwitdec, PPAMS chooses the first approach and achieves II_{static} of 12.86 with a large $II_{CflDelay}$ of (15.58 - 12.86)=2.72. For unepic, PPAMS chooses the second approach and achieves II_{static} of 10.74 with a small $II_{CflDelay}$ of (10.91-10.74)=0.17.

Columns 16 and 17 show the achieved runtime initiation interval ($II_{dynamic}$), and the relative error between $II_{dynamic}$ and $II_{expected}$ computed as $(II_{dynamic} - II_{expected})/II_{dynamic}$. Note that for the last ('Average') row, we show the average of the absolute values of the relative errors to avoid mutual cancellation of the negative and positive errors. We see that for most of the applications the error is quite small, less than 3%. In unepic and pegwitenc on the 4-wide machine, the errors are 4.67% and 5.66%, respectively. This larger error occurs because some of the predicates that map to the same resource violate the independence assumption that the compiler made during scheduling. These predicates turns out to be positively correlated for semantic reasons within the conditional tests which the compiler does not attempt to analyze; these correlations result in more runtime conflict than was originally estimated by the compiler.

Columns 5, 10 and 18 show the size of the epilogue (in terms of the stage count) of the modulo scheduled loops for the three schedulers. We can see that on average DPAMS increases the baseline schedule epilogue by a factor of 1.9, and PPAMS

more than triples the baseline schedule epilogue. The reason for this increase in the epilogue size is twofold: first, as cmpp latency increases, the SRT schedule length increases, and second, both DPAMS and PPAMS intentionally stretch the schedule by moving operations further away from their producers so as to allow more aggressive combining.

Finally, columns 6, 11 and 19 show the average number of rotating registers [10] required by each of the three schemes. Rotating registers are used to allocate the variables with multiple lifetimes which occur when the variable is live across several loop iterations [44]. The farther away the variable consumer is scheduled from its producer, the more rotating registers the producer will require. Hence, the increase in the required number of rotating registers that we see with DPAMS and PPAMS occurs for the same two reasons that cause the increase in the size of the epilogue for these two schemes. These increases in register requirements are quite reasonable by today's standards. More experimental studies and insights into rotating register requirements for BAMS, DPAMS and PPAMS are provided in Section 4.4.1.

Table 3.4(b) shows the corresponding data for $P_{base}(6)$, $P_{dpas}(6, 3)$ and $P_{ppas}(6,3,1)$ machines for which conclusions similar to those for the three 4-wide machines can be drawn. However, as mentioned above, the 6-wide predicate-aware machine with 4 integer units has more resources and therefore achieves less benefit from operation sharing than the equivalent 4-wide machine with 2 integer units. DPAMS and PPAMS consequently achieve less speedup over the baseline for the 6-wide machines.

3.5.2.5 Overall Speedup

The full application results presented in this subsection assume that the ppa-improvable acyclic and cyclic regions are scheduled with 'first-fit' PPALS and PPAMS,

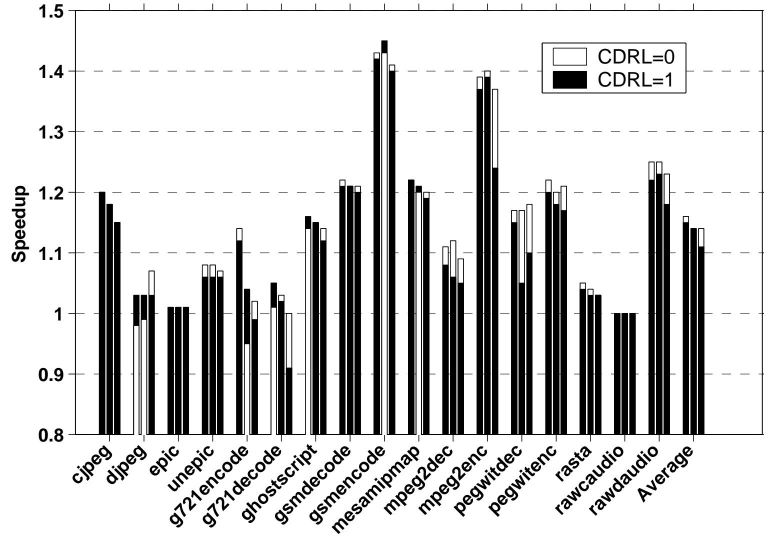
respectively. When deterministic scheduling is performed on dpa-improvable regions (for comparison with probabilistic), 'first-fit' DPALS and DPAMS are used. As we said earlier, the rest of the regions are scheduled using BALS and BAMS baseline scheduling algorithms.

Figure 3.25(a) shows the overall speedup achieved by 4-wide PPAS processor over the corresponding 4-wide baseline processor for each application. The left, middle and right bars correspond to cmpp latency 1, 2, and 3 cycles. The white and black bars for CDRL=0 and 1, respectively, are overlaid as in Figure 3.23. Figure 3.25(b) shows the corresponding data for the 6-wide processor.

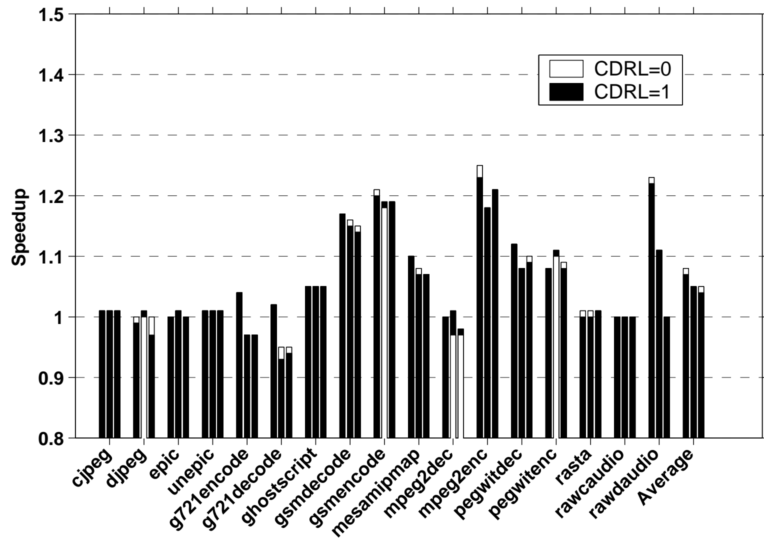
The average speedups achieved over all applications are 16%(15%), 14%(14%) and 14%(11%) for 4-wide machines with CDRL of 0(1) and cmpp latencies 1,2 and 3, respectively. The corresponding speedups for the 6-wide machines are 8%(7%), 5%(5%) and 5%(4%). Some applications have much higher speedup; for all three cmpp latencies; gsmdecode and mpeg2enc enjoy more than 35% speedup for the 4-wide machines and more than 18% speedup for the 6-wide machines. Each of these benchmarks spends more than 90% of its execution time in the modulo scheduled loops, which achieve very high speedup with PPAMS (see Figure 3.23). For all the applications on a 6-wide machine, most of the overall performance improvement comes from PPAMS; PPALS achieves very little speedup.

Figure 3.26(a) shows the overall speedup over a 4-wide baseline processor achieved by a 4-wide deterministic predicate-aware processor (left bar), as well as two 4-wide probabilistic predicate-aware processors with CDRL=0 (middle bar) and CDRL=1 (right bar), respectively. All three predicate-aware machines assume cmpp latency of 3 cycles. Figure 3.26(b) shows the corresponding data for the 6-wide processor.

As expected, in most cases, PPAS outperforms DPAS, resulting in an average

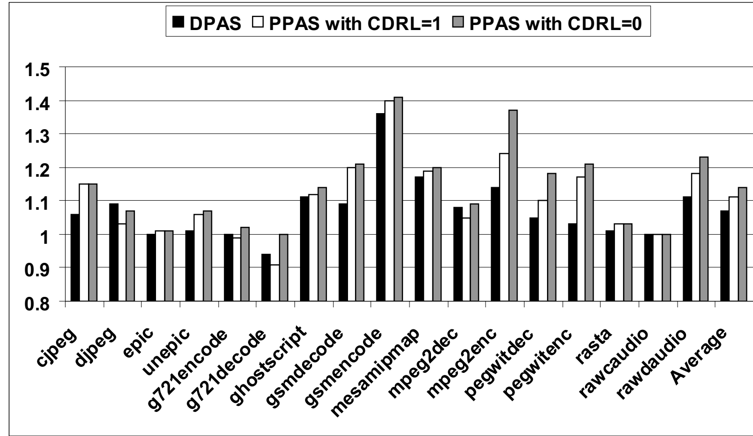


(a) Speedup of $P_{ppas}(4,1/2/3,0/1)$ over $P_{base}(4)$

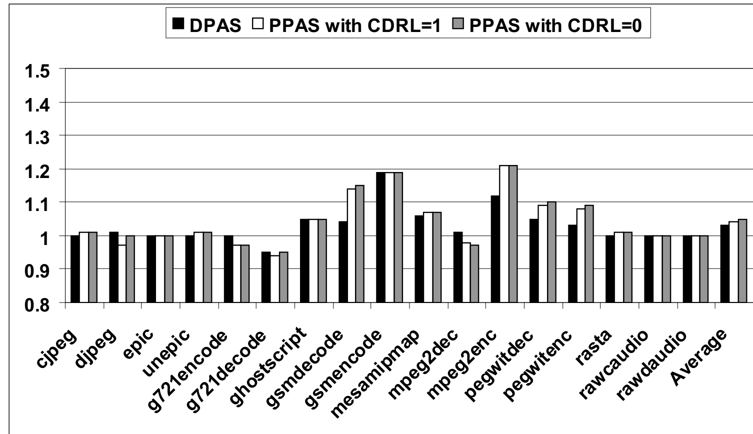


(b) Speedup of $P_{ppas}(6,1/2/3,0/1)$ over $P_{base}(6)$

Figure 3.25: Overall speedup of PPAS over baseline for cmpplat=1, 2 and 3



(a) Speedup of $P_{dpas}(4,3)$, $P_{ppas}(4,3,1)$, $P_{ppas}(4,3,0)$ over $P_{base}(4)$



(b) Speedup of $P_{dpas}(6,3)$, $P_{ppas}(6,3,1)$, $P_{ppas}(6,3,0)$ over $P_{base}(6)$

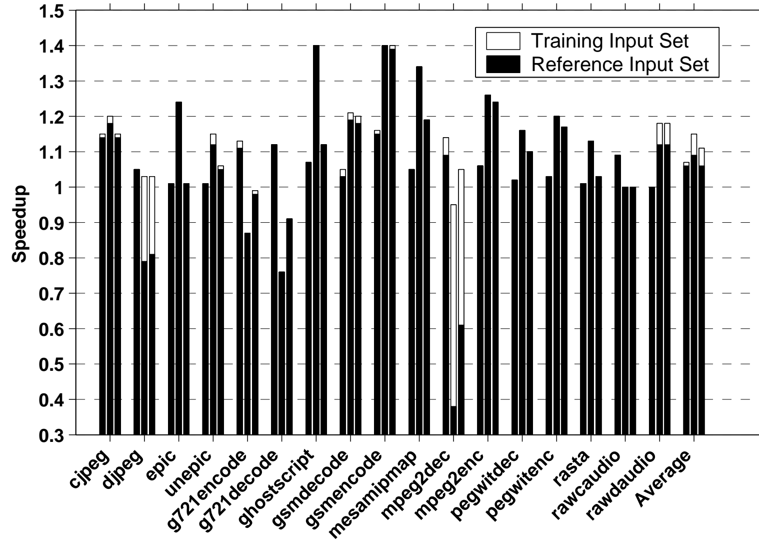
Figure 3.26: Overall speedups of DPAS and PPAS over baseline (cmpplat = 3)

6.5% gain for the 4-wide machine with CDRL=0, and 2% gain for the 6-wide machine. When CDRL is increased to 1, the overall performance gain of PPAS over DPAS drops to 4% for the 4-wide machine and 1% for the 6-wide machine. We have seen that larger CDRL has negligible impact on acyclic performance. We have also seen that most of the cyclic regions are more sensitive to the value of CDRL; the expected II, and the actual PPAMS speedup over BAMS decreases as CDRL increases (see Figure 3.23(a)). Thus the overall decreas in speedup for CDRL=1 is due to these cyclic regions.

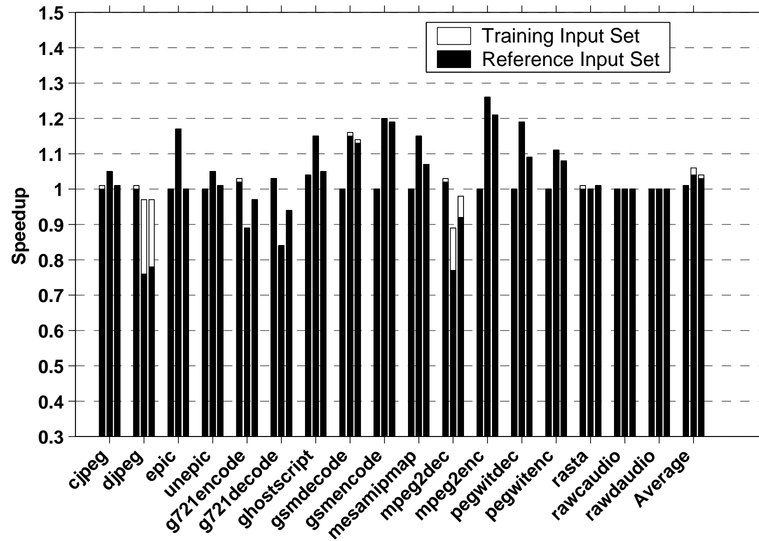
For some of the benchmarks, DPAS achieves higher performance than PPAS. For example, for mpeg2dec, on a 4-wide machine, DPAS achieves 3% more speedup than PPAS on a 4-wide machine with CDRL=1. This happens because the mpeg2dec application is dominated by loops with a short trip count. In this case, as Table 3.4 shows, PPAMS achieves a smaller II, but results in a longer epilogue than DPAMS which, due to the low trip of the loops, results in the better performance of DPAMS.

The results presented so far have been collected using the training input set for each benchmark. Figure 3.27(a) compares the speedups achieved by a 4-wide PPAS processor over the corresponding 4-wide baseline processor for both the training and reference input sets. A cmpp latency of 3 cycles and CDRL=1 are assumed. The left, middle and right bars correspond, respectively, to the speedup due to acyclic regions only (scheduled with 'first-fit' PPALS), cyclic regions only (scheduled with PPAMS), and the overall application. Each bar is composed of the overlaid white and black sub-bars for training and reference input sets, respectively. Figure 3.27(b) shows the corresponding data for the 6-wide processor.

We see that the PPAS processor achieves better speedup on the training input, which was used for gathering profiling information for the scheduler, than on the reference input. However, as the left bar shows, 'first-fit' PPALS loses very little



(a) Speedup of $P_{ppas}(4,3,1)$ over $P_{base}(4)$



(b) Speedup of $P_{ppas}(6,3,1)$ over $P_{base}(6)$

Figure 3.27: Speedup of PPAS over baseline that correspond to training and reference input sets (left bar corresponds to acyclic regions speedup, middle bar to cyclic, and right bar to overall)

speedup when run on the training input. This is due to the fact that, as explained in Section 3.5.2.2, acyclic regions scheduled with PPALS have a very small delay due to conflicts. Using a different input set, does not increase this delay significantly. However, we do observe some speedup degradation for 5 applications on a 4-wide machine. Out of these 5 applications, mpeg2dec exhibits the largest loss in speedup, down to 1.09 from 1.14. 5%. On average, however, in comparison to the training input, PPALS, when run on the reference input, exibits reduction in speedup from 1.07 to 1.06 on the 4-wide and no loss in speedup on the 6-wide machine.

PPAMS, on the other hand, loses more of its speedup than PPALS does when run on the reference input, rather than the training input. This is true for both machine models. However the most substantial speedup loss is due to two benchmarks: djpeg and mpeg2dec. For example, for djpeg when PPAMS is run on the reference input, rather than the training input, its speedup is reduced from 1.3 to 0.79 on the 4-wide machine and from 0.97 to 0.76 on the 6-wide machine. This significant loss of speedup for these two benchmarks is due to the fact that the reference input results in a large number of positively correlated predicates which were not correlated in the training input. These correlations result in more runtime conflict for the reference input in these two benchmarks. For the other 15 benchmarks, PPAMS loses very little of its speedup relative to the training input, when run on the reference input.

Over the entire application suite, PPAS's speedup, when run on the reference rather than the training input, is reduced down to 1.06 from 1.11 on the 4-wide machine and to 1.03 from 1.04 on the 6-wide machine. However, if djpeg and mpeg2dec were omitted when computing the average results, then PPAS would only lose speedup down to 1.10 from 1.11 on the 4-wide machine, and would maintain its speedup on a 6-wide machine when the reference input is used.

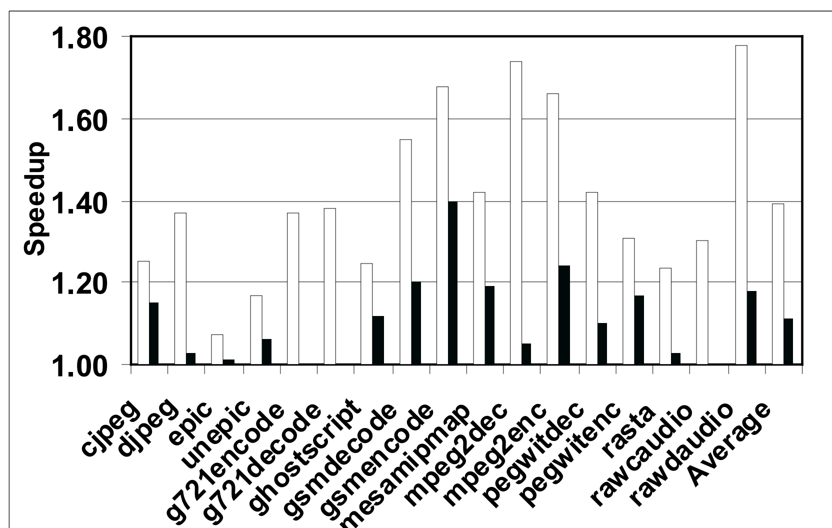


Figure 3.28: Overall speedups of the 6-wide baseline (left bar) and the 4-wide PPAS with cmpplat=3 and CDRL=1 (right bar), relative to the 4-wide baseline

Finally, the Figure 3.28 compares the overall speedup over a 4-wide baseline processor that is achieved by a 6-wide baseline processor (left bars) and by a 4-wide probabilistic predicate-aware processor with a cmpp latency of 3 cycles and CDRL = 1 (right bars).

We see that the wider baseline machine with more resources and no resource sharing achieves an average of 25% higher performance than the narrower predicate-aware machine resource with sharing capability, but fewer resources. This result was expected for two reasons. First, a wider processor generally achieves some improvement in performance for every type of code region (typically less for acyclic regions which are latency bound, and more for cyclic regions which are resource, rather than latency bound), in contrast to a predicate-aware processor that can only improve the performance of a ppa-improvable code region. Second, within a given ppa-improvable region, there are operations for which there exist no other operations with which they can share a resource on the narrower PPAS processor. Such operations might still be

scheduled in parallel (resulting in a shorter schedule) on the wider baseline machine, but might not be scheduled in parallel on the narrower baseline machine due to its tighter resource constraints.

Note that it is possible for a narrower PPAS processor to outperform a baseline processor that is twice as wide. This will happen if sufficiently often 3 or more operations get assigned to share the same resource on a PPAS processor. For example, 3 floating-point operations that share a single floating-point unit on a 4-wide PPAS processor will execute in 1 cycle; however, it will take 2 cycles to execute these 3 floating-point operations on a 6-wide baseline processor with 2 floating-point units. However, none of the applications in our experiments achieve such an overall speedup, since the number of operations that get combined to share the same resource in the same cycle very rarely exceeds 2 operations.

3.5.2.6 Pipeline Utilization

A key hardware feature for supporting PPAS (and DPAS) is the ability to read predicates early (during the predicate-read and dispatch stage) and discard operations guarded under False. Discarding operations early in the pipeline prevents them from superfluously reserving the resources that are used later in the pipeline.

Table 3.5 quantifies the utilization of individual pipeline stage resources per predicated operation. For a given benchmark (row) and pipeline resource (column), the corresponding table entry shows the percentage of dynamic operations in the benchmark that utilize this pipeline resource. A given pipeline resource is utilized by a given operation only if the operation is not nullified at some earlier stage due to a False predicate.

Table 3.5(a) shows the resource requirements for the 4-wide baseline processor.

Benchmark	FETCH	DISPATCH	DECODE	REGISTER READ	EXECUTE INT	EXECUTE FP	EXECUTE MEMORY	EXECUTE BRANCH	WRITE BACK
cjpeg	100	100	100	100	68.1	0	26.29	5.61	33.52
djpeg	100	100	100	100	71.98	0	25.64	2.38	82.75
epic	100	100	100	100	63.21	14.44	11.89	10.46	96.19
unepic	100	100	100	100	58.26	6.97	28.82	5.96	62.01
g721encode	100	100	100	100	79.71	0	17.31	2.98	66.34
g721decode	100	100	100	100	80.53	0	16.77	2.69	70.95
ghostscript	100	100	100	100	64.59	0.7	26.48	8.22	64.53
gsmdecode	100	100	100	100	87.68	0	9.94	2.37	78.09
gsmencode	100	100	100	100	84.87	0	14.37	0.76	50.14
mesamipmap	100	100	100	100	43.33	15.21	35.97	5.5	81.96
mpeg2dec	100	100	100	100	66.23	0	27.24	6.53	68.25
mpeg2enc	100	100	100	100	90.2	0.43	6.14	3.23	64.16
pegwitdec	100	100	100	100	81.65	0	15.36	2.99	76.91
pegwitenc	100	100	100	100	77.29	0	19.91	2.8	52.92
rasta	100	100	100	100	58.84	9.93	21.62	9.6	89.56
rawcaudio	100	100	100	100	90.53	0	7.56	1.9	73.13
rawdaudio	100	100	100	100	88.59	0	9.11	2.3	74.88
Average	**100**	**100**	**100**	**100**	**73.86**	**2.8**	**18.85**	**4.49**	**69.78**

(a) Resource requirements for the baseline processor, $P_{base}(4)$

Benchmark	FETCH	DECODE	PRED. READ & DISPATCH	REGISTER READ	EXECUTE INT	EXECUTE FP	EXECUTE MEMORY	EXECUTE BRANCH	WRITE BACK
cjpeg	100	100	100	53.93	39.81	0	11.71	2.41	33.52
djpeg	100	100	100	83.12	59.01	0	21.78	2.33	82.75
epic	100	100	100	98.39	62.41	14.18	11.49	10.31	96.19
unepic	100	100	100	68.78	43.01	2.55	17.68	5.54	62.01
g721encode	100	100	100	74.46	58.07	0	13.63	2.75	66.34
g721decode	100	100	100	77.68	61.55	0	13.56	2.56	70.95
ghostscript	100	100	100	77.98	51.25	0.4	20.45	5.87	64.53
gsmdecode	100	100	100	78.66	69.5	0	6.9	2.26	78.09
gsmencode	100	100	100	50.56	43.46	0	6.41	0.69	50.14
mesamipmap	100	100	100	83.11	34.77	13.51	29.35	5.47	81.96
mpeg2dec	100	100	100	77.45	49.76	0	23.1	4.59	68.25
mpeg2enc	100	100	100	65.21	56.2	0.43	5.84	2.75	64.16
pegwitdec	100	100	100	78.26	61.28	0	14.21	2.77	76.91
pegwitenc	100	100	100	53.87	40.59	0	10.59	2.7	52.92
rasta	100	100	100	93.16	55.04	9.45	19.75	8.92	89.56
rawcaudio	100	100	100	73.15	64.64	0	6.62	1.89	73.13
rawdaudio	100	100	100	74.88	64.61	0	7.97	2.3	74.88
Average	**100**	**100**	**100**	**74.27**	**53.82**	**2.38**	**14.18**	**3.89**	**69.78**

(b) Resource requirements for the probabilistic predicate-aware processor, $P_{ppas}(4,3,1)$

Table 3.5: Pipeline utilization statistics per predicated operation for baseline and probabilistic predicate-aware processors (using 'first-fit' PPALS and PPAMS)

Since in the baseline processor the operations guarded under False predicates are only nullified in the execution stage of the pipeline, all pipeline stages up to and including execution stage are required by all operations regardless of the value of their predicates. Note that the execution stage is broken down into four individual function unit resources: integer, floating-point, memory and branch; the sum of the requirements over the four function units is equal to 100%. Recall that Figure 3.3 shows that 30% of all dynamic operations have their predicates evaluate to False. This is consistent with Table 3.5(a) which shows that the write back stage of the pipeline is effectively required by the remaining 70% of the operations.

Table 3.5(b) shows resource requirements for the 4-wide probabilistic predicate-aware processor with a cmpp latency of 3 cycles and CDRL=1, using 'first-fit' PPALS and PPAMS scheduling. Operations whose predicates are read and found to be False in the predicate read and dispatch pipeline stage will be nullified in this stage (see Section 3.3). However, recall that the first three stages (fetch, decode, and predicate read and dispatch) are must-use stages that are required by all operations. The register read and execute stages (that come after predicate read and dispatch) are required by only 74% of the operations. The remaining 26% of the operations therefore must have been nullified during the predicate read and dispatch pipeline stage. Note that not all 30% of the operations guarded under False predicates get nullified in the predicate read and dispatch stage; the remaining 4% of the operations guarded under False are not nullified until the execute stage, as in the baseline machine. Consequently both machines show that 69.8% of the operations require the write-back stage.

The 4% of all operations that have Flase predicates, but do not get nullified during the predicate read and dispatch stage could not be nullified early because

their predicates were not available for early read during this stage. This occurs when an operation is scheduled less than *extendedlatency* cycles after its cmpp (less than 3 cycles after in our case). Recall that if operation's predicate is not available, the operation must reserve all of its resources (up to and including the execute stage of the pipeline) unconditionally, which is what happens for this 4% of all operations. Overall, in addition to preventing 30% of all dynamic operations from superfluously reserving the write back stage, PPAS is able to effectively prevent 26% of all dynamic operations from superfluously reserving the register read and execute stages of the pipeline.

3.6 Related Work

In this section we discuss some hardware and compiler approaches to predication that are most relevant to the predicate-aware scheduling (PAS) described in the previous two chapters.

3.6.1 Hardware Support for Predication

Predication is a widely used technique. A number of past and present VLIW/EPIC machines, both embedded and general-purpose, have hardware support for predication.

Cydra 5 has two modes of execution. In the first mode it can read predicates in the decode stage and nullify the operations guarded under False. In the second mode Cydra 5 performs *eager execution* of nearly all operations (except branches and memory writes) in which an operation is executed prior to resolving its predicate [10, 46]. When predicates are read in the later execution stage of the pipeline, Cydra

5 eliminates all side effects of the operation so as to allow simply disregarding the operation if it should never have executed, i.e. no recovery action is required despite its speculative execution.

To the best of our knowledge Cydra 5, is the only machine which is similar to our predicate-aware architecture in that it allows predicates to be read and operations guarded under False to be discarded early. Although existing machines do vary as to which stage in the pipeline reads predicate values and when operations are nullified, no architecture that we are aware of allows disjoint operations to reserve the same resource in the same clock cycle, as in our predicate-aware technique.

The Texas Instruments TMS320C6000 [56] accesses its predicate register file (PRF) early in the execution stage, before accessing the register file (GPR). Ideally, this could provide an opportunity to select among several disjoint operations, and let only those operations that are guarded under a True predicate proceed with execution. Nevertheless, this is not done. A predicate-aware TI 'C6x would require disjoint operations to be scheduled at least two cycles later than their corresponding cmpp operations (vs. three, as assumed in our processor model, see Section 2.3). This reduction is possible since the TI 'C6x combines register (and predicates) read and execute in the same pipeline stage, which we felt would jeopardize clock speed in our architecture.

Intel's Itanium [26, 48] processor accesses its PRF in the execution stage of the pipeline. Itanium also accesses the PRF in the register read pipeline stage in parallel with general registers. This simultaneous access is done in order to nullify an operation that is in the PRF stage and is waiting on the result from a long latency operation that is currently being executed: if the read predicate is False, the consuming operation is squashed and execution can continue. However, the impact of this particular optimization on overall performance is small [48]. Since the PRF

is accessed in parallel with the general registers in the register read stage, distinct GPR register ports must still be reserved by all simultaneously scheduled predicated operations in the Itanium processor, regardless of whether they will be nullified.

3.6.2 Compiler Support for Predication

A general dicussion of compiler support and the prior techniques on which we depend is presented in Section 2.4.2. We know of no predicate-aware techniques for compiling acyclic regions. This section therefore addresses only compiler support for cyclic regions. Predication is used before modulo scheduling to convert loop bodies with internal control flow into a single basic block which can later be modulo scheduled. Predicate-aware scheduling improves the performance of modulo scheduled loops with internal control flow by reducing the additional resource requirements introduced by the predicated operations. A number of other compiler approaches have been proposed to achieve efficient software pipelined schedules of loops with control flow. These approaches can be classified into two broad categories: fixed II approaches and variable II approaches. We next discuss each of these approaches and compare them with predicate-aware modulo scheduling (PAMS).

Fixed-II Approaches

Hierarchical reduction is a technique that converts code with control flow into straight-line code by collapsing each conditional construct into a reduced pseudo operation [31, 32]. It is a hierarchical technique since nested conditional constructs are reduced by collapsing from the innermost to outermost. A reduced operation is formed by first list-scheduling both paths of the conditional construct. As in DPAMS, the resource usage of a reduced operation is determined by the union, rather than

the sum, of the resource usages of both paths after list scheduling. The dependencies between operations within the conditional construct and those outside are replaced by dependencies between the reduced operations and the outside operations. After hierarchical reduction, the reduced operations can be modulo scheduled along with the other operations of the loops. Hierarchical reduction assumes no special hardware support. Thus, the conditional constructs are regenerated after modulo scheduling, and all operations that have been scheduled with the reduced operation are duplicated to both paths of the conditional construct. While hierarchical reduction allows a loop with conditional constructs to be modulo scheduled and in general, as in PAMS, reduces the resource requirements of the modulo scheduled loop, it places some artificial scheduling constraints on the loop by list scheduling the operations of the conditional construct. The list schedule causes the reduced operations to have a complex resource usage which is more likely to conflict with already scheduled operations during modulo scheduling. In addition, if a reduced operation spans more than one *II*, it may actually conflict with itself in which case no schedule for that *II* can be found. Finally, regenerating the conditional construct and duplicating operations on all paths can result in significant code growth that can be exponential in the worst case. Increase in the code size may lead to more cache misses and degraded performance. In addition, reverse if-conversion reintroduces the branches into the loop kernel that can potentially result in branch mispredictions and also lower performance.

As discussed at length in this and previous chapters, conventional modulo scheduling with if-conversion [41] uses predicates to convert the loop body into a single basic block, which is then scheduled using a traditional modulo scheduling algorithm. The main advantage of modulo scheduling with if-conversion over hierarchical reduction is that operation placement is more flexible since these operations are not list scheduled

prior to modulo scheduling. However, its main drawback is that by forcing operations to reserve resources unconditionally, it sums the resource requirements over all paths, leading to resource pressure problems and schedule length increase as described in the previous two sections. PAMS reduces the resource constraints in predicated code by allowing several predicated operations to conditionally reserve and share the same resource in the same cycle.

Enhanced Modulo Scheduling [59] initially modulo schedules predicated code in which disjoint operations are allowed to share resources. This is similar to DPAMS, as well as to hierarchical reduction, in that this sharing enables the resource requirements of operations from disjoint paths to be the union rather than the sum of the requirements of each individual operation. However, this scheme assumes no hardware support for predication. Therefore in the next step, the control flow is regenerated from the intermediate schedule to obtain the final pipelined schedule. The intermediate schedule is then discarded; the idea of executing the shared-resource schedule with predicate-aware hardware is not explored in that work. Reverse if-conversion, as in hierarchical reduction, can cause performance degradation due to increased code size and branch mispredictions.

As opposed to hierarchical reduction and enhanced modulo scheduling, predicate-aware modulo scheduling takes advantage of hardware support for predication and therefore avoids performance degradation due to these two factors. On the other hand, as in hierarchical reduction and enhanced modulo scheduling, predicate-aware modulo scheduling reduces the resource constraints by allowing resource sharing among several predicated operations. Note that DPAMS, as in the other two techniques, only allows provably disjoint operations to share a resource. PPAMS further reduces resource requirements, compared to DPAMS, hierarchical reduction and enhanced

modulo scheduling, by allowing arbitrary predicated operations to share the same resource, but does so only when the expected resource conflict penalty is acceptable.

3.6.2.1 Variable-II Approaches

The APP (All Path Pipelining) approach [54] pipelines each path separately using a software pipelining technique for straightline loops and then merges together the pipelined kernels of those paths. Since each of these kernels is scheduled based on the constraints of a single path only, a variable II can result. Since all path pipelining pipelines each path separately, the pipeline schedule of each path is limited by the resource requirements only of this path only. Therefore, the best case performance of all path pipelining is achieved when only the control path with the least resource requirements executes repeatedly and dominates the execution of the entire loop.

In contrast, the best case performance of DPAMS is bound by the resource requirements of the busiest control path (and the instruction width of the machine) regardless of which path executes. In all path pipelining, although each path can be pipelined tightly, the transitions between paths, which are likely to occur during actual execution on a realistic input set, are loosely pipelined, leading to poor performance if these transitions are frequent. The overall II can vary depending on the execution frequency of each path and the frequency of transitions among the paths. In PPAMS the overall II also varies with conflict delays that depend on the activation frequencies of the predicates; however, PPAMS controls this variability by including the expected conflict penalty in its objective function while scheduling.

Modulo Scheduling with Multiple Initiation Intervals (MSMII) [60] schedules if-converted code so that control paths with higher execution frequencies have shorter IIs than paths with lower frequencies. MSMII creates a partitioned loop kernel with

several predicated loop-back branches (potentially one per iteration path). Path priorities are assigned based upon dynamic profiling of the loop, and operations are then placed in the kernel so that the highest priority paths use an initiation interval close to the minimum II for that path, and the lower priority paths are then scheduled in turn as well as possible using only the resource slots still available to them. Predicated operations are used to execute the correct operations and the correct loop-back branch based upon which path is actually executed dynamically.

There are two main drawbacks of the MSMII approach. First, both hardware support and dynamic profile information (as in the case of PPAMS) about the program's execution are required for the algorithm to be effective. Second, the overall II may increase because the operation scheduling is now more restricted than in single II (traditional) modulo scheduling: the particular operation can only be scheduled in the partition that corresponds to its control path, or any of the earlier partitions, but not in any of the later partitions.

For MSII, the best performance is achieved when the highest priority path also happens to be the path with the least resource requirements, and this path executes repeatedly and dominates during the execution of the entire loop. In contrast, DPAMS achieves the same performance regardless of which path executes. Furthermore, the best case performance of DPAMS is bound only by the resource requirements of the busiest control path and the instruction width of the machine.

3.7 Summary

In this chapter, we have proposed and evaluated a generalization to the deterministic predicate-aware scheduling (DPAS) technique described in Chapter 2, called

probabilistic predicate-aware scheduling (PPAS). PPAS can achieve better schedules on predicated code regions than DPAS by further reducing wasted resources in VLIW/EPIC processors with predicated execution. In contrast to DPAS, which can only combine disjoint predicated operations to share the same resource in the same runtime cycle, PPAS enables the compiler to combine two or more arbitrary predicated operations to share the same processor resource. Unlike DPAS, where in any cycle the predicate of at most one of several combined disjoint operations is True, assigning several arbitrary predicated operations to the same resource in PPAS will result in runtime resource conflicts, whenever two or more predicates of a set of combined operations evaluate to True in same cycle. PPAS performs its assignment in a probabilistic manner using a combination of predicate profile information and predicate analysis aimed at maximizing the benefits of combining in view of the expected degree of conflict.

Both acyclic and cyclic probabilistic predicate-aware modulo schedulers have been implemented and evaluated on a suite of Mediabench applications. For a cmpp latency of 3 cycles and CDRL=0, when 'first-fit' PPALS is used on acyclic regions and PPAMS is used on cyclic regions, probabilistic predicate-aware scheduling achieves a 14% speedup for a 4-wide predicate-aware machine, and 5% for a 6-wide predicate-aware machine. For CDRL=1, the corresponding speedups are reduced to 11% and 4%.

Finally, note that in our experiments 30% of all dynamic operations had False predicates. The baseline machine reserves execution resources unconditionally for all of them. The 'extend-all' versions of PPALS and PPAMS eliminate these dynamic operations prior to the register access and execute stages. They achieve this early elimination by extending the cmpp latency for almost every predicated operation, thus enabling the probabilistic predicate-aware machine to read an operation's predicate

early during the predicate read and dispatch pipeline stage and immediately discard operations guarded under False; however, by extending almost all cmpp latencies, 'extend-all' PPALS can lengthen critical paths by postponing the schedule slots of all predicated operations.

By employing 'first-fit' PPALS, each operation is scheduled into an early schedule slot whenever possible to avoid unnecessarily lengthening the critical paths of acyclic regions, but if it ends up being scheduled sufficiently later than its cmpp operation, its predicate is read early and if it is False, the operation is nullified prior to the register read and execute stages. The 30% of all dynamic operations that have False predicates break down further into 10% belonging to acyclic regions, and the other 20% belonging to cyclic regions. By using 'first-fit', rather then 'extend-all' PPALS, for list scheduling the acyclic regions, PPAS allows 26% of all operations with False predicates (6% from acyclic regions scheduled with 'first-fit' PPALS + 20% from cyclic regions scheduled with PPAMS) to in fact be scheduled late enough to be nullified before execution. Only 4% of all operations (which belong to acyclic regions scheduled with 'first-fit' PPALS) are nullified after execution.

CHAPTER 4

A STORAGE MECHANISM FOR EFFICIENT SOFTWARE PIPELINING

4.1 Introduction

In the last two chapters we described predicate-aware scheduling which improves performance of predicated code on machines with limited resources, such as function units and register ports, by allowing several predicated operations to share the same resource in the same cycle. In this chapter we describe Register Queues - a novel technique to improve the performance of software pipelined loops on a machine with a limited number of architected registers, by allowing several simultaneously live values to share same architected register.

Many code transformations performed in optimizing compilers trade off an increase in register pressure for some desirable effect (lower cycle count, larger basic block size, etc.). Perhaps this is most clearly shown in the **software pipelining** [7, 8, 22, 32, 42, 43] of a loop, which interleaves instructions from multiple iterations of the original loop into a restructured loop kernel. This restructuring improves pipeline throughput by enabling more instructions to be scheduled between a value

being defined by a high latency operation (e.g., multiplication or memory load) and its subsequent use. Software pipelining thus decreases the time between successive loop iterations by spreading the *def-use* chains in time. This rescheduling increases the number of simultaneously live instances of loop variables from different iterations of the original loop body. To accommodate these variables, each of the simultaneously live instances needs its own register. Furthermore, each instance must be uniquely identified to permit matching a use of a variable to the correct definition; there must be some mechanism to differentiate among live instances of a variable defined in previous iterations and the definition in the current iteration.

Two common schemes that support this form of register naming are **modulo variable expansion** (MVE) [32], and the *rotating register* file (RR) [44, 46]. MVE is a software-only approach which gives each simultaneously live variable instance its own name, unrolling the loop body as necessary to insure that any later uses can directly specify the correct instance (more on this later). MVE both increases the architected register requirements and expands the loop body to accommodate the register naming constraints of the software pipelined loops. In contrast, RR is a hardware-managed register renaming scheme that eliminates the code expansion problem by dynamically renaming the register specifier for each instance of a loop variable. This renaming is achieved by adding an additional level of indirection to the register specification to incorporate the loop iteration count; this makes it possible to explicitly access a variable instance that was defined n iterations ago. However, since the rotating register file contains architected registers, RR still requires a large number of architected registers to permit generating efficient schedules.

Each of these techniques satisfy the register requirements for a variable by assigning the instances defined in successive loop iterations to distinct architected registers

in some round-robin fashion. The number of architected registers required for a software pipelined (SP) loop therefore grows linearly with increased functional unit latencies [37], i.e., a longer latency operation within the loop will lead to a greater number of interleaved instances of the original loop in the SP loop kernel, and therefore more live instances of the loop variables. Therefore, a shortage of architected registers either limits the number of interleaved loop iterations or introduces spill code, each of which degrades performance.

As pipelines get deeper and wider, longer latencies (in particular memory latencies) and higher concurrency used to hide these latencies lead to more overlap between lifetimes, which leads to increased register pressure. DPAMS and PPAMS software pipelining scheduling algorithms exacerbate this problem by intentionally moving variable definitions farther away from their uses to allow more operations to share the resource (and therefore achieve higher concurrency), which also leads to more overlap between lifetimes and increased register pressure.

Therefore, efficient highly concurrent software pipeline schedules that account for realistic memory latencies are difficult, and often impossible, to achieve with MVE or RR for architectures with small or moderate sized register files. One solution is to dramatically increase the number of architected registers available. This may be achieved when a new instruction set architecture is proposed (e.g., the IA-64 or EPIC instruction set [25] which supports 128 integer and 128 floating point registers). In this work, we propose an alternative register addressing mechanism which can be integrated into existing instruction set architectures with minimal modification while alleviating the register pressure and register naming issues that are inherent in SP.

In this work we demonstrate that by introducing **Register Queues** (RQs) and *rq-connect* instructions, the architected register space is no longer a limiting factor

in achieving efficient software pipelined loop schedules. The design of these register queues is derived from the interprocessor queues that support asynchronous communication in decoupled architectures[53, 61]. Software pipelining using queues has also been studied in VLIW/EPIC architectures [15, 16, 34] and decoupled processors [58], but not in general purpose superscalar designs. In particular, [16] proposes the use of a queue register file (QRF) to support the execution of software pipelined loops in VLIW machines. This extends prior work on VLIW processors [28] by making the queues architecturally visible; earlier work scheduled values in pipeline registers, also organized as queues, for a specific VLIW implementation. By making the queues architecturally visible, portability between VLIW implementations is provided. Our work proposes the register queue mechanism for conventional superscalar processors, as well as software/hardware techniques to ease the integration of RQs into existing instruction set architectures and machine implementations with out-of-order pipelines.

In the context of this research, register queues can most clearly be viewed as a combination of the rotating register file ([6, 46]) and register connection [30] concepts. This combination enables a decoupling of the total register space for SP into a small set of architected registers and a large set of physical registers that are organized as circular buffers and accessed indirectly. By using register queues, the architected register requirements of a software pipelined loop are independent of the latencies of the scheduled instructions. Integrating RQs into an existing architecture is also straightforward. We will show that the inclusion of a single new instruction, *rq-connect*, is all that is necessary to add RQs to any instruction set architecture while maintaining full backward compatibility. Experimental results show that the RQs method significantly reduces both the architected register and the code size requirements of software pipelined loops.

	time	iteration 1	iteration 2	iteration 3
Prologue	1	iadd r1, r1 #4		
	2	fload f2, 0(r1)		
	3		iadd r1, r1 #4	
	4		fload f2, 0(r1)	
Kernel	5	fadd f6, f6, f2		iadd r1, r1 #4
	6			fload f2, 0(r1)
Epilogue	7		fadd f6, f6, f2	
	8			
	9			fadd f6, f6, f2
	10			

Figure 4.1: Software pipeline example. This sample program adds elements of a floating-point array and stores the sum in a scalar. Shown are multiple iterations of the loop with an initiation interval of 2 cycles ($II = 2$).

The remainder of this chapter is organized as follows: Section 4.2 provides a brief introduction to software pipelining and describes previous work in both software pipelining and register file organization. Section 4.3 describes the concept of register queues and the architectural modifications required to support this approach. Section 4.4 presents our experimental evidence of the performance advantage of register queues over existing schemes including the application of register queues to both deterministic and probabilistic predicate-aware scheduling techniques. We offer conclusions in Section 4.5.

4.2 Prior Work

As a simple example of SP, consider Figure 4.1 which shows the intermediate level code of one iteration of a loop that accumulates the elements of a floating point array into a scalar (loop control instructions have been eliminated for clarity). For this

example, we assume a two-wide issue machine with a latency of 3 for the load operation, 2 for floating-point addition, and 1 for integer addition. The scheduling process is governed by two constraints: **resource constraints** determined by the resource usage requirements of the computation, and **precedence constraints** derived from the latency calculations around elementary circuits when they exist in the dependence graph for the loop body due to a loop-carried dependence. With an issue width of 2 and a loop body consisting of 3 instructions, we do not have the resources (issue width in this case) to start a new loop iteration more often than once every 2 cycles. The interval between starting new instances of a loop is termed the initiation interval or II of the loop (in this case we must make $II \geq 2$). Furthermore this loop contains a loop-carried dependence between successive instances of the floating-point add which has a latency of 2; for this reason as well, we must make $II \geq 2$.

Figure 4.1 shows a software pipelined code sequence, for $II = 2$. Instructions at time steps 1-4 form the prologue of the software pipelined loop, time steps 5-6 are the steady-state segment (or kernel of the loop), and 7-10 form the loop epilogue. The prologue and epilogue are executed once and the steady-state kernel is executed repeatedly (n-2 times to execute n iterations of the original loop).

The example in Figure 4.1 demonstrates a problem with register names in software pipelined schedules. The *fload* instruction from iteration $i + 1$ starts executing before the *fadd* instruction from iteration i uses the value created by the *fload* of iteration i. This creates two simultaneously live instances of the register *f2*. One way to overcome the register overwrite problem (WAR hazard) is to increase the initiation interval to 4 to allow the *fadd* operation from the ith iteration to complete before the *fload* of iteration i+1 is issued. However, this would halve the loop throughput to one iteration every four cycles. We now describe several alternative solutions that have

been proposed to address this register naming problem.

Modulo variable expansion (MVE) [32] is a compiler transformation (requiring no hardware support) which schedules a software pipelined loop. The purpose of MVE is to manage the naming problem by making sure that instances of a variable whose lifetimes overlap are allocated to distinct architected registers. So, if the lifetime of a value spans three iterations of the pipelined loop and its lifetime overlaps the instances of that variable in the next two iterations, three registers will be allocated in the loop kernel for that variable. In general, at least $\lceil \frac{l}{II} \rceil$ registers are required for each variable in the loop, where l is the variable's lifetime in cycles. Since successive definitions of a variable must be assigned to different registers (since they are simultaneously live), the kernel has to be unrolled, thus lengthening the steady state loop body. The kernel of the loop must therefore be expanded by a factor of at least $\lceil \frac{l}{II} \rceil$ to account for the different register specifiers required for successive definitions of the variable. The actual degree of unrolling is, however, determined by the requirements for all the variables, given the minimum number of registers required for each variable.

When expanding the loop kernel, two techniques are examined. One technique (which we will call MVE1) minimizes register pressure at the expense of increasing the degree of loop unrolling that is necessary. Each variable v_i is allocated its minimum number of registers, q_i, and the degree of unrolling, u_{mve1}, is given by the **least common multiple** (lcm) of the q_i. The other schedule (which we will call MVE2) favors minimizing the number of times that the loop is unrolled, at the expense of more register pressure. This minimum degree of unrolling, u_{mve2}, is the max q_i, which, of course, is never more than $\text{lcm}(q_i)$ required by MVE1. However, rather than requiring exactly q_i registers for each variable as in MVE1, MVE2 requires q_i for a variable if and only if $u_{mve2} \bmod q_i = 0$, but otherwise requires that the number of

registers allocated to store the instances of a variable increase from q_i to the smallest divisor of u_{mve2} that is greater than q_i.

Several additional techniques have been proposed to minimize register requirements in SP loops. In [23], Huff proposes a heuristic based on a bidirectional slack-scheduling method that schedules operations early or late depending on their number of stretchable input and output flow dependences. Integer programming has been used in [13, 21] to lower register requirements by optimizing according to several potentially conflicting constraints and objectives, such as resource constraints, scheduling operations along critical dependence cycles, maximizing the throughput, and minimizing the schedule length of the critical path. Stage scheduling [12] breaks the schedule into two steps. In the first step, a modulo scheduler generates a schedule with high throughput and a short schedule length. In the second step, a stage scheduler reduces the register requirements of a modulo schedule by reassigning some operations to different stages. All of these schemes aim at reducing the number of architected registers in the software pipelined loops. The best of these schemes can reduce register pressure by as much as 25% in the configurations studied. However, since all live values must be allocated to architected registers, these schemes are unable to decouple the architected register requirements from the physical requirements. In this work, we concentrate on modulo scheduling, while recognizing that our results can be applied to other scheduling algorithms as well.

Rau [46] addressed the naming problem in software pipelined loops by employing a new method of addressing a processor register file in the Cydra-5 minisupercomputer [6]. The Rotating Register File (RR) is a register file that supports compiler managed hardware renaming by adding the register address (specified in the instruction) to the contents of an Iteration Control Pointer (ICP) (modulo the number of registers in the

RR). This register specifier is then used to index into the architected register space. A special loop control operation decrements the ICP each time a new iteration starts, giving each loop iteration a distinct set of physical registers from those used by the previous iteration (thus a value referenced as r5 in iteration i will be addressed as r6 in i+1). Since register access includes an additional indirection (i.e. adding the ICP to the specifier), unrolling is unnecessary and the loop kernel is not expanded from its original form. RR can therefore eliminate the code expansion problem from SP, but it still requires a large number of architected registers because all of the physically addressable registers are part of the architected rotating register file [44].

The problem of increasing a limited architected register space without dramatically changing an existing instruction set has also been explored. The Register Connection (RC) [30] method tolerates high demand for the architected registers by adding a set of extended registers to the core register set, and incorporating a set of instructions to remap architected register specifiers into the extended set of physical registers. RC architectures use these instructions to dynamically connect architected registers to extended registers. Accesses to an architected register are automatically directed to its most recently connected physical register of the extended register file. A register mapping table with one entry per architected register is used to map each architected register to its own core physical register (by default) or to any register in the extended register file (as setup by a connect instruction). The indirection of RC is similar to that found in register renaming tables [29] used in many superscalar architectures, except that the mapping is performed under compiler control which enables more live values to reside in the extended register file than can be addressed at any point in time by the operand specifiers of the current instruction. This RC work did not target software pipelined loops; however, we show that by decoupling the

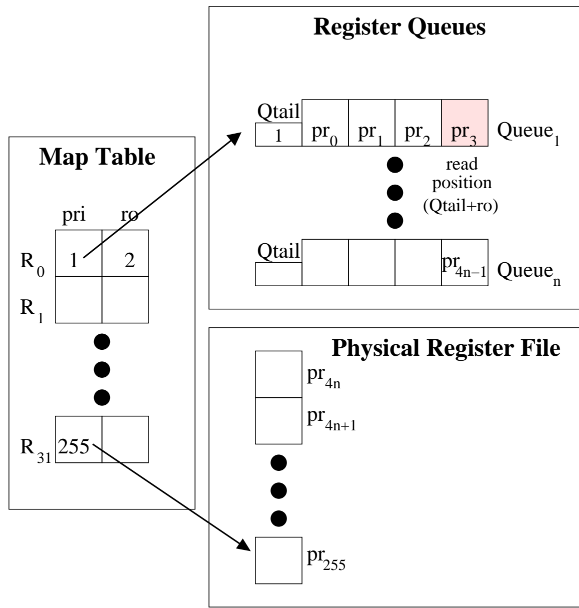


Figure 4.2: Microarchitectural extensions to support RQs for a machine with 32 architected registers, n queues of length 4, and 256 physical registers.

architected register set from a much larger physical register file, the RC method can greatly reduce the architected register requirements of these loops. Although using RC to perform SP in the context of modulo variable expansion significantly reduces architected register requirements, RC (like MVE) still requires loop unrolling to solve the register naming problem. Furthermore, RC adds some connect instructions to the loop kernel, prologue and epilogue.

4.3 Register Queues

We now propose an alternative scheme called Register Queues (RQs). RQs incorporate both a hardware-managed register renaming feature similar to RR and the register decoupling of RC to ameliorate both the code size and the architected register problems from SP. When scheduling for an SP loop, variables with multiple live instances will be placed in a queue; all other variables in the loop will be assigned to

conventional registers. The register file in an RQs design consists of three parts as shown in Figure 4.2:

- **a set of register queues**: Each queue has a *Qtail* pointer, analogous to the ICP in the RR, and a set of contiguous registers which share a common name-space with the physical register file, but are logically (and probably physically) separate. In Figure 4.2 the registers that constitute register queue 1 are physical registers pr0 through pr3; physical registers pr_4 through pr_7 make up register queue 2, etc. These registers are analogous to the registers in the RR, and use the same modulo arithmetic to index into the queue. They differ from RR registers in that registers in the queue must be explicitly mapped to an architected register before being accessed. Like the RR registers, the registers in the queues become part of the state of the processor and must be saved during context switch.

- **a physical register file**: The physical register file contains the remaining set of physical registers not allocated to a register queue. This set of registers is equivalent to the physical register file found on most superscalar processors. In Figure 4.2 the physical register file contains registers pr_{4n} through pr_{255}.

- **an architected register map table**: This table maps each architected register either to a physical register (using standard register renaming logic) or to a register queue (using an *rq-connect* instruction). Each entry in the map table contains a particular physical register index (pri) and a read offset (ro). The index specifies that either some free physical register or a particular register queue is to be mapped to the architected register. The read offset, used only for register queue mappings, contains an offset into the queue specifying which

register in the queue is mapped to the architected register.

A single *rq-connect* instruction is added to the ISA to manage the RQs: *rq-connect* maps, remaps or unmaps an architected register to one of the register queues. The semantics of the *rq-connect* instructions are:

- **rq-connect $rq, $ar, imm**: maps an architected register $ar to register queue $rq ($rq = 1,2,...,n) by writing the queue number into the *pri* field of the map table. Furthermore, the read offset (ro) in the queue is specified by the immediate field *imm*. Subsequent reads of architected register $ar will now map to the *imm*th entry from the *Qtail* of register queue $rq. Note that the semantics for a read are different than for real queues; instead of destructively reading from the head of the queue, an architected register is mapped to some location in the queue and reads occur from that location in a nondestructive manner. This greatly increases the flexibility of using register queues (though it makes the term queue somewhat of a misnomer).

- **rq-connect $0,$ar, 0**: remaps architected register $ar to a free register from the physical register file. By numbering the register queues from 1 to n, we leave the $rq = 0 operand in the *rq-connect* instruction free to indicate that the architected register $ar should be disconnected from its register queue.

A read access to an architected register that is mapped to a register queue causes the following events to take place:

1. Use the register specifier in the operand field of the machine instruction to index into the Map Table and extract the register queue identifier (the *pri* field of the register map table entry) and an offset into the queue (the *ro* field).

2. Index into the queue specified by *pri* at the specified read offset. To compute the offset, the *Qtail* is added to *ro*, modulo the number of registers in the queue. The physical register specifier is the index bits in the *pri* field with the least significant 2 bits replaced by the computed offset. Note that in this example, the circuit used to perform the mapping is a 2-bit adder – not a 7-bit adder as used in the Cydra-5 RR.

3. Read the contents of that physical register or pass that physical register identifier to later pipeline stages if the results must be forwarded from an earlier instruction that has yet to retire.

A write into the register queue involves the following sequence of steps:

1. Use the register specifier in the operand field of the machine instruction to index into the map table and extract a register queue identifier (the *pri* field). This selects the register queue; the read offset is not needed since a write value is always appended to the tail of the queue.

2. Decrement the *Qtail* pointer for the queue. This is analogous to decrementing the ICP in the RR. Note that in RQs the update of *Qtail* automatically occurs on each write to the queue, whereas in the RR the ICP is updated using a special branch instruction. Both solutions effectively manage the register naming problem.

3. Pass the physical register identifier of this new *Qtail* position in the queue (along with the instruction) to the appropriate reservation station. Note that at this point all register identifiers found in the reservation station are standard physical register specifiers, leaving the reservation station and operand forwarding logic unchanged.

It should now be apparent that the offset (ro) field of the map table entry is used only for reads from queues; it should be 0 to reference the most recently defined variable instance, 1 to reference the previous instance, etc. Furthermore, since there is only one *ro* field for each architected register it is not possible to read two different queue offsets using a single architected register except by using an intervening connect instruction, and at most one connect instruction can be issued in one cycle for the same architected register. Finally, if a read and a write are issued in the same cycle to the same architected register which is mapped to a queue, which physical register is accessed by that read is unaffected by that write. Furthermore, if the read and the write are to the same physical register, the value in that register prior to that write will be read.

4.3.1 SP Scheduling Using Register Queues

Managing the dynamic mapping of variable instances as a queue enables implementing efficient software pipeline schedules with little change in code size or architected register requirements. Each register queue, like a rotating register in RR, provides a set of registers to contain instances of a variable for several successive iterations. RR uses a contiguous set of RR architected registers to enable unconstrained access to any physical register. By contrast, RQs assigns each variable that is read one or more iterations after its definition in a software pipelined loop to a distinct register queue that holds all live instances of that variable. [1] Architected registers with unique operand specifiers are then connected to the particular locations in the queue that contain the live instances that are to be read. If a value is read three

[1] Several variants of this scheme are discussed below. In particular, Section 4.3.2 discusses how to use two register queues to accommodate a variable when one queue cannot accommodate all of its simultaneously live instances.

iterations after its definition, i.e., after 2 other intervening writes to the same variable, its architected register is mapped to the third most recent definition by using an *rq-connect* instruction to set the offset, *ro*, for that architected register to 2. In general, for a particular use of a variable, *ro* is set to the number of intervening writes that occur to that variable (or in general to that register queue to cover the case when multiple variables share the same queue) between the definition of interest and the use of that defined value.

The two more recent definitions are (at the time of this read) associated with positions of that queue that now have offsets of 0 and 1; no architected registers need ever be mapped to these queue locations if they will not be referenced until a later iteration. This mapping mechanism eliminates the need for unrolling the software pipelined loop kernel since architected registers are only mapped to offset positions in the queue that are actually read; writes to variables assigned to queues are always to the decremented *Qtail*. This scheme reduces the pressure on the register operand bits of the instruction set architecture since the number of architected registers, which must have unique operand bit patterns, is determined by the number of registers at any one time that are actually connected to queues at various read offsets. Since at most one register is needed for each active (queues, read offset) pair, the total register requirements are much less than the total of all simultaneously live instances of all variables.

The functionality of RQs can be demonstrated by re-examining the loop fragment from Figure 4.1. Figure 4.3 (a) shows the code fragment after SP is applied – including the prologue, kernel and epilogue of the loop. The prologue code includes instructions 1 through 5. Instruction 1 creates a mapping between architected register *f2* and register queue *q1* at read position 1. Writes to *f2* will now decrement *q1*'s *Qtail* and

Line	Assembly Code
1	rq−connect q1, f2, 1
2	iadd r1, r1, #4
3	fload f2, 0(r1)
4	iadd r1, r1, #4
5	fload f2, 0(r1)
6	**fadd f6, f6, f2**
7	**iadd r1, r1, #4**
8	**fload f2, 0(r1)**
9	fadd f6, f6, f2
10	rq−connect q1, f2, 0
11	fadd f6, f6, f2
12	rq−connect 0, f2, 0

(a) Scheduled Code

Cyc	Iteration A	Iteration B	Iteration C	Iteration D	Iteration E
1	rq−con q1,f2,1				
2	iadd r1, r1, #4				
3	fload f2, 0(r1)				
4		iadd r1, r1, #4			
5		fload f2, 0(r1)			
6	fadd f6, f6, f2		iadd r1, r1, #4		
7			fload f2, 0(r1)		
8		fadd f6, f6, f2		iadd r1, r1, #4	
9				fload f2, 0(r1)	
10			fadd f6, f6, f2		iadd r1, r1, #4
11					fload f2, 0(r1)
12				fadd f6, f6, f2	
13					rq−con q1,f2,0
14					fadd f6, f6, f2
15					rq−con 0,f2,0

(b) SP execution of 5 original loop iterations

Cyc	Instruction	queue 1 before instruction					queue 1 after instruction					Value read
3	fload f2, 0(r1)	0	–	–	–	–	3	–	–	–	$f2_1$	–
5	fload f2, 0(r1)	3	–	–	–	$f2_1$	2	–	–	$f2_2$	$f2_1$	–
6	fadd f6, f6, f2	2	–	–	$f2_2$	$f2_1$	2	–	–	$f2_2$	$f2_1$	$f2_1$
7	fload f2, 0(r1)	2	–	–	$f2_2$	$f2_1$	1	–	$f2_3$	$f2_2$	$f2_1$	–
8	fadd f6, f6, f2	1	–	$f2_3$	$f2_2$	$f2_1$	1	–	$f2_3$	$f2_2$	$f2_1$	$f2_2$
9	fload f2, 0(r1)	1	–	$f2_3$	$f2_2$	$f2_1$	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	–
10	fadd f6, f6, f2	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	$f2_3$
11	fload f2, 0(r1)	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	–
12	fadd f6, f6, f2	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	$f2_4$
14	fadd f6, f6, f2	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	$f2_5$

(c) Reads and writes of register (f2) in sample execution

Figure 4.3: Software pipeline schedule for the sample code (see Figure 1) using RQs

overwrite the register pointed to by the new value of *Qtail*. Once the read offset is set to 1, subsequent reads of *f2* will retrieve the contents of *q1* register (*Qtail*+1) mod queue size. The remaining instructions in the prologue load the first two memory values and increment the pointer (r1) twice.

Instructions 6-8 in Figure 4.3 (a) represent the loop kernel. The read from register *f2* in instruction 6 returns the second most recent write to *q1* (i.e. ($Qtail$ + 1) mod queue size). The variable in f6 has only one live instance at any time and does not require a queue. Instruction 7 increments the pointer (r1). Instruction 8 writes the next load value into register *f2*, decrements *Qtail* for register queue *q1*, and puts the loaded value into the register pointed to by the new *Qtail*. This loop kernel iterates until the last load operation is performed, leaving uses of the final two memory data for the epilogue.

Instructions 9-12 form the epilogue of the SP loop schedule. Instruction 9 uses the second to last memory value in the same manner as the loop kernel access. However, the last value loaded will remain in the tail of the queue since no further writes to the queue are performed. To use this value, we need to remap *f2* to reference the offset 0 position in *q1*. This is performed with another *rq-connect* instruction (instruction 10). Instruction 11 can then read from *f2* and access the final load value from the *Qtail* position. Finally, architected register *f2* is remapped to a free register in the physical register file, completing the SP schedule.

Figure 4.3 (b) shows how the SP schedule interleaves instructions from different loop iterations. The prologue instructions are issued on cycles (time) 1-5; these instructions include the *rq-connect* and *iadd*s and *fload*s for the first two iterations of the original (unscheduled) loop (we will refer to the original loop iterations by uppercase letters – A and B in this case). The kernel of the loop is shaded and in

this example executes three times (cycles 6-7, 8-9, 10-11); the first iteration of the SP loop kernel executes the *fadd* instruction from the original loop iteration A along with the *iadd* and *fload* from iteration C. The second time through the kernel will execute instructions from original loop iterations B and D at cycles 8-9; the third time through the kernel executes instructions from iterations C and E at cycles 10-11. Finally, the epilogue of the SP schedule is executed at times 12-15 performing the remaining *fadd* instructions (for iterations D and E).

Figure 4.3 (c) shows how data is accessed in register queue *q1* for the sample code. Queue reads do not change the state of the register queue; *fload* instructions at times 3,5,7,9 and 11 are shown to decrement the *Qtail* and write the new data in the queue. At cycle time 6, the first *iadd* instruction reads the first value written to *q1*. It retrieves the value $f2_1$ by adding the current *Qtail* (2) to the read offset (which was set to 1 by the initial *rq-connect* instruction), and accesses *q1* at position 3 (the rightmost, shaded position in *q1* in Figure 4.3 (c)). The remaining reads are as shown in Figure 4.3 (c) and operate similarly. Recall that the *rq-connect* at cycle 13 has changed the read offset to 0 just before the final read at cycle 14, and that this connect instruction was necessitated by the lack of a store to the queue between the read at 12 and the read at 14. Note that a dummy write to the queue could have been used at cycle 13 instead of the *rq-connect* instruction, as it would achieve the desired effect by increasing the *Qtail*, rather than the offset.

In this example, only a single variable is allocated to a queue and it contains two live instances. In general, there may be many variables with multiple simultaneously live instances. One simple connect strategy employs a single unique register queue for each such variable to hold all live instances of that variable. This mechanism works well in reducing the architected register pressure of SP schedules, but may require a

large number of register queues (one for each variable containing multiple instances). Furthermore, with fixed length queues, many of the registers in the queue may not be required (if there are fewer live instances of a variable than registers in a particular queue), whereas some variables with many live instances may not be accommodated by a single queue.

Fortunately, since the read offset values may be changed, the RQs access capabilities for a single queue are flexible enough to hold instances of more than one variable. Thus, we can assign all instances of several variables to the same queue, connecting read offsets accordingly. It is only necessary to determine the read offset for each use, given the sequence of writes for all the variable instances mapped to the queue. This is simple when writes for each variable are unconditionally performed; it is then simply a matter of counting the definitions that occur between the definition and use of a particular instance. It becomes more challenging when writes to a variable are conditionally executed (e.g., instructions in an *if-then-else* statement). In this case, we must carefully determine the possible read offsets or assign the conditionally defined variable to a queue that is not shared. Alternately, a dummy write on the alternate path can be inserted to insure that a value will be written to the queue regardless of the execution path.

A second queue register allocation issue arises when the variables assigned to a particular queue contain more live instances than the number of registers in the queue. In this case, we can either increase the initiation interval of the SP schedule (so as to reduce the number of instances), or we can concatenate two or more physical queues into a larger logical queue by copying the head of the first queue to the tail of the second queue. Each copy costs one extra instruction in the loop body to perform the copy, and one additional architected register to read the oldest value in the first

queue (offset 3) as the source field of the copy instruction (any register mapped to the following queue can be used as the destination of the copy since all writes append to the queue tail regardless of the read offset). This mechanism is discussed further in Section 4.3.2.

Finally, it is possible to run out of architected registers, even when using RQs. In this case, we can avoid spilling values to memory by reconnecting architected registers inside the loop body. Indeed, it is possible to use a single architected register throughout the SP schedule by reconnecting prior to each definition or use of a variable that is allocated to a register queue. This strategy would lead to a large number of connect instructions in the loop body (one for each read and write), but it would correctly implement the register requirements of a software pipelined loop. This mechanism is discussed further in Section 4.3.3.

4.3.2 Managing Queue Overflow

If a variable assigned to a particular queue has more live instances than the number of registers in the queues, we can concatenate two or more queues by copying the head of the first queue to the tail of the second queue before it is overwritten by a new data instance. Suppose, for example, that the load latency of the machine executing the loop fragment in Figure 4.1 is increased to 11 cycles. To maintain an initiation interval of 2, the time between a definition (*fload*) and its use (*fadd*) would span six iterations of the SP kernel, resulting in 6 (rather than just 2) simultaneously live instances of the variable, which exceeds the queue size (4 elements).

In scheduling this loop, the prologue code would expand to 14 instructions (1-14) spanning 6 iterations of the original loop (A-F), as shown in Figure 4.4 (a). The first connect instruction (at position [A,1] in the figure) creates a mapping between

Cyc	Iteration A	Iteration B	Iteration C	Iteration D	Iteration E	Iteration F	Iteration G
1	rq–con q1,f2,3						
2	rq–con q2,f4,1						
3	iadd r1, r1, #4						
4	fload f2, 0(r1)						
5		iadd r1, r1, #4					
6		fload f2, 0(r1)					
7			iadd r1, r1, #4				
8			fload f2, 0(r1)				
9				iadd r1, r1, #4			
10				fload f2, 0(r1)			
11	fmove f2, f4				iadd r1, r1, #4		
12					fload f2, 0(r1)		
13		fmove f2, f4				iadd r1, r1, #4	
14						fload f2, 0(r1)	
15	fadd f6, f6, f4		fmove f2, f4				iadd r1, r1, #4
16							fload f2, 0(r1)
17		fadd f6, f6, f4					
18			rq–con q2,f4,0				
19			fadd f6, f6, f4				
20							
21				fadd f6, f6, f2			
22					rq–con q2,f2,2		
23					fadd f6, f6, f2		
24						rq–con q1,f2,1	
25						fadd f6, f6, f2	
26							rq–con q1,f2,0
27							fadd f6, f6, f2

(a) SP schedule

Cyc	Instruction	queue 1 after instruction					queue 2 after instruction				
4	fload f2, 0(r1)	3	–	–	–	$f2_1$	0	–	–	–	–
6	fload f2, 0(r1)	2	–	–	$f2_2$	$f2_1$	0	–	–	–	–
8	fload f2, 0(r1)	1	–	$f2_3$	$f2_2$	$f2_1$	0	–	–	–	–
10	fload f2, 0(r1)	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	0	–	–	–	–
11	fmove f2, f4	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	3	–	–	–	$f2_1$
12	fload f2, 0(r1)	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	3	–	–	–	$f2_1$
13	fmove f2, f4	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	2	–	–	$f2_2$	$f2_1$
14	fload f2, 0(r1)	2	$f2_4$	$f2_3$	$f2_6$	$f2_5$	2	–	–	$f2_2$	$f2_1$

(b) contents of queue1 and queue 2

Figure 4.4: Software pipeline schedule with queue overflow (*fload* latency = 11).

architected register *f2* and register queue *q1* with a read offset of 3. Writes to *f2* enqueue data at the tail of the queue, while reads from *f2* access the oldest element in the queue. Since the queue size is insufficient to store six items, a second queue must be allocated to live instances of this variable; the second connect instruction (at [A,2]) maps architected register *f4* to *q2* to provide the remaining queue storage for this variable. After 4 loads to *q1*, the *fmove* pseudo-instructions at [A,11], [B,13], and [C,15] copy the oldest data from *q0* to the tail of *q1*. Once the oldest element in q0 is copied to *q1*, it is safe to overwrite it by executing another *fload* instruction; another *fmove* is then required sometime before the next *fload*.

The queue management performed in the prologue of this SP scheduled loop is shown in Figure 4.4 (b). The first four elements are written into *q1* by the *fload* instructions (at [A,4], [B,6], [C,8] and [D,10]). The first element is then copied to *q2* (at [A,11]) freeing that storage for the next write to *q1* (at [E,12]). This process is repeated to move the second element written to *q1* (at [B,13]) and allow the final write to *q1* in the prologue (at [F,14]).

A fourth instruction (at [C,15]) is added to the loop kernel to continue moving head elements from the head of*q1* to the tail of *q2*. The kernel is otherwise unchanged from the schedule in Figure 4.3 except that the *fadd* instruction now employs architected register *f4* and reads from *q2*. Assuming that this kernel is executed once (i.e. that 7 iterations of the original loop are required), the epilogue starts at cycle time 17. To perform more than 7 iterations of the original loop, cycles 15 and 16 are simply repeated once per additional iteration. Note that architected register *f4* and hence queue *q2* is both read and written at cycle 15. Following common design for multiported register files, we assume that the read uses the old value of *Qtail* while the write uses the decremented value as its index, but does not write into the *Qtail*

register until after the read access is completed (for the opposite sequence the offset, *ro* for the read, would simply have to be set to 2 rather than 1). Thus at cycle 15, [A,15] reads $f2_1$ from position (2+1) of *q2*, while [C,15] writes $f2_3$ into position (2-1) of *q2*. The *fload* at cycle 16 then simply overwrites $f2_3$ in position (2-1) of *q1* with $f2_7$.

Queue accesses in the epilogue differ from earlier references. Since the epilogue code will not enqueue new data into *q1* (i.e. there are no *fload* instructions in the epilogue) *fmove* instructions are no longer required; instead, *rq-connect* instructions are added to change the read offset to access the correct entry in the queues. Notice that the first two *fadd*s in the epilogue reference *q2* (through *f4*) and the final four references access *q1* (through *f2*). Two final *rq-connect* instructions (not shown in Figure 4.4 (a)) could be added if necessary to reconnect *f2* and *f4* to free registers (as done by instruction 12 of Figure 4.3 (a)).

4.3.3 Further Reducing Architected Register Pressure

In the event that too few architected registers are available to support the SP schedule, it is possible to reconnect architected registers inside the loop body. Figure 4.5 illustrates this approach by re-examining a modified version of the loop fragment from Figure 4.1. For demonstration purposes we added a new instruction (*fadd f8, f6, f2*) requiring an additional read of the queue mapped to *f2* with a different offset. [2] Normally, each different read location in the queue would be mapped to a different architected register so as to eliminate reconnecting; however, in this example, we assume that only a single architected register is free for use by this variable.

[2] We also change the latency for *fadd* to 1 for this example in order to simplify the discussion of the resulting schedule.

Line	Assembly Code
1	**iadd r1, r1, #4**
2	**fload f2, 0(r1)**
3	**fadd f6, f6, f2**
4	**fadd f8, f6, f2**

(a) Original loop body before SP scheduling

Cyc	Iteration A	Iteration B	Iteration C	Iteration D
1	rq-con q1,f2,1			
2	iadd r1, r1, #4			
3	fload f2, 0(r1)			
4		iadd r1, r1, #4		
5		fload f2, 0(r1)		
6	fadd f6, f6, f2		iadd r1, r1, #4	
7			fload f2, 0(r1)	
8	rq-con q1,f2,2 fadd f8, f6, f2			iadd r1, r1, #4
9		rq-con q1,f2,1 fadd f6, f6, f2		fload f2, 0(r1)
10		rqcon q1,f2,2 fadd f8, f6, f2		
11			rq-con q1,f2,1 fadd f6, f6, f2	
12			fadd f8, f6, f2	
13				rq-con q1,f2,0 fadd f6, f6, f2
14				fadd f8, f6, f2

(b) SP schedule

Cyc	Instruction	queue 1 before instruction					queue 1 after instruction					Value read
3	fload f2, 0(r1)	0	-	-	-	-	3	-	-	-	$f2_1$	-
5	fload f2, 0(r1)	3	-	-	-	$f2_1$	2	-	-	$f2_2$	$f2_1$	-
6	fadd f6, f6, f2	2	-	-	$f2_2$	$f2_1$	2	-	-	$f2_2$	$f2_1$	$f2_1$
7	fload f2, 0(r1)	2	-	-	$f2_2$	$f2_1$	1	-	$f2_3$	$f2_2$	$f2_1$	-
8	rq-con q1,f2,2 fadd f8, f6, f2	1	-	$f2_3$	$f2_2$	$f2_1$	1	-	$f2_3$	$f2_2$	$f2_1$	$f2_1$
9a	rq-con q1,f2,1 fadd f6, f6, f2	1	-	$f2_3$	$f2_2$	$f2_1$			$f2_3$	$f2_2$	$f2_1$	$f2_2$
9b	fload f2, 0(r1)	1	-	$f2_3$	$f2_2$	$f2_1$	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	-

(c) Contents of queue 1

Figure 4.5: Software pipeline schedule with architected register pressure (*fadd* latency = 1).

The modified source code is shown in Figure 4.5 (a) and the resulting SP scheduled loop is shown in Figure 4.5 (b).

In the modified loop body there are two reads from register *f2* with different read offsets. If we had an additional architected register, it would simply be connected to the second read position in the same queue. Instead, we keep reconnecting register *f2* inside the loop body to alternate between the two desired read positions. Prologue instruction [A,1] (Figure 4.5 (b)) maps architected register *f2* to register queue *q1* with read offset 1. The *fload* instructions enqueue data in *q1*. The first *fadd* instruction (at [A,6]) accesses *q1* with read offset1 (to access the value loaded at [A,3] despite the intervening *fload* at [B,5]). The second *fadd* instruction (at [A,8]) accesses the same data in *q1* with read offset 2 (since there has been another intervening *fload* at [C,7]). The following iterations connect similarly.

The loop kernel requires six instructions: the original four instructions and two additional *rq-connect* instructions. In the first cycle of the loop kernel (cycle 8), *f2* is mapped to *q1* with a read offset of 2 to enable access to the value required for the *fadd f8, f6, f2* instruction in [A,8]. The second cycle of the loop kernel (cycle 9) remaps *f2* to access *q1* with a read offset of 1, in order to access the value required for the *fadd f6, f6, f2* instruction in [B,9]. The *fload* instruction writes a new element, as before, into the queue and updates the *Qtail*.

Keeping the initiation interval at 2 cycles requires careful management of the queue resources; for example, in cycle 9 we are changing the mapping of register *f2*, reading the queue specified by *f2*, and writing a new element into the queue specified by *f2*. The ordering of these operations is as follows:

- the read from *f2* uses the queue mapping (*q1*) and the read offset (1) forwarded from the *rq-connect* instruction that is issued in the same cycle (instead of

reading the current *ro* field from the map table) and adds that forwarded offset to the *Qtail* value (1) that was available at the start of the cycle, and thereby accessing element $f2_2$.

- the write to *f2* also uses the queue mapping forwarded from the *rq-connect* instruction issued in the same cycle, decrements the *Qtail* value (to 0) and writes $f2_4$ into *q1* at position 0.

Figure 4.5 (c) shows the queue requests that occur in the prologue and the first iteration of the loop kernel. Writes to the queue occur in cycles 3,5,7 and 9. Reads occur on cycles 6 and 9 for the *fadd f6,f6,f2* instruction (using read offset 1) and on cycle 8 for the *fadd f8,f6,f2* instruction (using read offset 2). The ninth cycle is separated into two parts to illustrate the details of the queue access: cycle 9a shows the queue read performed by *fadd f6, f6, f2* using an offset of 1 from the current *Qtail* of 1, returning element $f2_2$; cycle 9b shows the second phase of that cycle in which the *fload* alters the queue state, in particular $f2_4$ is written into *q1* and its *Qtail* is decremented. Remapping *f2* continues as appropriate for the accesses by the *fadd* instructions in the epilog, as in the previous examples, without any intervening *fload* instructions.

4.4 Performance Evaluation

Our experimental study consists of two parts. In the first part we demonstrate the capability of the RQs approach by comparing the register space and kernel code requirements for various load latencies in the RR method and both MVE methods (labeled MVE1 and MVE2) and comparing the results to those with the RQs scheme. In the second part, we demonstrate the capability of the RQs approach by comparing

the register space requirements of the RQs and RR schemes when either deterministic or probabilistic predicate aware modulo scheduling is used with various cmpp latencies.

In both experiments we use an iterative modulo scheduler (IMS) [41] which produces a near optimal steady-state throughput for machines with realistic machine models. However, note that we have not implemented a register queue allocation algorithm within the IMS scheduler. The register queues requirements for a given loop in both studies are estimated by counting the number of register definitions in the loop body. When the loop contains predicated operations, which is always the case for loops scheduled with DPAMS and PPAMS, it often happens that a register consumer depends on several distinct producers that are guarded under different predicates. All these producers define the same destination register that is associated with a single queue. Since predicates only become known at runtime, it is not possible for a compiler to correctly insert an rq-connect instruction to connect the consumer of this register with a fixed position in the queue that will always correspond to the correct value written by the correct producer. The read offset into the queue will vary at runtime depending on which producer's predicate is True in a given loop iteration. This is a well-known naming problem in predicated code which, in order to match a register consumer with the correct producer, requires that the predicates of all the possibly correct producers be known; obviously, no such predicate values are available at compile time. We have not addressed this problem in this dissertation. Instead, we assume that each producer of the same register that is guarded under a distinct predicate will go into a distinct queue. This assumption will result in overestimating the number of queues required to perform software pipelining using RQs in those cases where there are several producers of the same register that are guarded under

distinct predicates.

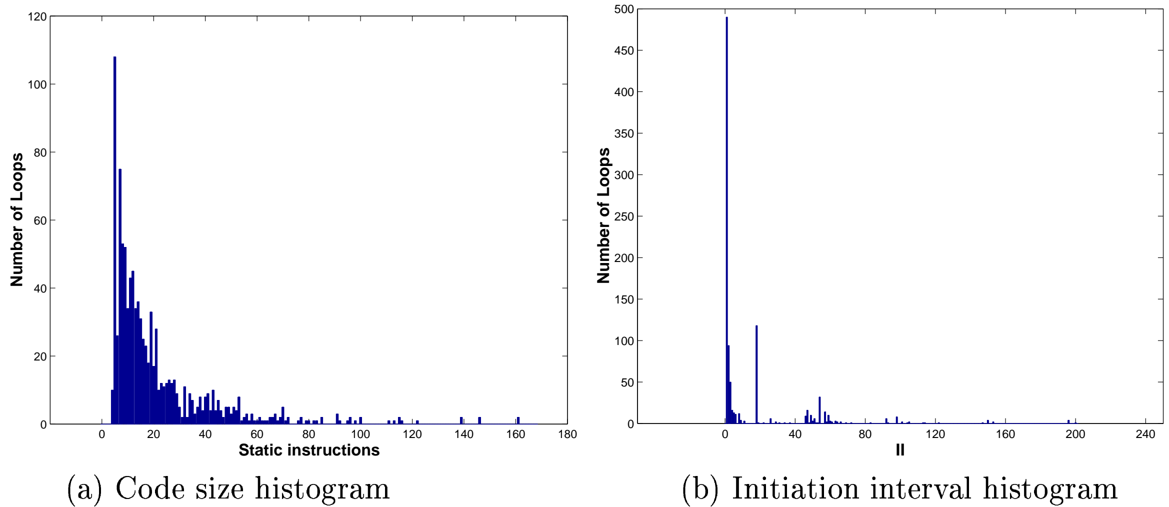
4.4.1 Register Queues Study for Various Load Latencies

As we described in Section 4.2, MVE1 minimizes register pressure at the expense of increasing the degree of loop unrolling, whereas MVE2 minimizes degree of loop unrolling at the expense of using more registers. For each technique we vary the load latency from 1 cycle to 45 cycles to assess how the resource requirements might vary across a wide variety of machine models.

The benchmark loops studied in this series of experiments were obtained from the Perfect Club Suite, SPEC, and the Livermore Kernels. These loop kernels were provided by B.R. Rau from HP Labs. Loops were compiled by the Cydra 5 Fortran77 compiler performing load-store elimination, recurrence back-substitution, and IF-conversion. The input to our scheduler consists of the intermediate representation; SP is then performed, generating a new intermediate representation with support for RQs. Of the 1327 loops extracted from these applications, 983 were selected for this study; the remaining 344 loops did not perform memory references.

In these experiments we used two target machine models. One machine model has limited resources (similar to the 6-wide machine baseline machine introduced in Section 2.5.1), while the other has no resource constraints. The code sizes (static instruction count) of the 983 loops studied (before SP was performed) are shown in Figure 4.6(a). A majority of the loops ranged from 5 to 20 instructions, with the largest loops exceeding 150 instructions.

Figure 4.6 (b) shows the initiation intervals (after software pipelining) for the loops, assuming a load latency of 13 cycles and no resource dependencies. The vast majority of the loops have *II* between 2 and 15 cycles, with a few loops requiring up



(a) Code size histogram (b) Initiation interval histogram

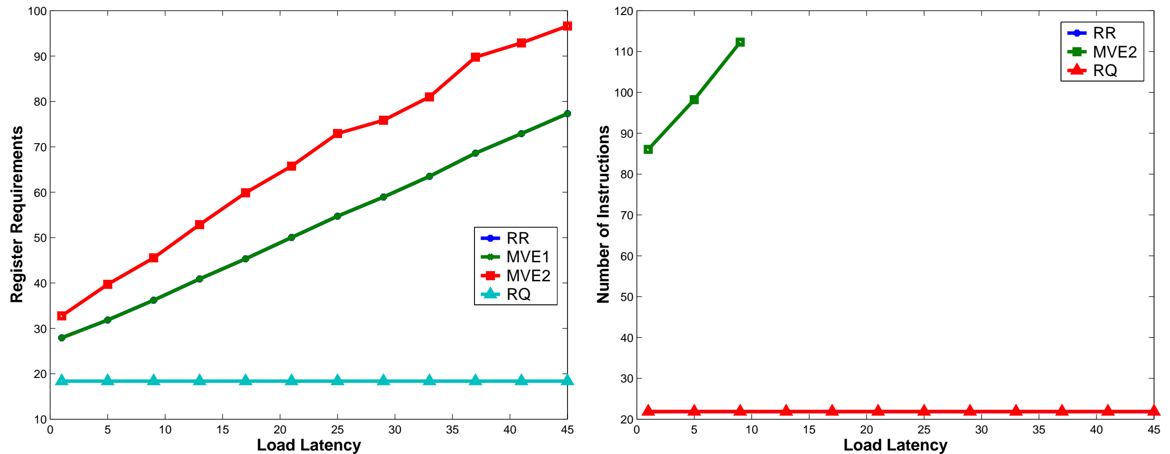
Figure 4.6: Loop statistics

to 200 cycles.

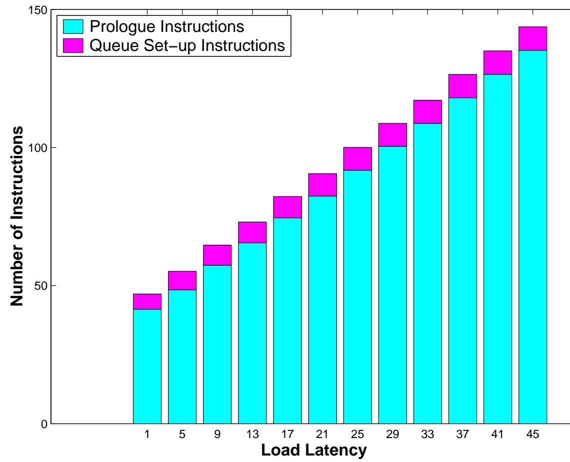
4.4.2 Software Pipelining Using MVE, RR and RQs

The results of the experiments in this section show the effects on architected and physical register requirements, as well as the code expansion of the loop due to software pipelining. Software pipelining was performed using both methods of MVE (minimizing register requirements (MVE1) and minimizing unrolling (MVE2)) with no hardware support. SP was also performed targeting each of the two machine configurations with hardware support: RR, and RQs. These results are presented in Figure 4.7 and Figure 4.8 for the two machine models with unlimited and limited resources, respectively.

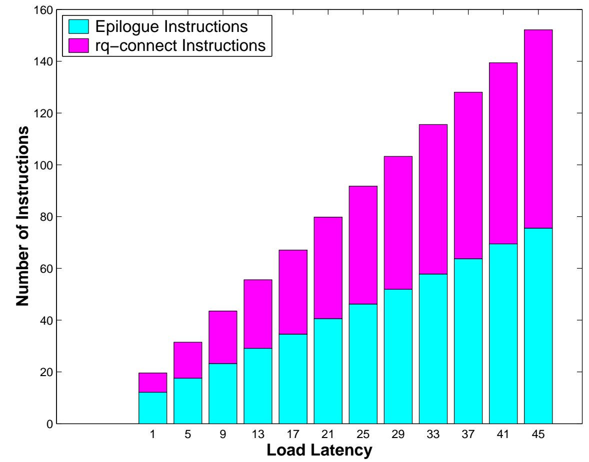
Figure 4.7 (a) and Figure 4.8 (a) show the architected register requirements after performing software pipelining on each loop (averaged over all loops). The graphs show the increase in register requirements and code expansion of the loop kernel as memory latency is increased from 1 to 45 cycles. The two models differ



(a) Architected register requirements (b) Code size requirements



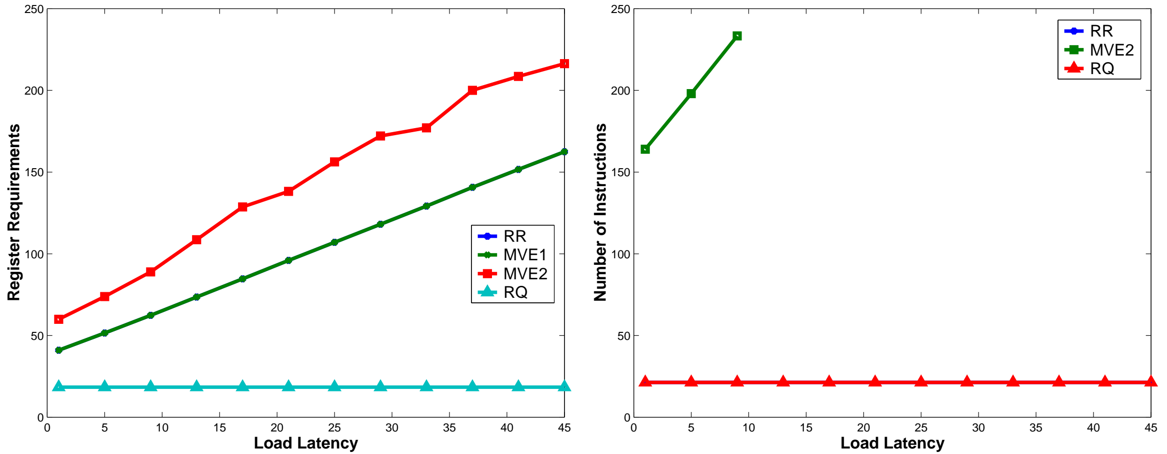
(c) Prologue code size



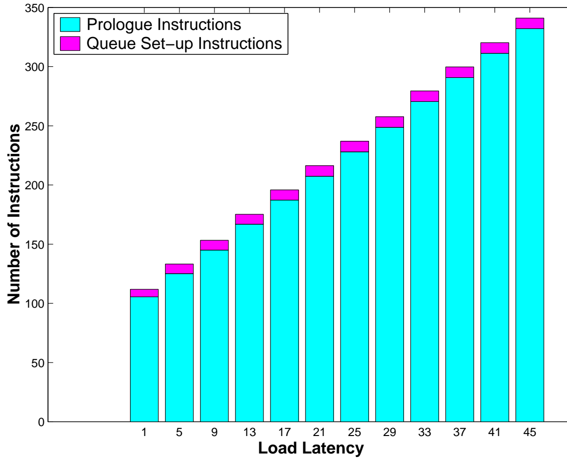
(b) Epilogue code size

Figure 4.7: A study of RR, MVE and RQs schemes for machine model 2 (with limited resources).

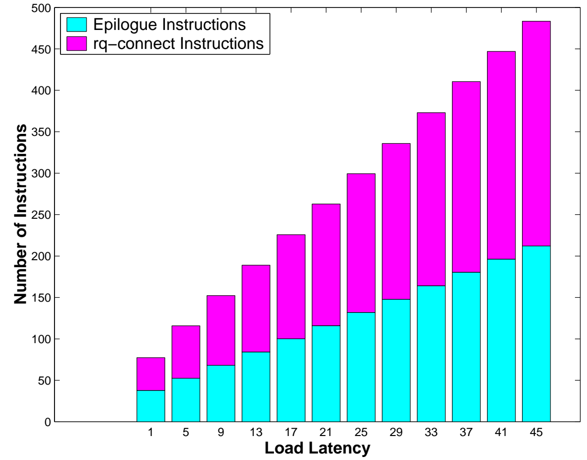
significantly in the number of registers required to achieve the best (minimum *II*)) software pipelining. For the three schemes other than RQs, the unlimited resource model (which has no resource constraints and therefore a small *II*) typically requires 2 to 3 times as many registers as the more realistic machine model. However, the trends seen in both models are similar. In the RR and MVE1 schemes, the numbers of architected registers are identical, growing at a linear rate with load latency (middle curve). The architected register requirements for MVE2 increase more rapidly, since extra registers are added to reduce the code expansion; this growth rate is also fairly



(a) Architected register requirements (b) Code size requirements



(c) Prologue code size



(b) Epilogue code size

Figure 4.8: A study of RR, MVE and RQs schemes for machine model 1 (with unlimited resources).

linear.

Architected register requirements for the RQs scheme remain constant as long as all live instances of each variable can fit in one register queue. The increased latency affects the RQs schedule only by increasing the offsets specified in the *rq-connect* instructions in the loop prologue; as more instances of a variable are needed to support higher latencies the offset is increased to account for the change in the location of the instance that is read. The number of architected registers in the RQs scheme is bounded by the number of consumers (instructions in the loop body that

read from the queue) and is not affected by the latency of the instructions.

Figure 4.7 (b) and Figure 4.8 (b) show the code expansion caused by SP as memory latency increases. Code size remains unaffected by memory latency for both RR and RQs due to the hardware support for renaming the instances of a variable. The code size drastically increases in the MVE schemes because of the additional unrolling required to handle the explicit, distinct naming of the additional live instances of the variables defined by load instructions as latency increases. MVE1 is not shown on these graphs because of its tremendous code expansion; for a load latency of 13 on the machine with limited resources, the kernel code size in MVE1 averages 149,256 instructions.

Figure 4.7 (c) and Figure 4.8 (c) show the code expansion of the prologue code as latency increases. Each bar shows the number of instructions moved from early iterations of the original loop to initialize the software pipeline, as well as extra instructions required in the RQs model to connect architected registers to register queues (the darker shaded portion at the top of each bar). The additional overhead in the prologue to initialize the register mappings required in the RQs scheme is seen to be minimal. Figure 4.7 (d) and Figure 4.8 (d) show similar code requirements for the loop epilogue. Here the overhead for the RQs scheme is higher, which is caused by the necessity to remap architected registers to read the final instances in the queue, after no more writes to the queue are performed to preserve a constant queue read offset.

Figure 4.9 shows the number of variables with multiple simultaneously live instances over all loops for the machine with unlimited resources. The vertical axis shows how many loops have a specified number of variables with multiple live instances. For instance, the leftmost column (at 2 on the horizontal axis) shows that

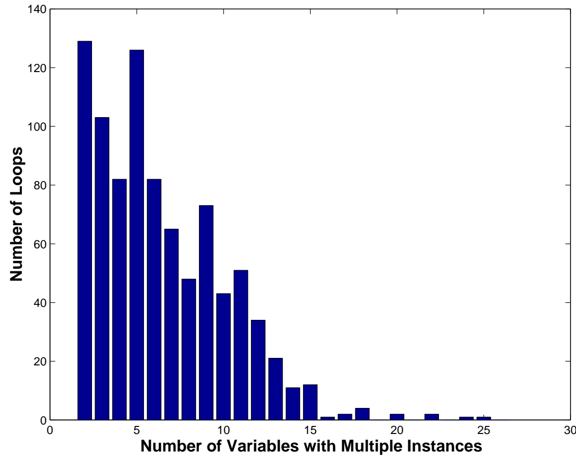


Figure 4.9: Histogram of the number of multi-instance variables in a loop.

130 loops have exactly 2 variables with multiple live instances. Almost all of the loops have fewer than 16 variables with multiple live instances. Since the register queues are allocated only to those variables with multiple live instances, the register queue allocation problem need only address those (few) variables.

Figure 4.10 shows the number of simultaneously live instances for each of the variables identified in Figure 4.9 for the machine model with unlimited resources. Over half of the variables require only 2 instances, resulting in little physical register pressure in the queue. The number of live instances becomes even less as resources are more limited. This result also makes finding a very close to optimal bin-packing solution to register queue mapping quite easy. The largest number of live instances found was 13. Unlike the number of variables with multiple instances (Figure 4.9), the number of instances for each of those variables will increase in proportion to the memory latency.

Figure 4.11 shows the rate of increase in the number of instances (averaged over all variables in all loops) as load latency increases. The growth is linear, ranging from 2.5 with low latency (since we count only variables with multiple instances, 2

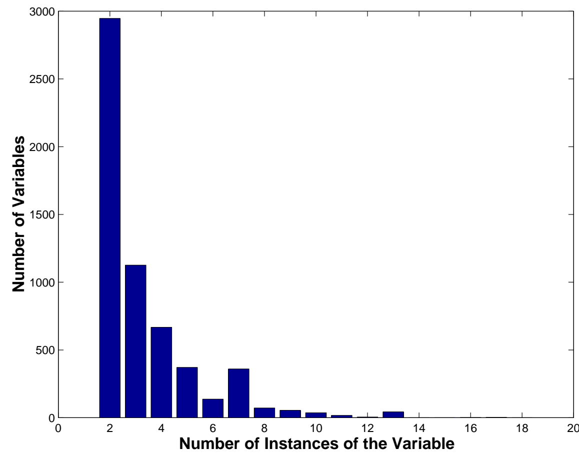


Figure 4.10: Histogram of the number of instances of variables containing multiple live instances (averaged over all loops).

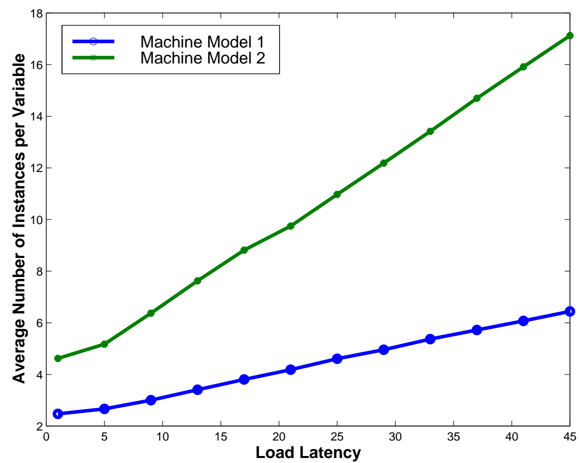


Figure 4.11: Average queue size as latency increases for machine models 1 (limited resources) and 2 (unlimited).

is an absolute minimum) to 6.5 when memory latency is 45 (for the machine model with limited resources), or 4.5 to 17 (for the machine with unlimited resources). This number is very large when allocating the small number of physical registers found on most machines, making SP intractable. However, since these are only physical register requirements in the RQs model, RQs makes it much more feasible to perform SP.

4.4.3 Scheduling Multiple-use Lifetimes for FIFO Queues With and Without Destructive Reads

The implementation of register queues presented in this work might more descriptively be called circular register buffers. Unlike FIFO queues, reads can access any element in the buffer and reads are nondestructive. We chose this design to enable more flexible access to live variable instances. In this section we examine the effects of this flexible access mechanism on the SP schedules by comparing it with conventional FIFO organizations which read from the head of the queue. We examine two FIFO designs, one which utilizes a destructive read and a second which employs both destructive and nondestructive reads of the queue.

The first experiment performed to evaluate the effectiveness of our more flexible queue access mechanism was to characterize the usage of all variables that are allocated to a queue in the benchmark loops. For each write to a queue, the number of reads of that element are counted. If there is a single read, then a FIFO organization with destructive reads is sufficient to access the element; destroying the data is allowed since it will not be re-accessed and the data will be located at the head of the queue when accessed provided that the queue does not hold instances of some other variable. However, writes of a variable that is read multiple times make destructive

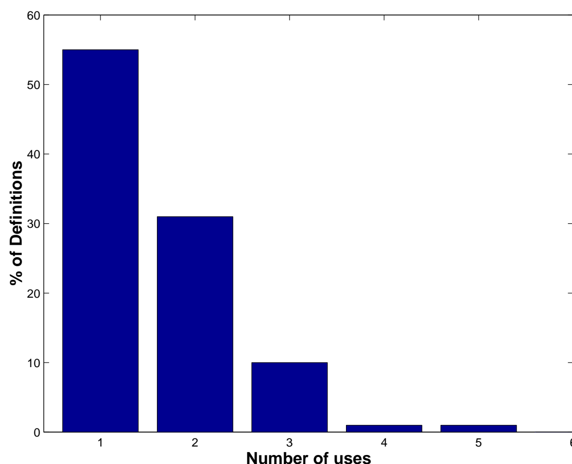


Figure 4.12: Number of uses for each instance of variables with multiple instances.

reads unattractive since a destructive read of the first read access eliminates the data, preventing further reads. In this event, additional code must be inserted to reconstruct the data or to retain it in some other storage (e.g. a general purpose register) to support multiple reads.

Figure 4.12 shows how many variables have multiple readers. The horizontal axis shows the number of readers for a variable and the vertical axis shows what percent of all the variables allocated to register queues have a given number of readers. Note that 55% of the variables written to a register queue are read only once; these read references require no access method more sophisticated than a FIFO queue structure. The remaining 45% of the variables written to queues require at least 2 reads, making destructive reads problematic.

At least one FIFO queue design [58] proposed allowing both destructive and non-destructive reads from the head of a queue to provide more flexible access to queue elements. This approach increases the number of variables that can be allocated to a queue without the overhead of moving them to a conventional register prior to using the data; however, it is still too restrictive for some variables in an SP pipelined

code. About one third of the variables with multiple reads require reads of the same instance in different iterations of the SP kernel. This means that the first read will very likely not occur when the element is at the head of the queue, since it must remain in the queue until the last read. In this event, a FIFO queue structure would require that multiple queues be allocated for such a variable: one queue to store the data from the write to the first read, a second to store the data between successive reads in different SP kernel iterations, and possibly additional queues if reads occur in three or more different iterations.

To determine the instruction overhead required within the SP kernel if FIFO queues were used, we rescheduled each of the loops that had at least one variable that was written to a queue and read two or more times. The loop kernel instruction count was increased by 60% for those loops when only destructive FIFO accesses were provided. This overhead included register queue to general purpose register moves and re-queuing instructions when reads occurred in different loop iterations. The overhead was reduced to 19% when nondestructive reads from the head of the queue were also allowed. This overhead consisted of register queue to register queue moves for variables with reads in different SP kernel iterations. For this reason we feel that allowing nondestructive reads from any queue position is a much preferred solution; it offers a much more flexible solution for queue sharing, etc. as discussed at the end of Section 4.3.

4.4.4 Use of Register Queues in Predicate-aware Machines

In this section we demonstrate the capability of the RQs approach by running some experiments to compare the register space requirements of the RQs and RR schemes on machines that employ deterministic or probabilistic predicate aware mod-

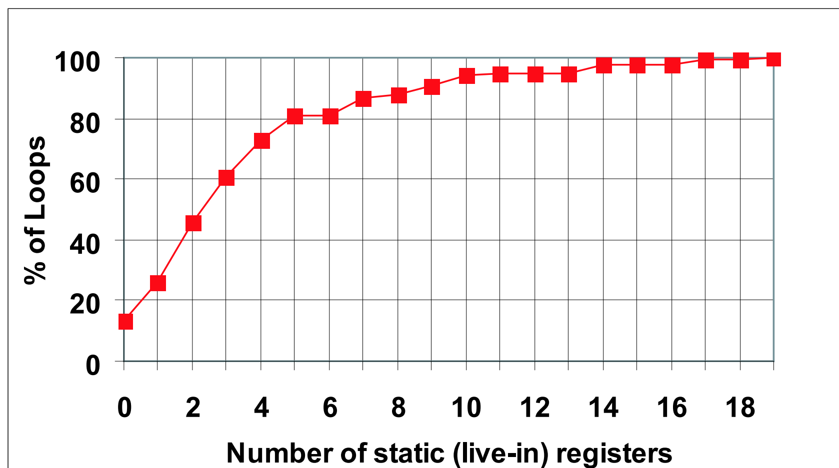


Figure 4.13: Histogram of the number of live-in values

ulo scheduling (DPAMS or PPAMS) with various cmpp latencies. Both scheduling schemes attempt to reduce the initiation interval of the loops relative to baseline modulo scheduling (BAMS) by allowing some predicated operations to share the same resource in the same cycle. To maximize the degree of sharing, the predicated operations are often moved further away from their consumers. This can create a large number of simultaneously live instances of loop variables from different iterations of the original loop body, each of which needs its own physical register. As we have seen in Section 3.5.2.4 (Table 3.4(a)), compared to BAMS on a 4-wide machine with cmpplat=3, DPAMS and PPAMS increase the rotating register requirements of the modulo scheduled loops by 30% and 50%, respectively.

For this study, we use a total of 122 if-converted loops extracted from the seventeen MediaBench [33] applications used in Chapter 2 and Chapter 3. We use 4- and 6-wide baseline, deterministic predicate-aware (see Section 2.5), and probabilistic predicate-aware processor models (see Section 3.5). In addition, we assume that register queues are deep enough to accommodate all simultaneously live variable instances.

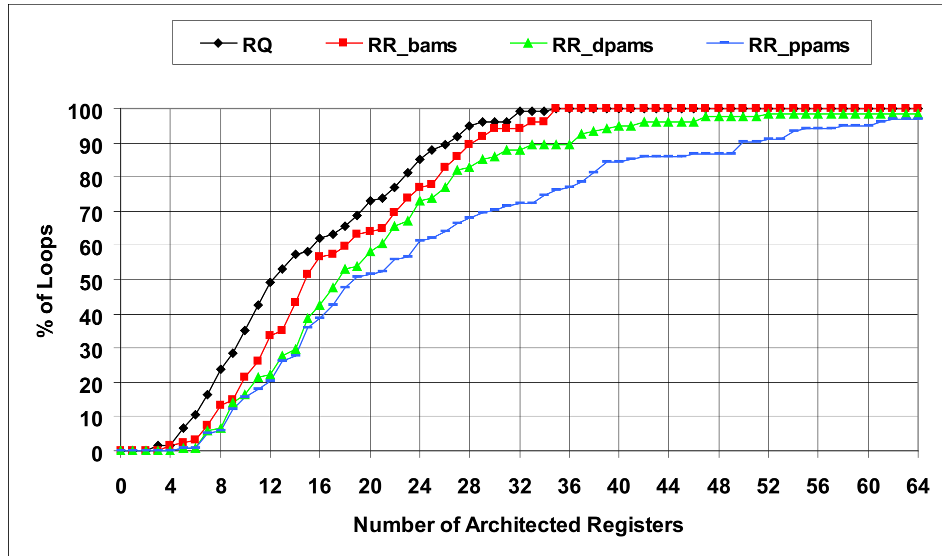
We first study the number of static registers required. The static registers result

from live-in static values, which are defined outside of the loop and never defined within it. All these values must be allocated to architected registers to avoid spilling within the loop body. Figure 4.13 shows the cumulative histogram of the number of static registers in the loops. The point (x, y) on the curve indicates that y% of all loops contain at most x static registers. We see that 80% of all the loops require at most 6 architected registers to accommodate their static values. The average number of static registers required is 3.8; the maximum is 19. The architected register requirement studies presented in the rest of this section include these static registers.

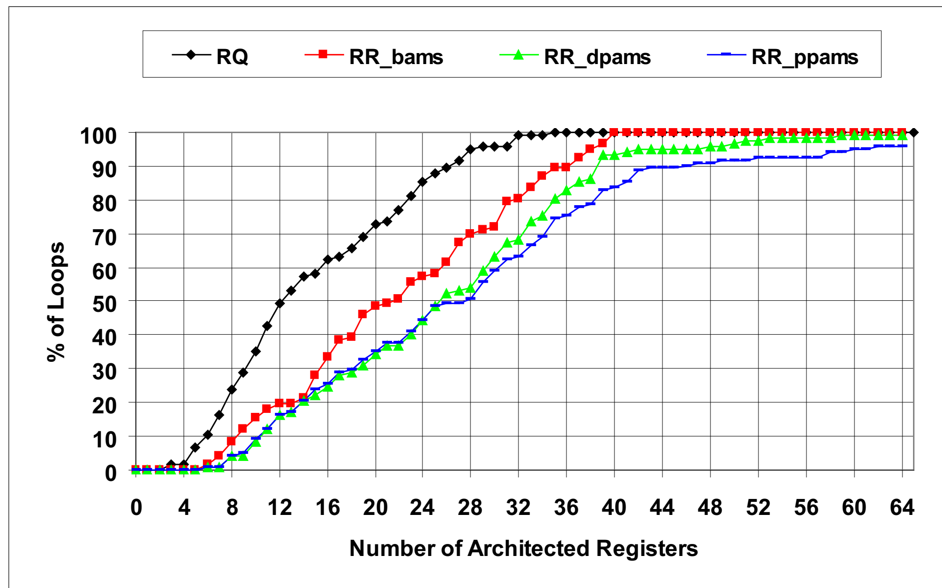
Figure 4.14(a) shows the histogram of the architected register requirements for the BAMS, DPAMS and PPAMS schemes on the 4-wide machines. Both predicate-aware machines have a cmpp latency of 3 cycles, and the probabilistic predicate-aware machine has a conflict detection and recovery latency (CDRL) of 1 cycle. The point (x,y) on the curve indicates that y% of the loops require x or fewer architected registers to achieve the same II as on the machine with an infinite number of architected registers. The bottom three curves show the histogram of the architected register requirements with the RR scheme for BAMS, DPAMS and PPAMS. The top curve shows the architected register requirements for all three schedulers when RQs is used. Figure 4.14(b) shows the corresponding data for the 6-wide machines.

Note that the architected register requirements with the RQs scheme are identical for all three schedulers and thus are represented just one curve. As we said earlier, the number of architected registers in the RQs scheme is bounded by the number of consumers and is not affected by how far the consumer is from the producer in a single iteration schedule; thus it is unaffected by which of the three schedulers is used.

When the RR scheme is used with a fixed number of architected registers, DPAMS



(a) $P_{dpas}(4, 3)$ and $P_{ppas}(4,3,1)$



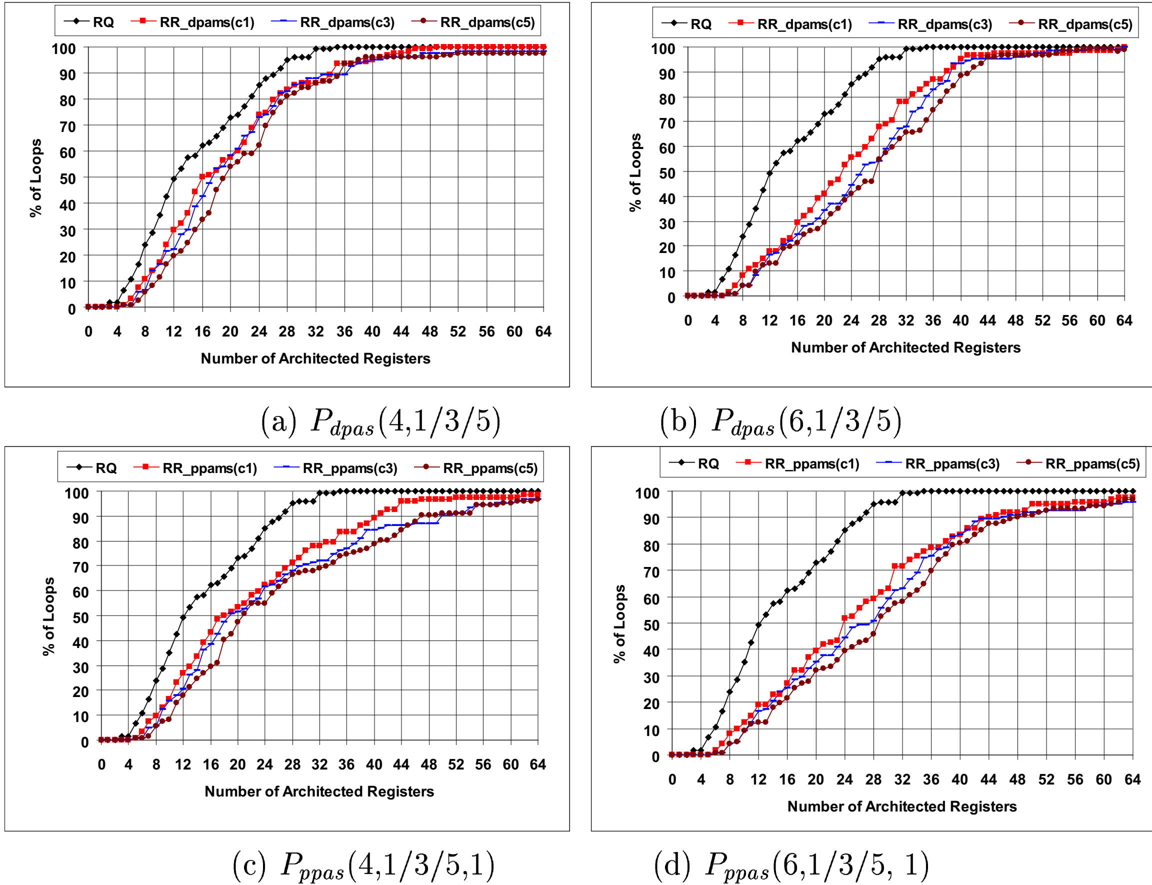
(b) $P_{dpas}(6, 3)$ and $P_{ppas}(6,3,1)$

Figure 4.14: Architected register (RQs vs. RR) requirements for DPAMS and PPAMS with cmpp latency 3 and CDRL=1

can schedule fewer loops than BAMS, and PPAMS can schedule fewer loops than DPAMS. For example, for a 4-wide machine with 32 architected registers, BAMS can schedule 94% of all loops with the RR scheme, whereas DPAMS and PPAMS with RR can schedule only 87% and 72% of all loops, respectively. As expected, PPAMS requires largest number of architected registers with RR, since it aggressively moves operations from different iterations to minimize the expected schedule length (see Section 3.4.4). Since DPAMS can only combine operations from the same iteration, it is less aggressive than PPAMS, and hence requires fewer architected registers with the RR scheme. Note that for the same 4-wide machine with 32 architected registers, we can schedule 99% of the loops with all three schedulers by using the RQs scheme.

As we go to a 6-wide machine with the RR scheme, the required number of architected registers increases. For example, with 32 architected registers, the RR scheme permits BAMS, DPAMS and PPAMS, respectively, to schedule only 80%, 68% and 63% of the loops on the 6-wide machine compared to 94%, 88% and 72% of the loops that same scheduler can schedule on the 4-wide machine. This reduction is due to the fact that the machine with more resources allows higher throughput which is achieved by increasing the number of overlapped iterations and lengthening def-use chains. This, in turn, increases the number of simultaneously live instances of loop variables. However, when using the RQs scheme, the number of architected registers required on 4- and 6-wide machines remains the same.

Figure 4.15 shows the effect of cmpp latency by comparing RQs and RR requirements for DPAMS and PPAMS with cmpp latencies of 1, 3 and 5 cycles. The bottom three curves of Figure 4.15(a) show the histogram of the architected register requirements for the DPAMS scheduler with RR on three 4-wide deterministic predicate-aware machines with cmpp latencies of 1,3 and 5, respectively. The top



(a) $P_{dpas}(4,1/3/5)$ (b) $P_{dpas}(6,1/3/5)$

(c) $P_{ppas}(4,1/3/5,1)$ (d) $P_{ppas}(6,1/3/5, 1)$

Figure 4.15: Architected register requirements (RQs vs. RR) for DPAMS and PPAMS with cmpp latencies 1, 3 and 5 (CDRL=1 for PPAMS)

curve of this figure shows the architected register requirements for DPAMS when RQs is used. Figure 4.15(b) shows the corresponding data for the 6-wide deterministic predicate-aware machines. Figure 4.15(c) and (d) show data corresponding to Figure 4.15(a) and (b) for the PPAMS scheduler on the 4- and 6-wide probabilistic predicate-aware machines with CDRL=1.

We see that as we increase cmpp latency, the DPAMS and PPAMS schedulers require more architected registers with the RR scheme, whereas the register requirements with RQs remain constant for all three latencies. This increase in the number of registers with RR is expected since increasing cmpp latency moves consumers fur-

ther away from their cmpp producers. However, the number of architected registers with the RQs scheme is unaffected by this distance.

4.5 Summary

Existing software pipelining implementations have limited effectiveness due to their high architected register requirements, particularly as operation latencies grow. In this chapter, we have introduced the RQs technique which almost completely eliminates architected register pressure and code size increases from software pipeline schedules by combining a modification to the microarchitecture of a processor with a minor extension of its ISA and a modified register allocation algorithm in the compiler.

RQs achieves this goal by combining the features of RR (to enable instances of a variable defined in earlier iterations to be accessed efficiently) with the features of RC (to decouple architected registers from the physical registers holding live variable instances). By including the dynamic register name mechanisms found in RR, we can achieve a software pipelined loop without unrolling the kernel, and by adding the register decoupling capabilities of RC we can allocate multiple instances of a loop variable without increasing architected register pressure. This enables RQs to schedule loops for expected memory latencies when a cache miss occurs; the alternative is to assume that all memory accesses will hit in the L1 cache and stall the processor when a miss occurs which leads to non-optimal schedules, particularly when cache miss rates are high.

Our experiments on the loops from a large benchmark suite showed that RQs provides a significant reduction in the number of architected registers and code size requirements (compared to RR and MVE). Furthermore, with RQs, memory latency

increases have little effect on either code size or architected register requirements. RQs thus enables more aggressive implementation of software pipelining. Finally, by allowing reads to occur nondestructively and from any location in the queue, RQs significantly reduces the instruction overhead required to access values stored in conventional FIFO queues.

RQs can be incorporated into existing instruction set architectures with the addition of a single new instruction and a modification of the register renaming microarchitecture. Furthermore, the complexity of the implementation approximates that of RR, requiring a single level of indirection and modulo arithmetic of small (4 or 5 bit) offsets to address the physical registers in the queue (for queues of length 16 or 32). The physical register requirements of RQs can also be scaled by reducing the number of registers in a queue and/or by restricting the number of queues. The results show that a small number of modest size queues is sufficient to support software pipelining, even as instruction latencies increase.

We have also demonstrated the potential benefits of RQs in the context of the DPAS and PPAS modulo scheduling schemes. We have seen that using rotating registers, on the 4-wide predicate-aware machine with a cmpp latency of 3 cycles and 32 architected registers, we could only schedule 87% of the loops from the seventeen Mediabench applications with DPAMS, and only 72% with PPAMS without register spilling or increased II. For the corresponding 6-wide machine DPAMS and PPAMS can only schedule 68% and 63% of these loops, respectively. On the other hand, by using RQs with 32 architected registers, we can schedule 99% of the loops no increased II and no register spilling by using either DPAMS or PPAMS on either 4- or 6-wide predicate-aware machines with a cmpp latency of 1, 3 or 5 cycles.

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

5.1 Summary

To expose higher levels of instruction level parallelism, the VLIW/EPIC compilers uses aggressive optimization techniques, such as predication and software pipelining, which exploit the increased instruction level parallelism provided among several successive basic blocks. To take full advantage of the parallelism offered by these optimizations requires increasing the number of function units and other processor resources. Increasing function units to meet the computation demands of the optimized code regions, leads in turn to increases in large centralized on-chip structures, such as the register file and bypass interconnect. These larger structures are, however, difficult to design without compromising clock speed, and increasing the design complexity and power consumption of the processor. As such they make the promised higher performance levels difficult to deliver in practice.

This dissertation is concerned with processor performance in the context of predicated execution and software pipelining, two of the most common and effective techniques for increasing instruction level parallelism. In particular, we address the issue

of how to improve the processor performance without increasing the processor's critical resources. To this end, we propose three schemes, together with necessary and recommended adaptations of the processor pipeline that enable predication and software pipelining to achieve higher performance by means of more efficient utilization of the existing processor resources.

In Chapter 2, we describe deterministic predicate-aware scheduling (DPAS) which allows several disjoint predicated operations to share the same resource in the same cycle, thus increasing the resource utilization and decreasing the schedule length in predicated code regions where resources are a bottleneck. The processor pipeline modifications allow predicated operations that share a resource to read their predicates early and thereupon discard operations guarded under False predicates. The main consequence of this modification is that in order to combine several operations to share the same resource, they must be scheduled a sufficient distance (no less than *extendedlatency* cycles) away from their corresponding *cmpp* operations. The distance of *extendedlatency* cycles between an operation and its cmpp is sufficient to ensure that when the operation's predicate is read early in the pipeline (during the predicate read and dispatch stage), the corresponding cmpp operation has already completed execution and the predicate is available.

We have proposed and evaluated three deterministic predicate-aware scheduling algorithms: two deterministic predicate-aware list scheduling algorithms ('extend-all' DPALS and 'best-fit' DPALS) to scheduling acyclic regions, and one deterministic predicate-aware modulo scheduling algorithm (DPAMS) to schedule cyclic regions.

'Extend-all' DPALS aggressively extends the cmpp latency for each predicated operation in the region to *extendedlatency* cycles whenever that may potentially improve the performance. Over all predicated regions, 'extend-all' DPALS on 4-wide

deterministic predicate-aware machine with the realistic extended cmpp latency of 3 cycles achieves a 2% speedup over the 4-wide baseline machine. On a 6-wide machine with the same cmpp latency, 'extend-all' DPALS achieves no speedup. Such a modest performance gain is due to the fact the acyclic regions are latency-bound: increasing cmpp latency increases the critical path length and reduces the benefits of DPALS.

In contrast to 'extend-all', 'first-fit' DPALS does not extend cmpp latency. Instead, an operation can be placed close to its cmpp. However, it is prohibited from sharing the resource with any other operation if it is scheduled closer than *extendedlatency* cycles to its cmpp. 'First-fit' DPALS on the 4-wide deterministic predicate-aware machine with a cmpp latency of 3 cycles achieves a 4% speedup over the 4-wide baseline machine, and a 1% speedup over the 6-wide baseline machine. By not extending cmpp latency, 'first-fit' DPALS has a better chance of avoiding increasing a region's critical path length, and thereby generally outperforms 'extend-all' DPALS. Of course as the cmpp latency grows, less early reading is possible and less resource sharing will occur in 'first-fit' DPALS, and hence it too will achieve less performance gain relative to baseline scheduler.

DPAMS, as in 'extend-all' DPALS, aggressively extends cmpp latency for each predicated operation in the region whenever that may potentially improve the performance. DPAMS on the 4-wide deterministic predicate-aware machine with a 3 cycle cmpp latency achieves a 10% speedup over the 4-wide baseline machine. This substantial speedup for DPAMS, compared to DPALS, is due to the fact that cyclic region schedules are generally constrained by resources rather than by latencies: DPAS directly targets this resource constraint problem. As we go to a 6-wide machine, the DPAMS speedup with a cmpp latency of 3 cycles is reduced to 4%. Note that both DPALS and DPAMS achieve less speedup on a 6-wide than on a 4-wide machine. This

is due to the fact that a 6-wide machine has more resources than a 4-wide machine and thus resource sharing is less helpful and less is done.

When 'first-fit' DPALS is used on acyclic regions and DPAMS is used on cyclic regions, DPAS achieves a 7% speedup over the entire MediaBench application suite on a 4-wide deterministic predicate-aware machine with a cmpp latency of 3, and 3% for the corresponding 6-wide machine with the same latency. For the 6-wide machine, the performance benefits of DPAS come almost entirely from DPAMS.

Although DPAS is effective for a number of regions, its application is limited only to the regions that contain at least two disjoint operations. Probabilistic predicate-aware scheduling (PPAS), however, can assign *arbitrary* predicated operations to share the same resource in the same runtime cycle. Contrary to DPAS, PPAS does allow more than one predicate to be True in the same cycle, thus resulting in runtime conflicts. Assignment is performed in a probabilistic manner using a combination of predicate profile information and predicate analysis aimed at maximizing the benefits of sharing in view of the expected degree of conflict. The processor pipeline is further modified to detect and recover from such conflicts.

The delay experienced due to conflict is determined by two factors: (i) an application-specific factor, which depends on the frequency with which more than one resource sharing predicated operation has its predicate evaluate to True, and (ii) the machine-dependent conflict detection and recovery latency (CDRL), which depends on whether the first conflicting operation can be dispatched in the conflict cycle itself (CDRL=0) or whether the conflict cycle is lost and the first conflicting operation is not dispatched until the cycle following the conflict cycle (CDRL=1).

As in DPAS, we have proposed and evaluated three probabilistic predicate-aware scheduling algorithms: two probabilistic predicate-aware list scheduling algorithms

('extend-all' PPALS and 'first-fit' PPALS) and one probabilistic predicate-aware modulo scheduling algorithm (PPAMS).

Over all predicated regions, 'extend-all' PPALS on a 4-wide probabilistic predicate-aware machine with cmpp latency of 3 cycles achieves a 3% speedup over the 4-wide baseline machine. On a 6-wide machine with the same cmpp latency, 'extend-all' PPALS achieves no speedup. 'Extend-all' PPALS achieves less than a 1% speedup over 'extend-all' DPALS because the gains from the more aggressive operation sharing offered by 'extend-all' PPALS are reduced by the effect of the increased critical path length due to the 3 cycle cmpp latency.

However, 'first-fit' PPALS, for a 4-wide probabilistic predicate-aware machine with a cmpp latency of 3 cycles, achieves a speedup of 7% over the 4-wide baseline machine. On a 6-wide machine with the same cmpp latency, 'first-fit' PPALS achieves a 1% speedup. The 3% speedup that 'first-fit' PPALS gains over 'first-fit' DPALS on a 4-wide machine is due to the increase in sharing opportunities that PPALS can explore across arbitrary predicated operations while it simultaneously limits the increase in the critical path length, in contrast with 'extend-all' PPALS whose critical path is increased by the latency extension of every cmpp operation on the critical path before scheduling even begins. Furthermore, the PPALS speedup is not very sensitive to CDRL latency and is virtually identical for both CDRL values (0 and 1 cycle).

PPAMS on the 4-wide probabilistic predicate-aware machine with a cmpp latency of 3 cycles achieves substantial speedups of 19% for CDRL=0 and 15% for CDRL=1 over the corresponding 4-wide baseline machine. In general, PPAMS can explore many more sharing opportunities than PPALS; in addition to being able to combine arbitrary operations from a single iteration, it can also combine operations from across different loop iterations. This enables PPAMS to achieve a substantial performance

gain over DPAMS for the same machine with the same cmpp latency. However, PPAMS does lose some performance as CDRL latency increases; due to its more aggressive combining, PPAMS is much more sensitive to the value of CDRL than PPALS. For a CDRL of 1 cycle, PPAMS's performance drops by 3%, relative to a CDRL of 0 cycles on a 4-wide machine. On a 6-wide machine, with a cmpp latency of 3 cycles, PPAMS achieves 7% and 6% speedup with a CDRL of 0 and 1 cycles, respectively. As we have said, the speedup is more modest on a 6-wide machine due to the fact that a 6-wide machine has more resources than a 4-wide machine and its schedules thus do less resource sharing. Because less resource sharing also means fewer resource conflicts on a 6-wide machine, increasing its CDRL up to 1 cycle degrades the performance by only 1%, versus 3.5% for a 4-wide machine.

When 'first-fit' PPALS is used on acyclic regions and PPAMS is used on cyclic regions, PPAS achieves a 14% speedup over our entire MediaBench application suite, on a 4-wide deterministic predicate-aware machine with cmpp latency of 3 cycles and CDRL=0, and 5% for the corresponding 6-wide machine. For CDRL=1, the corresponding speedups are reduced to 11% and 4%.

Given the performance results for both DPAS and PPAS, we believe that it is advantageous to build a PPAS machine, rather than a DPAS machine. The PPAS clearly achieves higher speedup over the baseline than DPAS. On the other hand, the PPAS machine is not substantially more complex to design than the corresponding DPAS machine. Also, as our results indicate, 'first-fit' PPALS is clearly preferable to 'extend-all' PPALS, as 'first-fit' PPALS achieves higher speedup over the baseline than 'extend-all' PPALS for almost all applications.

Finally, to deal effectively with the architected register pressure problem in software-pipelined loops, we have developed a hardware/software mechanism called Register

Queues (RQs). Our experimental results for a benchmark suite of 983 loops from the Perfect Club, SPEC-89, and the Livermore Fortran Kernels show that by decoupling the architected register space from the physical registers, RQs achieves efficient software-pipeline schedules for high operation latencies with almost no increase in architected registers and code size.

We have also found that both DPAMS and PPAMS significantly increase the register requirements of predicated cyclic regions. Therefore, we applied the RQs scheme in conjunction with DPAMS and PPAMS to the predicated cyclic regions that were extracted from the MediaBench applications. If we used rotating registers with 32 architected registers, we could only schedule 87% of these loops with DPAMS and 72% with PPAMS on the 4-wide predicate-aware machine with a cmpp latency of 3 cycles. For the corresponding 6-wide machine, the rotating registers scheme can only schedule 68% with DPAMS and 63% with PPAMS. However, by using RQs with only 32 architected registers, both DPAMS and PPAMS schemes can schedule 99% of these loops on both 4- and 6-wide predicate-aware machines with a cmpp latency of 3 cycles.

5.2 Future Directions

The work presented in this dissertation can be extended in numerous directions. First, both DPAMS and PPAMS extend the cmpp latency of modulo scheduled loops prior to scheduling. As we have seen in Section 2.5 and Section 3.5, for some benchmarks (such as rawcaudio and rawdaudio) increasing the cmpp latency may cause $RecMII$ to exceed the baseline initiation interval. In such cases, the predicate-aware modulo scheduler was not applied since no speedup would be gained despite

the fact that many predicated operations in these loops can still share resources.

The performance of these loops can still be improved with either DPAMS or PPAMS by modifying predicate-aware modulo scheduling to increase the latency of only those cmpp operations that are not in a critical recurrence cycle. One approach is to decide dynamically during scheduling whether an operation's cmpp latency should be extended. The main drawback of such a dynamic scheme is that the decision is local, and does not take into account the cumulative effect on $RecMII$ by other operations.

Another approach would be to perform the pre-pass cmpp latency extension so that cmpp latency is only extended for a subset of operations. The goal here is to perform a global analysis and choose the best selection of operations so that increasing their cmpp latency maximizes the total combining opportunity for these operations, but causes a minimum increase in $RecMII$. One possible metric to assess a total combining opportunity for a set of predicated operations is the cumulative delay due to conflict, which is the sum of individual delays due to conflicts resulting from combining each predicated operation in the set with every other operation in the same set. The combining is only legal if the data dependences between the combined operations are satisfied. Hence the combining opportunity can be computed as the inverse of this cumulative delay.

Furthermore, to accomplish the actual operation selection, one possibility is to use a 0-1 knapsack formulation. A 0-1 knapsack formulation packs a knapsack of a certain capacity with items, each having some weight and some benefit, so that the capacity of the knapsack is not violated and the maximum total benefit is maximized. In this case, the knapsack's capacity would be some value of $RecMII$ that must be less than the initiation interval of the baseline schedule (to assure that some performance gain

over baseline is possible). The total weight of the set of selected operations is the value of $RecMII$ when each operation's cmpp latency is increased. The total benefit of the set of selected operations is their total combining opportunity.

In our experiments, the predicate-aware modulo scheduler (PPAMS) achieves very fast convergence to the dynamic expected initiation interval, $II_{expected}$. However, given the hindsight obtained from these experiments, an even faster search algorithm for $II_{expected}$ can be proposed.

Our current algorithm in Figure 3.17(a) of Section 3.4.4.2 chooses $II_{expected}$ as the midpoint between II_{low} and II_{high}. However, given that the optimum $II_{expected}$ tends to be 10% or less better than baseline II, we might better choose the $II_{expected}$ value to be somewhere around $0.95 \times II_{high} + 0.05 \times II_{low}$ (i.e. at the 95% point of the gap, rather than half way between II_{low} and II_{high}). Whenever a schedule is found we can drop down another 5% of the gap between II_{low} and II_{high} and try again, until no schedule is found. Once no schedule is found, we then set II_{low} to the current $II_{expected}$, set II_{high} to the $II_{expected}$ of the last schedule found, and begin a binary search of the gap between the new II_{low} and II_{high}, as in our current algorithm; however, whenever a schedule is found for $II_{expected}$, we gain a little bit more efficiency by setting II_{high} to the cost of the solution found rather than to the current value of $II_{expected}$, which eliminates the possibility of finding the same schedule twice (or even a slightly worse schedule) later in the search.

Furthermore, in the II_{static} loop of the search algorithm in **ppams_FindSchedule** (Figure 3.17(b)), we iterate II_{static} from the lowest value up to the highest, as in the algorithm shown, but once the very best solution is found by the given algorithm, instead of simply accepting that solution, we try to refine it as follows. At that final value of $II_{expected}$, we iterate through all possible values of II_{static}. If multiple solutions

with various values of II_{static} are found within the final gap, we choose the solution within this final small gap that has the highest value of II_{static}. This solution is chosen because it will have the smallest expected conflict penalty, and thus is expected to achieve the most stable behavior of any schedule with that $II_{expected}$.

Finally, if no solution at all is found, then we run the baseline algorithm and find the best baseline solution - but then rather than simply accepting that solution, we make one more try by setting II_{low} to the baseline bound, and II_{high} to the cost of the baseline solution found. If this gap is not negligibly small, we do a binary search for the best PPAMS solution in this range. This final search might discover one or more PPAMS solutions somewhere within this previously unexplored gap. Note that this gap was generally quite small in our experiments, as the baseline solutions found were in fact quite close to their bounds.

Another direction is to develop an epilogue-conscious predicate-aware modulo scheduler. Currently, no attempt has been made to control the length of a single iteration schedule during the predicate-aware scheduling of a cyclic region: the operations are moved as far as necessary (as long as resource and data constraints are satisfied) if they can be combined to share the same resource. This aggressive combining often results in highly improved predicate-aware initiation interval, but a long loop epilogue. As we have seen, long epilogues can be a problem for loops with a short trip count, and can significantly degrade the overall performance of DPAS and PPAS relative to the baseline schedule. An epilogue-conscious predicate-aware modulo scheduler would try to maximize the degree of operation combining subject to controlling the size of the loop epilogue.

Our work on predicate-aware scheduling can also be extended to deal with data dependences in a probabilistic manner. In a data dependence graph, outgoing edges

from operations with relatively low execution frequency could be removed, thus resulting in a graph with fewer latency constraints and hence, hopefully, more instruction level parallelism and higher performance. The scheduler would ignore these removed edges when scheduling operations, thereby effectively assuming that the predicates of the operations with removed outbound edges will always evaluate to False. Of course, whenever the predicate of one of these operations does evaluate to True at runtime, a recovery process would be required to correctly execute the original code.

The central idea of this dissertation is to achieve higher performance by using existing resources more efficiently. The converse question is: "what is the minimum number of resources required to achieve a given level of performance?" For example, given a machine with a large number of resources, can we eliminate some or, hopefully, most of these resources and achieve the same (or similar) performance as the original machine? To illustrate, consider a machine with N arithmetic units and general purpose register files designed with $2 \times N$ register read ports. This typical configuration is used due to the conservative assumption that in the worst case each of the N operations will require 2 source operands from the register file. In general, however, design for worst case behavior is not the best design, since this behavior may rarely occur in practice. However, register ports are a critical factor in determining cycle delay, as well as the area and power requirements of the register file [47]. We observe that (i) many operations, such as loads, stores and operations with immediates, have fewer than two source operands, (ii) when two consumers of the same register are scheduled in the same cycle, they can share a port, and perhaps most significantly, (iii) many operands are read from the bypass network. Given these three observations, we believe that by using an intelligent scheduling algorithm that can capitalize on these observations, the number of read ports in the register file could be

reduced considerably without noticeable performance loss.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitecture. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248 – 259, June 2000.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–180, January 1983.

[4] Analog Devices. *ADSP-TS001M TigerSHARC DSP product description.*

[5] C. Basoglu, K. Zhao, K. Kojima, and A. Kawaguchi. The MAP-CA VLIW-based media processor. http://equator.com.

[6] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The Cydra-5 minisupercomputer: Architecture and implementation. *Journal of Supercomputing*, 7(1):143–180, May 1993.

[7] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(9):18–27, September 1981.

[8] R. G. Cytron. *Compiler-time Scheduling and Optimization for Asynchronous Machines.* Dept. of Computer Science Report UIUCDCS-R-84-1177, University of Illinois at Urbana-Champaign, Urbana, IL, 1984.

[9] E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel. Effective control for pipelined computers. In *Proceedings of COMPCON*, pages 181–184, February 1975.

[10] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1):181–227, May 1993.

[11] A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 180–191, November 1995.

[12] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, November 1995.

[13] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 75–84, November 1994.

[14] P. Faraboschi, G. Brown, J. A. Fisher, and G. Desoli. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 203–213, June 2000.

[15] M. Fernandes, J. Llosa., and N. Topham. Partitioned schedules for clustered VLIW architectures. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 386–391, March 1998.

[16] M. A. Fernandes. *Clustered VLIW Architecture Based on Queue Register File.* Ph.D. Dissertation, Department of Computer Science, University of Edinburgh, 1998.

[17] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(9):478–490, July 1981.

[18] J. Fisher. Very Long Instruction Word Architectures and the ELI-52. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[19] J. Fritts. *Architecture and Compiler Design Issues in Programmable Media Processors.* Ph.D. Dissertation, Department of Electrical Engineering, Princeton University, 1998.

[20] D. M. Gillies, R. D. Ju, R. C. Johnson, and M. S. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 114–125, December 1996.

[21] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 85–94, November 1994.

[22] P. Y.-T. Hsu. *Highly Concurrent Scalar Processing.* Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1986.

[23] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[24] W. W. Hwu, S. A. Mahlke, W. Y. Chena, P. P Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, May 1993.

[25] IA-64 Application Developer's Architecture Guide. Intel Document 245188, http:// developer.intel.com/design/ia64/, 1999.

[26] Itanium Processor Microarchitecture Reference for Software Optimization. Intel Documents, ftp://download.intel.com/design/Itanium/Downloads/, 2001.

[27] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 100–113, December 1996.

[28] V. Kathail, M. Schlansker, and B. R. Rau. *HPL PlayDoh Architecture Specifications: Version 1.0.* HP Laboratories Technical Report HPL-93-80, 1994.

[29] R. Keller. Lookahead processors. *ACM Computing Surveys*, pages 177–195, December 1975.

[30] K. Kiyohara, S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, S. Anik, and W. W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 247–256, May 1993.

[31] M. Lam. *A Systolic Array Optimizing Compiler.* PhD Dissertation, Carnegie Mellon University, 1986.

[32] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–327, 1988.

[33] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[34] J. Llosa, M. Valero, J. Fortes, and E. Ayguade. Using sacks to organize registers in VLIW machines. In *Proceedings of International Conference on Parallel and Vector Processing*, pages 628–639, September 1994.

[35] S. A. Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches.* Ph.D. Dissertation, Department Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1996.

[36] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[37] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. In *Proceedings of the International Conference on Supercomputing*, pages 260–271, July 1992.

[38] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[39] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206 – 218, June 1997.

[40] J. C. H. Park and M. S. Schlansker. *On predicated execution.* HP Laboratories Technical Report HPL-91-58, 1991.

[41] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[42] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: computer techniques and architectural support. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 131–139, 1982.

[43] B. R. Rau, P. J. Kuekes, and C. D. Glaeser. *A Statically Scheduled VLSI Interconnect for Parallel Processors.* Computer Science Press, 1981.

[44] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[45] B. R. Rau, Michael S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Michorarchitecture*, pages 158–169, December 1992.

[46] B. R. Rau, D. W. L. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 1(22):12–34, January 1989.

[47] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 375–386, January 2000.

[48] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, Vol. 20, No. 5:24–43, September/October 2000.

[49] J. W. Sias, D. I. August, and W. W. Hwu. Accurate and efficient predicate analysis with binary decision diagrams. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 112–123, December 2000.

[50] M. Smelyanskiy, S. A. Mahlke, and E. S. Davidson. Probabilistic predicate-aware modulo scheduling. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, March 2004. To appear.

[51] M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H. H. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 169–178, March 2003.

[52] M. Smelyanskiy, G. S. Tyson, and E. S. Davidson. Register queues: A new hardware/software approach to efficient software pipelining. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–12, October 2000.

[53] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, June 1982.

[54] M. G. Stoodley and C. G. Lee. Software pipelining loops with conditional branches. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 262–273, December 1996.

[55] The Trimaran System. www.trimaran.org, 1999.

[56] TMS320C62x/67x CPU and Instruction Set Reference Guide. Texas Instruments Documents, http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf, 1998.

[57] R. Towle. *Control and Data Dependence for Program Transformations*. Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 1976.

[58] G. S. Tyson. *Evaluation of a Scalable Decoupled Microprocessor Design*. Ph.D. Dissertation, University of California - Davis, 1997.

[59] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, 1992.

[60] N. J. Warter and N. Partamian. Modulo scheduling with multiple initiation intervals. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 111–118, 1995.

[61] W. A. Wulf. Evaluation of the WM architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 382–390, May 1992.

[62] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Improved spill code generation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'2000 Conference on Programming Language Design and Implementation*, pages 134–144, June 2000.