UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MAURÍCIO LIMA PILLA

RST: Reuse through Speculation on Traces

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Philippe Olivier Alexandre Navaux
Advisor

Prof. Dr. Felipe Maia Galvão França
Coadvisor

Porto Alegre, June 2004

CIP – CATALOGING-IN-PUBLICATION

Pilla, Maurício Lima

RST: Reuse through Speculation on Traces / Maurício Lima Pilla. – Porto Alegre: PPGC da UFRGS, 2004.

174 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2004. Advisor: Philippe Olivier Alexandre Navaux; Coadvisor: Felipe Maia Galvão França.

1. Speculative Trace Reuse, Superscalar Architectures, Parallel Processing, Value Reuse, Value Prediction. I. Navaux, Philippe Olivier Alexandre. II. França, Felipe Maia Galvão. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof^{a}. Wrana Maria Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitora Adjunta de Pós-Graduação: Prof^{a}. Jocélia Grazia
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

To those that were patient enough to tolerate my bad times.

ACKNOWLEDGMENTS

That's the part where I speak out of all the things that cannot be said on the other Chapters but must appear somewhere or I'll burst. I've spent some years on this process, and the least I can do for myself is to distribute part of the credits for finishing my PhD thesis. With the credits goes part of the burden, hence this is just an exercise of selfishness...

Easy to know that I am doomed to forget to thank some very important person here in the acknowledgments, or not give enough credit for another one. Therefore, before I start writing, I must say that I am already sorry for forgetting you – you who should be here but are not. By the way, how many pages can I write just for the acknowledgments? I guess that somebody must have asked this before, but I couldn't care less (wanna cookie?).

Most of the important people in my life just don't have the slightest idea of how important they are for me. Thanks to everybody that have made my life a little bit more bearable when I was too bored and wanting to run away from the graduate life. Special thanks to the 249's crew, Mozart, Émerson, Tatiana, Roberta, Bohrer, Mônica, Ricardo, Patrícia, Nicolas, Juliana, and everybody else that lost his or her time sipping coffee and eating chocolate in our most famous coffee breaks. Gotta thank the friends from the outside world too, like Nácul, Mattia, Thais, Paulo. I'd like to thank my dear dog, Toby, which was always happy to see me regardless of how much meat I had for him and never bothered me with problems. And I can't forget my love Vivi. She has just recently entered my life, but was able to make it much more enjoyable.

There are not enough words to acknowledge the crew from my research group for all that they have done for me. Rafael, Bohrer, Mozart, Tatiana (yes, there are repeated names here), Pizzol, and all the other, please accept my gratitude for your help in many aspects of my graduate student life.

Thanks to the sysadmins that never made my work too hard (nor too easy). I'd like also to thank the mysterious Mr. L.O., head of the labs, for all the times that he had to find me some parts to replace faulty hardware. Special thanks to Silvania and Sula, that constantly had to endure me asking for my advisor or for some fix on the bloody air-conditioning.

This thesis is also for my advisor, Dr. Navaux, that trusted me and my qualifications to be enough to enter a PhD program without going for a Master first, and patiently heard my complains and guided me through the shadows of the thesis. Thanks also to Dr. Geyer and Dr. Ana Price, that also believed me to be fit for the work. Thanks also to my co-advisor, Dr. França, and to Dr. da Costa, which helped me out to determine my thesis subject and pointed me out the people to whom I

should talk about it.

And I'll never forget the people from Pittsburgh, Dr. Soffa, Dr. Childers, Dr. Mock, Dr. Mossé, Naveen, Min, Xiang, Ricardo, Mirisolla, Carol, Evandro, Carolina, and all the other nice people I've met there. They made my graduate internship there one of the most valuable experiences that I'll keep for the rest of my life.

Finally, to my family, which helped me to abstract some financial issues and concentrate on the thesis, besides all the other not-so-evident helps that we are not able to understand until we are far away from them.

From here, I must assume the hard task of finding my way through all the paths that life is opening for me. For each choice, I have a 50/50 chance of being right (or being wrong). Where's that bloody coin?

Advertisement: this work has been produced with grants from CNPq, CAPES, LabTeC/Dell, and HP projects, as well as with some parental funding.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 11

LIST OF FIGURES 13

LIST OF TABLES 17

ABSTRACT 19

RESUMO 21

1 INTRODUCTION 23

2 PREVIOUS WORK 27
2.1 Superscalar and VLIW architectures 27
2.1.1 Data dependencies 30
2.1.2 Control dependencies 30
2.1.3 Resource conflicts 31
2.2 Value locality and repetition 32
2.3 Value prediction 33
2.3.1 Common steps of value prediction techniques 34
2.3.2 Last value prediction 35
2.3.3 Stride prediction 36
2.3.4 Trace-level prediction 37
2.3.5 Speculation control and confidence 37
2.3.6 Recovery mechanisms for mispeculations 39
2.4 Value reuse 40
2.4.1 Common steps of value reuse techniques 42
2.4.2 Value reuse mechanisms 42
2.4.3 Instruction reuse 42
2.4.4 Basic block reuse 44
2.4.5 Trace reuse 44
2.5 Combined value reuse and prediction 45
2.5.1 Result cache based mechanisms 45
2.5.2 Region based mechanisms 46
2.5.3 Instruction reuse based mechanisms 46

3 REUSE THROUGH SPECULATION ON TRACES 47
3.1 Motivation 47
3.2 Reuse through Speculation on Traces 49
3.3 Trace construction and reuse 50
3.3.1 Reuse domains and memoization tables 51
3.3.2 Trace construction and reuse for DTM 54
3.3.3 Trace construction and reuse for RST 56
3.3.4 Misprediction detection and recovery 57
3.4 Comparison with related techniques 57
3.5 Contributions 58

4 A RST ARCHITECTURE 61
4.1 DTM's pipeline 61
4.2 RST's pipeline 62
4.2.1 Stage RS1 63
4.2.2 Stage RS2 63
4.2.3 Stage RS3 67
4.2.4 Stage RS4 70
4.2.5 Integrating all stages 71

5 EXPERIMENTAL ENVIRONMENT 73
5.1 Simulator 73
5.2 Workload 74
5.3 Baseline configuration 75

6 LIMITS OF SPECULATIVE TRACE REUSE 79
6.1 RST with maximum table sizes 80
6.1.1 Speedup for DTM (DTC an RTC) 80
6.1.2 Speedup 80
6.1.3 Contribution to committed instructions 82
6.1.4 Trace length 84
6.1.5 Branches per trace 84
6.1.6 Input and output scope sizes 86
6.1.7 Critical paths 86
6.1.8 Reason for finishing trace formation 88
6.1.9 Remarks for this Section 89
6.2 Memoization table associativity 90
6.2.1 Speedup 90
6.3 Memoization table size 91
6.3.1 Speedup 92
6.4 Number of inputs and outputs 92
6.4.1 Speedup 94
6.5 Number of predicted inputs 95
6.5.1 Speedup 95
6.6 Varying pipeline depth 97
6.6.1 Speedup varying fetch depth 97
6.6.2 Speedup varying dispatch depth 98
6.6.3 Speedup varying issue depth 98

6.6.4 Speedup varying writeback depth 98
6.6.5 Speedup without superpipeline 100
6.6.6 Remarks for this Section 100
6.7 Varying pipeline width 101
6.7.1 Speedup 101
6.8 Varying number of functional units 102
6.8.1 Speedup 102
6.9 Varying cache configurations 103
6.9.1 Speedup 104
6.10 Stride prediction 104
6.10.1 Speedup 105
6.11 Varying reuse domains 106
6.11.1 Speedup 107
6.11.2 Reuse contribution to committed instructions 107
6.12 Summary 109

7 RESULTS FOR A RST ARCHITECTURE 111
7.1 RST without confidence mechanisms 111
7.1.1 Speedup 112
7.1.2 Misprediction rates 113
7.1.3 Mispeculation penalty 113
7.1.4 Remarks about RST without confidence mechanisms 114
7.2 RST with oracle confidence 115
7.2.1 Speedup 115
7.3 RST with simple confidence mechanisms 116
7.3.1 Speedup with limitation on the number of alternative traces 117
7.3.2 Speedup with limitation on the minimal critical path 117
7.3.3 Prediction rates with limitation on the minimal critical path 119
7.3.4 Speedup with limitation by the number of ready sources 120
7.3.5 Remarks for this Section 121
7.4 RST with confidence counter 4096 3 3 3 1 3 121
7.4.1 Speedup 122
7.4.2 Reuse rate 123
7.4.3 Confidence reliability 126
7.4.4 Misprediction rates and penalties 126
7.4.5 Adding path information to confidence 128
7.5 RST with confidence counter 4096 7 7 7 1 0 128
7.5.1 Speedup 129
7.6 Confidence table sizes 129
7.6.1 Speedup 129
7.6.2 Speedups for a 6-stage pipeline 130
7.7 RST with stride prediction and last-value prediction 131
7.7.1 Speedup 131
7.8 RST with confidence by trace 132
7.8.1 Speedup 133
7.9 Comparison to alternatives 134
7.9.1 Speedups over a doubled first-level cache 134
7.9.2 Speedups over an instruction reuse technique 135

7.9.3 Other comparisons 135
7.10 Summary 136

8 CONCLUSION AND FUTURE WORK 137
8.1 Thesis summary 137
8.2 Conclusions from the limits study 138
8.3 Conclusions from the RST architecture study 139
8.4 Contributions 140
8.5 Future works 141

9 RESUMO DA TESE E SUAS CONTRIBUIÇÕES 143
9.1 Resumo da Tese 143
9.2 Análise do estudo de limites 144
9.3 Análise do estudo da arquitetura 146
9.4 Contribuições 147
9.5 Trabalhos futuros 147

REFERENCES 149

APPENDIX A PUBLISHED PAPERS 157

LIST OF ABBREVIATIONS AND ACRONYMS

AM	Arithmetic Mean
BHB	Block History Buffer
CDP	Combined Dynamic Prediction
CRB	Computation Region Buffer
CISC	Complex Instruction Set Code
CMOS	Complementary Metal-Oxide Semiconductor
DTC	DTM Trace Construction policy
DTM	Dynamic Trace Memoization
DTMm	Dynamic Trace Memoization with Memory reuse
EPIC	Explicitly Parallel Instruction Computing
FA	Fully Associative
FP	Floating Point
HM	Harmonic Mean
IA	Intel Architecture
IBM	International Business Machines
icr	Input Context Registers
icv	Input Context Values
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LRU	Least Recently Used
LSQ	Load/Store Queue
LVPT	Load Value Prediction Table
MIPS	Microprocessor without Interlocked Pipeline Stages
npc	Next Program Counter
ocr	Output Context Registers

ocv	Output Context Values
PC	Program Counter
PISA	Portable Instruction Set Architecture
RAW	Read After Write
RB	Reuse Buffer
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
RST	Reuse through Speculation on Traces
RSTm	Reuse through Speculation on Traces with Memory reuse
RT	Recovery Table
RTC	RST Trace Construction policy
RUU	Register Update Unit
SPEC	Standard Performance Evaluation Corporation
SRC	Speculative Result Cache
VLIW	Very Large Instruction Word

LIST OF FIGURES

Figure 2.1: Conceptual superscalar pipeline 28
Figure 2.2: Organization of a superscalar pipeline 28
Figure 2.3: Example of control dependency 30
Figure 2.4: Control dependency and independency 31
Figure 2.5: Last value predictor 35
Figure 2.6: Load value prediction 36
Figure 2.7: Stride predictor 37
Figure 2.8: Value predictor based on traces 38
Figure 2.9: A simple automate for branch prediction 38
Figure 2.10: Value prediction with confidence 39
Figure 2.11: A reorder buffer 40
Figure 2.12: Generic Reuse Buffer 43
Figure 2.13: Implementation of a SRC table 45

Figure 3.1: Comparison of traces reused and not reused in DTM 48
Figure 3.2: Entry in Memo_Table_G, no memory accesses allowed 51
Figure 3.3: Entry in Memo_Table_G, memory accesses allowed 52
Figure 3.4: Entry in Memo_Table_T, no memory accesses allowed 52
Figure 3.5: Entry in Memo_Table_T, memory accesses allowed 53
Figure 3.6: Precedence of different reuse types in RST 53
Figure 3.7: Code snippet 54
Figure 3.8: Storing reusable instructions 54
Figure 3.9: Trace formation 55
Figure 3.10: Trace construction, reusing a trace 55
Figure 3.11: Storing reusable instructions 56
Figure 3.12: Trace construction, reusing a trace 57

Figure 4.1: Pipeline for a DTM architecture 62
Figure 4.2: Pipeline for a RST architecture 63
Figure 4.3: Details of stage RS1 64
Figure 4.4: Details of stage RS2 65
Figure 4.5: Details of stage RS3 67
Figure 4.6: Recovery Table entry 68
Figure 4.7: RT with one entry 69
Figure 4.8: RT with three entries 69
Figure 4.9: Resolving a RT entry 70
Figure 4.10: Details of stage RS4 70
Figure 4.11: Connections for RST stages 72

Figure 5.1: Superpipeline for baseline architecture 77

Figure 6.1: Speedups over baseline for DTM (RTC and DTC), Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 81
Figure 6.2: Speedup over baseline, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 81
Figure 6.3: Speedup over DTM, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 82
Figure 6.4: Contribution to committed instructions, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 83
Figure 6.5: Average trace length for RST and DTM, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 85
Figure 6.6: Average number of branches per trace, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 85
Figure 6.7: Average number of trace inputs, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 86
Figure 6.8: Average number of trace outputs, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 87
Figure 6.9: Average critical paths for reused traces, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 87
Figure 6.10: Average reason for finishing traces, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 88
Figure 6.11: Average reason for finishing stored traces, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative 89
Figure 6.12: Speedup for RST over baseline, varying Memo_Table_T associativity 90
Figure 6.13: Speedup for RST (DTC) over baseline, varying Memo_Table_T associativity 91
Figure 6.14: Speedup for DTM over baseline, varying Memo_Table_T associativity 92
Figure 6.15: Speedup over baseline architecture varying Memo_Table_T size, Memo_Table_T in31 out31 FA, Memo_Table_G 16384 entries FA 93
Figure 6.16: Speedup over the baseline architecture, Memo_Table_T 512 entries (31 outputs) 4-way, Memo_Table_G 2048 entries 4-way, varying number of inputs 94

Figure 6.17: Speedup over the baseline architecture, Memo_Table_T 512 entries (31 inputs) 4-way, Memo_Table_G 2048 entries 4-way, varying number of outputs . . . 95
Figure 6.18: Speedup of RST (RTC) over the baseline architecture, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way, varying the number of predictable inputs . . . 96
Figure 6.19: Speedup of RST (DTC) over the baseline architecture, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way, varying the number of predictable inputs . . . 96
Figure 6.20: Speedup of RST (RTC, VP3) over DTM, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 97
Figure 6.21: Speedup over baseline architecture varying fetch depth, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 98
Figure 6.22: Speedup over baseline architecture varying dispatch depth, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 99
Figure 6.23: Speedup over baseline architecture varying issue depth, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 99
Figure 6.24: Speedup over baseline architecture varying writeback depth, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 100
Figure 6.25: Speedup over baseline architecture with original *sim-outorder*'s pipeline, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 101
Figure 6.26: Speedup over baseline with 2-wide pipeline, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 102
Figure 6.27: Speedup over baseline with original number of functional units, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 103
Figure 6.28: Speedup over baseline varying first-level cache size, VP31 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 104
Figure 6.29: Speedup over baseline architecture, RST with and without stride prediction, VP1 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 105
Figure 6.30: Speedup over baseline architecture, RST with and without stride prediction, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way . . . 106
Figure 6.31: Speedups over baseline, VP31 Memo_Table_T 1024 entries (31 inputs, 31 outputs) FA, Memo_Table_G 4096 entries FA . . 107

Figure 6.32: Speedups over baseline for RST, VP31 Memo_Table_T 1024 entries (31 inputs, 31 outputs) FA, Memo_Table_G 4096 entries FA . . . 108
Figure 6.33: Reuse contribution to committed instructions for RST (with and without memory reuse), Memo_Table_T 1024 in31 out31 FA VP31, Memo_Table_G 4096 FA . . . 108

Figure 7.1: Speedup over DTM architecture, without confidence mechanisms 112
Figure 7.2: Misprediction penalties, RST with DTC policy . . . 114
Figure 7.3: Misprediction penalties, RST with RTC policy . . . 115
Figure 7.4: Speedup for RST over baseline with perfect confidence . . . 116
Figure 7.5: Speedup for RST over DTM with perfect confidence . . . 117
Figure 7.6: Speedup over DTM, varying the maximum number of alternative traces . . . 118
Figure 7.7: Speedup over DTM, varying the minimum critical path . . . 118
Figure 7.8: Traces predicted or mispredicted, varying the minimum critical path . . . 119
Figure 7.9: Speedup of RST (DTC) over DTM, restricting by number of ready sources . . . 120
Figure 7.10: Speedup over baseline architecture, counter 4096 3 3 3 1 3 . . . 122
Figure 7.11: Speedup over DTM, counter 4096 3 3 3 1 3 . . . 123
Figure 7.12: Instructions reused (isolated or in traces), counter 4096 3 3 3 1 3 124
Figure 7.13: Instructions bypassed by traces, counter 4096 3 3 3 1 3 . . . 125
Figure 7.14: Reuse contribution to committed instructions, counter 4096 3 3 3 1 3125
Figure 7.15: Percent of correct confidence lookups, counter 4096 3 3 3 1 3 . . . 126
Figure 7.16: Cumulative distribution of misprediction penalties, RST (VP1) confid 4096 3 3 3 1 3 . . . 127
Figure 7.17: Speedup over baseline, counter 4096 7 7 7 1 0 . . . 129
Figure 7.18: Speedup over baseline architecture varying confidence table sizes, RST 3 3 3 1 3 . . . 130
Figure 7.19: Speedup over the baseline with 6-stage pipeline, Memo_Table_T 512 in4 out4 4-assoc VP2, Memo_Table_G 2048 4-assoc . . . 131
Figure 7.20: Speedups of strided RST 4096 3 3 3 1 3 over DTM . . . 132
Figure 7.21: Speedup over DTM, RST (DTC) VP1 and VP2, confidence by trace 3 3 3 1 3 . . . 133
Figure 7.22: Comparison of speedups over baseline for a doubled-l1 cache architecture and DTM/RST architectures . . . 134
Figure 7.23: Speedup over a instruction reuse architecture, RST VP2 . . . 135

LIST OF TABLES

Table 4.1: Sequence of instructions in the rename stage 65
Table 4.2: Mappings from register renaming 66
Table 4.3: Discovering mappings to be freed 66
Table 4.4: State after freeing unnecessary mappings 67

Table 5.1: Simulated workloads – SPEC95int 74
Table 5.2: Simulated workloads – SPEC2000int 74
Table 5.3: Main parameters for baseline architecture 76
Table 5.4: Memory hierarchy for baseline architecture 76
Table 5.5: Functional units for baseline architecture 77
Table 5.6: Memoization tables 78

Table 6.1: Configuration for maximum table size 80
Table 6.2: Configuration for Memo_Table_T and Memo_Table_G, experiments with number of inputs, outputs, predicted inputs 93
Table 6.3: Configuration for Memo_Table_T and Memo_Table_G, experiments with memory reuse 106

Table 7.1: Configuration for Memo_Table_T and Memo_Table_G for Chapter 7 112
Table 7.2: Misprediction rates for RST 113
Table 7.3: Confidence configuration for counter 4096 3 3 3 1 3 121
Table 7.4: Misprediction rates for RST counter 4096 3 3 3 1 3 127
Table 7.5: Average speedups over baseline varying path depth, RST (VP2) confid 4096 3 3 3 1 3 128
Table 7.6: Confidence configuration for counter 4096 7 7 7 1 0 128

ABSTRACT

In this thesis, we present a novel approach to combine both reuse and prediction of dynamic sequences of instructions called **Reuse through Speculation on Traces** (RST). Our technique allows the dynamic identification of instruction traces that are redundant or predictable, and the reuse (speculative or not) of these traces. RST addresses the issue, present on Dynamic Trace Memoization (DTM), of traces not being reused because some of their inputs are not ready for the reuse test. These traces were measured to be 69% of all reusable traces in previous studies.

One of the main advantages of RST over just combining a value prediction technique with an unrelated reuse technique is that RST does not require extra tables to store the values to be predicted. Applying reuse and value prediction in unrelated mechanisms but at the same time may require a prohibitive amount of storage in tables. In RST, the values are already stored in the Trace Memoization Table, and there is no extra cost in reading them if compared with a non-speculative trace reuse technique. The input context of each trace (the input values of all instructions in the trace) already stores the values for the reuse test, which may also be used for prediction.

Our main contributions include: *(i)* a speculative trace reuse framework that can be adapted to different processor architectures; *(ii)* specification of the modifications in a superscalar, superpipelined processor in order to implement our mechanism; *(iii)* study of implementation issues related to this architecture; *(iv)* study of the performance limits of our technique; *(v)* a performance study of a realistic, constrained implementation of RST; and *(vi)* simulation tools that can be used in other studies which represent a superscalar, superpipelined processor in detail.

In a constrained architecture with realistic confidence, our RST technique is able to achieve average speedups (harmonic means) of 1.29 over the baseline architecture without reuse and 1.09 over a non-speculative trace reuse technique (DTM).

Keywords: Speculative Trace Reuse, Superscalar Architectures, Parallel Processing, Value Reuse, Value Prediction.

RST: Reuso Especulativo de *Traces*

RESUMO

Na presente tese, apresentamos uma nova abordagem para combinar reuso e previsão de seqüências dinâmicas de instruções, chamada **Reuso por Especulação em Traces** (RST). Esta técnica permite a identificação dinâmica de *traces* de instruções redundantes ou previsíveis e o reuso (especulativo ou não) desses *traces*. RST procura resolver a questão de *traces* que não são reusados por seus valores de entradas não estarem prontos para o teste de reuso, observada na Memorização Dinâmica de *Traces* (DTM). Em estudos anteriores, esses *traces* foram contabilizados como sendo cerca de 69% de todos os *traces* reusáveis.

Uma das maiores vantages de RST sobre a combinação de um mecanismo de previsão com uma técnica de reuso de valores em que os mecanismos não são relacionados é que RST não necessita de tabelas adicionais para o armazenamento dos valores a serem previstos. A aplicação de reuso e previsão de valores pela simples combinação de mecanismos pode necessitar de uma quantidade proibitiva de espaço de armazenamento. No mecanismo RST, os valores já estão presentes na Tabela de Memorização de Traces, não incorrendo em custos adicionais para lê-los se comparado com uma técnica não-especulativa de reuso de *traces*. O contexto de entrada de cada *trace* (os valores de entrada de todas as instruções contidas no *trace*) já armazenam os valores para o teste de reuso, os quais podem ser também utilizados para previsão de valores.

As principais contribuições de nosso trabalho incluem: *(i)* um *framework* de reuso especulativo de *traces* que pode ser modificado para diferentes arquiteturas de processadores; *(ii)* definição das modificações necessárias em um processador superescalar e *superpipeline* para implementar nosso mecanismo; *(iii)* estudo de questões de implementação relacionadas à essa arquitetura; *(iv)* estudo dos limites de desempenho da nossa técnica; *(v)* estudo de desempenho de uma implementaçãode RST limitada por fatores realísticos; e *(vi)* ferramentas de simulação que podem ser utilizadas em outros estudos, representando um processador superescalar e *superpipeline* em detalhes.

Salientamos que, em uma arquitetura utilizando mecanismos realistas de estimativa de confiança das previsões, nossa técnica RST consegue atingir *speedups* médios (médias harmônicas) de 1.29 sobre uma arquitetura sem reuso e 1.09 sobre uma técnica não-especulativa de reuso de *traces* (DTM).

Palavras-chave: Arquiteturas Superescalares, Processamento Paralelo, Reuso de Valores, Previsão de Valores.

1 INTRODUCTION

"I too am a Seeker," said Kim, using one of the lama's pet words.
"Though" – he forgot his northern dress for the moment–
"though Allah alone knoweth what I seek."

Rudyard Kipling, "Kim"

Control and true data dependencies are major impediments to obtaining better performance in modern processor architectures. Simply increasing available resources in order to improve performance may not be the best choice, because the increased hardware complexity may significantly impact cycle time. Hence, other ways of increasing performance, like the exploitation of instruction-level parallelism, are necessary in order to enhance the computing power of general-purpose processors.

Data dependencies also limit the exploitation of instruction-level parallelism, delineating the upper bound performance that can be obtained by using better branch prediction, larger caches, deeper pipelines, and more functional units. Therefore, new techniques that can successfully overcome these limits, and at the same time are not overly complex to be implemented, are necessary to further improve processor performance.

Two families of such techniques, although very distinct in nature and implementation, have been the subject of many studies in the last years: value reuse and value prediction.

Value reuse is a non-speculative way of exploiting the redundancy found in the execution of most programs. After the execution of a given set of computations, stored inputs and results may be used to avoid execution of redundant computations. If the inputs of a previously seen instance of the computations match the current architecture state, it may be reused and its outputs are directly written to the output registers, thus bypassing pipeline stages and saving important resources for other instructions.

Value prediction, on the other hand, uses previously seen values to estimate the next values, postponing the comparison to a point after the value is produced. But sometimes the value may not be the same as predicted, and in these cases the architecture state must be rolled back to its state before the misprediction. In this way, value prediction may allow computations to start earlier, but it may also cause disruptions to the execution flow and waste of resources when mispredictions happen. On the other hand, value reuse does not allow computation sets to be reused when

there are unknown inputs by the reuse test, but it does not suffer from mispeculations. Value prediction also occupies more resources because of mispeculations, while reused instructions may free many resources as they are not executed at all.

A balance between the two families of techniques, where resources are not greedily consumed by speculative instructions but that results can be provided to instructions as soon as possible, would be ideal.

Techniques of trace reuse (COSTA, 2001) have been proposed to take advantage of dynamic sequences of instructions with shared inputs and outputs. Even though these techniques can improve performance by reusing traces that encapsulate data and control dependencies, many of the reusable traces are not reused because some of their inputs are not known during the reuse test, and thus they cannot be compared to the stored values.

In many cases, these traces could be reused if the inputs were known, allowing further performance increases. For these cases, value prediction could be used to postpone the test of inputs to a later time, when the values are known.

Some approaches for combined value reuse and prediction as proposed in (LIAO; SHIEH, 2002) have the shortcoming of mixing two unrelated techniques, resulting in a large increase on hardware complexity. Other proposals, such as (HUANG; LILJA, 1999; WU; CHEN; FANG, 2001), do not have the generality to be used in different processor architectures and do not provide legacy compatibility, requiring modification on compilers and recompilation of applications which usually are not desirable or feasible in production environments.

In this work, we introduce **Reuse through Speculation on Traces** (RST), a novel approach for dynamic and speculative reuse of traces. RST addresses the major challenge of achieving significant performance improvements from simple trace reuse with just a small additional amount of hardware.

The advantage of our integrated approach over combining other value prediction and value reuse mechanisms is clear: there is no need for additional tables to store values to be predicted, because they are already stored in a memoization table. RST can speculatively reuse traces that are already fetched from the memoization table, regardless of having all their inputs ready or not. Therefore, RST does not increase pressure on the memoization tables if the same trace construction policies of trace reuse are employed.

For the recovery of mispredictions, the same mechanism used for branch mispredictions (squashing all instructions after the misprediction) can be used, dispensing another execution engine to deal with mispredictions such as required by other techniques (NAKRA; GUPTA; SOFFA, 1999; WU; CHEN; FANG, 2001; KOUSHIRO; SATO; ARITA, 2003).

RST is not so intensive on resource usage as traditional value prediction, because speculatively reused traces are not executed, but only reused. Hence, although instructions following a speculatively reused trace execute in a speculative way, the resource occupation is smaller than if the instructions inside the trace were also to be speculatively executed. Trace reuse (speculative or not) allows to bypass dispatch, issue, and execute stages, freeing important resources and providing inputs for other instructions sooner than in an architecture without reuse.

Our approach does not require modifications in the instruction set, in the compiler or in the operating system, allowing for integral support of legacy applications. Compiler and profiler support can be used for further improvements in performance,

but they are not necessary to obtain speedups as our results will show.

RST can be combined with other mechanisms designed to facilitate exploitation of instruction level parallelism, such as branch prediction, trace caches and others. The advantages of RST over non-speculative reuse techniques are also clear: many reusable computations that would not be considered because of not ready inputs are now eligible for speculative reuse.

Besides all of these advantages over non-speculative trace reuse techniques and value prediction mechanisms, RST also features most of the advantages of trace reuse, as collapsing data and control dependencies. RST reduces the pressure on the dispatch, issue, and execute stages, bypassing them when a trace is reused (speculatively or not). As in DTM, RST's pipeline is implemented in parallel with the instruction pipeline. Thus, RST should not decrease the clock rate by the addition of extra logic to the critical path in the instruction pipeline.

In this thesis, we developed a novel speculative trace reuse framework that can be modified to fit different architectures and requires minimal extra hardware compared to regular trace reuse. RST is not restricted to superscalar architectures and its concepts can potentially be implemented in other kinds of processor architectures such as VLIW processors.

To test this framework, we deployed a detailed superscalar simulator representing an implementation of RST on a superpipelined processor, and we simulated several configurations – both for a limits study and a realistic processor. Using this simulator, we were able to present new insights and to study the effects of both trace reuse and speculative reuse in face of new constraints not approached by previous work.

Many different parameters are varied in this work, like cache sizes, pipeline lengths, memoization table sizes, confidence strategies, reuse domains, number of inputs and outputs for traces, and number of predicted inputs.

In our experiments, RST reached average speedups (harmonic means) of 1.29 over a baseline architecture without reuse or prediction, and 1.09 over an architecture with non-speculative trace reuse when realistic configurations and confidence mechanisms are simulated. RST was also able to outperform alternatives such as doubling the first-level caches and only reusing instructions employing the same on-chip area that was necessary to implement the reuse tables of RST's configuration.

Our main contributions can be resumed as follows: *(i)* a speculative trace reuse framework that can be adapted to different processor architectures; *(ii)* definition of the modifications in a superscalar, superpipelined processor in order to implement our mechanism; *(iii)* study of implementation issues related to this architecture; *(iv)* study of the performance limits of our technique; *(v)* a performance study of a realistic, constrained implementation of RST; and *(vi)* simulation tools that can be used in other studies which represent a superscalar, superpipelined processor in detail.

This work is organized as follows. In Chapter 2, we introduce value reuse and value prediction, presenting the most significant related work. After that, RST is proposed in Chapter 3. In Chapter 4, we depict an architecture implementing RST over a superscalar architecture. Chapter 5 presents the simulation tools and workloads used for this work.

In the next two chapters, we discuss simulation results. Chapter 6 presents a limits study, where we start with an architecture with lots of resources and perfect confidence estimation, and then we constrain certain parameters to discover per-

formance limits of RST. Then, in Chapter 7 we show our results for constrained configurations with real or no confidence mechanisms.

Finally, we draw conclusions and discuss future work in Chapter 8, and present a summary of this thesis written in Portuguese in Chapter 9 as required by the Graduate Program.

2 PREVIOUS WORK

"Be careful whose advice you buy, but be patient with those who supply it. Advice is a form of nostalgia. Dispensing it is a way of fishing the past from the disposal, wiping it off, painting over the ugly parts and recycling it for more than it's worth."

Mary Schmich, "Everybody's Free to Wear Sunscreen"

In this Chapter, we present definitions and previous work that are required to understand this thesis. We start presenting superscalar and VLIW architectures in Section 2.1. Then, we introduce the fundamentals on how value reuse and value prediction work in Section 2.2.

In Section 2.3, we present value prediction, and after that, we introduce value reuse in Section 2.4. Finally, we present techniques that employ both value reuse and value prediction in Section 2.5.

2.1 Superscalar and VLIW architectures

Superscalar Architectures are characterized by the possibility of simultaneous execution of multiple scalar instructions (JOHNSON, 1991; HWANG, 1993; SMITH; SOHI, 1995; FLYNN, 1995; DE ROSE; NAVAUX, 2003). These architectures have multiple functional units, fed by an instruction pipeline. Superscalar architectures have been developed since the beginning of the 90's and, even if they are considered by some researches as an extension of RISC architectures, their implementations usually present a tendency towards increasing complexity.

Figure 2.1 shows the conceptual design of a superscalar pipeline (SMITH; SOHI, 1995). Instructions from a static program are fetched into the pipeline by the fetch stage, where branch prediction is used in order to reduce the impact of control dependencies on performance. Then, instructions are decoded and dispatched. In this process, dependencies among instructions are detected. After that, the instructions wait for their inputs to be ready, and they may be executed in a different order than that they were fetched, as long as true dependencies are respected. The issue stage is responsible for sending instructions with ready inputs to the functional units for execution. Finally, the last stage is responsible for reordering and committing non-speculative instructions, in order to keep the semantics of a sequential program.

Figure 2.2 presents the typical organization of a superscalar pipeline (SMITH; SOHI, 1995). Fast memories are necessary to keep the pipeline feed, as the clock rate in the microprocessor core is many times faster than the current memory tech-

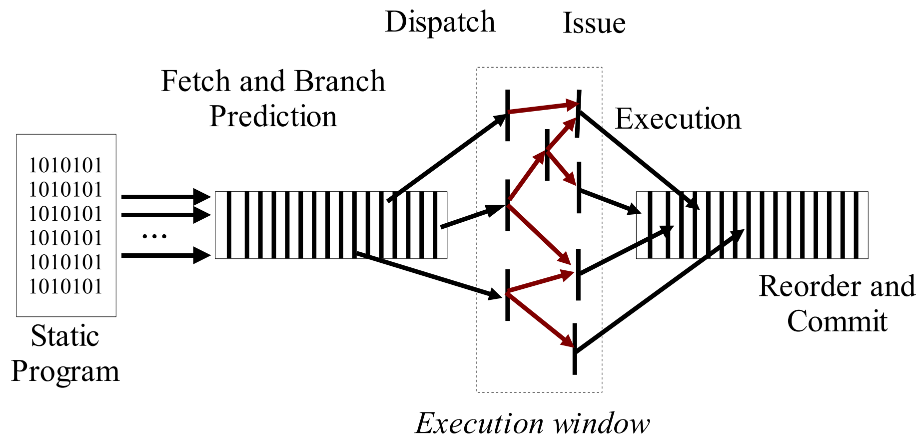

Figure 2.1: Conceptual superscalar pipeline

nologies.

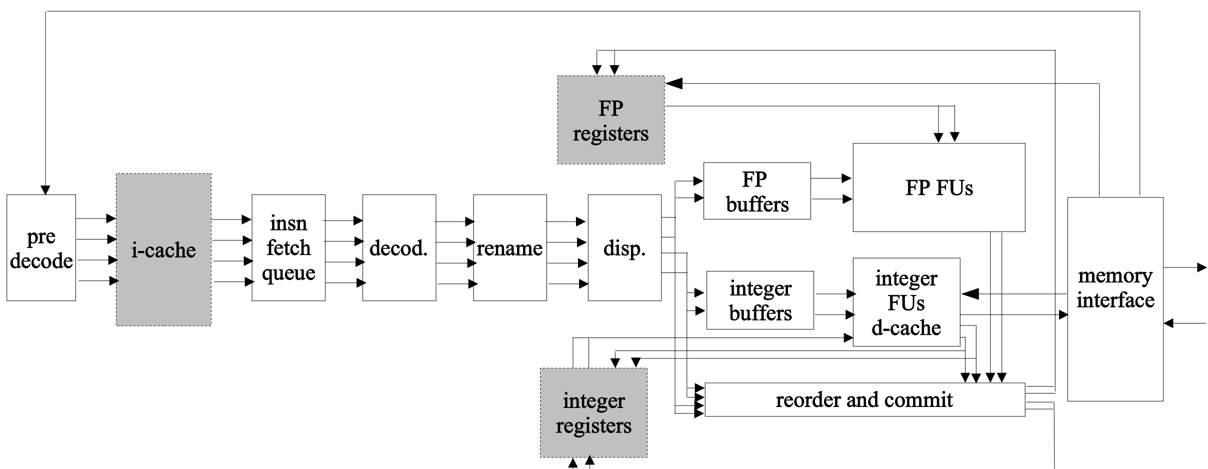

Figure 2.2: Organization of a superscalar pipeline

Today, most general purpose processors have superscalar architectures, like R10000, Pentium 4, Alpha 21264, Athlon, and UltraSparc-III (MIPS Technologies Inc., 1995; INTEL, 2001; KESSLER, 1999; ADVANCED MICRO DEVICES, 2000; HOREL; LAUTERBACH, 1999). In contrast with signal processing applications, which are efficiently executed by systolic architectures, and some scientific applications that present good performance in vector processors (JOHNSON, 1991), general purpose applications are hard to characterize, and thus improving their performance is a hard task.

VLIW Architectures (Very Large Instruction Word) (FERNANDES; SANTOS, 1992; DE ROSE; NAVAUX, 2003) are similar to superscalar architectures, with multiple functional units fed by a pipeline. The main difference between VLIW and superscalar architecture is that the former needs help from compiler to determine dependencies among instructions. All parallelism must be explicitly defined in compilation time, while parallelism is dynamically found in superscalar architectures and implemented in hardware. VLIW architectures are not popular in commercial

designs, suffering from the fact they can only exploit static parallelism and are very compiler-dependent. Each time a new processor with a new set of resources is created, the VLIW programs usually must be recompiled in order to exploit the new set of resources. The Crusoe processor from Transmeta (KLAIBER, 2000) is an example of a commercially available VLIW processor.

The new architecture IA-64 from Intel is also an example of VLIW architecture (HENNESSY; PATTERSON, 2003). It is commercially known as the EPIC architecture (Explicitly Parallel Instruction Computers) (SHARANGPANI; ARORA, 2000). But there are some differences to VLIW processors: EPIC computers can do part of some tasks in the hardware, while VLIW processors are completely dependent on the compilers. Part of the scheduling is done in hardware, but most of it is done by the compiler. VLIW processors have a strict in-order execution, while EPIC processors have their dependencies marked by the compiler, but the hardware is capable of running the instructions out-of-order, respecting the dependencies (HENNESSY; PATTERSON, 2003).

An important tendency for all processor architectures is the use of superpipelines in order to improve clock rates and keep up with the current CMOS technologies. In a superpipelined architecture, pipeline stages are broken into smaller stages (DE ROSE; NAVAUX, 2003). For example, a Pentium-4 processor has 20 stages, while a Pentium-III from the previous generation has only 10 stages (INTEL, 2001).

A typical superscalar processor fetches and decodes several instructions per cycle (SMITH; SOHI, 1995), while scalar architectures fetch and execute only one instruction per cycle (HWANG, 1993). Branch prediction is used to guess the address of the next instruction when a branch is found, allowing to fetch and execute instructions speculatively before the branch outcome is known.

Compared to other architectures, the superscalar architectures implement the largest number of techniques to exploit ILP in hardware. Superscalar architectures provide hardware to perform almost all tasks, including dynamic dependence determination, as well as resource allocation. On the other side, VLIW architectures present static instruction scheduling determined by the compiler, which becomes responsible to allocate resources for each instruction. Although VLIW processors rely more on an efficient and complex compiler, their hardware is simpler than the superscalar implementations.

In the superscalar architectures, the instructions fetched and decoded are analyzed and searched for data dependencies. Even if the superscalar architectures are able to fetch and execute multiple instructions per cycle, they are bound to the sequential model of scalar architectures to keep the compatibility. Thus, the analysis of dependencies is used to determine a partial ordering of instructions which preserves the sequential semantics and allows the exploitation of parallelism among instructions (SOHI; BREACH; VIJAYKUMAR, 1995), called **Instruction-Level Parallelism** (ILP).

Anyway, these architectures pay a penalty to enforce the sequential behavior. Their performance is affected mainly by three different kinds of hazards which prevent the ILP to be fully exploited: **data dependencies**, **control dependencies**, and **resource conflicts** (JOHNSON, 1991).

2.1.1 Data dependencies

Register renaming techniques are used to treat output dependencies and anti-dependencies in superscalar architectures, and only the true dependencies are maintained in order to keep the semantics. **True dependencies**, also known as read-after-write hazards (RAW), occur when an instruction needs values produced by previous instructions (STALINGS, 1996).

These dependencies may limit the number of instructions which can be executed in parallel, reducing the exploitation of available resources and, consequently, the overall performance achieved. Memory accesses also have data dependencies (WALL, 1993; POSTIFF et al., 1999), but their complexity and effects are even more difficult to cope than the ones caused by register dependencies because of the address calculation and the latencies to access memory. Memory renaming techniques are not practical, as the address ranges are very large.

Instructions without data dependencies may be executed in parallel and in a different order than the one they are fetched. Most superscalar processors use **Dynamic Instruction Schedule** (SMITH; SOHI, 1995) to allow out-of-order (o-o-o) execution of instructions.

2.1.2 Control dependencies

Branch instructions increase the operational complexity of instruction pipelines. Instructions following a branch instruction hold a **Control Dependence**, also known as procedural dependency (STALINGS, 1996) or control hazards (PATTERSON; HENNESSY, 1997).

Figure 2.3 shows an example of control dependency (WALL, 1993).The operations between the branch and its target if the condition `r1=0` holds will be executed only if `r1` is not zero. Therefore, these instructions must not change the state of the architecture until the branch output is known, and consequently, the right stream of instructions can appear as executed in the architecture state.

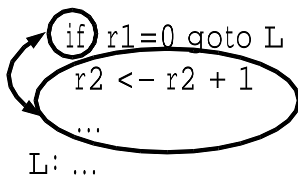

Figure 2.3: Example of control dependency

For many dynamic schedulers, all speculatively executed instructions after a mispredicted branch must be discarded (LAM; WILSON, 1992). But this approach is excessively restrictive. For example, in the code shown in Figure 2.4, the operation `r2` $\leftarrow$ `r1+3` is executed regardless of the `if` output. Even if the branch is mispredicted, instructions executed for this operation could be committed if they do not have data dependencies. Instructions in this situation are called **Control Independent** on `if`. The operation `r1` $\leftarrow$ `1` is **Control Dependent** on `if`, as it will not be committed if the branch is taken (`r1=0`).

Data dependencies may occur inside control independent blocks. For example, if the operation `r1` $\leftarrow$ `1` is executed in Figure 2.4, then there will be a true data dependency on the operation `r2` $\leftarrow$ `r1+3`. But if the previous command is not

```
  if r1=0 goto L
  r1 ← 1
L:r2 ← r1+3
```

Figure 2.4: Control dependency and independency

executed, there will be a false data dependency, which will be resolved only when the branch is resolved (ROTENBERG; SMITH, 1999). Control independent instructions executed with wrong inputs must be re-issued an re-executed, but then they would be ready to be issued inside the reservation stations or the scoreboard, and it would not be necessary to fetch, decode and dispatch them again if the architecture was smart enough to take advantage of control independencies.

The effect of control dependencies in superscalar processors is worse than in scalar processors, because more instructions are fetched per cycle (STALINGS, 1996). The speculative execution of instruction blocks is used to decrease this impact (WALL, 1993). Some new issues arise from this, as the need for mechanisms that prevent mispeculative instructions to change the state of the architecture, and there are also instructions that cannot be executed in speculative mode, like stores.

The most used technique to speculate through control dependencies is branch prediction (LEE; SMITH, 1984; JOHNSON, 1991; PATTERSON; HENNESSY, 1997; HWANG, 1993; MCFARLING, 1993; YEH; PATT, 1993; KESSLER, 1999). Even a small misprediction rate causes a huge impact on performance. As the pipeline must be flushed, all the instructions fetched after the branch squashed, and the fetch is redirected. Other techniques are possible, like multi-path execution (SANTOS, 1997; KLAUSER; GRUNWALD, 1999; HEIL; SMITH, 1996; KLAUSER; PAITHANKAR; GRUNWALD, 1998) and trace caches (PATEL; FRIENDLY; PATT, 1997; ROTENBERG et al., 1997; ROTENBERG; BENNETT; SMITH, 1999).

2.1.3 Resource conflicts

Resource Conflicts occur when two or more instructions compete to use the same resource at the same time (STALINGS, 1996), and are also known as structural hazards (PATTERSON; HENNESSY, 1997). For example, two instructions may compete for the same floating-point functional unit in a given cycle. In this case, one instruction must wait for a free resource, while the other executes.

Resource conflicts may be eliminated by increasing the number of instances of the resources which cause the contention (STALINGS, 1996). But processors have limitations in the number of transistors and interconnections, therefore the increase in the number of resources may result in a processor that cannot be implemented or that cannot take full advantage of the achievable clock rates for a certain implementation technology. Resource balance may be used to obtain better cost/performance rates, resulting in an architecture with practical implementation (SANTOS et al., 1999).

Another way to reduce the impact caused by resource conflicts is to pipeline resources (STALINGS, 1996). This technique is used mainly in functional units. But performance may decrease if there are not independent tasks to be executed in the different stages, or the task sizes are not regular.

2.2 Value locality and repetition

One of the first references to the redundant nature of most computations that occur on microprocessors was reported by Richardson (RICHARDSON, 1992). He noted a number of possible situations where the same computation would be repeated, as when the input is a node in a CMOS circuit (many nodes as 1 V or 5 V), conversions from different units (from inches to millimeters), or parsing program files (many occurrences of the same keywords).

Value Locality (LIPASTI; WILKERSON; SHEN, 1996; LIPASTI; SHEN, 1996) can be defined as the probability of value recurrence in a given storage position (memory addresses, caches, and registers). Gabbay and Mendelson (1996) have a different point of view for the same subject, defining **Value Predictability** as the potential in a program of successful predicting values generated during its execution.

Value predictability can be classified in two classes:

- **Temporal Value Predictability** is the probability that an instruction will generate the same result as a function of its more recent output; and
- **Spatial Value Predictability** is the probability that an instruction will generate the same results as an extrapolation of previously seen values.

This phenomena is already known as **Reference Locality** (HENNESSY; PATTERSON, 2003). For example, the use of caches to reduce the latency of memory accesses is only possible because there are both temporal and spatial value reference in their accesses.

Branch prediction mechanisms (LEE; SMITH, 1984; YEH; PATT, 1991, 1993; MCFARLING, 1993; WALLACE; BAGHERZADEH, 1997) also take advantage of reference locality to improve performance. These mechanisms store information about the branch outcomes, taken or not-taken. According to this information, the branch predictor guesses the result of branches and redirects fetch to the new, predicted target. The target address can also be predicted by using a buffer like a Branch Target Buffer (BTB). Again, predicting branches is only possible because their execution presents a tendency of behaving similarly across the execution of a program. Works as (LIPASTI; SHEN, 1996; GABBAY; MENDELSON, 1996; SODANI, 2000) show that value locality is a frequent phenomena in programs and can be used in order to increase performance.

A program can present many cases of value locality during its execution, with data instances repeating themselves or instructions being executed with the same inputs, and thus producing the same outputs. Sodany (2000) defined then **Dynamic Instruction Repetition** as the re-execution of an instruction with the same operands, producing the same result.

From this definition, we define **Value Redundancy** as the repetition of results, generated by instructions, basic blocks, traces or whatever other set of instructions when the same input values are submitted to execution. The value redundancy observed in programs waste resources with the re-execution of operations when the results have been calculated before.

One of the causes of value redundancy comes from the fact that these programs are developed to cope with all the possible inputs, having code to treat exceptions and wrong inputs. These codes are also developed to reuse previously developed code and to allow future expansions (LIPASTI; WILKERSON; SHEN, 1996).

Even employing compilers that apply complex optimizations on code, value redundancy still occurs, and can even be increased by them. Many values that are unknown during compilation time can turn to be constants during execution. For example, calls to virtual functions are implemented by compilers as code to load a pointer to a function, and this pointer is a constant during run time (LIPASTI; WILKERSON; SHEN, 1996). Other computations may become repetitive due to the inputs. A loop doing operations on an array may execute repeated times with the same inputs, and these should not necessarily be processed more than once.

There are many works in this area exploring value locality and redundancy in processors, and many methods to take advantage of their characteristics in order to improve resource use and performance. In most cases, the use of static mechanisms is not an option because of the dependencies in the input values, which are resolved only during runtime. For these cases, software-based techniques trying to provide reuse or prediction would be heavily prejudiced by the overhead imposed by the tests comparing values to be reused or the correctness of predictions. Therefore, for most cases we consider that reuse and prediction techniques must be implemented in hardware to achieve considerable improvements, but we also think that compiler techniques and profiling can be used to improve the effectiveness of such mechanisms.

2.3 Value prediction

Value Prediction (VP) consists of predicting register contents based on previously seen values (LIPASTI; SHEN, 1996). These values may present temporal and/or spatial locality, allowing to forecast their future values within a given rate of success.

Value prediction is a speculative technique used to increase processor performance. Different of value reuse techniques which do not execute but just reuse redundant instructions, value prediction allows the execution of instructions whose input values are not known yet. As a prediction may be wrong, recovery mechanisms are necessary to rollback the architecture state to the way it was before the misprediction, and then the instructions are re-executed with the correct inputs.

The most widely known and used value prediction technique is branch prediction. Although being a very specialized form of value prediction (and older than the formal definition of it), it is not more than predicting a bit (taken or not-taken) and the target address for the branches. All modern, general-purpose processors heavily employ branch prediction to keep pipelines full even in the occurrence of branches. Another difference is that branch prediction is used to deal with control dependencies, while the value prediction techniques that we discuss here are designed to overcome data dependencies.

As with any VP technique, branch predictors suffer from the effects of mispredictions, and must have a recovery mechanism to deal with mispeculations. Therefore, we can say that all current superscalar processors have recovery mechanisms for branch mispredictions, and these can be used with minor modifications to deal with other kinds of mispredictions and not only those caused by branches.

The main advantages of value prediction are:

- Value prediction can overcome true data dependencies (LIPASTI; SHEN, 1996; SAZEIDES; SMITH, 1997), as it allows instructions with inputs that have

not been calculated yet to execute in parallel with the instructions that are producing the values;

- These techniques can also reduce the latency of memory instructions or that have a high complexity (as floating-point division);
- Future processors are expected to have high latencies in their transmission lines, and value prediction can be employed to hide them (PARCERISA; GONZÁLEZ, 2000).

The main disadvantage of value prediction is the mispeculation penalty that occurs whenever a value is incorrectly predicted. Instructions that executed with incorrect inputs must be executed again with correct inputs.

Sazeides and Smith (1997) have studied value predictability and defined three main types of value sequences:

- **Constant**, where the same value occurs again;
- **Stride**, where there is a fixed difference (stride) between two subsequent values; and
- **Non-stride**, where there is a complex correlation or no correlation between two subsequent values.

Constant sequences are easier to explore, as there is no need of identifying the strides or the correlation among values and the current path. But in some cases, the stride sequences may allow better performance improvements, as for some loops. The non-stride sequences are usually not exploited because of their complexity.

2.3.1 Common steps of value prediction techniques

Even having differences on implementation like the way tables are organized, the access key and other details, value prediction mechanisms work based on the same general steps:

1. An instruction or set of instructions from the domain of instructions where predictions are allowed is detected;
2. The mechanism verifies if there are inputs that are not ready;
3. If there are inputs that are not ready, then the value prediction tables are searched;
4. If there is a prediction which has a confidence above a determined threshold, it is set as the current value for all the subsequent instructions that try to access that value, and the instructions are executed in speculative mode until the actual value is resolved;
5. When the predicted value is resolved, the actual value is compared with the predicted one. If they match, all the instructions that used the predicted value are set as non-speculative and can be committed. If the values mismatch, then at least all instructions on the dependence chain from the mispredicted value

must be executed again. Depending on the mechanism, even instructions that do not depend on it are squashed and executed again (for the sake of a simpler recovery mechanism);

6. Success or failure in predicting a value are used to update the mechanism, in order to improve the rate of correct predictions.

In the following subsections, we present some mechanisms based on value prediction and the main differences among them, and after that, schemes to correct mispeculations.

2.3.2 Last value prediction

A possible implementation of value prediction is the **Last Value Predictor** (GABBAY; MENDELSON, 1996, 1998) (Figure 2.5), which predicts a value based on the last value seen for a given instruction, thus exploring constant value sequences. The PC address is used to access the prediction table, where the prediction for the load value is stored.

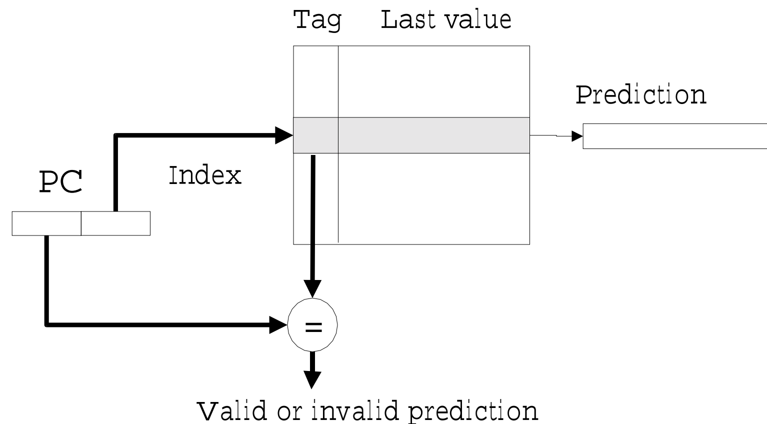

Figure 2.5: Last value predictor

A specialized version of last value prediction is the **Load Value Prediction** (LIPASTI; WILKERSON; SHEN, 1996; LIPASTI; SHEN, 1996), where only load instructions are predicted because they present large latencies and high value locality. Figure 2.6 depicts a possible implementation of load value prediction.

The Load Value Prediction Table (LVPT) holds previous values read by loads, addressed by the instruction address. A confidence table is used in order to keep information about correct predictions. Both tables receive the PC of a load instruction. The LVPT outputs a predicted value, while the confidence table says if this prediction should be accepted or not.

Another version predicts values of registers (GABBAY; MENDELSON, 1996), but addresses the history tables by the register index and not by the instruction address. This technique associates a prediction to all instructions that write to the same register and interfere with its prediction. The advantage of this mechanism is the small table needed to hold values to be predicted, a function of the number of registers.

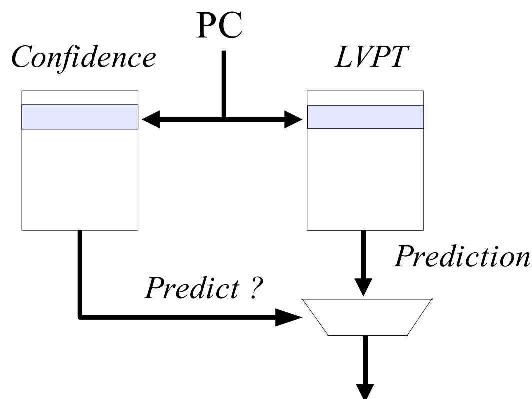

Figure 2.6: Load value prediction

Tullsen and Seng (1999) developed a diverse mechanism, without the need of additional tables to store previous values. In this mechanism, an instruction uses an old value in a register as the prediction for its next value. The hardware needed is dramatically reduced, but this mechanism cannot hold a large number of previous values neither capture more complex patterns of predictability. Therefore, only constant sequences of values can be successfully explored.

Based on the correlation among instructions and the instructions executed earlier, **Value Prediction Based on Correlation** (WANG; FRANKLIN, 1997a,b) stores the last n observed values and the branch history is used to select one of them. Sazeides and Smith (1997) proposed another value prediction mechanism based on correlation, but using the previously n seen values as a context.

Parcerisa and González (2000) proposed the use of value prediction to hide long latencies to be found in long wires for the future microprocessor chips, which are likely to have problems with capacitance because of the characteristics of the CMOS technology.

Many variations have been also proposed, like two-level value prediction (WANG; FRANKLIN, 1997a,b), hybrid value prediction (WANG; FRANKLIN, 1997a,b; SATHE; WANG; FRANKLIN, 1998), and others. Many of them are based on similar approaches for branch prediction, like changing the way tables are addressed, adding path information by using branch registers, and so on. In the next subsections, we discuss two of the most original works that do not rely on just a small change to improve value prediction accuracy and performance: *stride prediction* and *trace level prediction*.

2.3.3 Stride prediction

In the Last Value Predictor, only the last value seen can be used as a prediction for the next value, but the mechanism may be extended to allow multiple instances of the same instruction, or a stride prediction by having an extra field for the stride to be added to the value (**Stride Predictor**, Figure 2.7). This extension allows the prediction of values in loops, for example.

Even if a value has not been seen before, it can be successfully predicted based on the previous values for the same instruction or register if they follow a simple pattern that the stride predictor can identify, like:

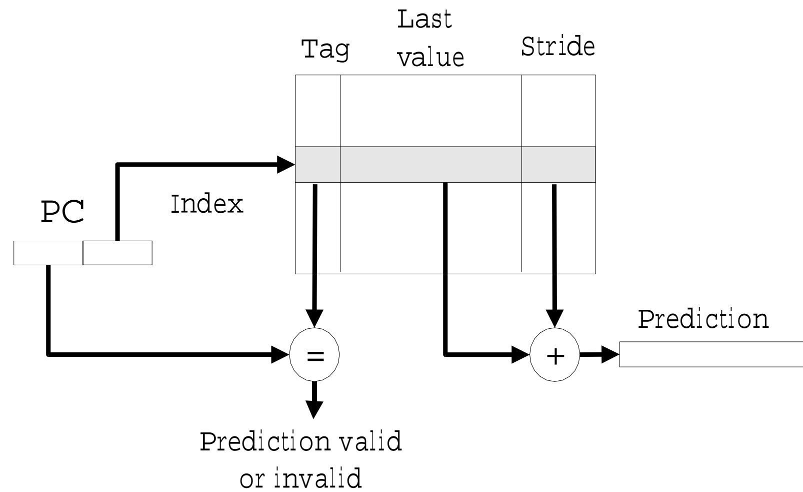

Figure 2.7: Stride predictor

`1, 2, 3, 4, 5, ...`
Or
`1, 3, 5, 7, 9, ...`

In these two cases, there is a fixed difference between two instances of the same value.

2.3.4 Trace-level prediction

An interesting mechanism was developed by Sathe, Wang and Franklin (1998), based on the prediction of multiple values for traces of instructions in a single cycle. Only the last updated values in a trace (live outputs) are predicted, reducing the necessary bandwidth in the predictor.

Figure 2.8 shows a value predictor based on traces. The trace address is used to index the value table. Each entry in this table may hold multiple values, and a bitmap field is used to store the mapping from values to instructions in the trace.

This technique allows multiple values to be predicted in parallel using only one access to the prediction table, thus increasing the possible performance improvements that would be otherwise limited by the bandwidth necessary to access the tables.

2.3.5 Speculation control and confidence

One of the main concerns when designing speculative techniques to improve performance in computer architectures is the cost involved in mispeculations. A mispredicted value has an associated penalty. All instructions that executed and depend on the mispeculated value must be executed again, now with the correct value. The penalty is composed not only by the execution time of those instructions, but also by the cost of redirecting fetch and waiting for instructions to reach the execution stage again. Besides, mispeculated instructions may occupy important resources that could be used to run non-speculative instructions. Therefore, it may be necessary to impose limits on how much an architecture may speculate in order to balance the costs and benefits of value prediction.

A possibility is to employ **confidence techniques** to obtain the best from spec-

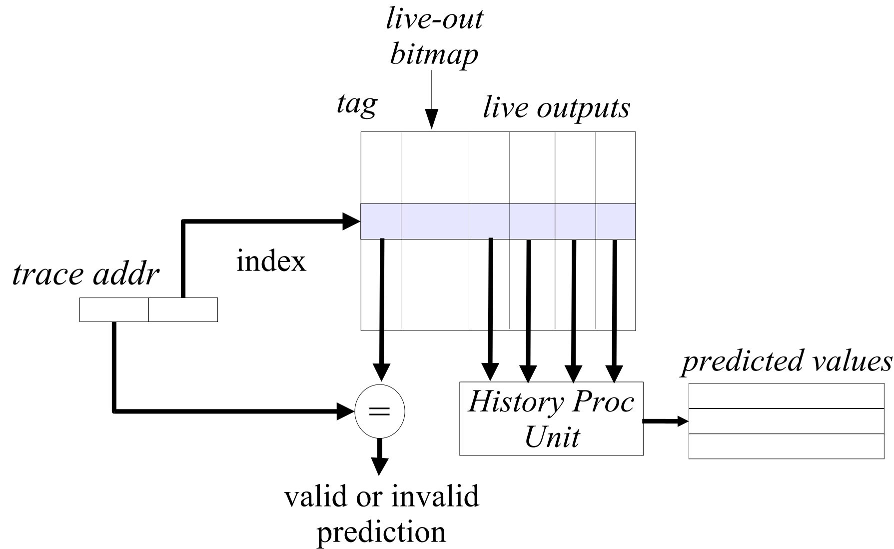

Figure 2.8: Value predictor based on traces

ulation. Confidence techniques usually employ simple finite automate to measure how predictable a certain value is, and then speculation is allowed only if the automate is at certain states where the prediction is supposed to be correct above a given confidence.

Figure 2.9 shows a simple automate for branch prediction with four states. Each state stores information about the last two predictions already resolved, which can be **T** (taken) or **N** (not-taken). Each new prediction can trigger a state change (the arrows) between two stages. The branch predictor may only predict a branch as taken if the last two branches were taken, for example. Then, the only state where a taken prediction would occur is in the **TT** state. This can be easily implemented by saturating counters which are increased or decreased based on the branch outputs. The same concept can be used for any value prediction.

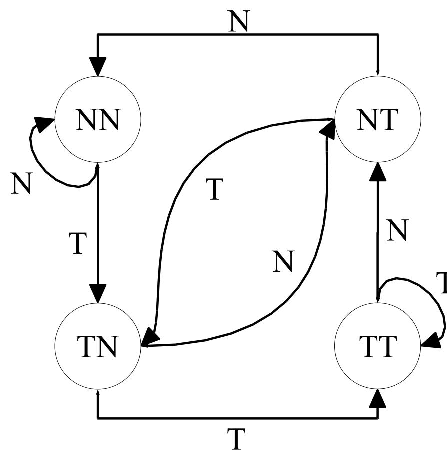

Figure 2.9: A simple automate for branch prediction

Grunwald et al (1998) compared the performance of different estimation meth-

ods, which can be used combined with many speculative techniques such as value prediction to determine which speculations are worth pursuing.

Calder, Reinman and Tullsen (1999) studied the use of different schemes to select predictions, using two levels of history and prediction tables, in the same way they are used in branch predictors. Confidence mechanisms assign a degree of certainty that a given prediction will be correct or not based on previous history and the current context, and prediction is only allowed when a certain threshold is reached.

Figure 2.10 shows a general structure for value prediction using confidence. A key is generated usually from the PC address, which may be hashed with other information as the current path, and it is used to access both the prediction tables and the confidence table. The confidence mechanism will select whether a prediction will be allowed or not.

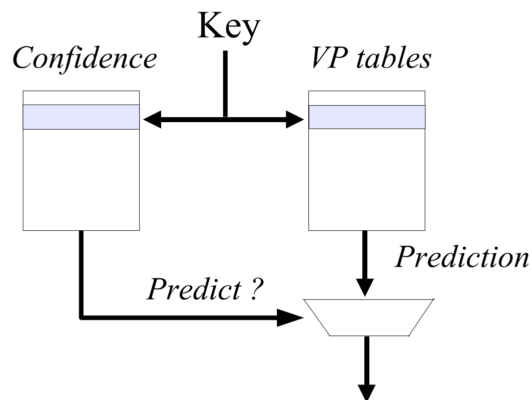

Figure 2.10: Value prediction with confidence

Tune et al (2001) have a different approach on how to control speculation. They find and predict chains of dependent instructions that are in the critical path to optimize performance, based on the previous behavior observed during execution.

2.3.6 Recovery mechanisms for mispeculations

Reinman and Calder (1998) developed a study focusing prediction mechanisms to improve performance. Besides the prediction mechanisms, they also studied two different ways of coping with mispeculations. Both mechanisms use a Re-Order Buffer (ROB) and reservation stations to keep the non-speculative state.

The first mechanism, **Squash Recovery**, is the basic scheme to deal with mispeculations, discarding all instructions following a mispredicted instruction. After that, the instructions are re-fetched and executed with the non-speculative values. It relies on the existence of a reorder buffer to determine which instructions were fetched after a misprediction, and they are all flushed. Instructions in a reorder buffer will be committed in order and only after they are on non-speculative state.

Figure 2.11 shows an example of a ROB. It is implemented as a circular queue, where instructions are retired from the head when all the previous instructions were retired, and the instruction itself was not squashed because of a misprediction or an exception. In Figure 2.11(a), there are six instructions in the ROB. Instruction `i3` was found to have its result mispredicted by the value prediction mechanism, therefore instructions after it may have executed with wrong values (Figure 2.11(b)).

Then, these instructions are squashed, fetch is redirected, and execution continues from that non-speculative state.

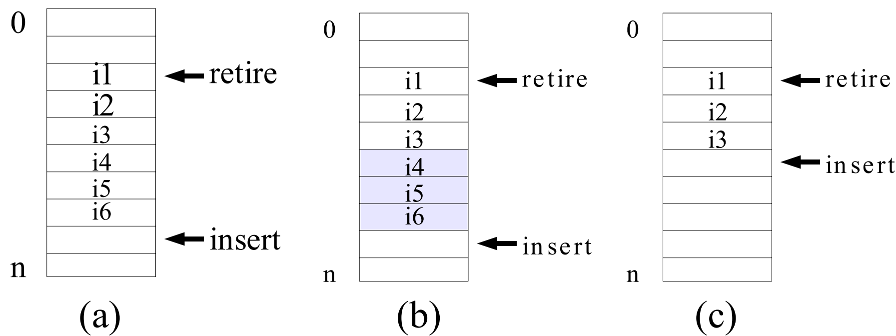

Figure 2.11: A reorder buffer

A more conservative mechanism is implemented by the **Re-execution Architecture**. When a mispeculation is detected, only the instructions which are dependent on the mispeculated instruction are re-executed. This mechanism is more complex and requires specialized hardware to detect dependencies.

Nakra, Gupta and Soffa (1999) developed a VLIW architecture with value prediction, using a compensation engine to execute code dynamically generated after a misprediction. The compensation code is executed in parallel with the regular code in the VLIW processor, thus reducing the penalty from mispeculations.

Another way of dealing with mispredictions is to exploit thread level parallelism to execute instructions with non-speculative values while executing with predicted values in another thread. An example of this approach was proposed by Koushiro, Sato, and Arita (2003), where a contrail processor divides the execution in two streams. One of the streams uses trace level value prediction to skip instructions, while a verification stream uses slower functional units to validate the predictions.

Wu, Chen, and Fang (2001) also use thread level parallelism to execute both speculative and non-speculative versions of the same stream to avoid mispeculation penalties.

The problem with the two approaches above is that more resources are necessary to implement the two cores than a single one, and there is also an increase in complexity.

2.4 Value reuse

The concept of **Value Reuse** – reusing a value that has already been calculated before – came from the observation that many instructions are executed with the same inputs, generating the same results (LIPASTI; WILKERSON; SHEN, 1996; SAZEIDES; SMITH, 1997; SODANI; SOHI, 1998; SODANI, 2000). Sometimes, the inputs are always the same, and then they could be transformed by the compiler, avoiding the unnecessary executions. But some issues prevent this optimization (SODANI, 2000):

- The dynamic path is not statically known. The repetition may depend on the input. Therefore, the repetition is not obvious in compiler time;

- The compiler may not optimize the code to keep the correction;
- The analysis needed to detect static repetition is complex and hard to perform;
- The space requisites may be prohibitive;
- Some repetitions are derived from instruction set limitations and cannot be eliminated.

It is not impossible to implement static reuse, but it would need compilers with constant propagation, function in-lining, loop unrolling, common subexpression elimination, and other techniques. All these techniques would be used globally, and a larger number of registers would be necessary. Memory address analysis would also be needed to allocate registers (SODANI; SOHI, 1998).

According to Sodani (2000), value reuse can improve performance in four ways:

- Instructions reused are not executed, which may save cycles for instructions with high latencies;
- The results are ready earlier, allowing dependent instructions to start execution earlier;
- Useful work in wrong paths may be preserved;
- Value reuse collapses data dependencies, and dependent instructions may be executed in parallel.

Additionally, some mechanisms may improve branch prediction by correcting mispredictions when instructions are reused (COSTA, 2001) and also use the redundancy of instructions in diverse locations that execute the same computations (MOLINA; GONZÁLEZ; TUBELLA, 1999).

Sodani and Sohi (1998) classified reuse opportunities in two groups: Squash Reuse and General Reuse. The former refers to reuse values produced by squashed instructions in mispredictions, while general reuse includes all the other possibilities.

The unit size to be reused may be an instruction (SODANI; SOHI, 1998), expressions and invariants (MOLINA; GONZÁLEZ; TUBELLA, 1999), basic blocks (HUANG; LILJA, 2000a), traces (GONZÁLEZ; TUBELLA; MOLINA, 1999; COSTA, 2001), as well as instruction blocks and sub-blocks of arbitrary size (HUANG; LILJA, 2000b). Costa (2001) also classified the reuse mechanisms based on the identification of redundancy in static or dynamic detection.

The main disadvantage of value reuse is that it requires all the inputs of a given reuse unit to be ready to compare with stored values, in the process called *reuse test*. Only when all inputs match a previous instance of the same reuse unit is that the reuse may actually happen. Thus, some units that would have a match are not reused because some of their inputs are not ready to be tested. In these cases, the reuse test could be postponed until the values are ready, but it would dramatically increase complexity and provide only small performance gains.

Studies show that, in some cases, 50% of the instructions executed in a program are redundant (SODANI; SOHI, 1998), therefore there is a large potential to increase performance using value reuse.

2.4.1 Common steps of value reuse techniques

As for value prediction, value reuse mechanisms also have the same general steps:

1. An instance of a set of computations is stored with its inputs and results;
2. In the next execution, a reuse opportunity is identified;
3. The reuse opportunity is tested by verifying if the inputs match any of the previously seen instances that are stored in the reuse tables;
4. If there is a match, the result values stored in the previous instance are reused, and all the instructions in the reused set of computations are skipped.

In the next subsections, we will discuss mechanisms based on value reuse.

2.4.2 Value reuse mechanisms

The first studies on dynamic reuse were started to reduce compiler costs. The **Value Cache** (HARBISON, 1982) eliminates some redundant expressions that dynamically occur and cannot be avoided with static support only. Simple expressions without branches, function calls or stores are analyzed. When an expression is executed, its input context and the result are stored in the value cache. If the same expression occurs again with the same inputs, it is reused. If it occurs with another set of inputs, it is invalidated (COSTA, 2001). Only one instance of each expression may be stored.

Memoization (ABELSON; SUSSMAN, 1985; MICHIE, 1968; RICHARDSON, 1992) allows the exploitation of redundant executions of functions when all inputs are repeated and there are no side effects, like global variables, memory positions and registers affected by other parts of the same program or externally.

When a function is executed, the input context is stored, as well as the output context. The next redundant executions of this function are changed to the stored output. An example of memoization is Richardson's **Result Cache** (RICHARDSON, 1992), aimed at long-latency floating-point instructions. His result cache is a direct-mapping table accessed at the same time that a long-latency instruction reaches the FP unity for execution. If there is an entry in the result cache with the same inputs, the operation can be halted and the stored value is reused. If an instruction misses, its inputs and results can be stored in the table after execution, so future, redundant executions of that instruction can benefit from it.

Sodani (2000) observed that the percentile of functions with the necessary characteristics for memoization in the integer benchmarks from SPEC95 is near to zero, but instructions grouped in other block sizes may present these characteristics. For the floating-point benchmarks, a significant number of function calls may be reused.

Citron, Feitelson and Rudolph (1998) developed a memoization technique to avoid executing long latency instructions in multimedia processing. Rebello (1997) proposed the use of a result cache to memoize the execution of functions, using both software and hardware support.

2.4.3 Instruction reuse

Instruction Reuse (SODANI; SOHI, 1998; SODANI, 2000) presents a **Reuse Buffer** (RB) to store redundant instructions. The **Reuse Test** selects instructions

that can be reused, comparing the current input values with the previously seen values.

Instruction reuse may employ different policies to store and invalidate information. Figure 2.12 shows a generic reuse buffer. The buffer is indexed using the PC address. Reused instructions are sent to the writeback stage, without requiring functional units or issue.

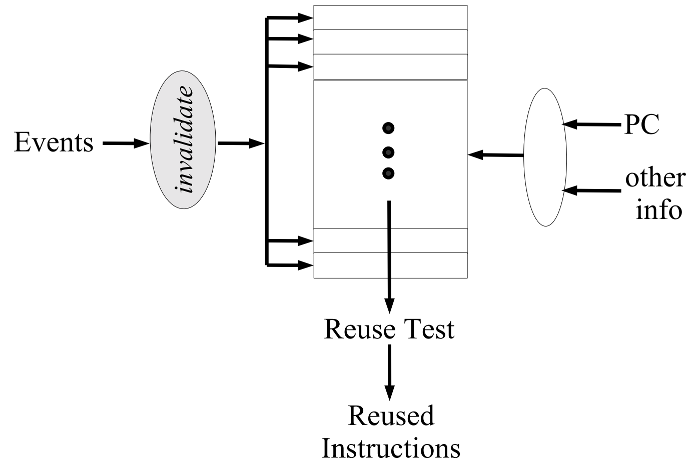

Figure 2.12: Generic Reuse Buffer

The reuse buffer associativity determines how many instances of the same instruction may be stored at the same time. Each instance may be distinguished by the instruction address and the input values. An increment in associativity also increases the number of simultaneous reuse tests, having an impact in hardware complexity. The issues involved in implementing reuse buffers are similar to the ones found in cache associativity and size.

Sodani (2000) defined four variations of this scheme:

- S_v, which verifies the input values for each instruction;
- S_n, which verifies the register names of inputs;
- S_{v+d}, extends the first scheme by adding tests of dependencies among instructions;
- S_{n+d}, extends the second scheme, also by adding tests of dependencies among instructions.

The last two variations present increased implementation complexity, as they also include the determination and testing of dependencies among the instructions.

Another reuse technique, called **Instruction Reuse by Register Integration** (ROTH; SOHI, 2000), takes advantage of instructions squashed because of mispredictions. It is simple to implement, and it basically needs modifications in the register renaming stage. It recognizes the validity of squashed values, and then the value stored in the register may be reused. It is only necessary to modify the register table for the next instructions that use the logical register so they point to the actual register. The drawback of this technique is the limitation to squashed instructions, but it can easily be used in conjunction with other reuse and prediction techniques.

2.4.4 Basic block reuse

An extension of dynamic instruction reuse is the **Basic Block Reuse** (HUANG; LILJA, 1999). Basic blocks (dynamic instruction sequences with branches only as the last instruction) are identified and stored with their inputs and live outputs, which are the outputs used by other instructions. Dead outputs are results that do not have consumers. The Block History Buffer (BHB) is used to store the basic blocks and their inputs and outputs. Annotations are written in instructions by compilers to determine live registers, modifying instruction format. Therefore, it does not provide legacy compatibility (COSTA; FRANÇA; CHAVES FILHO, 2000).

Huang and Lilja (1999; 2000b; 2000a) have detected that the behavior of SPEC95int benchmarks presents basic blocks with tendency to show good or no locality. Therefore, for these programs they determined that a history with depth one is enough to identify and reuse most of the basic blocks. Their work also shows that a 2048-entry BHB with four input registers, five output registers, four memory inputs, and two memory outputs covers 90% of the possible reuses, with a 9% overall performance improvement.

In the evolution of that work, Huang and Lilja (2000a) added limited compiler support to further increase performance. The **Sub-block Reuse** (HUANG; LILJA, 2000b) allows the reuse of blocks smaller than basic blocks, breaking the blocks based on constraints like the number of inputs and outputs. But still it is limited to the constraints of basic blocks, and a basic block is ended by a branch instruction.

Loads and stores are not reused in most mechanisms because of the side effects they may present. Some works have studied this problem. Bodík, Gupta and Soffa (1999) analyzed load-reuse,comparing the dynamic load-reuse with the amount of reuse statically found. Önder and Gupta (2001) proposed the explicit management of the physical register file contents as a level in the memory hierarchy. For each value in the register file involved in a load or store, the associated information, including the memory address, are stored. These stored values are used to implement non-speculative optimizations.

Another approach uses instruction reuse to exploit both same instruction and different instruction redundancy (YANG; GUPTA, 2000). This work focuses on memory reuse, and uses three different tables to store and correlate loads. The load reuse allowed the off-chip traffic to data cache to be reduced by 32%.

2.4.5 Trace reuse

A further step in value reuse is the **Trace Reuse** (GONZÁLEZ; TUBELLA; MOLINA, 1998; COSTA; FRANÇA, 1999; COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA, 2001). This family of techniques is based on the reuse of dynamic sequences of instructions (traces). Each trace has an input and an output context, composed by those registers that are read and written by the trace, respectively.

Trace instructions are not stored, and when a trace is reused, only the output scope is written in the output registers. Trace reuse is not limited by the same boundaries found in basic block reuse, and may even include multiple branches in a single trace. It does not require compiler help (although it may benefit from it) and supports legacy code without modifications.

One of these trace reuse techniques is the **Dynamic Trace Memoization** (DTM) (COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA; FRANÇA;

CHAVES FILHO, 2000; COSTA, 2001), where lie the roots of this work. DTM is a trace reuse technique for instruction and, more importantly, trace memoization. The reuse domain is composed of all integer instructions without side effects. Floating-point instructions are not reused because they show little redundancy (COSTA, 2001). For loads and stores, the address calculation is split from the actual memory access, and DTM can reuse the address calculation. Memory aliasing creates difficulties to reuse also the memory accesses, but Viana (2002) extended DTM by increasing the reuse domain to include memory accesses.

System calls are not reused because they require a mode change in the processor status, causing a pipeline flush.

2.5 Combined value reuse and prediction

In this Section, we present the works that come closer to our proposal, where both value prediction and value reuse are used to exploit the redundancy and predictability present in most programs.

2.5.1 Result cache based mechanisms

Huang, Choid, and Lilja (1999) proposed an extension for the Result Cache where entries can be speculatively reused, and called it **Speculative Result Cache** (SRC). Instead of reusing values only when the inputs are known, this technique uses the current value to index the result cache and reuses the value produced by the same instruction or any other instruction of the same type. Thus, it is not constrained to only same-instruction reuse. Figure 2.13 shows a possible implementation of a SRC table (HUANG; CHOI; LILJA, 1999). The two possible operands for an instruction (in the MIPS-IV instruction set) are hashed together and concatenated to the operand type to address the SRC table. Then, the operands are compared to the actual values, and in case of a match, the result is reused. If there are missing inputs, one of the instances for that specific type of instruction may be speculatively reused if the confidence is above a given threshold (field `conf`).

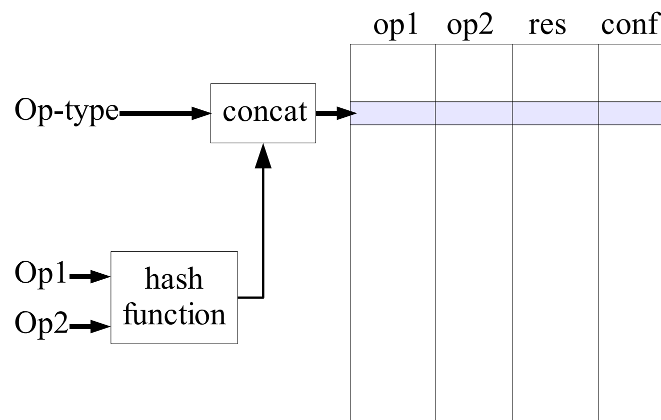

Figure 2.13: Implementation of a SRC table

Together with the Speculative Result Cache, they also proposed a mechanism called **Combined Dynamic Prediction** (CDP) (HUANG; CHOI; LILJA, 1999). It combines both SRC and a hybrid value predictor to exploit both value reuse and

value prediction. During execution, the chooser will pick a prediction from the value predictor or a speculatively reusable value from SRC. With this approach, they have achieved speedups of 10% in a 16-wide, 6-stage superscalar architecture and roughly 128 KB of storage for the CDP.

2.5.2 Region based mechanisms

Wu, Chen, and Fang (2001) proposed a speculative multi-threading scheme where both reuse and prediction are combined. Their approach uses a Computation Region Buffer (CRB) to store instances of a computation region, determined by the compiler, that have been recently executed. Each entry in the CRB has a set of input and output registers, whose values may be different for each instance. The Reuse Test is similar to what is done in trace-level reuse.

In the case of a region with unknown inputs by the time the reuse test is done, one of the instances may be chosen and the outputs are considered to be predicted values. The multi-threading hardware is used to execute both speculative and non-speculative instructions in parallel, in order to reduce misprediction penalties. This approach can increase performance of reuse from a speedup of 1.25 to 1.40 over an IA-64 baseline architecture without reuse or value prediction.

2.5.3 Instruction reuse based mechanisms

Liao and Shieh (2002) combined single instruction reuse and value prediction to achieve speculative reuse. Their architecture has two separate tables, a reuse buffer and a value prediction table, which are accessed in parallel for each instruction. If the operands are ready and there is a match in the reuse buffer, the instruction is reused. If there are inputs that are not ready, a value prediction may be employed in order to speculatively reuse the instruction. Instructions that have inputs in a dependence chain from a predicted value are marked as speculative and are re-executed in case of misprediction. This approach has the disadvantage of requiring both value prediction and value reuse tables.

In a six-stage pipeline architecture and simulating 50 million instructions, they achieved an average speedup of about 9% above their baseline architecture.

3 REUSE THROUGH SPECULATION ON TRACES

"The last thing one knows in constructing a work is what to put first."

Blaise Pascal

Reuse through Speculation on Traces (RST) (PILLA et al., 2001, 2002, 2003a,b) is a speculative trace reuse technique, integrating both trace reuse and value prediction as a complexity-effective approach to increase the number of reused traces.

In this Chapter, we present the concepts behind RST, leaving most of the architecture-dependent details to the next Chapter, whenever possible. First, we show the motivation for speculative reuse of traces in Section 3.1. After that, we present RST in Section 3.2. Trace construction, regular reuse and speculative reuse are explained in Section 3.3. In Section 3.4, we explain the main differences with other reuse and prediction techniques. Finally, in Section 3.5 we state the contributions of this work.

3.1 Motivation

Value reuse is a non-speculative way of exploiting the redundancy observed in the execution of most programs. After the input values of a set of instructions are verified against stored values and a match is found, their results can be reused without executing the instructions. Importantly, resources are not wasted due to reuse and are available to other instructions. The main disadvantage is that reuse must wait until all the input values are ready to be tested for reuse. Therefore, many cycles that could be saved by reusing instructions may be spent waiting for input values that were not ready at the time of the reuse test.

On the other hand, value prediction can hide the limits imposed by true data dependencies (LIPASTI; SHEN, 1996; SAZEIDES; SMITH, 1997). Instructions with true data dependencies may be executed in parallel when value prediction is employed. This technique may also hide latencies of instructions accessing memory or that present high computational complexity, but its main disadvantage is that mispredictions can incur a high recovery penalty. In fact, the misprediction penalty increases as pipelines get deeper.

Another disadvantage is that, since value prediction increases concurrency and demands for resources, instructions executing with mispredicted values may prevent

the execution of more useful instructions.

Many techniques for exploiting the redundancy and predictability are shown in Chapter 2. Trace reuse (COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA, 2001) presents the capacity of skipping sequences of redundant instructions, thus increasing performance. But much of its potential is still unrevealed, as many traces are not reused only because some of their input values are not ready to be compared to the stored values when the reuse test is done.

Figure 3.1 shows the percent of traces reused and not reused because of not ready inputs for DTM for a set of integer benchmarks from both SPEC 95 and 2000 in a 6-stage superscalar architecture (PILLA et al., 2001). The black bars show the percent of traces that are effectively reused of all possible traces with correct inputs; the remaining traces are not reused because their sources were not ready at the reuse test. In average, 69% of all reusable traces were not reused because trace reuse is conservative and does not allow traces with undetermined inputs to be reused.

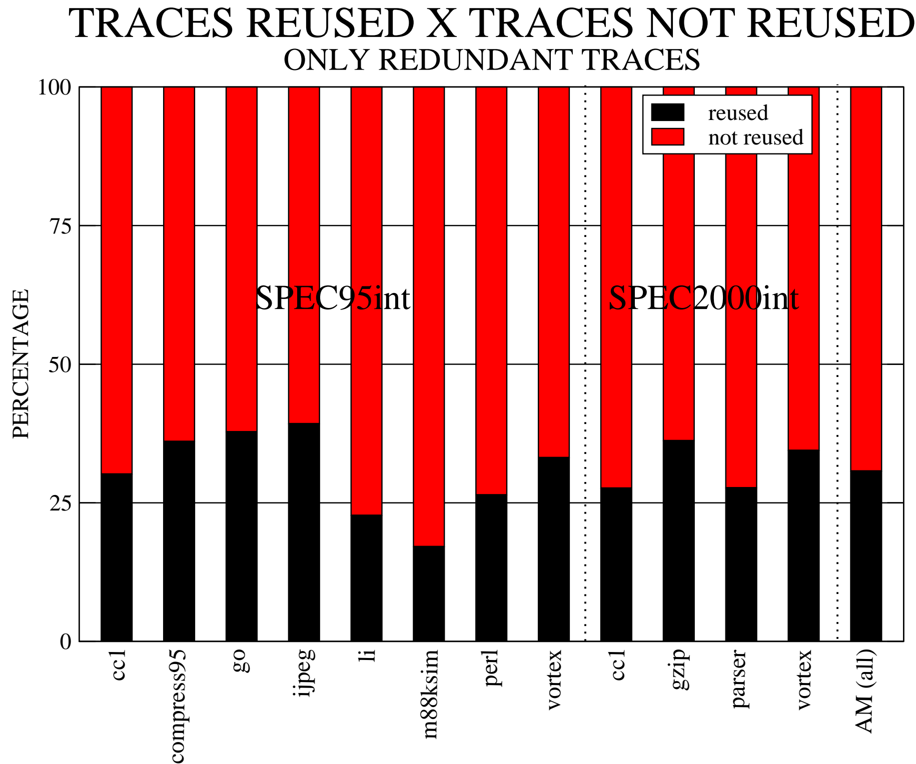

Figure 3.1: Comparison of traces reused and not reused in DTM

Even with most traces not being reused, DTM still can achieve speedups of 9.5% (PILLA et al., 2001). Based on this potential, RST was developed to reuse traces even when it is not possible to determine some of their input values by the time the reuse test is performed.

This situation tends to be even worse when deeper pipelines are considered. Traces are going to demand more cycles to be created, and there is the possibility that more inputs will not be available for the reuse test, as instructions will take more cycles to execute.

Sodani and Sohi (SODANI; SOHI, 1998) compared the benefits and drawbacks

of instruction reuse and value prediction to make their case on promoting instruction reuse. They verified that reuse can extract most of the existing redundancy in their workload (from 84 to 97% of all redundancy in some benchmarks of SPEC 95 int, in a six-stage pipeline architecture), showing that the characteristic of non-speculatively validating inputs does not significantly restrict the reuse capacity to capture redundancy. But their study does not count for other questions, like the new issues involved in using deep pipelines as in modern processors. Do late validation and speculation (value prediction) make any difference in performance when compared to early validation (value reuse)? Does the impact of misspeculations make value reuse a better alternative for these cases? Can only value reuse take advantage of most of the redundancy to improve performance?

3.2 Reuse through Speculation on Traces

Reuse through Speculation on Traces (RST) is a novel approach to improve trace reuse and hide true data dependencies. RST allows traces to be *(i)* **regularly reused**, when all inputs are ready and match the values stored in the input context of a trace; or *(ii)* **speculatively reused**, when there are unknown values in the input context of a trace. Therefore, any traces that could not be reused in previous approaches due to inputs not being ready for early validation may be reused in RST to exploit their predictability, and not only their redundancy.

RST combines the advantages of both value prediction and reuse. Unavailable inputs for memoized traces are predicted by RST based on the values already read from the input context of those traces. Thus, both late and early validation (SODANI; SOHI, 1998) are available to the architecture, with emphasis on early validation, that is value reuse. When the inputs of a trace are all ready by the reuse test time, then they are compared to the current values and, if they all match, the trace is reused. On the other side, late validation can be employed when some of the input values are not ready for early validation. Then, the value may be predicted, which causes the trace to be speculatively reused, and the inputs that were not known in the early validation are tested later.

One of the main advantages of RST over just combining a value prediction technique with an unrelated reuse technique is that RST does not require extra tables to store values to be predicted. Applying reuse and value prediction in unrelated mechanisms at the same time as in (LIAO; SHIEH, 2002) may require a prohibitive amount of storage in tables. In RST, the values are already stored in the trace memoization table (Memo_Table_T), and there is no extra cost in reading them if compared with a non-speculative trace reuse technique. The input context of each trace (the input values of all instructions in the trace) already stores the values for the reuse test, which may also be used for prediction. Thus, our proposed technique minimally increases the hardware to implement speculative trace reuse when compared to the hardware necessary for non-speculative trace reuse.

An important drawback of value prediction is the extra pressure on resources. As a value is predicted, more instructions become ready to execute and demand functional units, dispatch and issue bandwidth. Mispredictions cause instructions to be executed with incorrect inputs, which may avoid that other instructions with non-speculative inputs execute. When traces are speculatively reused in RST, the output values are sent directly to the writeback stage, reaching the instructions

waiting for these values and the register file. Dispatch, issue, and execution are bypassed for the *entire* trace in a single cycle after it has been determined to be reusable. Therefore, speculative reuse does not increase but reduces the pressure on valuable resources in many cases. Even when a misprediction occurs, many instructions may be reused instead of executed again, also reducing this overhead on the execution core.

RST does not increase the number of accesses to the memoization tables or to the register file. These accesses are already done in trace reuse techniques such as DTM, but the difference is that they are useless if some of the input values are not ready to be tested. Hence, RST does not increase the accesses to the reuse tables in order to obtain more reusable traces, but better exploits the traces that would be read anyway by non-speculative reuse.

RST may reuse both instructions and traces, but only traces are speculatively reused because they encapsulate more than one instruction and possibly critical paths, thus allowing more performance improvement than single instructions. Besides, allowing that only traces may be predicted also reduces the complexity of the resulting architecture and the number of tests that will be needed at the writeback stage, where mispredictions are detected.

Only same instruction reuse is treated by RST. The execution of the same instructions with the same inputs in a different PC address cannot be reused or speculatively reused in RST.

Another possibility for traces whose inputs are not ready is to postpone the reuse test, but this solution is very difficult to implement because it would need an exponential increase in the logic to compare the values for each extra cycle that a trace would be held for tests. Another problem would be the increased snooping in the result bus. Finally, most of the performance improvements could be lost by waiting for the values. Thus, we think that the best possible solution is to predict the inputs, and we developed RST in the present way.

3.3 Trace construction and reuse

In RST, traces are dynamically constructed from sequences of redundant instructions, stored in hardware tables called Memo_Table_G and Memo_Table_T like in DTM. In addition to including only reusable instructions found in Memo_Table_G or instructions whose inputs are produced by previous instructions in a new trace in formation as in DTM, RST can also include instructions that are not in Memo_Table_G but are part of the reuse domain. Thus, more traces can be created and reused later. The drawback of this approach is that instructions already found in the Memo_Table_G are more likely to be found again with the same inputs, therefore in some cases some traces may be expelled before they get the chance to be reused.

We will call the two ways of constructing traces as *(i) reusable mode*, for when instructions not in Memo_Table_G can be inserted in a trace, and *(ii) reused-only mode*, when only instructions found in Memo_Table_G are allowed in a trace. Anyway, it is easy to switch from one mode to the other, allowing the architecture to be adapted to the characteristics of the program running on it.

3.3.1 Reuse domains and memoization tables

Besides reusable and reused-only modes, different reuse domains are possible for RST, and they will determine different fields for the memoization tables. We will consider that memory accesses are not allowed in the regular reuse domain for RST, but in the cases that a reuse domain with memory accesses is considered, we will call it **RSTm** (RST with memory reuse).

When an instruction in the reuse domain is committed, RST may start to build a trace. The input context and output context are built with the values and register indexes from instructions in the reuse domain. A trace is finished when:

(i) a non-redundant instruction is found and the mechanism is in **reused-only** mode;

(ii) an instruction which does not belong to the reuse domain occurs;

(iii) a load/store instruction is found (in RST, but not in RSTm);

(iv) resource limits are reached (as the maximum number of inputs or outputs).

Figure 3.2 depicts an entry in the Memo_Table_G table for the case where memory accesses are not in the reuse domain (COSTA, 2001). This table holds reusable instructions, with their inputs and outputs.

bits	*30*	*32*	*32*	*32*	*1*	*1*	*1*
	pc	sv_1	sv_2	res/targ	jmp	brc	btk

Figure 3.2: Entry in Memo_Table_G, no memory accesses allowed

The fields are as follow:

- **PC**: the instruction address
- **sv1** and **sv2**: the source values for that instruction
- **res/targ**: the result value or the target address (in case of a branch)
- **jmp**: unconditional branch instruction when set
- **brc**: conditional branch instruction when set
- **btaken**: if `brc` is set, then this field signals if the branch is taken or not

Figure 3.3 shows the same Memo_Table_G entry but for the case where memory instructions can be fully reused (VIANA, 2002). It has two extra fields:

- **maddr**: the load or store address
- **mval**: used to mark the validity of a load value

Figure 3.4 shows the structure of an entry in the Memo_Table_T table for DTM without memory reuse. Each entry is divided in the following fields:

bits	30	32	32	32	32	1	1	1	1
	pc	sv_1	sv_2	res/targ	maddr	mval	jmp	brc	btk

Figure 3.3: Entry in Memo_Table_G, memory accesses allowed

- **pc**: the address of the first instruction in the trace
- **npc**: the address of the next instruction after the trace
- **icr**: the register names for each entry in the input context
- **icv**: values for the registers in the input context
- **ocr**: the register names for each entry in the output context (live registers)
- **ocv**: values for the output registers
- **bmsk**: bitmap used to mark branches inside the trace
- **btk**: direction of branches inside the trace

The address of the first instruction of a trace is stored in *pc*, and the address of the next instruction after the trace is stored in *npc*. The *npc* field is used to set the PC and to skip the instructions belonging to the trace when it is reused. Fields *icv* and *icr* store the input values and the register indexes, while *ocv* and *ocr* store the output values and the corresponding register indexes. *bmsk* is a bitmap to mark branches that occur inside the trace, while *btk* bits are set when branches are taken. These two last fields are used to update the branch prediction as in DTM (COSTA, 2001).

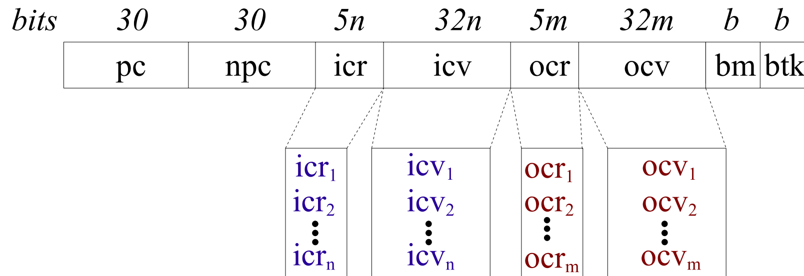

Figure 3.4: Entry in Memo_Table_T, no memory accesses allowed

There are some extra fields that are required when memory accesses are allowed in traces. Figure 3.5 shows a Memo_Table_T entry for RSTm. Most of the fields are the same as found in RST, with the following differences:

- For each *ocv* entry, there are more 3 fields:
  - *me*, marking values from memory accesses;
  - *l/s*, marking if the access is a load or a store;
  - *siz*, the size of the memory access (half, single, double).

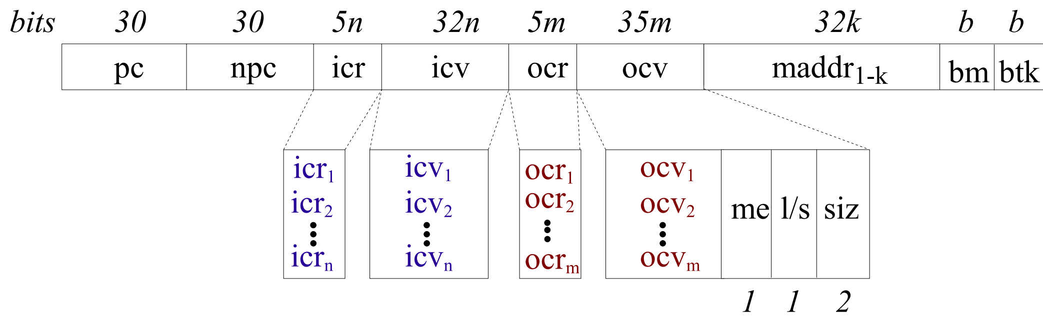

Figure 3.5: Entry in Memo_Table_T, memory accesses allowed

- The fields *maddr* hold the addresses accessed by the memory instructions in the order that they appear in the contexts.

Those fields are slightly different from the ones determined by Viana (VIANA, 2002), because he used a different ISA (SPARC v9, while we use the MIPS-IV-like PISA architecture).

Every instruction in the reuse domain is searched in both tables. If an entry is found in *Memo_Table_T* and its input scope matches the present register values, the trace beginning with that instruction is reused; if an entry is not found in *Memo_Table_T* but in *Memo_Table_G*, the instruction is reused. Trace reuse may cross branch boundaries and collapse true data dependencies in a single cycle, increasing performance. A trace will be speculatively reused if:

(i) There is no trace with all inputs known and correct;

(ii) There is at least a trace with unknown input values whose known input values match the current values; and

(iii) The confidence mechanism allows the prediction of the trace.

Therefore, the precedence of speculative trace reuse is above instruction reuse, but below trace reuse (Figure 3.6).

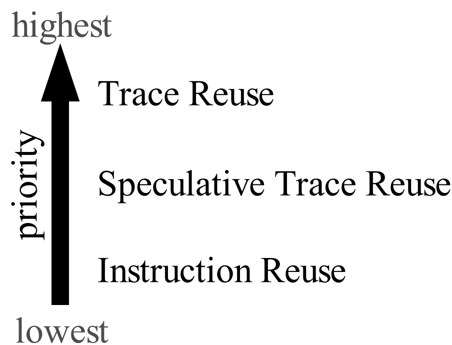

Figure 3.6: Precedence of different reuse types in RST

```
i1: add r1, r2, r3
i2: sub r4, r3, r5
b3: beq r4, r5, i7
i4: mul r2, r3, r4
i5: sw r2,  (r20)
i6: add r3, r4, r5
....
i7: div r4, r5, r7
b8: bne r4, r8, i11
i9: sub r5, r7, r8
i10: fadd_s r9, r1, r6
....
i11: xor r3, r4, r5
i12: lw r7, r3(100)
il3: sub r8, r7, r6
```

Figure 3.7: Code snippet

3.3.2 Trace construction and reuse for DTM

The next figures depict the trace formation process for the assembly code of Figure 3.7. For instructions with three operands, the first one is the destination register, and the other two are the input registers.

First, we will present how traces are created in DTM, allowing only **redundant** instructions (already in Memo_Table_G) in the traces. Figure 3.8 shows instructions in the reuse domain being identified (gray circles) and stored in Memo_Table_G. The black circle represents a load instruction, which has a different treatment as memory accesses are not being considered as part of the reuse domain.

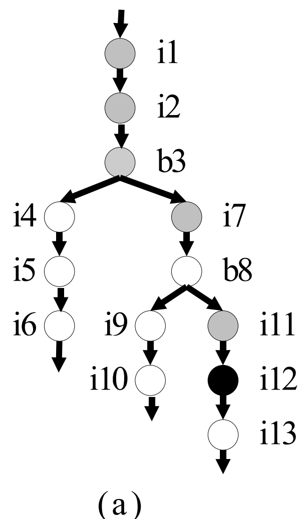

Figure 3.8: Storing reusable instructions

In the next execution shown in Figure 3.9(b), these instructions are reused, marked as redundant, and a trace is formed, until an instruction that does not belong

to the reuse domain or is not redundant is found. In this case, instruction *i7* terminates the trace construction as *b8* is not redundant (not found in Memo_Table_G). This trace is then memoized and stored in Memo_Table_T.

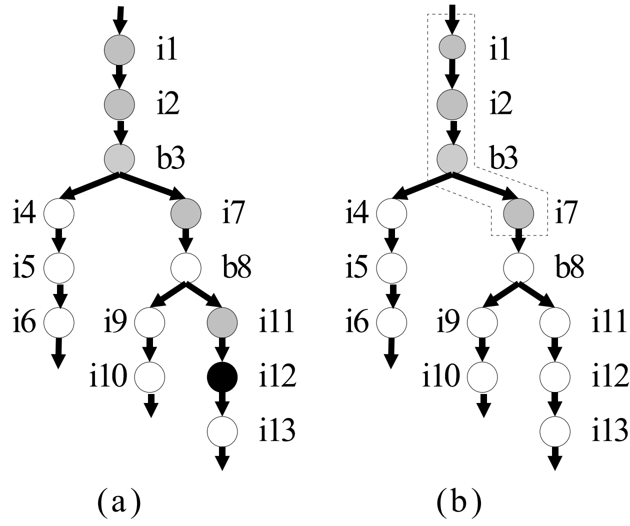

Figure 3.9: Trace formation

Figure 3.10(c) shows the next time execution reaches the beginning of this trace with the same inputs. At this point, the memoized trace is reused; i.e., the previous values are written in the output registers. In this example, the compared input registers are *r2*, *r3*, *r5*, *r7* and *r8*. If the inputs match the current values, the output context containing *r1* and *r4* is loaded into these registers as the outputs of the trace. Thus, all instructions inside the trace are essentially collapsed into the checking of the inputs and storing of the outputs. The instruction fetch is redirected to the next address after the trace.

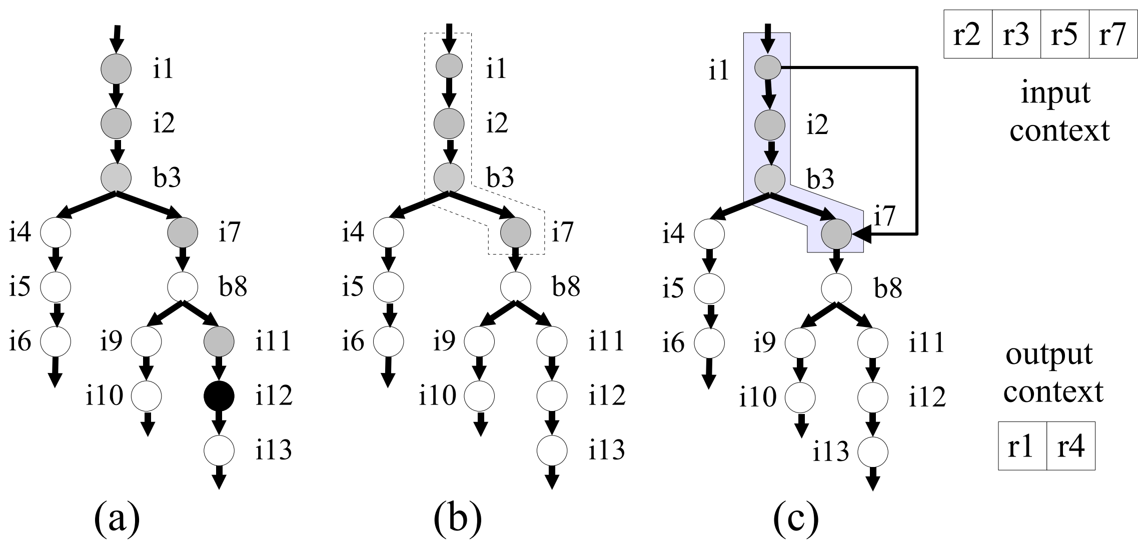

Figure 3.10: Trace construction, reusing a trace

3.3.3 Trace construction and reuse for RST

Figure 3.11 shows the process of trace creation using RST's policy of allowing even instructions that have not been reused yet to be included in a trace in formation. Instructions are included in the trace until an instruction that does not belong to the reuse domain is found. In this case, a load instruction (black circle) terminates trace construction, as memory accesses do not belong to the reuse domain. In this case, only the address calculation is kept in the trace, and the memory access will be issued as a regular load if the trace is reused. This trace is then memoized and stored in Memo_Table_T.

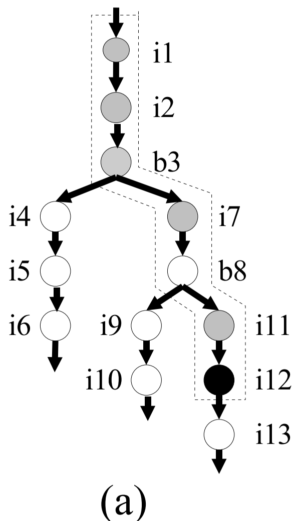

Figure 3.11: Storing reusable instructions

Figure 3.12(b) shows the next time execution reaches the beginning of this trace with the same inputs, when the memoized trace is reused; i.e., the previous values in the output context are written into the output registers. In this example, the compared input registers are *r2*, *r3*, *r5*, *r7* and *r8*. If *(i)* the inputs match the actual registers, or if *(ii)* there are inputs that are not ready and the known inputs match the registers and the confidence mechanism allows the prediction, then the values stored for *r1*, *r4* and *r3* are loaded into these registers as the outputs of the trace. Thus, all instructions inside the trace are essentially collapsed into the checking of the inputs and storing of the outputs. Instruction fetch is redirected to the next address after the trace, obtained from the field *npc*.

The main differences to DTM's policy and their consequences are:

- A trace is formed faster in RST than in DTM, because instructions do not need to be marked as redundant to be included in a trace, therefore they do not need to be included first in Memo_Table_G before making their way into a trace;

- Potentially, more traces are created in RST, as there are more candidate instructions;

- Traces may be less redundant than in DTM, as the included instructions are

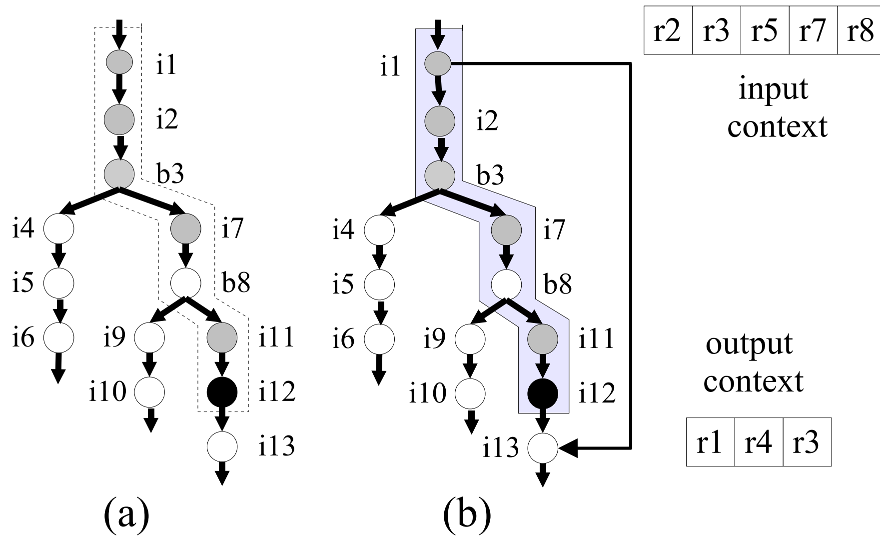

Figure 3.12: Trace construction, reusing a trace

not necessarily going to show redundancy (they may never occur again with the same inputs);

- On the other hand, instructions that show redundancy may be included earlier in a trace, and RST may benefit from traces that would not be formed in DTM;
- Memo_Table_T is accessed more often to store traces and it may become more polluted due to traces with small redundancy being stored.

3.3.4 Misprediction detection and recovery

The late tests are similar to those used for other value prediction mechanisms. When a trace is speculatively reused, the predicted value and information about the producers of predicted values are stored. When the producers finish executing, the predicted values and the actual outputs are compared. If they match, the commit stage is informed. The outputs of the speculatively reused trace can then be committed, and executed instructions following the trace can also be committed. If any of them does not match, then the recovery mechanism (the same used for branch mispredictions) is activated, instructions after the trace and trace outputs are discarded, and fetch is redirected.

The late test needs comparison hardware in the writeback stage to test for mispredictions. For each input that can be predicted in a trace, the mechanism will need a comparator after each functional unit that can produce a value of the same type of those that are predicted.

3.4 Comparison with related techniques

Compared with instruction reuse techniques (SODANI; SOHI, 1998), RST has all the benefits of trace reuse as detailed by Costa (COSTA, 2001), such as the potential for collapsing critical paths into a single cycle, improving branch predictions, and reducing the dispatch bandwidth. It also does not need to verify chains of dependent instructions during reuse times, as this information is encapsulated by the structure

adopted to store traces. Consequently, the reuse test tends to be simpler than instruction reuse with dependence checking.

Alternative schemes that just combine two unrelated mechanisms in a single architecture (LIAO; SHIEH, 2002) suffer from the increased table sizes required to provide low misprediction rates. Our approach does not waste resources on additional prediction tables, as it uses the values already stored in the memoization table.

RST also has the advantage of not being dependent on the compiler or ISA modifications such as needed in block and sub-block reuse (HUANG; LILJA, 1999; WU; CHEN; FANG, 2001), allowing the execution of legacy code without modifications.

Compared to the Combined Dynamic Predictor (HUANG; CHOI; LILJA, 1999), our approach has the advantage of having a more flexible reuse unit, the trace, thus being able to obtain benefits not only from long-latency instructions.

Unlike other trace reuse mechanisms (COSTA; FRANÇA; CHAVES FILHO, 2000; GONZÁLEZ; TUBELLA; MOLINA, 1999), RST can speculatively reuse traces even when inputs are not ready, allowing the exploration of a large amount of redundancy that would not be reusable otherwise. Correctly prediction of values also exposes more instruction level parallelism for the architecture, allowing to invest against the limits imposed by true data dependencies with increased efficiency.

Previous value prediction techniques like those developed by (GABBAY; MENDELSON, 1996; LIPASTI; SHEN, 1996; WANG; FRANKLIN, 1997a; SAZEIDES; SMITH, 1997) may increase resource contention problems as even misspeculations will need functional units and issue width, while RST is more conservative: predicted traces are not executed, but speculatively reused. Instructions inside a trace will not occupy dispatch bandwidth, issue bandwidth nor functional units.

RST does not require elaborated techniques to deal with mispredictions, as an additional core to compute a non-speculative version of the predicted code (NAKRA; GUPTA; SOFFA, 1999; WU; CHEN; FANG, 2001; KOUSHIRO; SATO; ARITA, 2003), or detecting the chains of dependent instructions to be re-executed as proposed by Reinman and Calder (REINMAN; CALDER, 1998), although these mechanisms could improve performance in the case of mispredictions. All our experiments considered a simple squash scheme where all instructions after a misspeculation are discarded, and the fetch is redirected to the first instruction that used a misspeculated value. This is the same way that branch misprediction is dealt in most processors. Hence, the additional cost for RST in terms of misprediction recovery is only the extra comparisons in the writeback stage to verify the predicted values against the computed ones.

The cost of extra hardware for RST in relation to DTM (COSTA, 2001) is composed by the extra tables for the confidence mechanism and the late tests in the writeback stage to verify the speculations. The confidence mechanism is a small addition in terms of hardware area for the configurations that we considered, less than 2% of additional hardware than necessary for DTM in most of the analyzed configurations.

3.5 Contributions

This Section briefly summarizes the main contributions of this thesis as follows:

1. A speculative trace reuse framework that can be modified to fit different ar-

chitectures and requires minimal extra hardware compared to regular trace reuse;

2. Experimental results for an instance of RST implemented in a 19-stage, superscalar architecture;
3. Characterization of traces and the predictability of their inputs;
4. Analysis of speculative trace reuse limits; and
5. Implementation details and constraints that were not fully explored in previous trace reuse works.

Besides these aspects, we detail in Chapter 5 all the improvements in the simulator that we implemented in order to develop this work but that can be used for other measurements.

4 A RST ARCHITECTURE

Person who say it cannot be done should not interrupt person doing it.

Chinese Proverb

DTM satisfies a number of requirements for reuse through speculation on traces, and previous studies have shown that it achieves good performance and presents advantages over other reuse techniques (COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA, 2001). Hence, we have chose DTM as a framework to implement RST, and extended it to include trace speculation, among other features to improve frequency and size of reused traces.

In this Chapter, we present an implementation of RST over DTM in a superscalar architecture. First, we show the pipeline for a DTM implementation in Section 4.1, then we compare our RST's pipeline to it in Section 4.2, pointing out the modifications that are required to add speculative trace reuse. Each stage necessary for RST is discussed, and details like how register renaming is performed and how mispredictions are detected and recovered are discussed.

4.1 DTM's pipeline

Figure 4.1 presents DTM's pipeline for a superscalar architecture. The fetch stage sends the program counter (PC) for the next instruction to both the memory hierarchy and the first DTM stage, called **DS1** (DTM Stage 1). This address is used to access Memo_Table_G and Memo_Table_T in parallel. Reuse candidates are sent to **DS2**, where their inputs are read. In this same stage, the actual values are compared with the candidates' inputs, and if there is a match, DTM can reuse an instruction or a trace. Traces have precedence over instructions, as they may provide better performance improvements than isolated instructions. Stage **DS3** is responsible for identifying instructions to be stored in Memo_Table_G and to create new traces from the reused instructions.

DTM's pipeline is parallel to the instruction pipeline, thus it does not add extra stages to the path that instructions must complete in order to be committed. The implementation of DTM does not significantly impart performance for instructions that are not reused, and improve performance for those that can be reused by skipping pipeline stages.

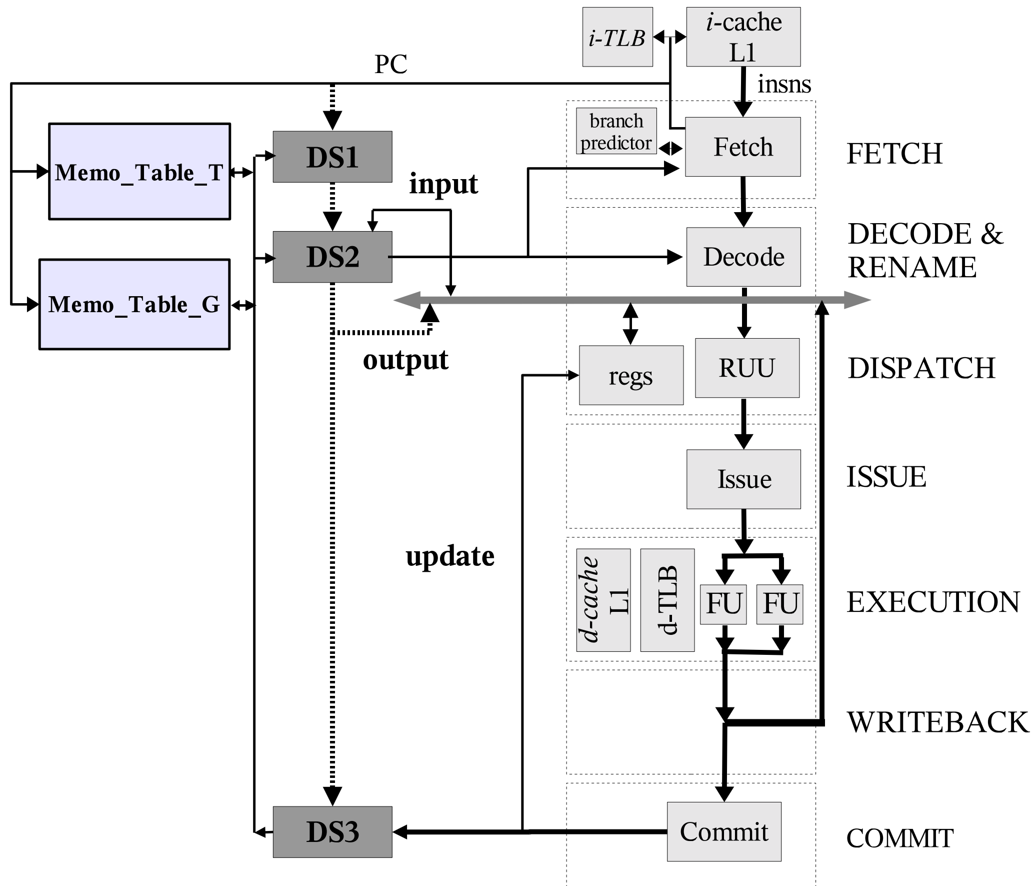

Figure 4.1: Pipeline for a DTM architecture

4.2 RST's pipeline

Figure 4.2 shows RST's pipeline based on a superscalar architecture (PILLA et al., 2002), built over the DTM architecture. There are four stages in RST, and they are parallel to the main instruction pipeline just like DTM's stages. Thus, the delay of the RST stages is not added to the main pipeline, and there is no increase in the number of cycles to execute an instruction neither decrease on the clock rate.

For RST, the main modifications occur in stages **RS2** and **RS4**. There is also an extra intermediate stage, RS3, in parallel with the writeback stage designed to identify mispredictions. The stages divide the work of speculative reuse as follow:

- Stage **RS1** works like DS1;
- Stage **RS2** is based on DS2, but with some differences in the reuse test;
- Stage **RS3** handles the misprediction test (and does not have equilavent in DTM); and
- Stage **RS4** is the equivalent of DS3.

In the next subsections, each stage is described, and in Subsection 4.2.5 the integration of all stages is depicted in an overview picture.

Figure 4.2: Pipeline for a RST architecture

4.2.1 Stage RS1

Stage **RS1** works just like DS1, fetching reuse candidates from both Memo_Table_G and Memo_Table_T. The PC address of the current instruction is sent to RS1 and the tables are accessed. Figure 4.3 shows the input (PC) and the outputs (candidates for reuse) of stage RS1. The outputs 1 and 2 are directly sent to the next RST stage, RS2, for the reuse test.

Inputs 5 and 8 provide updates on traces and instructions that were non-speculatively reused in stage RS2, or whose predictions were confirmed in stage RS3. This stage also receives reusable instructions and trace candidates to be stored from stage RS4 (inputs 11 and 12), and they are stored after the tables are searched and the entry is not found there. The search and comparison that are necessary to store a new trace or instruction are not detailed in Figure 4.3 as the operation is not different from what is required for other tables.

4.2.2 Stage RS2

Stage **RS2** is different from DS2, as the reuse test in RST allows that traces with unknown input values may be speculatively reused, using the previous values stored in the input context. In RS2, if one or more values in the input context are not ready (there is a true data dependency on an instruction which has not been finished yet) but the remaining values match the current values, then a confidence mecha-

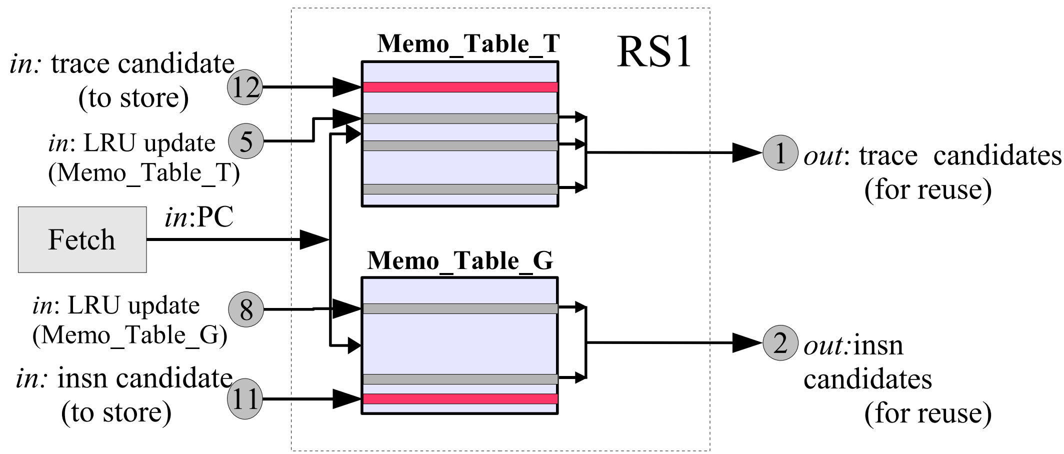

Figure 4.3: Details of stage RS1

nism (CALDER; REINMAN; TULLSEN, 1999) is accessed to verify if the value should be predicted or not. This mechanism is important to reduce misprediction and the associated penalties, as we show in Chapter 7.

If a confidence mechanism is to be used in RST, one of the stages RS1 or RS2 is modified to include it. The confidence tables may be accessed in parallel with the trace access (in RS1) or the access may be left for the case where the need for the confidence lookup is assured (in RS2). The implementation will depend on the pipeline depth that is being considered, and how many cycles are necessary to access the confidence mechanism and to update it. In this work, we assume that there is enough time to access the confidence mechanism in RS2, and therefore it is consulted only when strictly necessary.

Figure 4.4 depicts the structure of stage RS2. For traces, the registers to be accessed are indicated in the Memo_Table_T entry, but for instructions it uses the values that are decoded by the instruction pipeline in order to save circuit area. After consulting the register file for register values, they are compared with the instances from Memo_Table_T and Memo_Table_G. Trace inputs will be predicted only if the confidence allows it. The possible outputs are:

1. A reused trace – output values (output 7), LRU update to stage RS1 (output 5), and eventually a new PC (output 6);

2. A **speculatively** reused trace and its **predicted inputs** (outputs 3, 4, 6, 7);

3. A reused instruction – output value (output 7), LRU update to stage RS1 (output 8), and eventually a new PC (output 7).

If the architecture does not use register renaming to deal with data dependencies, then each value in the output context (whether it is speculatively reused or not) will occupy one entry in the Reorder Buffer (ROB) and another one in a reservation station. Subsequent instructions do not need to access intermediate values created inside a trace (and they would not be able, as they are not kept in Memo_Table_T, just the final value for each entry in the output context). For example, if a trace encloses 10 instructions but only two logical registers are written, then only two ROBs

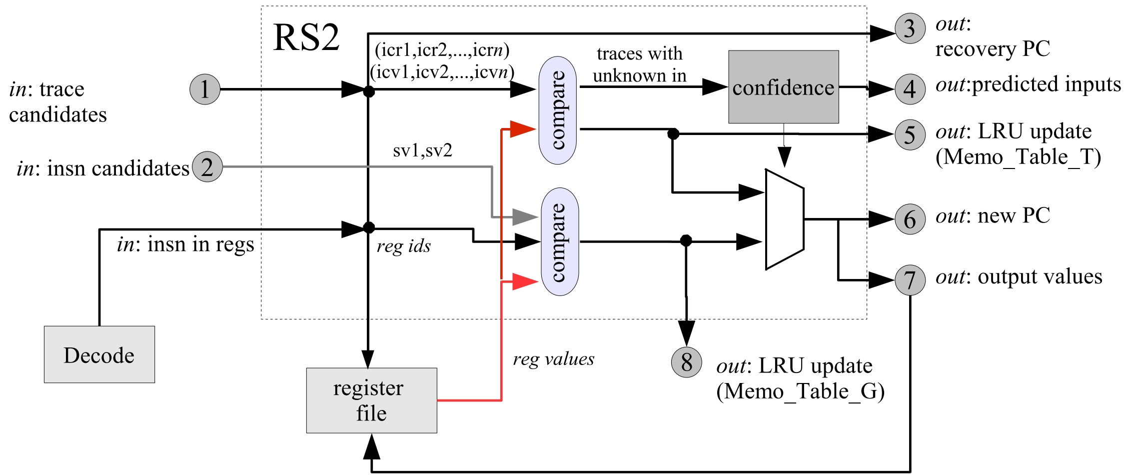

Figure 4.4: Details of stage RS2

and reservation stations will be necessary to hold all results. Therefore, 8 entries can be saved for other instructions in this case.

But if there is a register renaming stage (as in most modern architectures as Pentium 4, Alpha 21264, PowerPC and so on (HENNESSY; PATTERSON, 2003)), then there is an issue to be treated by RST. While the RS2 stage is testing trace candidates for reuse opportunities, the rename stage continues its register renaming. By the time that RST can decide if a trace is going to be reused or not, the tables that map logical registers onto physical registers and keep the active list will be in an inconsistent state.

For example, if some instructions enter rename in the following order and write the logical output registers (Table 4.1):

Table 4.1: Sequence of instructions in the rename stage

Order	**Instruction**	**Logical register**
1	i1	r3
2	i2	r2
3	i3	r3
4	i4	r2
5	i5	r2
6	i6	r5
7	i7	r12

Then, if a register renaming mechanism is used, the following mapping from logical to physical registers may occur (Table 4.2):

For each instruction writing a logical register, a free physical register is assigned to hold the value, preventing speculative update of the architecture state. Instructions after a mapping that read registers will always use the last assignment. Now, assume that instructions *i2* to *i5* are going to be part of a reused trace. To cope with the assignments that are not necessary because of trace reuse, a list of all mappings

Table 4.2: Mappings from register renaming

Order	Instruction	Logical register	Physical register
1	i1	r3	p54
2	i2	r2	p56
3	i3	r3	p57
4	i4	r2	p59
5	i5	r2	p70
6	i6	r5	p72
7	i7	r12	p75

done in the rename stages (similar to Table 4.2) can be analyzed just after the rename stage, and unnecessary bindings can be freed and rolled back to the previous mapping. Instructions after the trace are not affected.

For the current example, the list would be analyzed from the last instruction in the trace up to the first instruction in the trace. Logic registers *r2* and *r3* are written by the trace. Thus, we must search for the last write for each of these registers in the list (Table 4.3, where instructions belonging to the trace are marked with gray, and darker gray is used for instructions with the last mapping to the registers in the output context):

Table 4.3: Discovering mappings to be freed

Order	Instruction	Logical register	Physical register
1	i1	r3	p54
2	i2	r2	p56
3	i3	**r3**	p57
4	i4	r2	p59
5	i5	**r2**	p70
6	i6	r5	p72
7	i7	r12	p75

From this point, the processor knows that it can free the mappings for registers *p56* and *p59*, and that *p57* and *p70* provide the last values inside the trace for instructions after it. After freeing unnecessary mappings, we would have (Table 4.4):

The mappings from reused instructions are freed but for the last assignments, which will be used to write the output scope of the reused trace. Instructions before the trace are not affected, and the mapping for instruction *i1* is kept as it appears before the trace.

Another possible case is that the trace takes a different way than the one that was originally fetched by the processor. In this case, mappings after the trace will be removed, as these instructions are not going to be executed anyway.

RST and other reuse techniques that need to read register values to test against previously stored values may potentially increase the pressure on the register file, needing more read ports than architectures without reuse. In a previous study for

Table 4.4: State after freeing unnecessary mappings

Order	Instruction	Logical register	Physical register
1	i1	r3	p54
3	i3	r3	p57
5	i5	r2	p70
6	i6	r5	p72
7	i7	r12	p75

regular trace reuse (COSTA, 2001), DTM did not present an increase on demand for read ports, and even if RST presents an increased number of register reads per cycle, their access can be divided by the cycles before the reuse test. As RS2 is defined later in parallel to decode/rename with 4 stages, reading registers for RST could be divided among the stages before the reuse test without stalling the pipeline.

4.2.3 Stage RS3

Stage **RS3** handles the misprediction test. It looks for the results of predicted instructions from the writeback stage and compares the actual values with the predicted values. If a misprediction is found, then fetch is redirected to the beginning of the trace, and all instructions after the trace are squashed. This mechanism is the same used for branch mispredictions, which is already found in superscalar architectures. Instructions after the producer of a mispredicted value and before the mispeculated trace are not discarded, because they do not use the value predicted.

Figure 4.5 details the structure of stage RS3. The inputs that come from the previous stage RS2 are the recovery PC (3) and the predicted inputs (4) from a predicted trace. The predicted inputs are formed by the instruction id of the producer (the index of that instruction in the ROB or in the Register Update Unit RUU, depending on the architecture design), the register index and the value. The instruction id is used to index a table where recovery information is stored (the Recovery Table, discussed below). A trace with more than one predicted value will have multiple entries in this table.

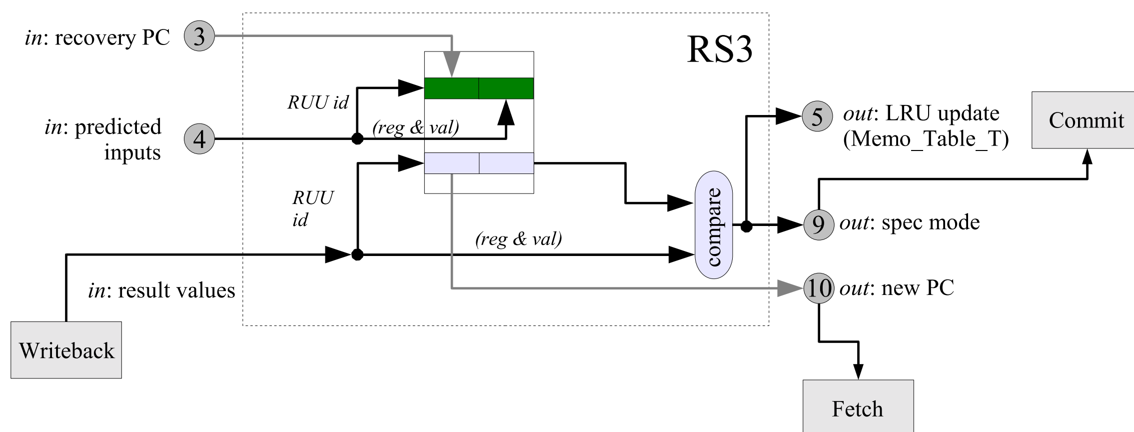

Figure 4.5: Details of stage RS3

Besides storing new predictions and recovery information, this stage is also responsible for testing the values provided by the writeback stage to detect mispredictions. The same instruction id is used to index the recovery table, and the values are compared to the predicted ones (stored in the table). The outputs of RS3 are the speculative mode for that trace (if it was mispredicted or not, output 9), which is sent to the commit stage; a possible LRU update to stage RS1 (output 5); and a possible new PC (output 10), to be sent to the fetch stage if a misprediction is found.

When more than one input can be predicted by trace, the table must handle multiple tests for the same trace. A counter can be added to the entries and decremented each time a correct value prediction is detected. When all the necessary tests for a trace were completed, the speculative mode of the instructions following the trace would be set accordingly.

In the case of a misprediction, the table should be searched to squash all entries that are related to traces predicted after the mispredicted one, and the mispredicted trace itself. This can be solved by using a ROB-like table, where information about predictions are store in order, which is called Recovery Table (RT). Each entry on this recovery table has the following fields (Figure 4.6):

- *prid* is the instruction of the producer for that value (size depends on the number of ROB or RUU entries, and it is $log_2(n)$ bits);
- *crid* is the instruction id of the consumer (first ROB or RUU entry occupied by the trace), where other recovery information may be found;
- *val* is the predicted value, to be checked against the calculated value;
- *lru* is a pointer for the trace entry in Memo_Table_T to be updated in the case of correct prediction (it may be unnecessary if LRU is updated in RS2);
- *cid* is a pointer for the confidence table entry to be updated (it may be unnecessary depending on the confidence mechanism);
- *np* is the number of predictions made for the current trace minus one (the size will be $log_2(n-1)$ bits, where n is the maximum number of predictions for a trace);
- *val* is a bit that marks if the entry is valid or not.

bits	*a*	*a*	*32*	*b*	*c*	*d*	*1*
	prid	crid	val	lru	cid	np	vl

Figure 4.6: Recovery Table entry

The *crid* field can be used not only to find the PC to recover in case of misprediction or the point in the ROB from where instructions should be squashed, but also to group entries in RT that are related to the same trace.

For example, if there is a speculatively reused trace with one predicted input, then the table may look like Figure 4.7. In this figure, the result of the operation in

	prid	crid	val	lru	cid	np	vl
0	4	5	0x8888	55	120	0	1
							0
							0
n							0

Figure 4.7: RT with one entry

the 4th ROB entry will be predicted as 0x8888 for the trace stored in the #55 entry in Memo_Table_T.

If another trace, but now with two predicted inputs, is stored in RT, it may look like Figure 4.8. Note that the field *np* is 1 for the two new entries, which is the number of predicted values for that trace minus 1.

	prid	crid	val	lru	cid	np	vl
0	4	5	0x8888	55	120	0	1
	21	30	0x0076	46	033	1	1
	9	30	0x8899	46	001	1	1
n							0

Figure 4.8: RT with three entries

Now, if the value of the first trace is mispredicted, then all entries in RT will be squashed, fetch will be redirected to the PC of the 5th ROB entry, all the entries in the ROB after the misprediction and the misprediction itself will be squashed, and the confidence entry #120 (*cid* field) will be updated with a misprediction.

On the other hand, if the predicted value matches the actual, then the confidence will be updated with a correct prediction, the consumer will be marked as non-speculative and may be committed from that point, LRU for trace #55 will be updated, and the RT entry will be released.

Figure 4.9 depicts another possible situation, where entry #1 in RT is resolved as correctly predicted. Then, the valid bit *vl* is reset, and the other entries for that specific trace (entry #2) will have the number of predicted values decreased. When the last prediction is confirmed (in the case of no mispredictions), then the LRU for the speculatively reused trace is updated, and all the rest of the treatment for correctly predicted traces is applied. If one of the predictions is incorrect, then the treatment is the same as specified before for a mispredicted trace with a single prediction.

The Recovery Table may also assume other configurations, such as a two-level structure to avoid multiple searches for the same trace id in a given cycle. However, this is not very significant as the RT is very small (less than 8 entries are necessary for prediction of two values per trace).

	prid	crid	val	lru	cid	np	vl
0	4	5	0x8888	55	120	0	1
	21	30	0x0076	46	033	1	0
	9	30	0x8899	46	001	0	1
n							0

Figure 4.9: Resolving a RT entry

4.2.4 Stage RS4

Stage **RS4** is the equivalent of DS3 in DTM. It detects and stores redundant instructions and traces. Depending on the policy, only reused instructions (as in DTM) may form traces, or instructions from the reuse domain that were not reused may also be included. All instructions after a mispredicted trace are squashed here, and the fetch is redirected to the address where the trace started. Figure 4.10 shows the structure of stage RS4, with two buffers dedicated to create the input and the output scopes, and a buffer to store branch bitmaps for the trace in construction.

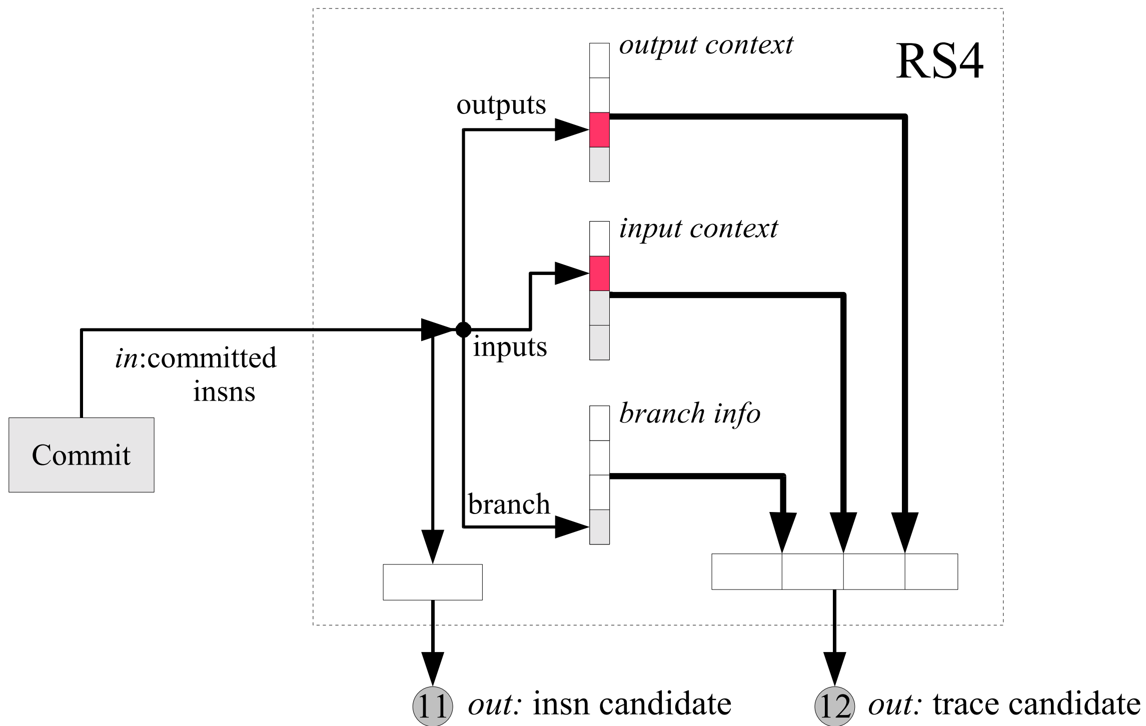

Figure 4.10: Details of stage RS4

The inputs received from commit are divided in: *(i)* inputs form the *input context*, *(ii)* outputs form the *output context*, and when the instruction is a branch, it will also be used to construct *(iii)* the branch mask. Trace construction is described in Chapter 3.

The output 11 is comprised by information about all the reusable instructions that were not reused but that are in the reuse domain. Stage RS1 will search for

them and, if they are not in Memo_Table_G yet, they will be included there. The same is valid for output 12, which is a finished trace. It will be searched in Memo_Table_T and stored there, if necessary.

Depending on the policy for entry allocation on Memo_Table_G and Memo_Table_T, this stage may also send information to RS1 about which instructions and traces should have their position on the LRU list updated.

4.2.5 Integrating all stages

Figure 4.11 shows how all the RST stages are connected. Some labels have been removed to better present the overall picture of RST. For details of each stage, refeer to the previous subsections.

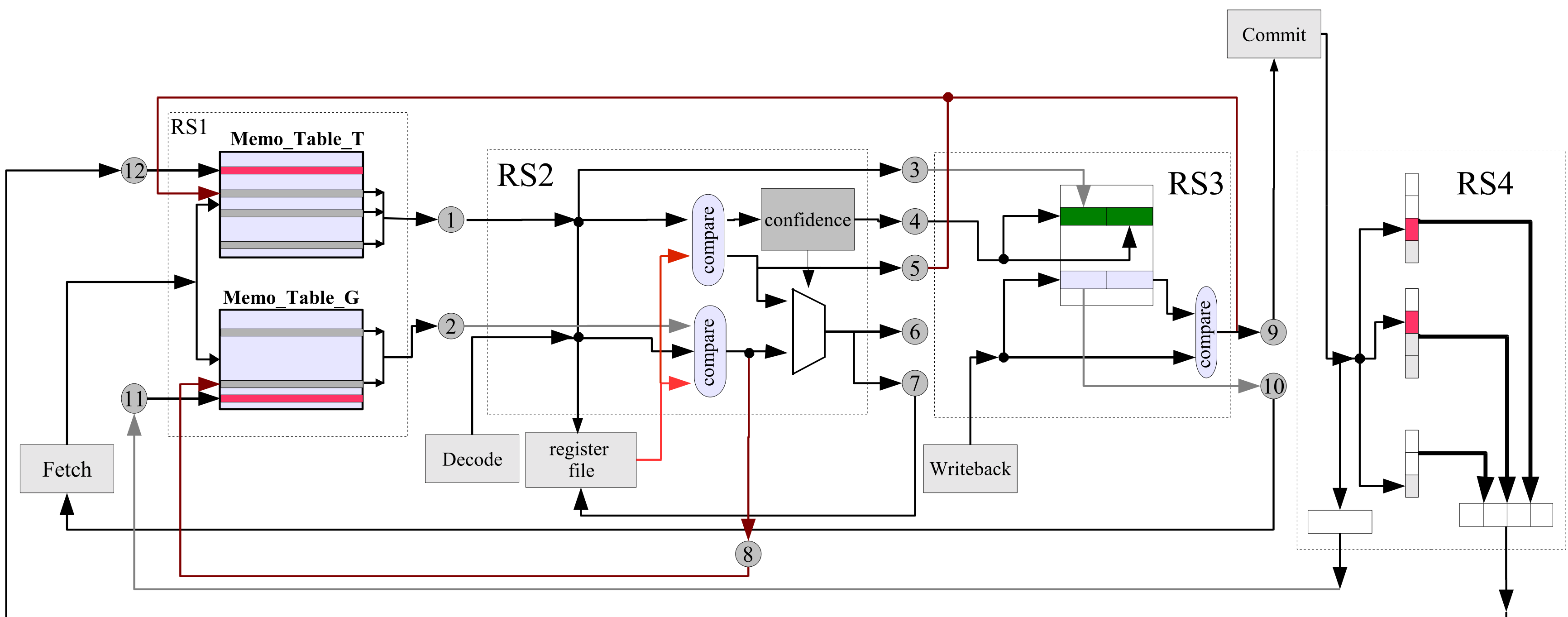

Figure 4.11: Connections for RST stages

5 EXPERIMENTAL ENVIRONMENT

All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of nitty frustrations drive away all but those who habitually focus on the end goal. Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists. But however the selection process works, the result is indisputable: "This time it will surely run," or "I just found the last bug."

Frederick Brooks, "The Mythical Man Month"

In this Chapter, we present the characteristics of the simulation environment used to obtain the results for our work. First, we introduce the simulator that we developed in Section 5.1. Then, we present the chosen workload in Section 5.2. Finally, the configurations for the simulated architectures are depicted in Section 5.3.

5.1 Simulator

Our RST simulator, called *sim-rst*, was developed on the *sim-outorder* simulator from the SimpleScalar Tool Set (BURGER; AUSTIN, 1997; AUSTIN; LARSON; ERNST, 2002), version 3.0b, using parts from the original DTM simulator implemented on the version 2.0 of the same tool set. It simulates in the instruction level, with most details of a superscalar processor such as out-of-order execution and in-order committing, memory hierarchy, and modeling of resources.

Besides the additional functions and modifications to allow the speculative reuse of traces, our simulator also extends the original six-stage pipeline from *sim-outorder* to deeper pipelines with arbitrary number of stages to better resemble modern, deeply pipelined processors. The extra stages are simulated by adding counters to some structures, as the instruction fetch queue. When an instruction is inserted there, the counter for that entry is set with the value of the current simulation cycle plus the number of stages until the next "real" stage, the decode stage; then, the decode stage will only pop instructions from the fetch queue when the current simulation cycle is greater than the stored value in the entries. The same process is used for the other stages.

We also divided the dispatch stage, where originally was simulated register renaming and dispatch, to allow reused instructions writing their results before the issue stage.

Reused traces write their outputs in the next cycle; therefore, they do not affect the reuse tests being done in the same cycle, and there is no chaining of reused

instructions nor traces. This is done in order to conform with a real pipeline implementation, where these local writebacks could increase hardware complexity and reduce clock rates.

Another modification was the extra third level of cache, also intended to simulate with more details the characteristics seen in state-of-art superscalar processors.

5.2 Workload

The benchmarks simulated in this work is composed by programs and input sets from the SPEC CPU 95 (SPEC – Standard Performance Evaluation Corporation, 1995) and SPEC CPU 2000 (HENNING, 2000), which have been developed to test processor and memory performance. The programs that we simulated are shown in Tables 5.1 and 5.2. The first column presents the name of the benchmark, the second column has a short summary of what the benchmark does, the third column shows the input set used in the simulations, and the last column presents the number of instructions that were simulated each time that specific benchmark was run.

The number of committed instructions refer to non-speculative instructions which have been retired normally in the commit stage. Each simulation was executed with a maximum of 500 million committed instructions or to its completion.

Table 5.1: Simulated workloads – SPEC95int

Benchmark	Description	Input	Committed
cc1	GNU C compiler	expr.i (ref)	225.7 millions
compress95	Compression and decompression	test.in (train)	35.5 millions
go	Plays the game of GO	9stone21 (ref)	131.8 millions
ijpeg	Image compression	vigo.ppm (train)	244.3 millions
li	LISP interpreter	deriv.lsp (ref)	500 millions
m88ksim	processor simulator	ctl.raw (ref)	244.7 millions
perl	Perl interpreter	primes.in (ref)	500 millions
vortex	Object-oriented database	vortex.in (ref)	500 millions

Table 5.2: Simulated workloads – SPEC2000int

Benchmark	Description	Input	Committed
cc1	GNU C compiler	cp-decl.i (train)	500 millions
gzip	Compression	lgred (random)	500 millions
parser	Word processing	lgred	500 millions
vortex	Object-oriented database	lgred	500 millions

Previous studied with regular trace reuse (COSTA, 2001) showed that floating-point benchmarks also benefit from the reuse of integer instructions, but programs based on integer operations are more suitable for trace reuse. In the present simulations, we decided to focus on the integer benchmarks.

All the benchmarks from SPEC95int were simulated, but only a subset of the SPEC2000int was chosen because *(i)* the current tools could not cleanly compile them for the target architecture, or *(ii)* there is no suitable input set for architecture simulation for the benchmark. Simulating the same number of instructions that are executed in the evaluation of real processors is not feasible with the current computing power, and as the benchmarks increase in complexity in the same pace as computing power, it is not likely to be feasible in a near future. On the other hand, simulating just a small number of instructions from those same benchmarks would also present meaningless results that could not resemble at all the targets being simulated. Hence our decision of not running some benchmarks because their results would not represent the characteristics that we wanted to test.

The chosen ISA was the PISA (Portable Instruction Set Architecture), a superset from the MIPS-IV ISA, due to the support in the SimpleScalar Tool Set, the possibility of running on many host architectures, and the compatibility of the tools with the x86 computers where they would be simulated. All benchmarks were compiled with *GCC* cross-compiler version 2.7.2.3 (CHONG, 1999) and binutils 2.5.2, little-endiam for x86 architectures.

Different computers with different versions of libraries or environment variables could lead to some degree of variation in the execution of our simulations. To increase the reliability of results, I/O traces were created for each workload, decreasing the difference among simulations in those heterogeneous computers.

For the simulations, we used three different PC clusters: an IBM Pentium III cluster with 12 nodes in the University of Pittsburgh (thanks to Dr. Mossé), an Itautec dual Pentium III cluster with 16 nodes in our laboratory (UFRGS) and the LabTeC/Dell dual Pentium III cluster with 20 nodes, also in our laboratory. We also used a dual Athlon workstation at the University of Pittsburgh to study some issues for very large tables(thanks to Dr. Soffa and Dr. Childers).

5.3 Baseline configuration

Our baseline architecture for most experiments is a 19-stage superscalar processor with a memory hierarchy of 3 levels, configured likewise a current commercial processor (INTEL, 2001). The configuration settings are described in the next tables, which are divided in four tables.

Table 5.3 presents the general parameters for all configurations, as the pipeline width, number of entries in the fetch queue, number of ROB entries, and branch prediction.

Table 5.4 describes the memory hierarchy for the baseline architecture, with three cache levels, latencies, and other parameters.

Table 5.3: Main parameters for baseline architecture

Parameter	**Value**
Pipeline width	4 instructions
Inst. Fetch Queue (IFQ)	16 instructions
Branch prediction First level Second level	two-level (gshare) 13-bit register (xored with PC) 8192 entries
Branch target buffer (BTB)	4096 entries, 2-associative
Return Address Stack (RAS)	none
RUU	128 entries
Load/Store queue (LSQ)	64 entries

Table 5.4: Memory hierarchy for baseline architecture

Hierarchy level	**Parameter**	**Value**
First Level Instruction	hit latency	1 cycle
	associativity	4
	Sets	128
	line length	64 bytes
	replacement policy	LRU
Total size		32KB
First Level Data	hit latency	1 cycle
	associativity	4
	sets	64
	line length	128 bytes
	replacement policy	LRU
Total size		32KB
Second Level Unified	hit latency	5 cycles
	associativity	8
	sets	256
	line length	256 bytes
	replacement policy	LRU
Total size		512 KB
Third Level Unified	hit latency	10 cycles
	associativity	8
	sets	1024
	line length	256 bytes
	replacement policy	LRU
Total size		2 MB
Main Memory	hit latency (first chunk)	100 cycles
	Hit latency (next chunks)	10 cycles
	Access width	16 bytes

Table 5.5 presents the configuration of functional units, with latency and number of instances. Please note that the latencies of the functional units are not the same as in the original *sim-outorder* simulator, in order to present the characteristics found in deeply-pipelined processors.

Table 5.5: Functional units for baseline architecture

Type	Parameter	Value
Integer ALUs (add/sub)	units	2
	latency	1 cycle
Integer ALUs (div/mul)	units	1
	latency	14 cycles
Memory Access	read ports	1
	write ports	1

Figure 5.1 depicts the division of stages for the superpipelined processor that we simulated, with 19 stages. The pipeline stages for RST are also shown, parallel to the instruction pipeline.

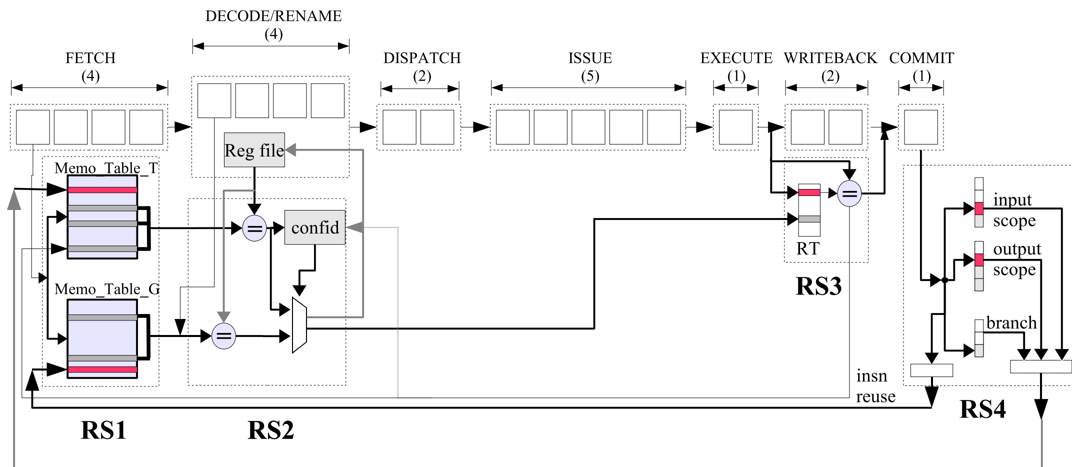

Figure 5.1: Superpipeline for baseline architecture

Table 5.6 presents the values used for Memo_Table_T and Memo_Table_G in both DTM and RST architectures for most simulations. These table sizes were determined based on two main restrictions: *(i)* the chip area should be similar to the cache sizes; and *(ii)* the number of entries should be a power of 2, to allow the simulation of associative tables.

Table 5.6: Memoization tables

Type	**Parameter**	**Value**
Memo_Table_G	entry size	129 bits
	number of entries	2048
	associativity	4
Total Size		32 KB
Memo_Table_T	entry size	388 bits
	number of entries	512
	associativity	4
Total Size		24 KB

For the trace construction policy, we employ two different algorithms in many simulations:

- **DTC policy:** the DTM Trace Construction policy is the same as the one originally employed by DTM, where only instructions already in Memo_Table_G can be inserted in a trace. This is the default for DTM's simulations in the rest of this work, but it will always be pointed out when used for RST.

- **RTC policy:** the RST Trace Construction policy allows that any instructions in the reuse domain, present or not in Memo_Table_G, to be part of a trace in construction. The RTC policy is the default for RST's simulation.

6 LIMITS OF SPECULATIVE TRACE REUSE

"If we knew what it was we were doing,
it would not be called research, would it?"

Albert Einstein

With this set of experiments, we want to verify the limits of performance that a speculative trace reuse architecture could achieve while constrained by other architecture features such as caches, number of functional units, pipeline width, and others. We start with 4K entries in the Memo_Table_T (about 1.22 MB) and 16K entries in the Memo_Table_G (about 256 KB), which represented tables with 8 times more entries than the ones that we use for the baseline results in the next Chapter. Both tables are fully associative unless stated otherwise, therefore there is no limit to the number of instances of a single instruction or trace for each table but for the number of entries in each table.

For these studies, we must restrict table sizes due to the time necessary for simulation. For each configuration, we have to simulate 12 benchmarks, and for this specific table sizes each simulation can take as long as 45 hours. Therefore, simulating larger tables is not practical with our current resources, but we think that these table sizes are more than enough to point out the desired performance trends.

Our first concern is to determine how changes in Memo_Table_T characteristics affect performance. In Section 6.1, we present results for the maximum simulated table sizes as the upper bounds for RST. After that, we restrict Memo_Table_T associativity in Section 6.2. Section 6.3 presents results obtained by constraining Memo_Table_T size. In Section 6.4, we vary the number of inputs and outputs in the trace contexts and analyze how they affect performance of speculative trace reuse. Then, we limit the number of values that can be predicted by trace in Section 6.5.

After testing multiple variations on the Memo_Table_T parameters, we start to vary other architecture configurations. Section 6.6 presents a study of how changes on the number of stages affect performance. Section 6.7 shows performance variations when the pipeline width is varied. In Section 6.8, we vary the number of functional units. In Section 6.9, we modify the memory hierarchy. Results for stride prediction are shown in Section 6.10. Section 6.11 presents results for different reuse domains (including memory reuse in RST). Finally, we summarize our experiments in Section 6.12.

6.1 RST with maximum table sizes

In this first study, we want to determine the upper bound speedups for RST when very large memoization tables are employed. Besides the number of entries, in this first experiment we do not restrict the number of registers in the input and output contexts, and the memoization tables are fully associative (FA). LRU policy is used to select a victim among all traces when the tables are full and a new entry is going to be inserted.

Table 6.1 presents the configuration for both Memo_Table_T and Memo_Table_G for the experiments featured in this Section.

Table 6.1: Configuration for maximum table size

Table	Parameter	Value
Memo_Table_T	Entries	4096
	Associativity	fully
	Size	1.22 MB
Memo_Table_G	Entries	16834
	Associativity	fully
	Size	256 KB

6.1.1 Speedup for DTM (DTC an RTC)

In this experiment, we compare performance of DTM with both trace creation policies, DTC and RTC, to choose one as a comparison standard for RST results. As our focus is RST and not DTM, we will present results for only one of the policies in the DTM case for the rest of this work.

Figure 6.1 shows a performance comparison of DTM (RTC) against DTM (DTC) using the speedups over the baseline architecture (defined in Chapter 5) without reuse. The vertical axis presents speedup, while the horizontal axis depicts the benchmarks we run. For each benchmark, the first bar is the speedup of DTM (RTC) over the baseline architecture, and the second bar is the speedup of DTM (DTC). The last set of bars presents the harmonic mean of all speedups for each case.

For almost all benchmarks, DTC does better than RTC for DTM, only in the case of *compress.95* and *m88ksim.95* there is a small advantage for RTC. In average (HM), results with DTC are slightly better than RTC. Thus, we choose DTM with the DTC policy as a comparison standard of trace reuse to compare with RST in the rest of this work.

6.1.2 Speedup

Figure 6.2 presents the speedups for both DTM and RST over the baseline architecture for the memoization resources presented in Table 6.1. For each benchmark, the first bar depicts the speedup for DTM, the second bar presents the speedup for RST using the same trace construction policy as DTM (DTC), and the last bar presents the speedup for RST for the more speculative RST trace construction policy (RTC). The last set of bars presents the harmonic mean of all speedups for each architecture (HM).

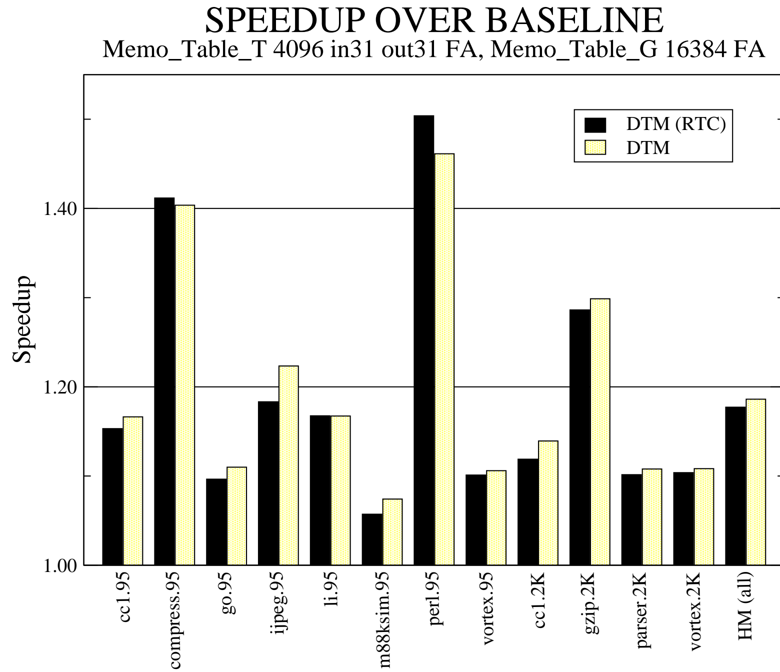

Figure 6.1: Speedups over baseline for DTM (RTC and DTC), Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

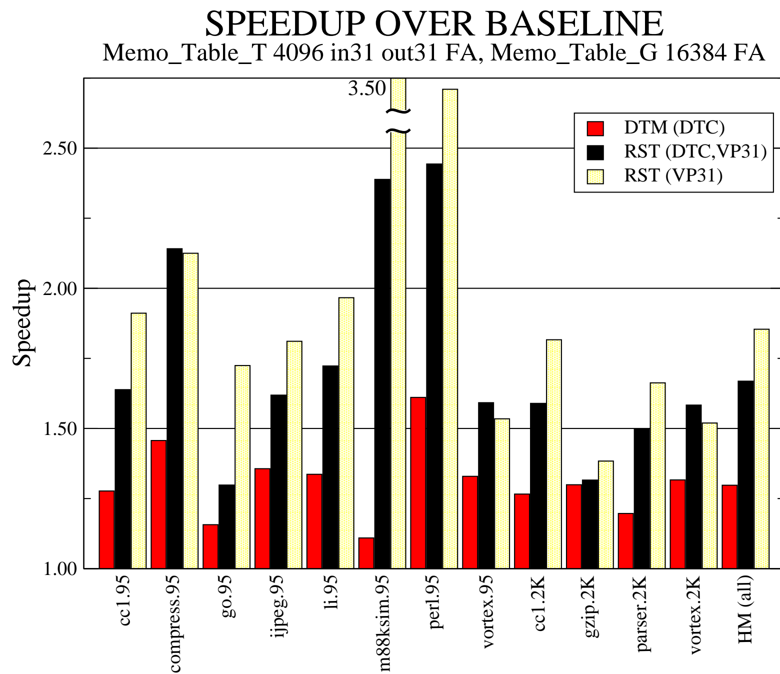

Figure 6.2: Speedup over baseline, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

RST using the DTC policy shows a speedup of 1.66 over the baseline architecture, while the RTC policy presents a speedup of 1.85 (harmonic means). The more speculative nature of RTC can provide more trace instances for the same PC addresses, and thus more choices for the reuse test. For 9 of the 12 benchmarks, this policy achieves improved performance over the DTC policy, with a speedup of 1.11 for RTC over DTC.

Figure 6.3 shows the speedup obtained for RST over DTM for the same configuration as for the previous graph. All benchmarks show performance gains when compared to DTM. The smallest speedups are observed for the benchmark *gzip.2K*, with speedups of 1.01 (RST with DTC) and 1.06 (RST with RTC). In average, RST is able to achieve speedups of 1.28 (DTC) and 1.42 (RTC) over DTM for this configuration.

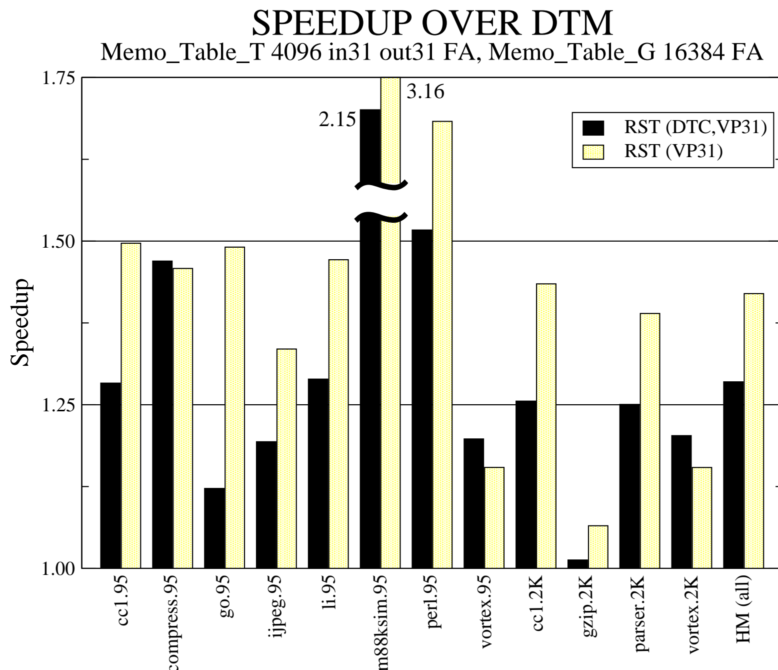

Figure 6.3: Speedup over DTM, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

From these results, we can infer that RST has a large potential for speedups over non-speculative reuse for most benchmarks, as only *gzip.2K* presents a speedup smaller than 1.15 over DTM. For this almost unconstrained configuration for the RST pipeline, but with a realistic instruction pipeline and memory hierarchy, we show the upper bound limits for performance of RST in deep pipeline processors.

In the next subsections, we analyze how RST improves performance by observing characteristics from the speculatively reused traces and comparing them to the traces reused in non-speculative trace reuse techniques (DTM).

6.1.3 Contribution to committed instructions

For this measurement, we want to determine how many instructions speculative trace reuse can reuse or skip by reusing traces when compared to regular trace reuse,

and then we correlate these results with the performance measured by the speedups of the previous subsection.

Figure 6.4 shows how instructions that are isolatedly reused, bypassed by traces or by speculatively reused traces contribute to the total number of committed instructions for the same configuration of Table 6.1. For each benchmark, the first column presents results for DTM, the second one presents the results for RST with the DTC policy, and the last column presents the results for RST with the RTC policy. Each column is divided in percentage of reused instructions, instructions reused inside traces, instructions speculatively reused inside traces (not shown in the DTM case), and finally the percentage of committed instructions that are not reused. The last set of columns presents the arithmetic means (AM) for each category.

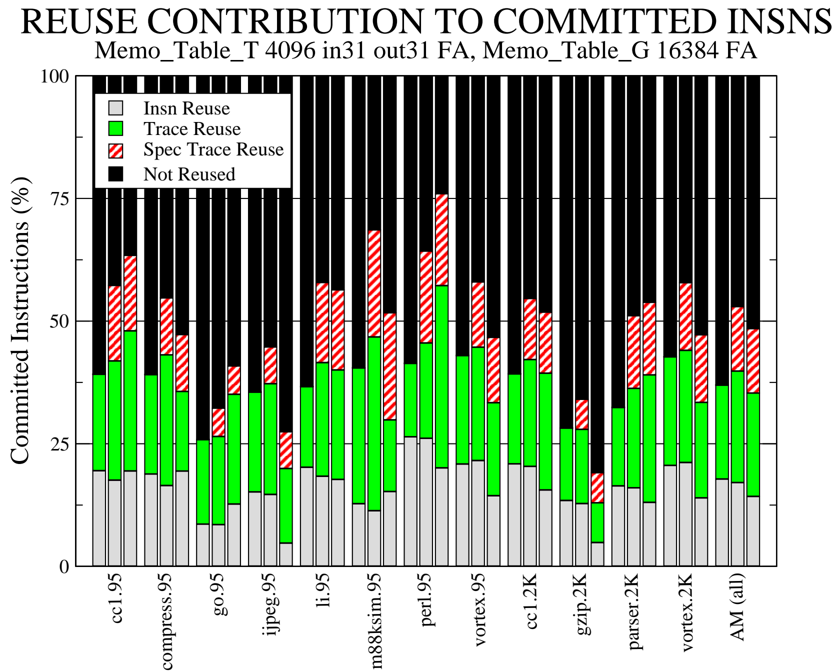

Figure 6.4: Contribution to committed instructions, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

In average (arithmetic mean), RST (DTC) can reuse 53% of all instructions by means of either trace or instruction reuse, while RST (RTC) reuses 48% and DTM reuses 37% of all instructions. This measurement is not directly proportional to performance, as the number of reused instructions has different impacts on different benchmarks. Comparing this graph with the speedup on Figure 6.2, we can clearly see that, in some cases, other characteristics that not the number of bypassed instructions affect performance. For example, the benchmark *m88ksim.95* has many more committed instructions coming from reuse using the DTC policy than for RTC (69% against 52% of all committed instructions). However, the speedup of RTC over DTC in this case is 1.47.

In almost all cases, RST reuses or bypasses more instructions than DTM with both DTC and RTC policies. Only for *ijpeg.95* and *gzip.2K* RST (RTC) presents a smaller number of reused instructions than DTM, but even reusing a smaller amount

of the total number of instructions RST still achieves better performance, as it can be seen in Figure 6.3.

From this, we can infer that not only the number of reused traces and instructions determines the achievable speedups, but also the quality of reused traces can interfere in the results. For example, if traces that are outside the global critical path are reused, they may not have the same impact in performance than a smaller number of reused traces that produce the inputs for other instructions and traces. Therefore, the number of reused and skipped instructions is an interesting metric but it cannot be used alone to justify one or another modification in the technique.

Another interesting trend is the reduction of reused instructions from Memo_Table_G for RST with the RTC policy. Many benchmarks present this trend, such as *ijpeg.95*, *li.95*, *perl.95*, *cc1.2K*, *gzip.2K*, *parser.2K*, and *vortex.2K*, where *perl.95* is the only benchmark that could recover the number of isolatedly reused instructions by reusing more instructions inside traces. In average, 17% of all committed instructions are reused by instruction reuse in RST (DTC), while RST (RTC) can only reuse 14% by instruction reuse.

6.1.4 Trace length

For the current measurements, our objective is to verify how trace lengths correlate with performance for both RST and DTM. For DTM, we considered the average trace length for all reused traces, not only those whose instructions are committed later. For RST, both reused and speculatively reused traces are considered.

Figure 6.5 shows the average number of reused and speculatively reused trace sizes for DTM and RST (arithmetic mean). The graph shows a trend towards longer traces for speculatively reused traces. However, the greatest difference appears between the two trace construction policies. While the average trace length for traces that are not speculatively reused is almost the same for RST (DTC) and DTM (2.32 instructions), the average number of instructions increases to 3.57 for RST (RTC). For the benchmark *m88ksim.95*, trace length grows from 2.4 to 7.5 instructions per trace when the RTC policy is used. Speculatively reused traces are longer than their non-speculative counter parts too.

From the current results, we can conclude that the RTC policy is able to provide longer traces that can be reused later, and that speculatively reused traces are likely to be longer.

6.1.5 Branches per trace

An important capability of trace reuse is to encapsulate control dependencies and resolve multiple branches per cycle. Figure 6.6 shows the average number of branches per trace for DTM and RST for both speculatively and regularly reused traces (AM).

Again, the trace construction policy determines characteristics of traces. There is not much difference between DTM and RST (DTC) in terms of average number of branches in reused traces, but traces for the RTC policy have about 58% more branches than the DTC policy. The benchmark *m88ksim.95* presents a very strong increase in the number of branches from DTC to RTC, with non-speculatively reused traces growing from an average of 0.8 to 2.6 branches per trace. This and the longer traces seen for *m88ksim.95* in the previous subsection result in better performance for the benchmark as the previously shown speedups confirm.

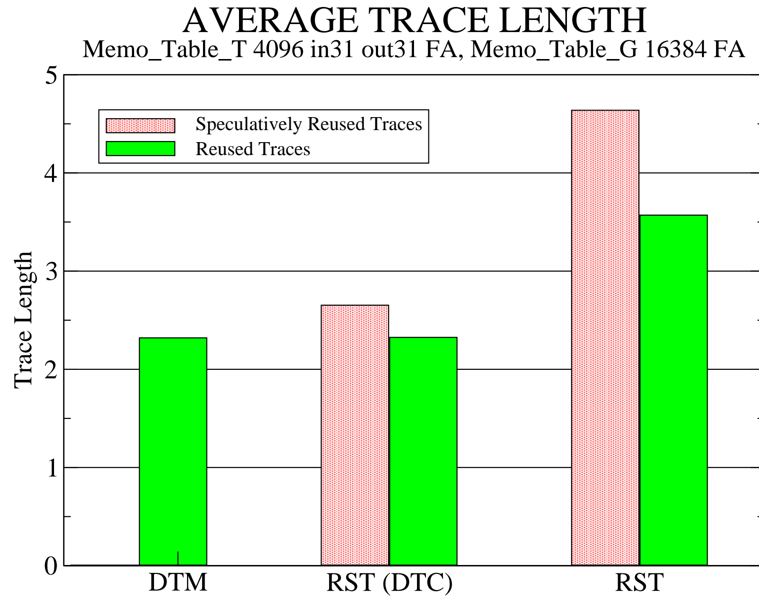

Figure 6.5: Average trace length for RST and DTM, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

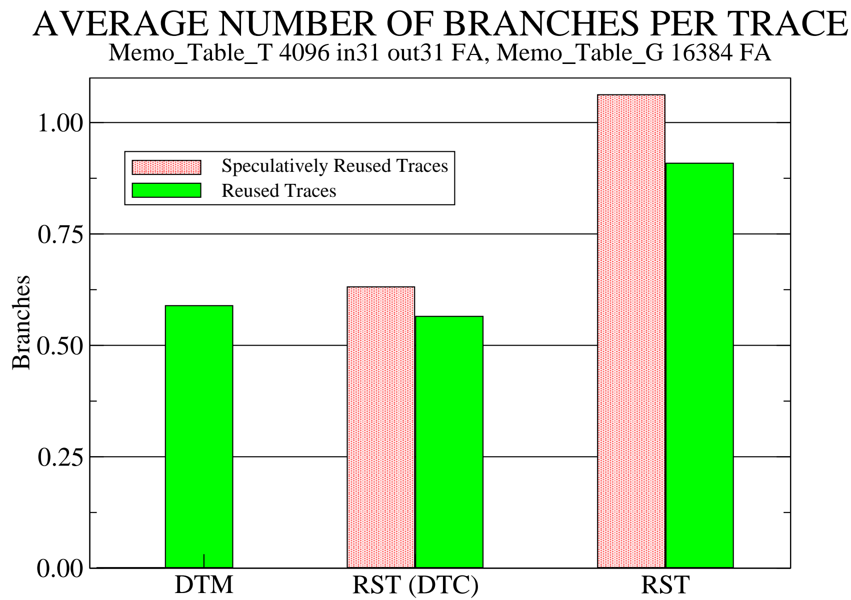

Figure 6.6: Average number of branches per trace, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

6.1.6 Input and output scope sizes

Another interesting measurement is the determination of the number of inputs and outputs in reused traces. Figure 6.7 depicts the averages (AM) for DTM and RST of the number of inputs of reused traces, speculatively reused or not.

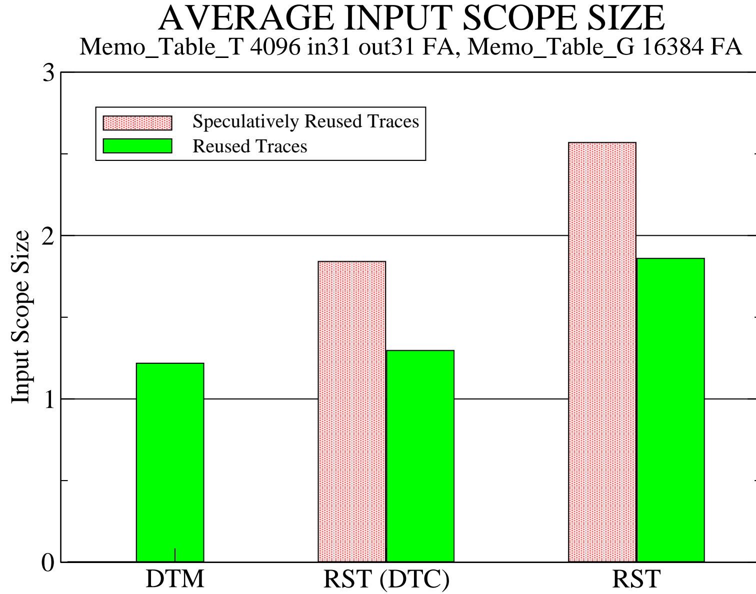

Figure 6.7: Average number of trace inputs, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

This is another measurement where trace construction policy plays an important role, but in this case all benchmarks experienced an uniform increase in the number of registers in the input scopes. The increase in trace length and number of inputs for reused traces occurs due to the more speculative nature of RTC; on the other hand, this more speculative nature produces traces with less redundancy than DTC, thus speculative trace reuse is even more important for RTC.

Figure 6.8 shows the average number of registers in the output context of reused traces. As for the previous graph where the input scope sizes was discussed, the trace construction policy also makes the difference here. Again, traces for the RTC policy have more outputs than for the DTC policy. If we consider that RTC produces traces with both more inputs and outputs in average, there is an increase in the likelihood of these traces being in critical paths.

6.1.7 Critical paths

As we stated before, traces can encapsulate true data dependencies and reuse them in a single cycle. In this measurement, we calculate the average critical path collapsed by reused traces, that is the maximum chain of instructions with true data dependencies in a trace.

Figure 6.9 shows the average critical path for DTM and RST. Speculatively reused traces present longer critical paths, about 61% longer for DTC and 72% longer

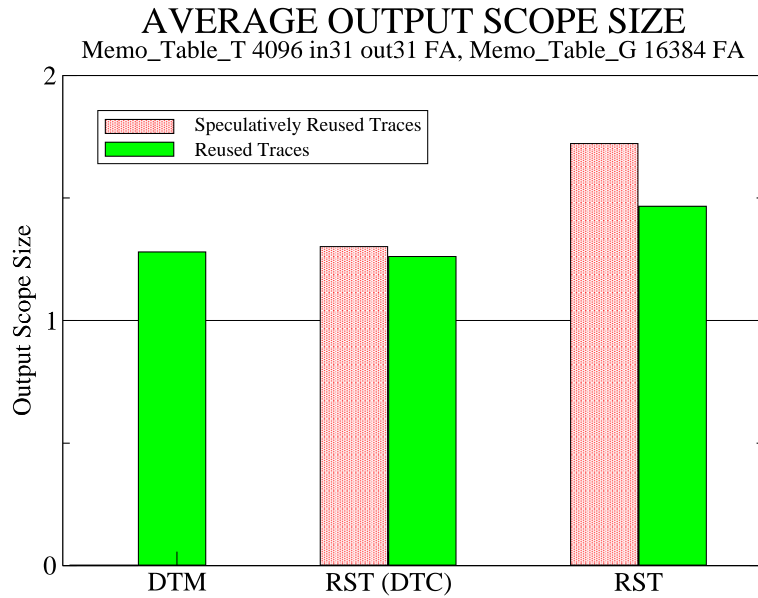

Figure 6.8: Average number of trace outputs, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

for RTC than the non-speculatively reused traces, and RTC traces have critical paths about 73% longer than DTC traces.

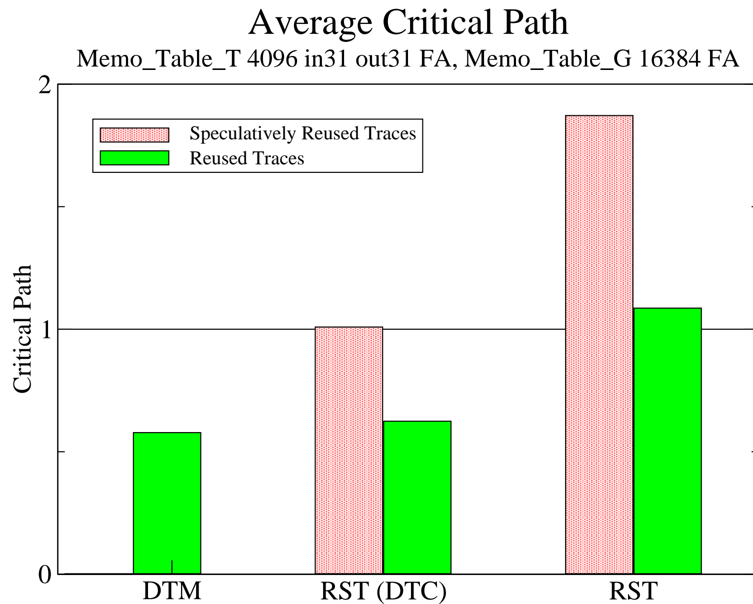

Figure 6.9: Average critical paths for reused traces, Memo_Table_T 4096 entries (31 inputs/31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

Therefore, we conclude that the RST policy for trace creation (RTC) is capable of building traces that encapsulate longer critical paths than the DTM policy (DTC).

6.1.8 Reason for finishing trace formation

Figure 6.10 shows the average (AM) reason for finishing trace formation. The average for each reason was obtained by adding up the rates for each reason for all benchmarks, and then dividing it by the number of benchmarks. The number of traces per benchmark was not used as weight. These statistics include all traces in creation, whether they are included in Memo_Table_T or just discarded because there was already an equal trace in Memo_Table_T or there was only one instruction on it (traces must have at least two instructions to be considered by RST and DTM).

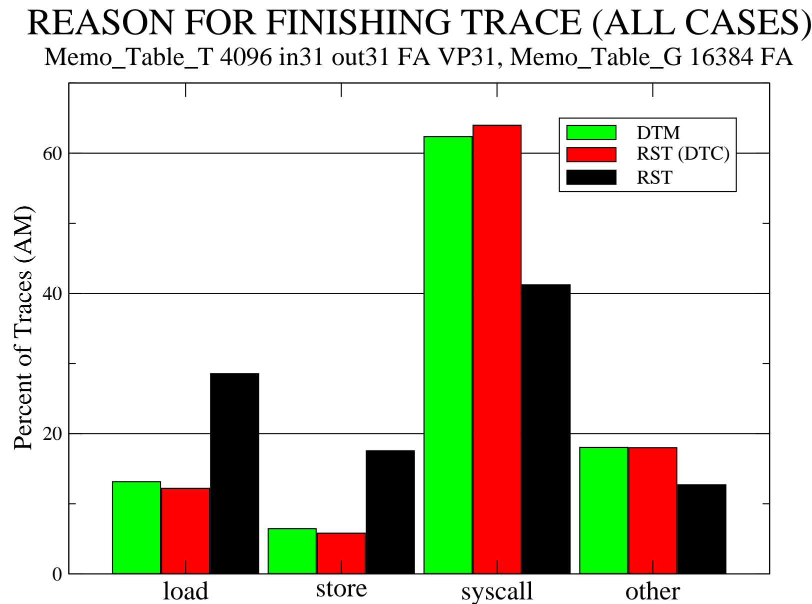

Figure 6.10: Average reason for finishing traces, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

The main difference that can be observed is not between RST and DTM, but between trace construction policies again. The number of traces that are finished by memory accesses increases and the number of traces finished by system calls decreases for RST when the RTC policy is employed.

The “other reasons"option also includes instructions that are not redundant (which does not affect RTC) and instructions that does not belong to the reuse domain and are not system calls.

Figure 6.11 shows the average (AM) reason for finishing traces that were stored in Memo_Table_T. For stored traces, there is a modification in the distribution of reasons to terminate traces. Memory accesses become the main issue for trace termination on RST (RTC), with system calls having a very small importance there. Even for RST (DTC) and DTM, memory accesses answer for 2/3 of all finished traces

that are stored. This can be explained by non-redundant instructions finishing traces in the DTC case, and by RTC producing more traces as it is not compromised by including only instructions reused from Memo_Table_G.

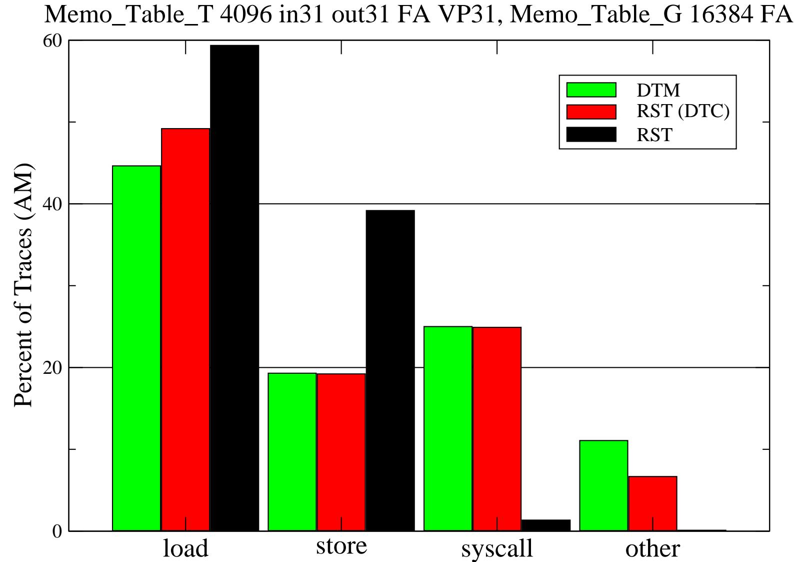

Figure 6.11: Average reason for finishing stored traces, Memo_Table_T 4096 entries (31 inputs, 31 outputs) fully associative, Memo_Table_G 16384 entries fully associative

From these reasons to finish traces, we conclude that memory accesses and system calls are the two main issues to be approached in order to build longer traces. The former factor will be treated in the last Section of this Chapter, where we vary reuse domains.

6.1.9 Remarks for this Section

From the current results, we conclude that varying the trace creation policy creates traces with diverse characteristics, and this variation can change performance in ways that not only by increasing or reducing the percentage of committed instructions that belong to reused traces.

Although RTC presents speedups much better for RST over the baseline architecture (1.85 against 1.66 for the DTC policy), the same does not happen with DTM. Therefore, we infer that the RTC policy depends on being able to anticipate trace inputs in order to be effective. The increase in the number of entries in the input contexts also points out this direction; increasing the number of inputs also increases the likelihood that some of these inputs will not be available for the reuse test, hence prediction plays an important role here.

We also show that not only the number of reused instructions but also traces characteristics like trace lengths, encapsulated dependencies, and number of branches are very important for the final performance.

6.2 Memoization table associativity

For this experiment, we want to verify the impact of different Memo_Table_T associativities in performance. Until now, all experiments used a fully associative Memo_Table_T, thus the entire table could be used to hold different instances of traces starting by the same instruction. This configuration has two problems: *(i)* a set of instructions with low redundancy but heavily executed could greedily take many Memo_Table_T entries, avoiding that more useful traces could be stored and reused; and *(ii)* the maximum number of traces to be tested is directly proportional to the associativity of Memo_Table_T; thus, a fully associative table would hardly be feasible in a real processor. Then, in the next experiments we vary Memo_Table_T associativity from fully associative (FA) to 16, 8, 4, 2, and 1 (direct-mapped), and then we analyze the variation of performance among the different configurations.

6.2.1 Speedup

Figure 6.12 presents the speedups for RST over the baseline architecture when Memo_Table_T associativity is varied. For most benchmarks, performance decreases when associativity is decreased, as expected. A smaller number of instances of traces determines a smaller number of reuse possibilities, thus decreasing performance. The benchmark *m88ksim.95* experienced a very strong performance decrease from a 4-way to a 2-way Memo_Table_T, decreasing speedups from 3.47 to 2.27. The benchmark *perl.95* also has a strong reduction on speedups, as can be seen in the graph. All benchmarks present reductions on speedups, and the average speedup (harmonic mean) depicted by the line with chess-squares decreases from 1.85 (for fully-associative Memo_Table_T) to 1.32 (for direct-mapped Memo_Table_T).

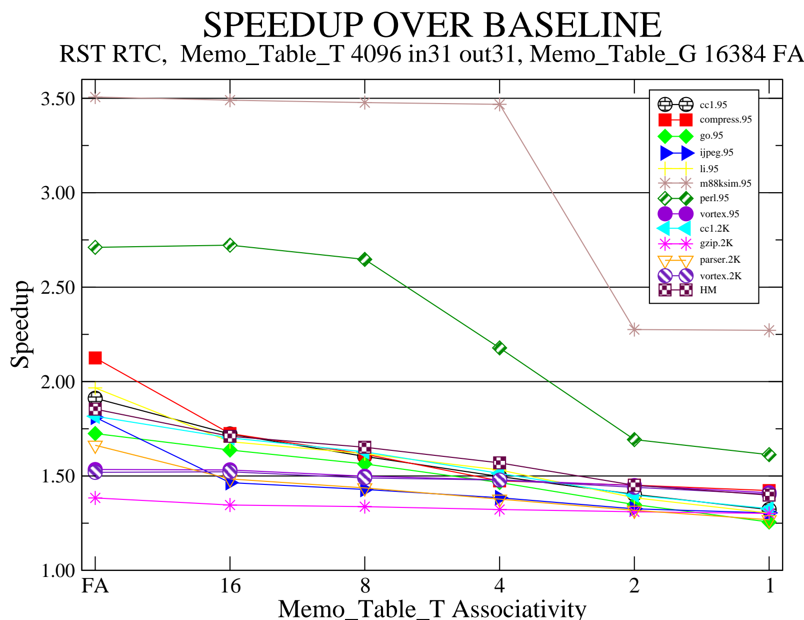

Figure 6.12: Speedup for RST over baseline, varying Memo_Table_T associativity

Figure 6.13 depicts speedups over baseline for RST with the DTC policy for trace construction. For this trace construction policy, *m88ksim.95* presents almost constant performance for the different associativities tested. But as for RTC, all benchmarks are affected by the reduction on associativity of Memo_Table_T. In average (harmonic mean), speedup decreased from 1.67 to 1.43 for RST (DTC).

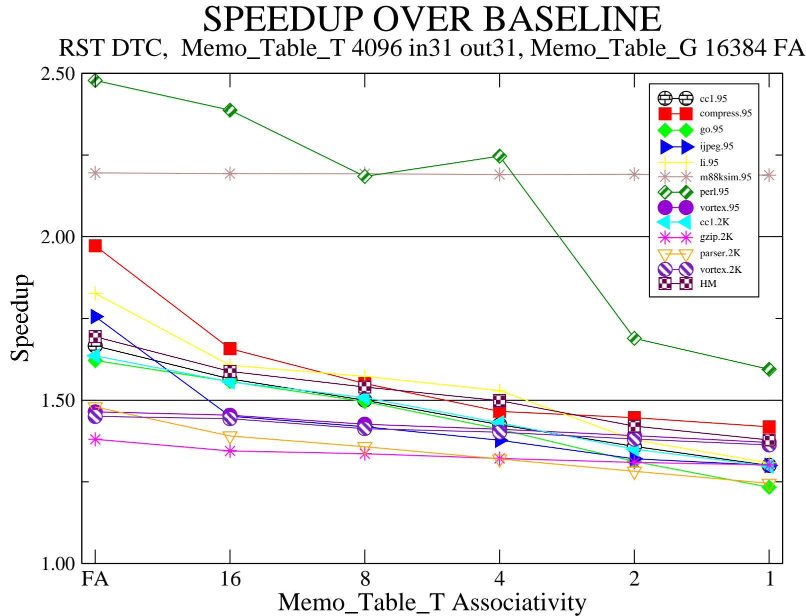

Figure 6.13: Speedup for RST (DTC) over baseline, varying Memo_Table_T associativity

Figure 6.14 presents the speedups for DTM over the baseline architecture. DTM results show a different tendency when the associativity is reduced. Performance does not suffer much from fully associative tables to direct-mapped tables. This points out that many trace instances for the same PC address do not help increasing performance when speculative reuse is not allowed.

From these results, we conclude that the associativity of Memo_Table_T is important for RST's performance, but the same is not true for DTM in the simulated configurations. We also show that some issues like entry aliasing can also affect performance by evicting key traces from Memo_Table_T, sometimes making a larger associativity to produce a smaller performance for some benchmarks.

6.3 Memoization table size

After experimenting on the Memo_Table_T associativity, in this experiment we wish to verify the impact of different Memo_Table_T sizes on performance. We start with the original number of entries in Memo_Table_T (4096), and then we reduce it to 2048, 1024, and finally 512 entries, which is the size that will be used later in the next Chapter for the constrained architecture configuration.

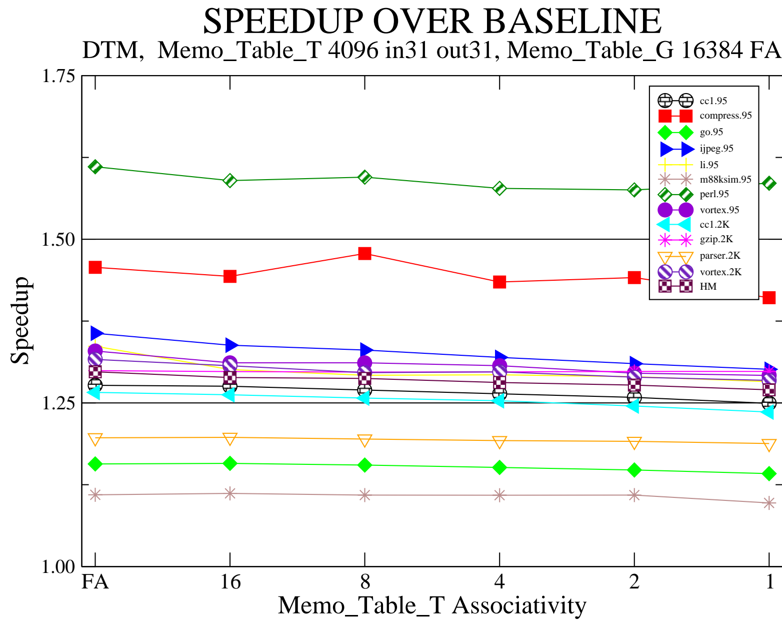

Figure 6.14: Speedup for DTM over baseline, varying Memo_Table_T associativity

6.3.1 Speedup

Figure 6.15 shows the variation on average (HM) speedup over the baseline architecture for RST and DTM when the number of entries in Memo_Table_T is varied.

All architectures have performance reductions when the number of entries are reduced as expected. The fall on performance is stronger for RST (DTC) from 4096 to 2048 entries. For the other cases, the speedups are reduced in an almost linear way.

Therefore, we conclude that Memo_Table_T size is very important to define the speedups that can be achieved by trace reuse in general, and speculative trace reuse in particular.

6.4 Number of inputs and outputs

In the following measurements, we want to study how the maximum number of inputs and outputs in trace contexts may affect performance. Reducing the number of inputs in a trace may cause shorter traces to be created; on the other hand, shorter traces with less inputs are more likely to be reused more times than longer traces, as they depend on less inputs to be redundant. In terms of hardware area and complexity, reducing the number of inputs that must be tested for trace reuse also reduces Memo_Table_T size and the number of comparisons that are done at the RS2 stage for the reuse test, resulting in potentially less registers being read at each cycle. Therefore, the smaller the number of inputs, more traces can be stored and the smaller the hardware complexity for the reuse test.

As presented in Chapter 4, each entry in the input or the output context requires

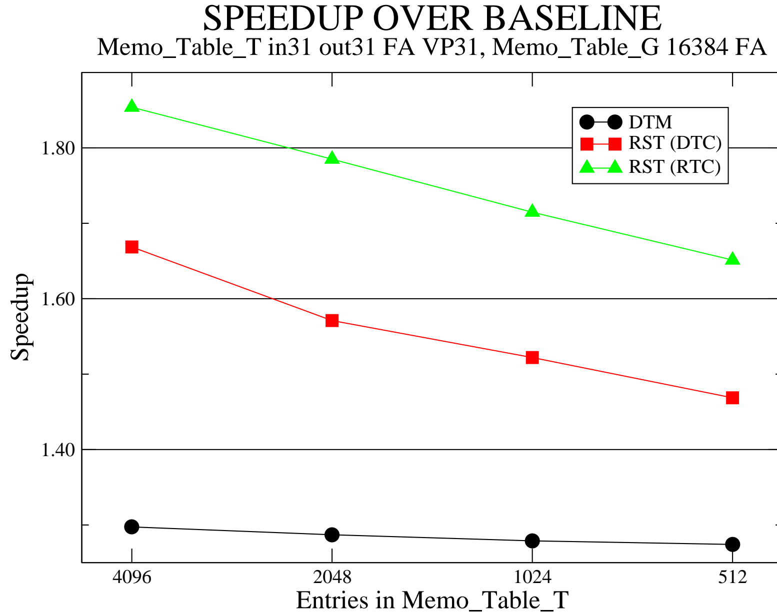

Figure 6.15: Speedup over baseline architecture varying Memo_Table_T size, Memo_Table_T in31 out31 FA, Memo_Table_G 16384 entries FA

32 bits for the value (*icv* or *ocv*) plus 5 bits for the register name (*icr* or *ocr*). For the input context, it also changes the number of comparison circuits in the RS2 stage.

The simulations are done varying the number of inputs or outputs from 31 to 8, 6, 5, 4, 3, 2 and 1 inputs. While a parameter was varied, the other one was fixed as the maximum possible (31) to avoid influence on results. For the confidence mechanism, RST was simulated with oracle confidence, and the maximum number of predicted inputs was always equal to the input scope size.

From this point, we reduce table size and associativity in order to reduce simulation time. Table 6.2 presents the configuration for both Memo_Table_T and Memo_Table_G for the next experiments.

Table 6.2: Configuration for Memo_Table_T and Memo_Table_G, experiments with number of inputs, outputs, predicted inputs

Table	**Parameter**	**Value**
Memo_Table_T	Entries	512
	Associativity	4
	Size	155 KB
Memo_Table_G	Entries	2048
	Associativity	4
	Size	32 KB

6.4.1 Speedup

Figure 6.16 presents the average speedups (harmonic mean) over the baseline architecture for DTM and RST with perfect confidence estimation. The maximum number of outputs is fixed at 31 outputs, while the number of inputs is varied from 31 to only 1 input, as well as the number of inputs that RST may predict.

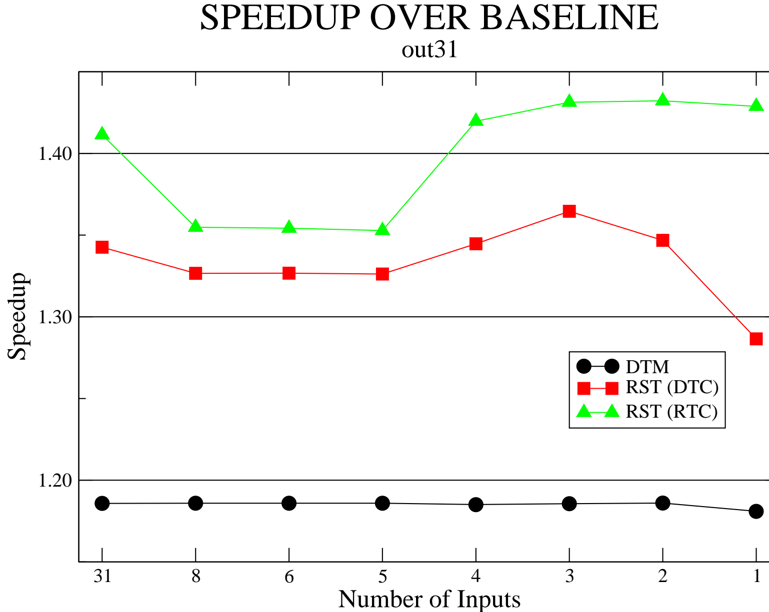

Figure 6.16: Speedup over the baseline architecture, Memo_Table_T 512 entries (31 outputs) 4-way, Memo_Table_G 2048 entries 4-way, varying number of inputs

DTM does not present much difference on performance when the number of inputs is reduced, while RST presents its better average performance (harmonic mean) between 3 and 1 inputs. For the DTC policy, performance decreased significantly from 3 to 1 inputs, while for the more speculative RTC policy allows RST to achieve similar inputs with 1, 2, or 3 inputs.

Figure 6.17 also shows the speedups for DTM and RST, but when the number of inputs is kept fixed at 31 inputs and the number of outputs is varied. Again, DTM presents an almost constant behavior regardless of the number of outputs, with a small decrease performance from 4 to 1 outputs. RST (DTC) achieves its peak performance at 3 outputs, while RST (RTC) can do even better with 2 outputs.

These speedups point that traces with less inputs and outputs can be very important for performance. A trace may have a small number of inputs or outputs, hence being “narrow", but even then it can encapsulate as many instructions as a “wider"trace with more inputs and outputs. Even more than the horizontal dimension of these traces (number of live inputs and outputs), participation on critical paths and high reusabilitity may be the most important factors for performance improvements in RST.

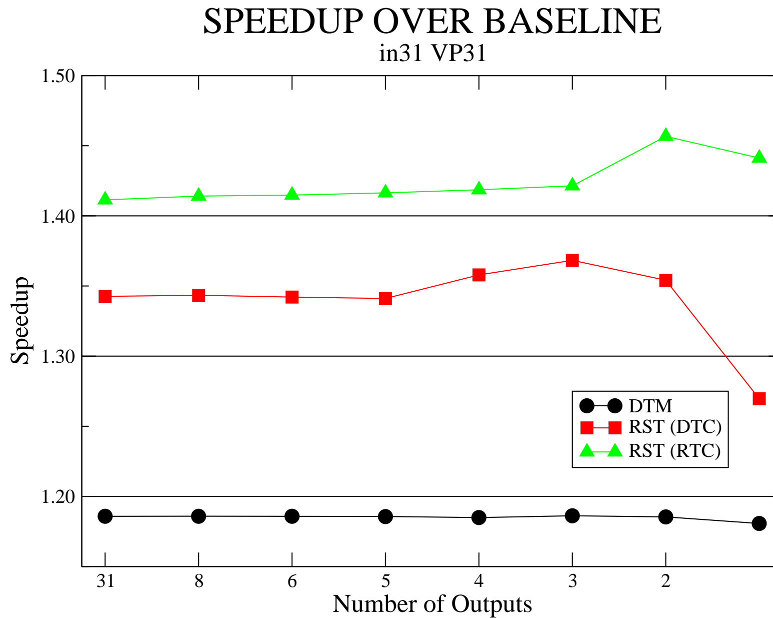

Figure 6.17: Speedup over the baseline architecture, Memo_Table_T 512 entries (31 inputs) 4-way, Memo_Table_G 2048 entries 4-way, varying number of outputs

6.5 Number of predicted inputs

For this experiment, we want to verify the impact of the number of predicted inputs in RST's performance. We start predicting at most 31 inputs (the maximum number of available registers, as *r0* is the constant zero), after what we reduce it to 8, 4, 2, and 1 predicted inputs. The remaining parameters are kept the same throughout the experiment.

6.5.1 Speedup

Figure 6.18 depicts the speedup of RST (RTC) over the baseline architecture for the simulated benchmarks. The vertical axis shows the speedup, while the horizontal axis presents the variation of the maximum number of predicted inputs.

As can be seen in the graph, the best results for RST (RTC) are obtained when at most 3 inputs may be predicted (VP3). The benchmark *perl.95* presents a small increase in performance above the other points for VP3, and the benchmark *m88ksim.95* has a drop in performance from VP2 to VP1, but the average speedup (HM) is very similar in all points until VP2.

Figure 6.19 depicts the speedup of RST (RTC) over the baseline architecture for the simulated benchmarks. The vertical axis shows the speedup, while the horizontal axis presents the variation of the maximum number of predicted inputs.

Again, the performance peak for *perl.95* is situated at VP3 (prediction of at most 3 values), but now *m88ksim.95* shows a peak on VP2, thus moving the best average speedup to VP2. Performance on VP1 is also less than in the other points, as expected. Another benchmark that benefits for more than one value being predicted is *parser.2K*, whose performance drops from VP2 to VP1, although it is almost

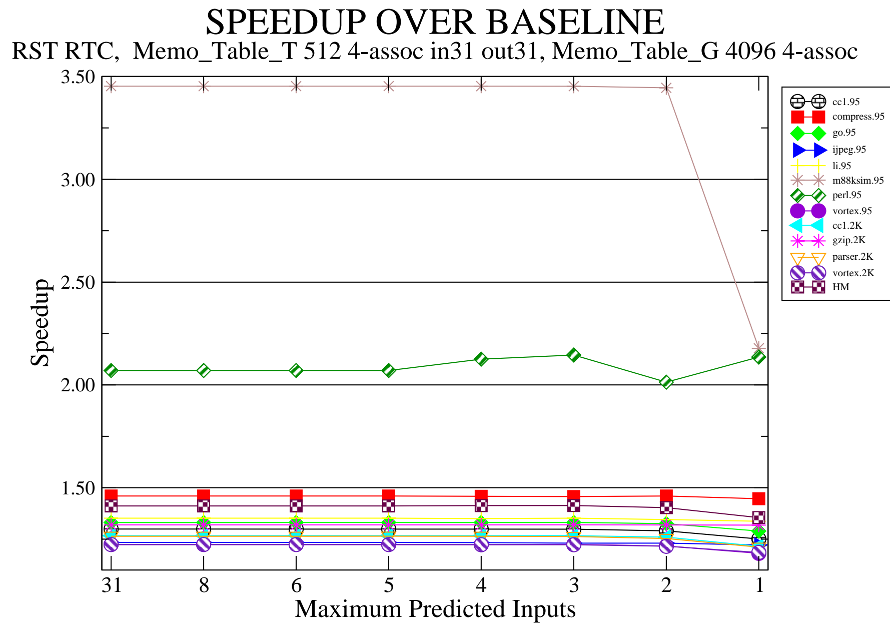

Figure 6.18: Speedup of RST (RTC) over the baseline architecture, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way, varying the number of predictable inputs

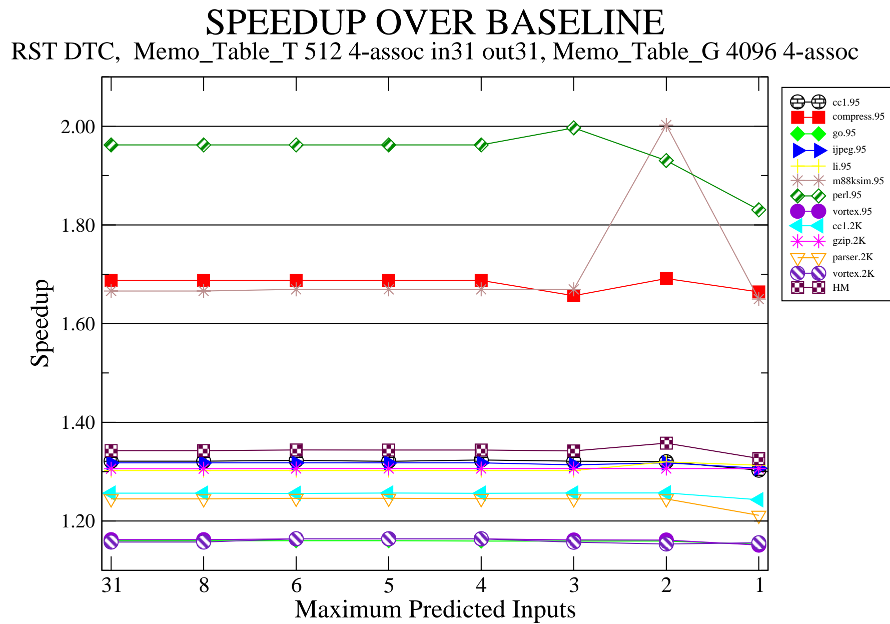

Figure 6.19: Speedup of RST (DTC) over the baseline architecture, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way, varying the number of predictable inputs

constant for the other points.

Figure 6.20 presents the speedup of RST over DTM for the best RST case (RTC and two predicted inputs at most) for reference.

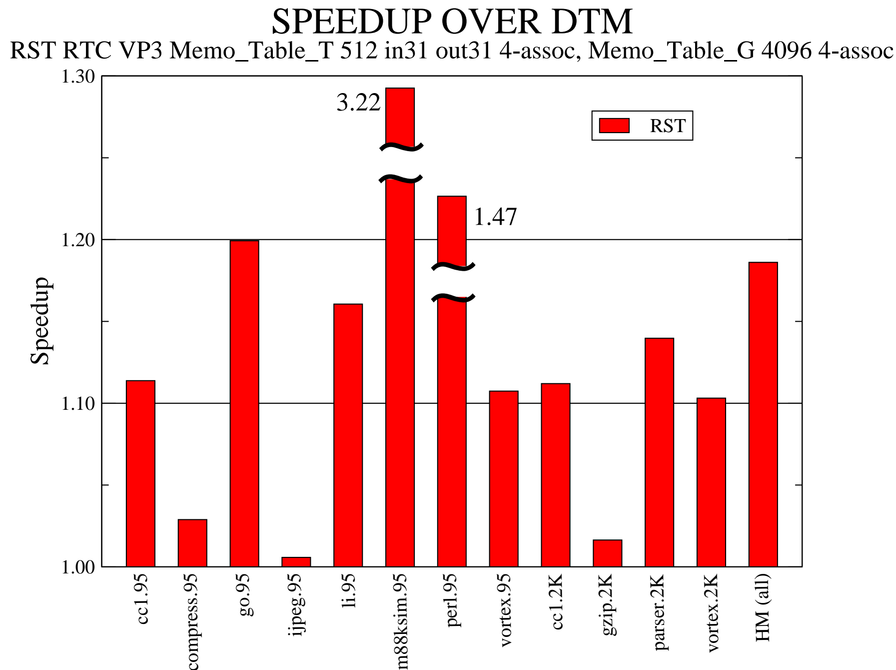

Figure 6.20: Speedup of RST (RTC, VP3) over DTM, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

This setup is able to achieve a slightly better performance than predicting 31 inputs, with an average speedup (HM) of 1.1861 against 1.1838 (both over DTM). Although the difference is minimal, it shows that there is no need for predicting many inputs to achieve good performance in RST, thus simplifying the hardware required for misprediction tests.

6.6 Varying pipeline depth

Until now, we have simulated a pipeline with 19 stages. In this set of experiments, we vary pipeline depth in different stages as to see how it affects RST's performance. In each experiment, we change the number of stages of fetch, dispatch, issue or writeback, and then we compare performance to the baseline architecture. In the last experiment, we reduce the superpipeline to a regular pipeline with 6 stages. The graphs present the average speedup (harmonic mean) for each architecture.

6.6.1 Speedup varying fetch depth

Figure 6.21 depicts the performance effects of reducing the number of fetch stages from 4 to 3, and then to 1 stage.

For all the architectures, decreasing the number of stages for fetching instructions and for branch prediction increased the average IPC (Instructions Per Cycle) in a similar way.

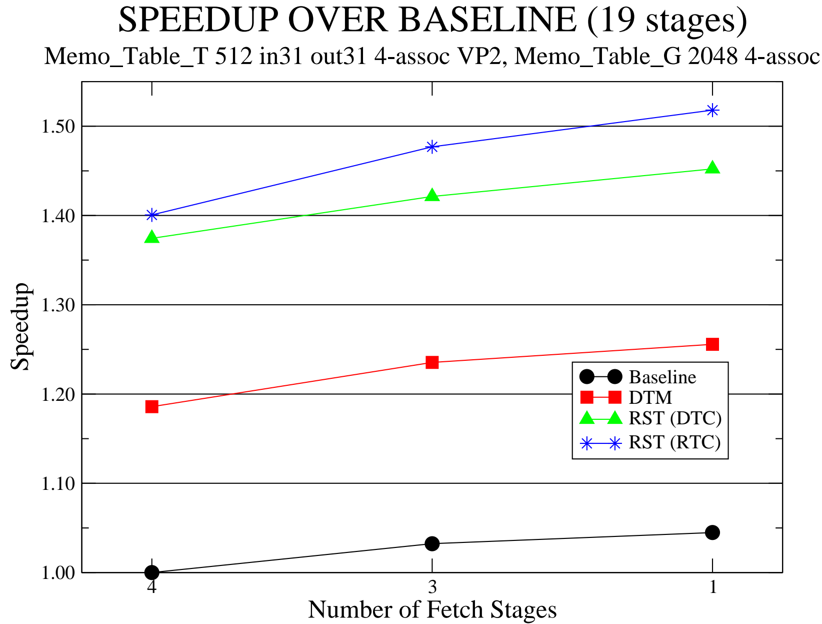

Figure 6.21: Speedup over baseline architecture varying fetch depth, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

6.6.2 Speedup varying dispatch depth

Figure 6.22 depicts the speedups over the baseline architecture with 19 stages when the number of dispatch stages is changed from 3 to 1 stage.

Again, all the average speedups increase when the number of dispatch stages are reduced from 3 to only 1. However, the effect of reducing 2 stages in dispatch is smaller than reducing only 1 stage in the fetch. For RST, speedup increases from 1.40 to 1.47 when 3 fetch stages are considered, but only to 1.41 when 1 dispatch stage is used.

6.6.3 Speedup varying issue depth

In Figure 6.23, we present the speedup over the baseline architecture when the number of stages involved in instruction issue is reduced from 5 to 3, and then to 1 stage.

The difference between RST (DTC) and DTM decreases when the number of stages decrease from 3 to only 1. This result points to a tendency of inputs getting ready early enough for the reuse test, thus reducing the number of cases where prediction is necessary. But even in this case, RST provides speedups over DTM for both DTC and RTC policies.

6.6.4 Speedup varying writeback depth

Figure 6.24 shows the effect on performance of reducing the number of writeback stages from 2 to only 1 stage.

All speedups increase from 2 to 1 writeback stage, but RST (DTC) increases are

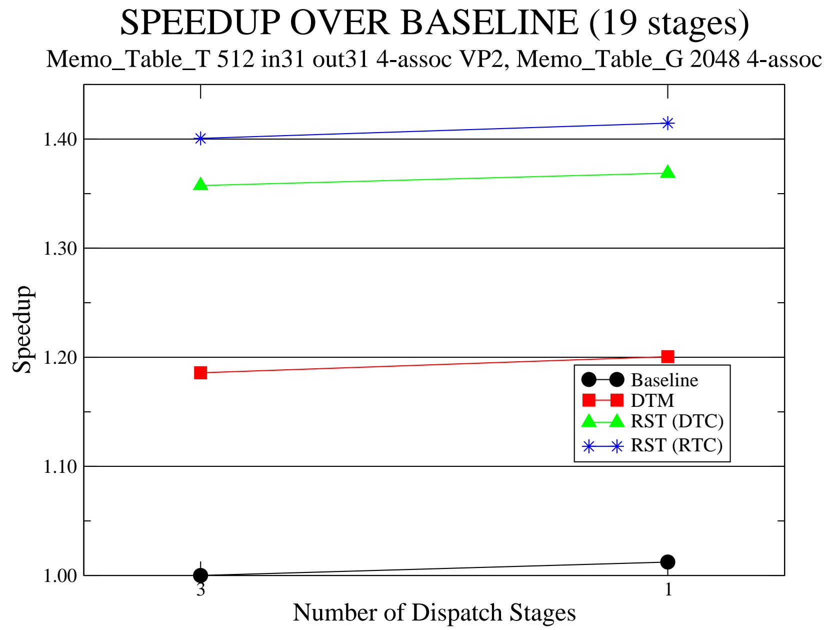

Figure 6.22: Speedup over baseline architecture varying dispatch depth, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

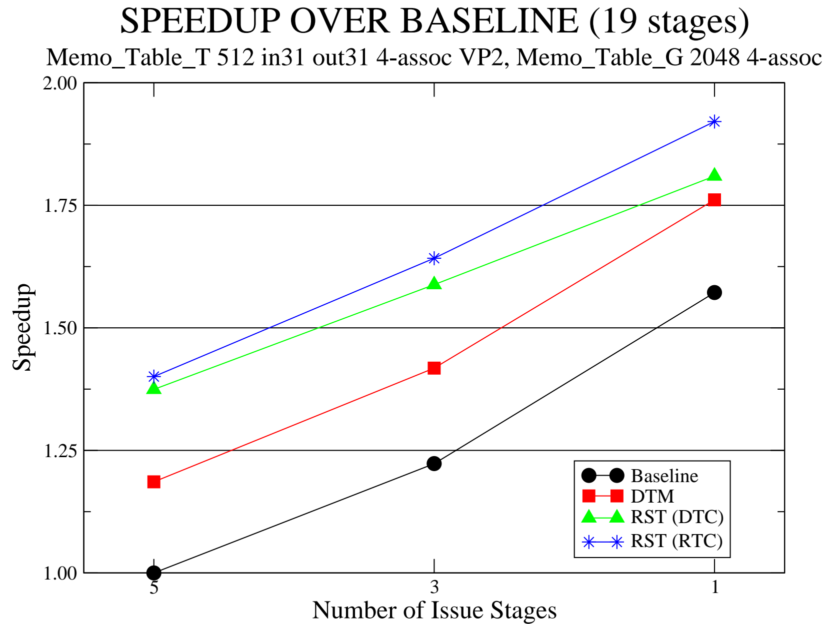

Figure 6.23: Speedup over baseline architecture varying issue depth, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

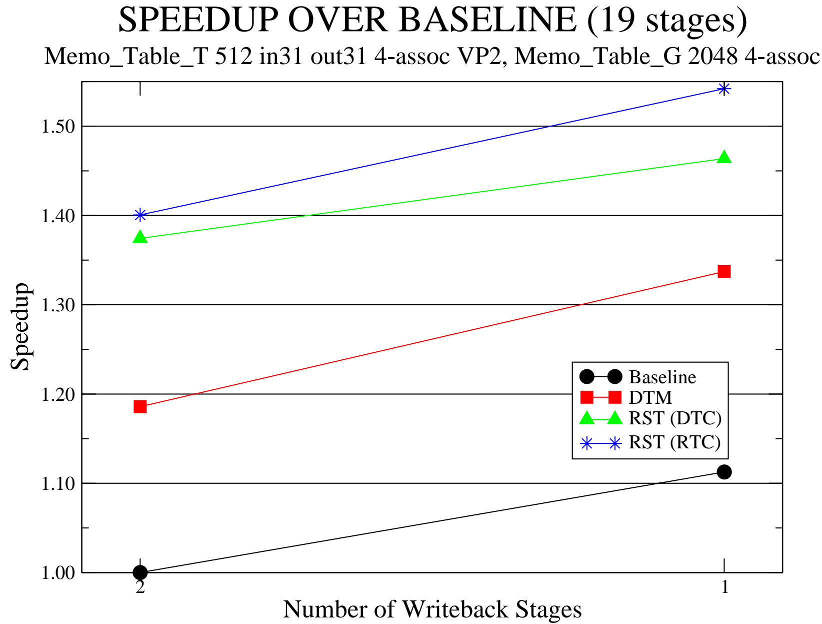

Figure 6.24: Speedup over baseline architecture varying writeback depth, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

lagging a little when compared to the other architectures.

6.6.5 Speedup without superpipeline

For the following simulations, we change the pipeline to a configuration similar to the original *sim-outorder*'s pipeline (BURGER; AUSTIN, 1997), with only 6 stages, each one representing a major pipeline function. In this pipeline, dispatch and decode are combined to better resemble the original *sim-outorder*.

Figure 6.25 presents the speedups over the baseline architecture with the same pipeline configuration for RST predicting at most two inputs (VP2) and DTM.

As for the 19-pipelined architecture, the results for the original *sim-outorder*'s pipeline also present significant speedups over the baseline architecture without reuse and over trace reuse (DTM). RST (RTC) presents the best results for most benchmarks, and clearly it is the best choice for both *m88ksim.95* and *perl.95*.

6.6.6 Remarks for this Section

In this Section, we show that RST can provide potential speedups over both regular trace reuse and architectures without reuse when different pipeline depths are considered. For some cases, the trace policy DTC limits the gains, but RTC performs well in all cases.

We also learn that reducing issue or the writeback depth are the most effective ways to improve performance for RST in all the tested pipeline depth decreases. Reducing from 5 to 3 issue stages allows RST (RTC) to increase speedups over the baseline from 1.40 to 1.64, and reducing from 2 to 1 writeback stage increases

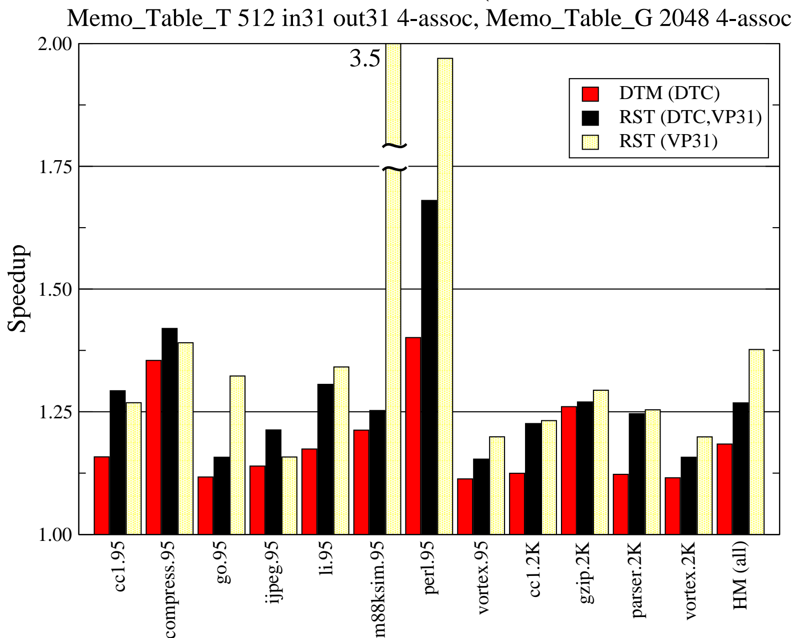

Figure 6.25: Speedup over baseline architecture with original *sim-outorder*'s pipeline, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

speedups from 1.40 to 1.54, while reducing from 3 to 1 dispatch stage provides speedups of 1.41, and reducing from 4 to 1 fetch stage allows speedups of 1.52. The gains from less writeback stages were expected, because they decrease the number of cycles that are necessary to make inputs to be ready. The gains from reducing the number of issue stages were expected for the same reasons, although they could also reduce the gains from trace reuse as issue stages are bypassed by both RST and DTM.

6.7 Varying pipeline width

For this experiment, our objective is to verify the effects of varying the pipeline width in RST's performance. As pipeline width we mean the number of instructions that can be fetched, decoded, dispatched, issued, and committed by cycle on each stage.

We first simulate a pipeline of width 2, then 4 and 8. After that, we correlate results by calculating the speedup over the baseline architecture with pipeline of width 2. The choice of these values is determined by the simulator limitation that the pipeline width must be a power of two. The other parameters, as the number of functional units, are kept the same for this study.

6.7.1 Speedup

Figure 6.26 presents the harmonic mean speedups for the baseline architecture, DTM, RST (DTC) and RST (RTC) varying the pipeline width over the baseline

architecture with a 2-wide pipeline. The horizontal axis shows the pipeline width, while the vertical axis depicts speedup.

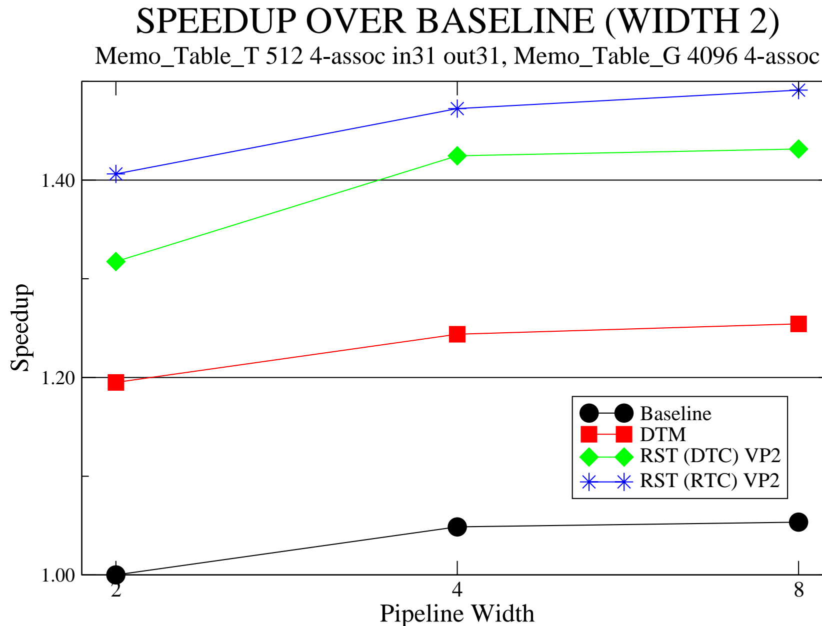

Figure 6.26: Speedup over baseline with 2-wide pipeline, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

The performance improvements are more pronounced when the pipeline width is increased from 2 to 4; increasing the pipeline width to 8 does not increase performance in the same way in any of the simulated processors. We suspect that this is due, among other things, to the number of functional units, which has not been raised accordingly to fit the wider pipeline. We study this aspect in the next Section.

6.8 Varying number of functional units

As in the previous experiment of Section 6.7 we verified that doubling pipeline width does not increase performance in the same way from 2 to 4 and from 4 to 8 wide pipelines, we designed the following measurements to verify if this behavior was caused mainly by the lack of functional units to execute.

We simulate the different processors with the same pipeline width but varying the number of functional units. For the baseline, we use the same configuration as before, 2 integer ALUs for add and subtract, and 1 integer ALU for multiply and divide; then, we double these parameters, while keeping constant all the other configuration aspects.

6.8.1 Speedup

Figure 6.27 depicts the average (harmonic mean) speedups over the baseline architecture with the same with the number of functional units set in Chapter 5. For each configuration, we present both the results for the previously set number

of functional units and the results with twice as much functional units (labeled as *2 x FUs*).

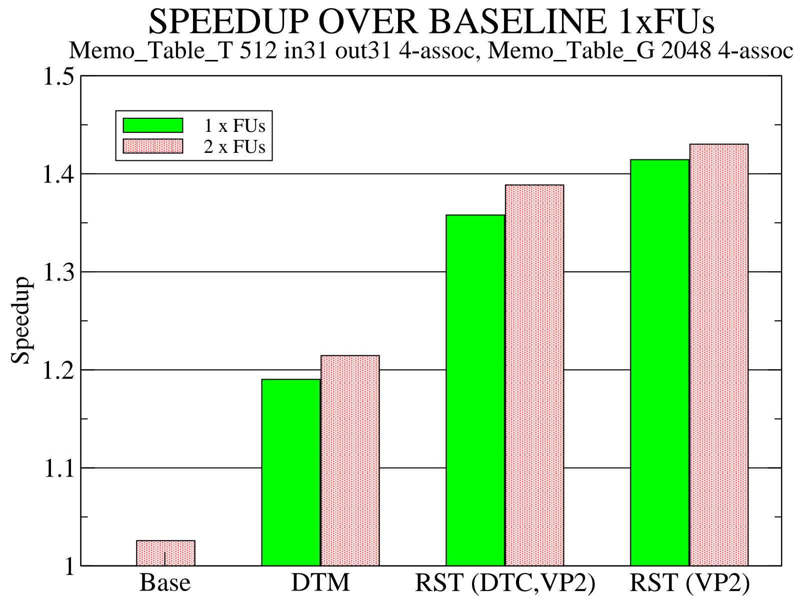

Figure 6.27: Speedup over baseline with original number of functional units, Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

The graph shows that RST (DTC) presents the larger difference in performance when the number of functional units is increased. The speedup increases from 1.36 to 1.39 in this case (difference of 0.03), while for the baseline architecture this increase is 0.025. The differences in the performance improvements are almost the same for all the configurations, but increasing the number of functional units also increases the complexity of issue and writeback, hence these small speedups hardly would pay off in a real processor.

We conclude that the limitations previously seen in Section 6.7 when the pipeline width was increased are not imposed by the lack of functional units but by all the limitations on memory access, control dependencies and true data dependencies, and increasing the number of functional units in this case would not solve the problem of how to increase performance in the wider pipeline.

6.9 Varying cache configurations

In the following experiment, we study the effect of different cache configurations in the performance of RST. We start with the baseline configuration specified in Table 5.4; after that, we double the first-level cache sizes by increasing the number of sets. In the next step, we also increase the second-level cache size in the same way. Finally, we analyze the effects of running the same configuration without a third level of cache. Other aspects such as latency are kept constant.

6.9.1 Speedup

Figure 6.28 features the average speedup (harmonic mean) over the baseline architecture. Bars with the caption *2 x L1* depict speedups when the double-sized first-level caches are used; the remaining bars are provided for comparison and use the baseline cache configuration.

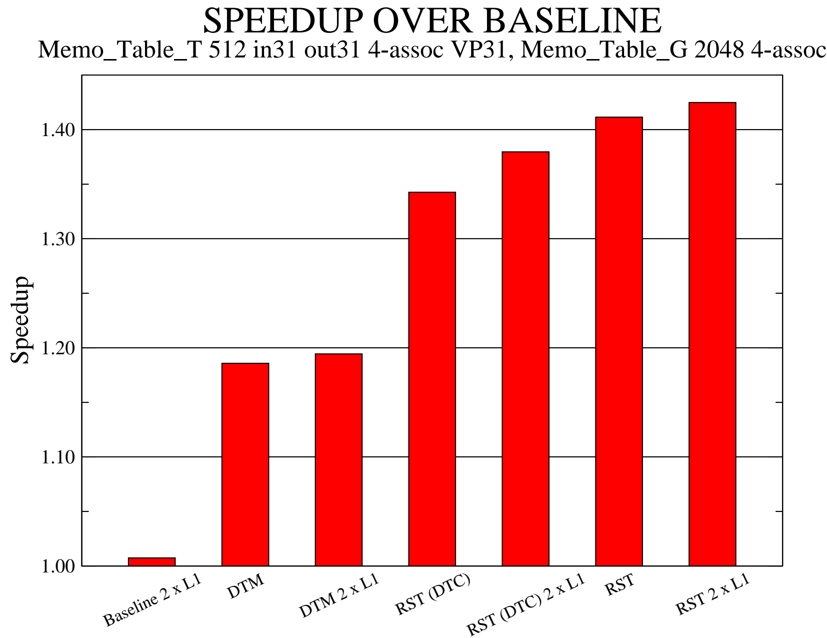

Figure 6.28: Speedup over baseline varying first-level cache size, VP31 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

All architectures with doubled first-level caches present improved performance when compared to the same architecture with the original cache size. However, only for RST (DTC) this increase in cache size resulted in a speedup of more than 1.015, actually 1.037.

Doubling the second-level cache also frustrates our expectations of speedups, with less than 1% of improvements in performance when compared to the doubled first-level cache results. As results are almost the same, we do not present the graph for doubled second-level caches. The same doubled second-level configuration, but without the third-level cache, resulted in a performance reduced by around 1.01 for all architectures.

From these results, we can conclude that variations on cache configuration affect RST in the same way they affect DTM or the baseline architecture without reuse in a deeply pipelined architecture.

6.10 Stride prediction

In the next experiment, we want to verify if stride prediction can be used to improve RST's performance. The primary definition of RST includes only last-

value prediction, but it can be extended to detect strides among traces created on sequence and then extrapolate these strides to dynamically create traces even with values that have not been seen yet.

6.10.1 Speedup

Figure 6.29 presents the speedups over the baseline for RST (DTC), RST (RTC) and RST (RTC) with stride and last-value prediction, predicting at most 1 input (VP1). In this graph, we can see that RST with stride trace creation can further improve performance when compared to RST (RTC), with an average speedup over the baseline of 1.40, while RST (RTC) obtains only 1.36. The difference is even larger when compared to RST (DTC), which can achieve only a speedup of 1.34 for this configuration.

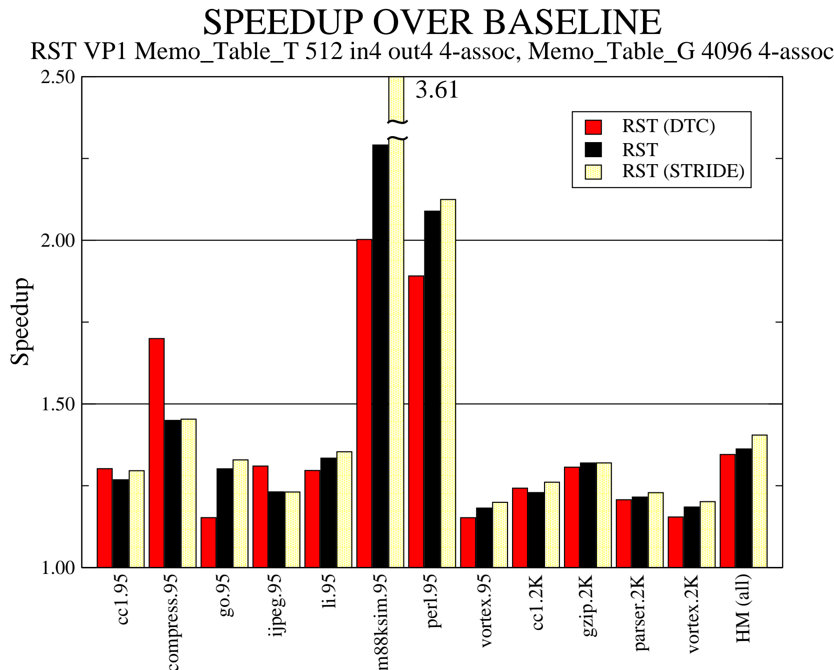

Figure 6.29: Speedup over baseline architecture, RST with and without stride prediction, VP1 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

Figure 6.30 presents the speedups over the baseline for RST (DTC), RST (RTC) and RST (RTC) with stride and last-value prediction, predicting at most 2 inputs (VP2). For all benchmarks and predicting at most 2 inputs, RST with stride trace creation does as well as or better than RST (RTC). In average (HM), the speedup over the baseline is 1.44 for RST (STRIDE), against 1.42 for RST (RTC). The difference is not large, thus the extra complexity for implementing the stride trace creation may not pay off for VP2.

From these results, we conclude that stride trace creation has the potential for further improving performance, but in the current implementation these gains are not as high as expected, probably because there are many cycles between the beginning of a stride sequence, the stride identification, and the stride reuse.

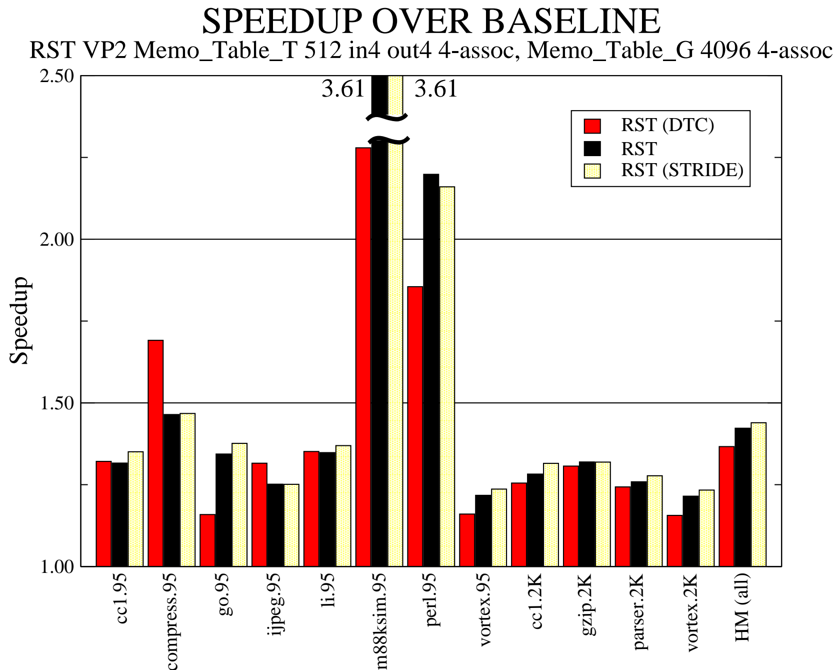

Figure 6.30: Speedup over baseline architecture, RST with and without stride prediction, VP2 Memo_Table_T 512 entries (31 inputs, 31 outputs) 4-way, Memo_Table_G 2048 entries 4-way

6.11 Varying reuse domains

In the following experiments, we introduce memory access in the reuse domain and then we analyze the resulting performance.

The configuration of memoization tables for these simulations uses smaller tables than most of the other limits studies in this Chapter because of the extra computational effort needed to simulate memory reuse. Setups for both Memo_Table_T and Memo_Table_G are depicted in Table 6.3. The number of loads and stores that could be inserted into traces is limited to the maximum number of instructions in a trace (64 instructions) because of memory footprint issues for our simulations.

Table 6.3: Configuration for Memo_Table_T and Memo_Table_G, experiments with memory reuse

Table	Parameter	Value
Memo_Table_T	Entries	1024
	Associativity	fully-associative
	Size	822 KB
Memo_Table_G	Entries	4096
	Associativity	fully-associative
	Size	64 KB

6.11.1 Speedup

Figure 6.31 shows the average speedups (harmonic means) over the baseline architecture when the reuse domain is varied (with or without memory reuse).

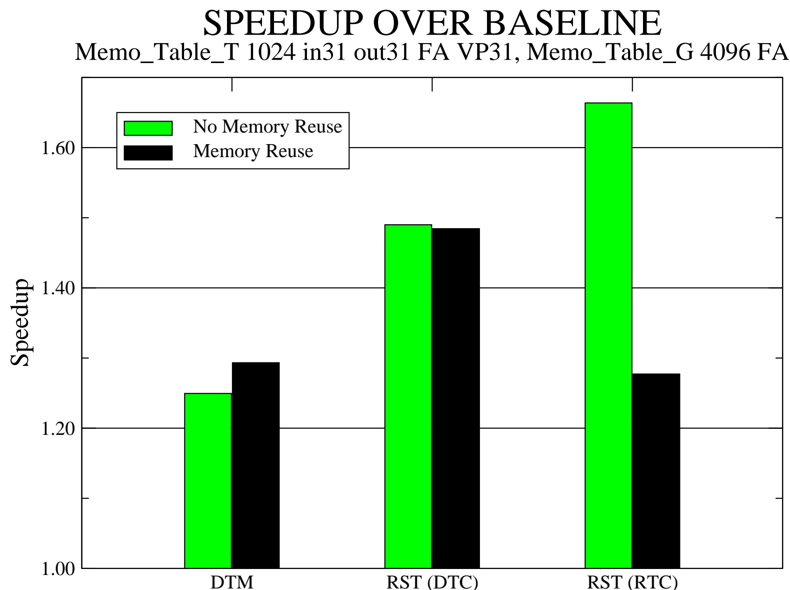

Figure 6.31: Speedups over baseline, VP31 Memo_Table_T 1024 entries (31 inputs, 31 outputs) FA, Memo_Table_G 4096 entries FA

The average speedups show that DTM can benefit from memory reuse, but RST does not show the same capability. For RST with the RTC policy, this is even clear: average performance is much worse when memory accesses are reused, dropping from a speedup of 1.66 over the baseline to only 1.28. Figure 6.32 shows the speedups over the baseline for only RST (RTC) with and without memory reuse to demonstrate it is a tendency not only of one or two, but all benchmarks.

In the next subsection, we will explain this behavior, but from the current results we can say that RST has the potential for better speedups than DTM with memory reuse. As memory reuse greatly increases hardware complexity, it is another point in favor of RST.

6.11.2 Reuse contribution to committed instructions

In this measurement, we want to verify why memory reuse reduces performance for RST, mainly for the RTC policy. Figure 6.33 shows the contribution to committed instructions of the different types of reuse (isolated instruction reuse, trace reuse, speculative trace reuse). For each benchmark, the first bar presents statistics for RST without memory access reuse, while the second bar depicts results for RST with memory reuse.

From this graph, we can see that RST is having problems to speculatively reuse traces. In some benchmarks, the number of committed instructions that are bypassed by speculative trace reuse drops to almost none. This is due to

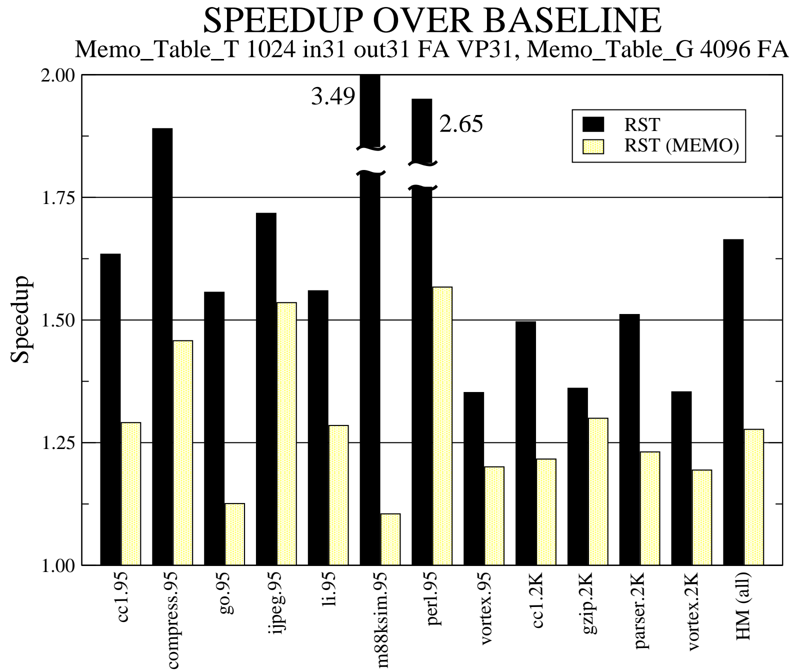

Figure 6.32: Speedups over baseline for RST, VP31 Memo_Table_T 1024 entries (31 inputs, 31 outputs) FA, Memo_Table_G 4096 entries FA

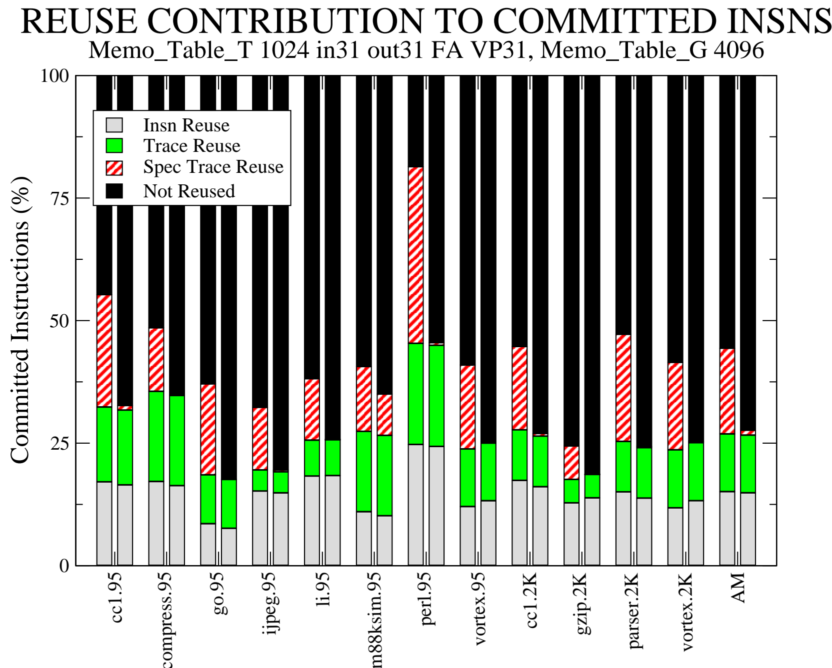

Figure 6.33: Reuse contribution to committed instructions for RST (with and without memory reuse), Memo_Table_T 1024 in31 out31 FA VP31, Memo_Table_G 4096 FA

the small redundancy presented by traces constructed with instructions not in Memo_Table_G and that include memory accesses. While for other architectures average trace length (AM) is around 2.5 instructions, in RSTm the average length is 10.7 instructions. Therefore, including memory accesses reduce trace redundancy, and therefore a less speculative trace construction technique as DTC should be used if memory access reuse is considered important for a given architecture.

6.12 Summary

In this Chapter, we studied how many configuration aspects change performance and the characteristics of reused traces in RST. We started with an almost unconstrained architecture, then we reduced some simulation parameters and analyzed the effects on RST. Our main measurements and analysis can be summarized as follows:

- A comparison of DTM with two different trace construction policies, where we discovered that the DTC policy could provide better results (HM speedup of 1.30 over the baseline);

- A study of performance and trace characteristics with very large memoization and fully associative memoization tables, where we determined the speedup upper bound for RST over the baseline as 1.85, and 1.41 over DTM;

- The measurement of contribution to committed instructions of the different types of reuse;

- A study about the effects of RTC in trace length, critical path length, trace inputs, trace outputs, and number of branches, where we found that this trace construction policy increased all the studied aspects;

- A study of the reasons for terminating traces in constructions, where we learned that system calls and memory accesses are the major impediments for longer traces;

- Measurements of effects of Memo_Table_T associativity in performance, determining that it is very important for performance in RST but not for DTM;

- A study of Memo_Table_T number of entries, where we showed that it is important for both RST and DTM;

- A comparison of different input and output scopes, showing that small scopes could reach performances even better than the larger scope sizes;

- A study of the number of inputs being predicted by trace, where we determined that 2 or 3 predictions per trace achieved the best results;

- Studies of variations in pipeline depth, pipeline width, and number of functional units;

- Different cache configurations, where we showed that RST could provide better speedups than doubling first-level caches;

- A study of stride-aware RST, showing that the technique can provide additional speedups over RST, with a speedup of 1.44 over the baseline against 1.42 for RST without stride trace creation;

- Variation on the reuse domain, showing that the current RST implementation did not achieve better performance by reusing memory accesses, but even without them, it could provide better speedups than DTM with memory reuse (DTMm).

7 RESULTS FOR A RST ARCHITECTURE

"Have you guessed the riddle yet?" the Hatter said, turning to Alice again.
"No, I give it up," Alice replied. "What's the answer?"
"I haven't the slightest idea," said the Hatter.

Lewis Carrol, "Alice's Adventure In Wonderland"

In this Chapter, we study the effects of many architecture parameters in RST. We start without any confidence mechanisms in Section 7.1 to show that confidence mechanisms are necessary to select traces to be speculatively reused for almost all studied cases. Then, we run experiments with perfect (oracle) confidence to determine the speedup upper bounds that may be obtained with RST with the resource restrictions of Chapter 5, and then we present the results in Section 7.2.

After presenting both the top and the bottom in terms of confidence mechanisms, we start trying very simple confidence heuristics based on information already available from traces to restrict mispredictions in Section 7.3. In the sections following Section 7.3, we present results for different confidence mechanisms based on saturated counters in Sections 7.4 and 7.5.

Section 7.6 discusses how confidence table sizes affect performance. In Section 7.7, we present results for a stride trace construction version for RST. Section 7.8 shows another variation of confidence, based on counters stored on Memo_Table_T.

Finally, we briefly compare RST to other works that not DTM in Section 7.9.

For most configurations, we show only the comparison of performance against the baseline or DTM architectures by using speedup graphs. But in some cases where there are interesting trends to study we also present other measurements for further discussion. Table 7.1 shows the configurations for the memoization tables used in this Chapter. The Memo_Table_T table size is for 4 inputs and 4 outputs in the trace contexts.

7.1 RST without confidence mechanisms

In this first experiment, we want to verify the results for RST without any confidence mechanisms to determine whether there is a necessity for confidence mechanisms to reduce mispredictions. We compare the results to DTM, as RST is intended to improve performance of conventional trace reuse by speculatively reusing traces. Hence, if RST presents worse performance than DTM even when using more resources, there are no benefits of employing it.

Table 7.1: Configuration for Memo_Table_T and Memo_Table_G for Chapter 7

Table	Parameter	Value
Memo_Table_T	Entries	512
	Associativity	4
	Size	23.5 KB
Memo_Table_G	Entries	2048
	Associativity	4
	Size	32 KB

We start presenting a performance comparison against a regular trace reuse technique (DTM). After that, we show how the misprediction rates and penalties

7.1.1 Speedup

Figure 7.1 shows the speedups over DTM for RST without confidence mechanisms. A column lower than the 1.0 line means that a benchmark experiences a performance decrease when compared to DTM. For each benchmark, the first two bars present the results for RST using the DTM Trace Construction policy (DTC); both prediction of one value (VP1) and two values (VP2) per speculatively reused trace are presented for both trace construction policies (DTC and RTC).

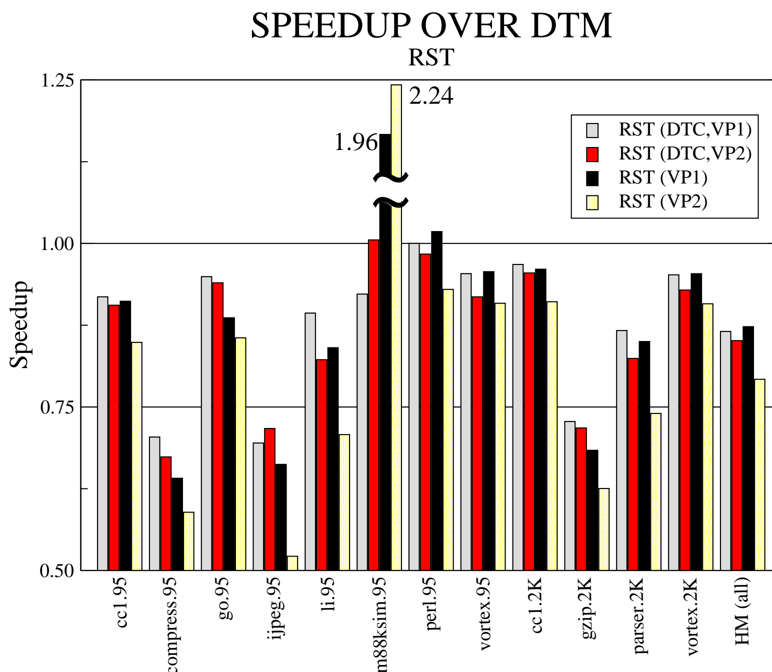

Figure 7.1: Speedup over DTM architecture, without confidence mechanisms

For most cases, there is a considerable decrease in performance when RST without confidence is used. Increasing the number of predicted values per trace also decreases performance, and in most cases, RTC policy presents worse results than

DTC because it creates traces in a more speculative way. Therefore, traces are less likely to be redundant, and more mispredictions occur. The notable exception is *m88ksim.95*, which presents large speedups even without any confidence mechanism using the RTC policy (1.96 for VP1 and 2.24 for VP2). As Table 7.2 shows, this benchmark has the smallest misprediction rates over all simulated benchmarks for the RTC policy.

7.1.2 Misprediction rates

From the misprediction rates shown in Table 7.2, we can also point that there is an overall decrease on misprediction rates when varying the trace creation policy. For VP1, RTC provides a decrease from 61% to 55%, and for VP2 the misprediction rates go from 63% to 60% when compared to the DTC policy. Therefore, in average RST using the RTC policy is able to be correct more often than with the DTC policy.

Table 7.2: Misprediction rates for RST

BENCHMARK	**misprediction rate (%)**			
	DTC		**RTC**	
	VP1	*VP2*	*VP1*	*VP2*
cc1 (95)	55.0	56.8	54.2	61.0
compress95 (95)	78.0	82.4	77.7	83.2
go (95)	78.8	79.0	75.5	77.9
ijpeg (95)	88.2	88.1	83.3	89.6
li (95)	59.1	67.5	65.0	70.2
m88ksim (95)	45.5	25.3	**13.2**	**20.6**
perl (95)	46.4	49.0	34.0	40.0
vortex (95)	41.6	51.9	26.2	36.7
cc1 (2K)	55.0	59.4	53.4	61.7
gzip (2K)	84.8	85.4	85.4	86.0
parser (2K)	61.0	68.9	59.7	68.0
vortex (2K)	39.2	46.2	25.7	35.9
AM (ALL)	**61.1**	**63.3**	**54.5**	**60.9**

For most cases, increasing the number of inputs also increases misprediction rates, which can be explained by traces with more unknown inputs being less predictable. On the other hand, increasing the number of inputs that can be predicted provides more speculative trace reuse opportunities, allowing further increases in performance.

7.1.3 Mispeculation penalty

Measuring the misprediction penalty of speculative trace reuse in out-of-order, superscalar architectures is a complex matter. It is hard to define where starts the impact of a given mispeculation on the performance (it may be measured from the cycle where the misprediction occurs, or from the cycle where it is detected, and there is also the problem of defining where it ends).

We chose to measure the number of cycles that RST takes to recover from a mispeculation since the value is predicted until fetch is redirected to the recovery address. Besides this, a mispredicted trace will also incur in filling the pipeline again with instructions that would be available if it was not predicted, but we leave this out of the mispeculation penalty calculation to reduce the complexity of the metric.

Figures 7.2 and 7.3 depict the misprediction penalties for RST using the different trace creation policies. For each benchmark, the first bar depicts the average misprediction penalty for VP1 (prediction of one value), and the second bar shows the same results for VP2.

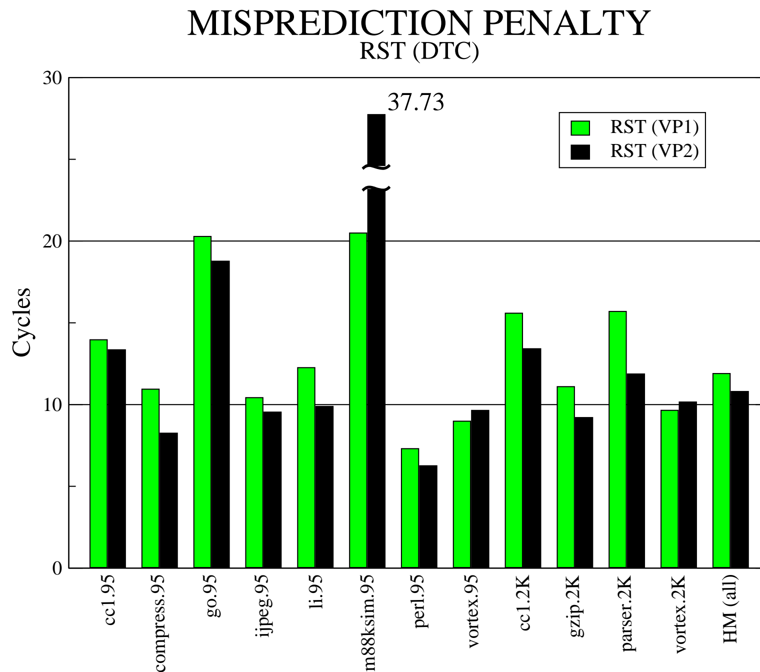

Figure 7.2: Misprediction penalties, RST with DTC policy

The harmonic mean of all average misprediction penalties is around 10 cycles for both trace construction policies. For most benchmarks, there is not a large difference in misprediction penalties when varying the trace construction policy. On the other hand, the benchmark *m88ksim.95* shows a very strong reduction on misprediction penalties (from about 37, for DTC VP2, to 5 cycles). This, combined with the low misprediction rate (20% for VP2), may explain in part the surprising speedups observed for *m88ksim.95* (2.24 over DTM). *perl.95* also presents some of the lower misprediction penalties, but the misprediction rates are more than twice as high as *m88ksim.95*, which impares RST capacity of increasing performance in this case.

7.1.4 Remarks about RST without confidence mechanisms

From these initial results for RST without confidence mechanisms, we verify that for most benchmarks it is not possible to achieve acceptable performance by means of speculative trace reuse without limiting in some way the misprediction occurrences and the associated penalty. The only benchmark that presents a very predictable sequence of traces is *m88ksim.95*, which achieves speedups of more than 2 over a non-speculative trace reuse technique without confidence mechanisms.

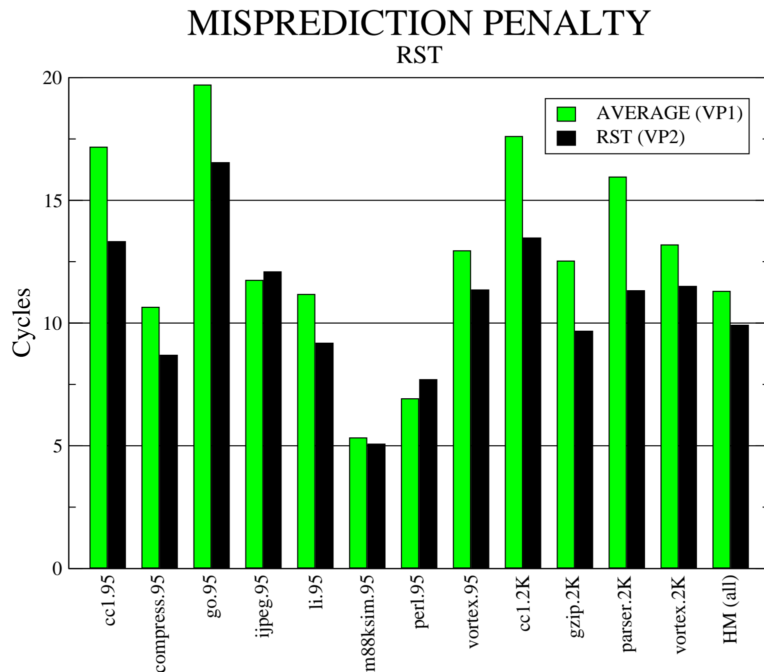

Figure 7.3: Misprediction penalties, RST with RTC policy

Consequently, we conclude that RST requires a confidence method to avoid mispredictions, in order to get speedups over non-speculative trace reuse techniques, or another technique that can either reduce misprediction rates or misprediction penalties. As confidence mechanisms are well-tested and simple ways to reduce misprediction rates, we study some confidence configurations designed to unleash the potential performance of RST in the next Sections.

7.2 RST with oracle confidence

The experiments in this Section are designed to determine the speedup upper bounds for RST when the number of inputs, outputs, predicted values, and memoization table sizes are restricted to values that could be implemented in current microprocessors. For these experiments, only the confidence is not configured as a feasible mechanism, but perfect confidence is used.

In the following graphs, we present results for both RTC and DTC policies for trace construction, as well as for VP1 (prediction of one value) and VP2 (prediction of at most two values). The choice for not predicting more than two values is related to the increasing complexity for testing more values at the writeback stage and also because the limits study in the previous Chapter does not show significant improvements for prediction of more than two inputs.

7.2.1 Speedup

Figure 7.4 shows the speedups over baseline for RST with perfect confidence (oracle) and also for DTM.

For all benchmarks, RST and DTM are able to produce significant speedups over

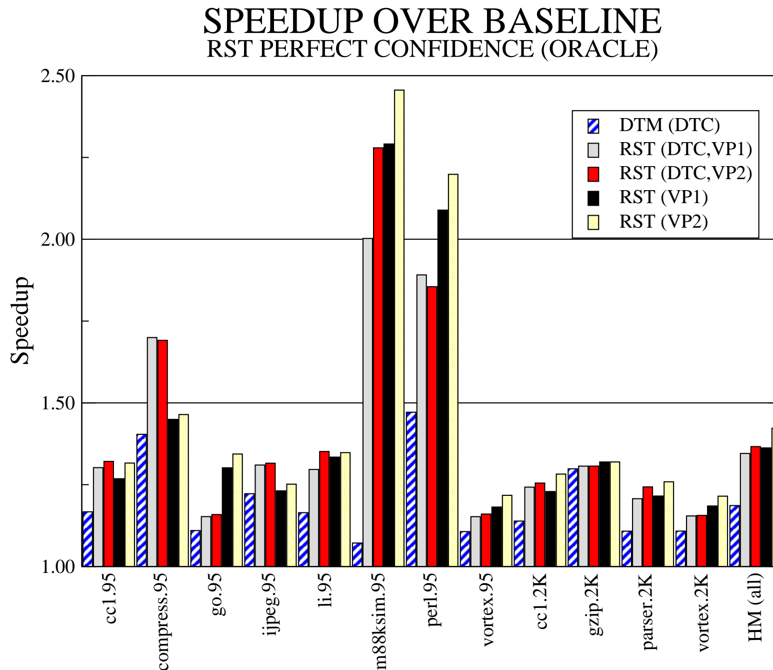

Figure 7.4: Speedup for RST over baseline with perfect confidence

the baseline architecture. The best average result (harmonic mean) is obtained by RST with the RTC policy and prediction of two inputs, with a speedup of 1.42 over the baseline. The other RST results are in the range of 1.34 to 1.36 over the baseline architecture.

Figure 7.5 represents the speedups for RST over DTM when perfect confidence estimation (oracle) is employed. Again, RST VP2 produces the best average speedup, 1.19 over DTM. This clearly shows the potential for RST improving performance in a constrained configuration.

From these results, we conclude that RST has a large potential for performance improvements for this architecture configuration when mispredictions are avoided. In the following Sections, we use different confidence mechanisms instead of perfect confidence to measure performance for realistic processors.

7.3 RST with simple confidence mechanisms

For these experiments, we try to use thresholds on certain trace characteristics to determine in which cases prediction is allowed. Different from counter-based mechanisms, there is no need for extra tables or fields in Memo_Table_T, as the information used to determine if the current trace is a good choice to be predicted or not is already stored along with the trace candidate. The advantage of such mechanisms is that there is almost no extra cost for their implementation, but the disadvantage is that the determination of parameters that make a trace more likely to be predictable or more likely to improve performance is a though task.

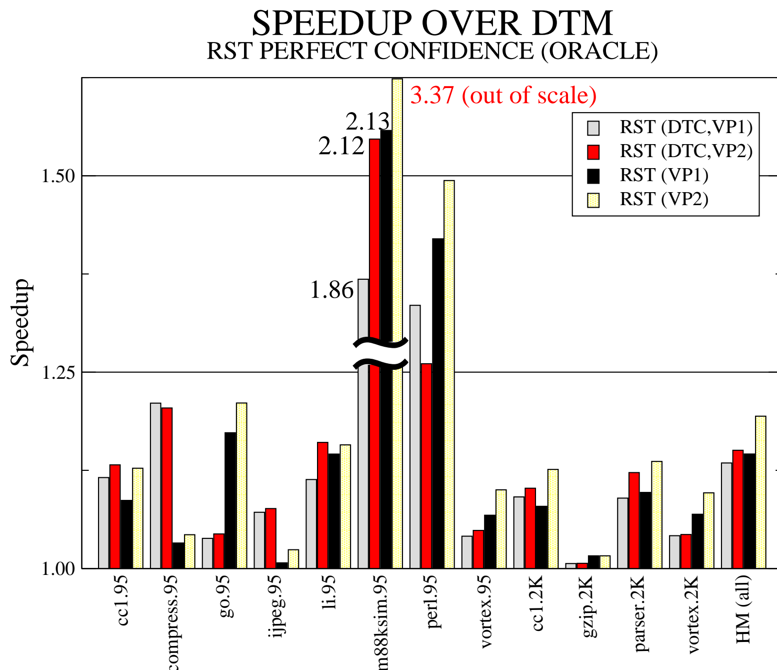

Figure 7.5: Speedup for RST over DTM with perfect confidence

7.3.1 Speedup with limitation on the number of alternative traces

In the current experiment, we select prediction opportunities by the number of alternative traces found for a given PC address. The rationale of this confidence scheme is that PC addresses with less trace candidates for reuse are more likely to be redundant and easier to choose.

Figure 7.6 shows the speedup over DTM for RST, predicting at most one input. For almost all benchmarks but for *perl.95* and *m88ksim.95*, pruning predictable traces by the number of candidates does not reduce losses with mispeculations. The results are the opposite of what was expected: performance decreases when the number of maximum alternative traces is constrained, showing that most predictable traces are found in the cases where there are more candidates. Results for RST with the DTM trace construction policy are similar to the presented ones, thus we will not present them.

As this policy does not produce a performance better than non-speculative trace reuse, we conclude that it is not enough to be used alone as a confidence scheme for RST.

7.3.2 Speedup with limitation on the minimal critical path

Another possibility is to use information about the critical path inside a trace to determine whether it is worth predicting it or not. If a trace encloses a long critical path, it may provide performance improvements that make prediction worth in the average case, even when some mispredictions occur. On the other hand, as for the study with limitation on the number of alternative traces, this policy may also limit the prediction of desirable traces that do not achieve the minimum criteria.

Figure 7.7 presents the speedup for RST over DTM when prediction is restricted

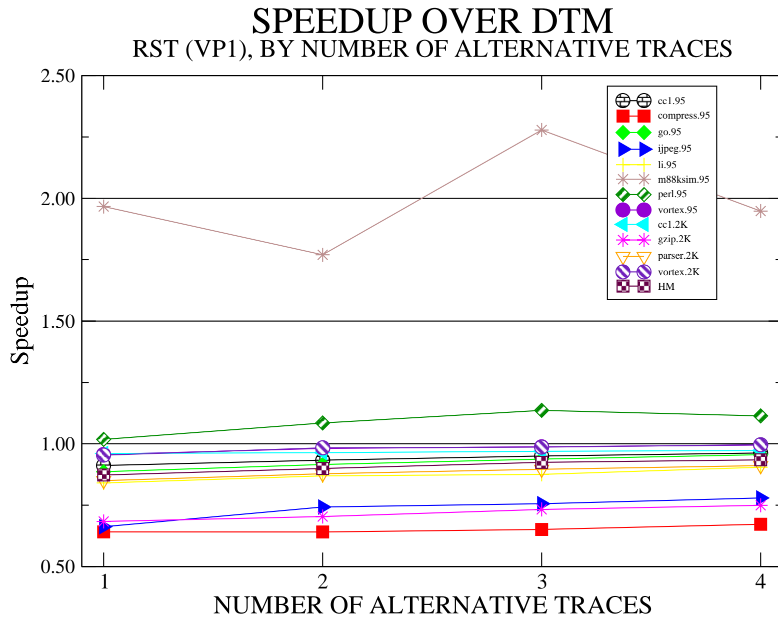

Figure 7.6: Speedup over DTM, varying the maximum number of alternative traces

based on the critical path of trace candidates.

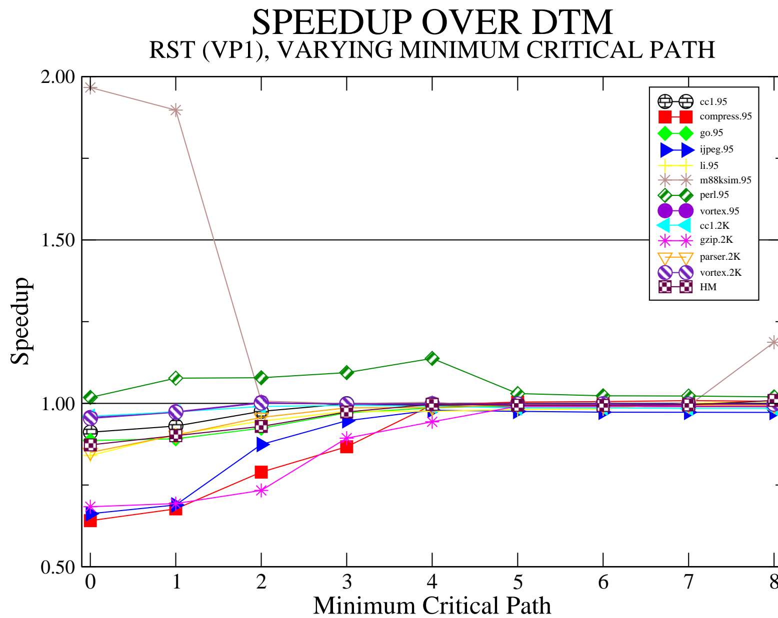

Figure 7.7: Speedup over DTM, varying the minimum critical path

The benchmark *m88ksim.95* presents a decline on performance, showing that even traces with small critical paths can have a large impact on performance. On

the other hand, its performance starts to increase again when the threshold for critical paths is set to 8 instructions. A similar lack of linearity in decrease or increase of performance can be seen in *perl.95*, in different points of the graph. This trend seems to be related to some specific traces that can be very important to increase or decrease performance (key traces), and that are sensitive to this parameter. Therefore, the tuning of this threshold can affect performance on an unpredictable way for different values and for different benchmarks.

As it can be seen by the harmonic mean and by most of the benchmarks, increasing the length of the critical path that allows a trace to be predicted reduces the loses on performance from mispredictions, but it still does not help RST to achieve performance better than non-speculative reuse techniques. Therefore, it cannot be used alone as a solution for misprediction penalties imposed by RST.

7.3.3 Prediction rates with limitation on the minimal critical path

Figure 7.8 depicts the average (arithmetic mean) prediction and misprediction rates over all traces reused, predicted or mispredicted. This graph shows how constraining predictions by the critical paths affects both predictions and mispredictions.

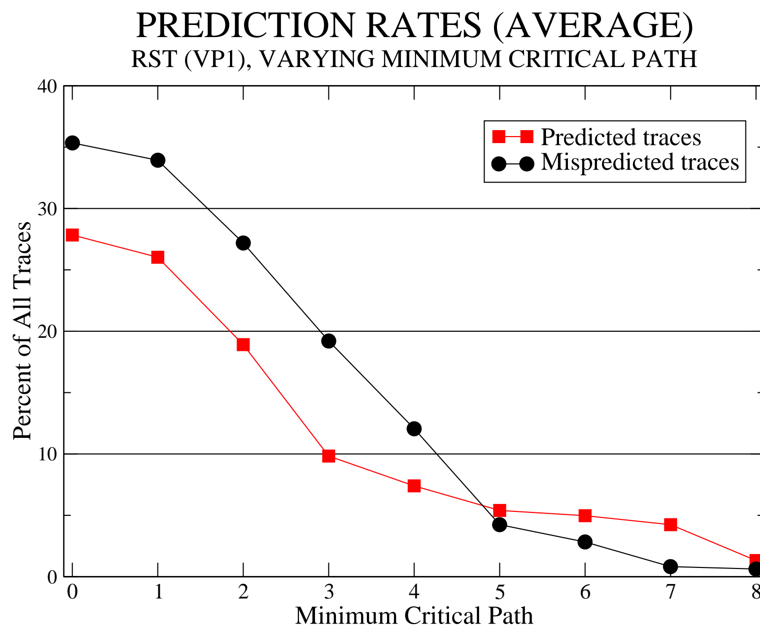

Figure 7.8: Traces predicted or mispredicted, varying the minimum critical path

As we can see, increasing the critical path threshold reduces misprediction rates but at the same it reduces the number of correctly predicted traces. The amount of predicted traces overcomes the number of mispredicted traces when the critical path threshold reaches 5 instructions, but they are still very similar (5.4% and 4.2% of all traces). By the time that the critical path threshold reaches 8, only 1.3% of all traces are correctly predicted, and only 0.6% are mispredicted. Although the misprediction rate is very low, the prediction rate is low too, reducing any potential performance gains to a very small fraction of their upper bounds.

7.3.4 Speedup with limitation by the number of ready sources

Traces with more ready input values may present more predictability than other traces. In this experiment, we test this hypothesis by restricting predictions to traces with a minimal number of ready sources. We vary the threshold from zero (any trace may be predicted) to 3 (only traces with 4 inputs and 1 unknown input may be predicted).

Figure 7.9 depicts the speedups of RST (DTC) over DTM. We do not present results with RST with the RTC policy because they are similar to RST DTC, but slightly lower.

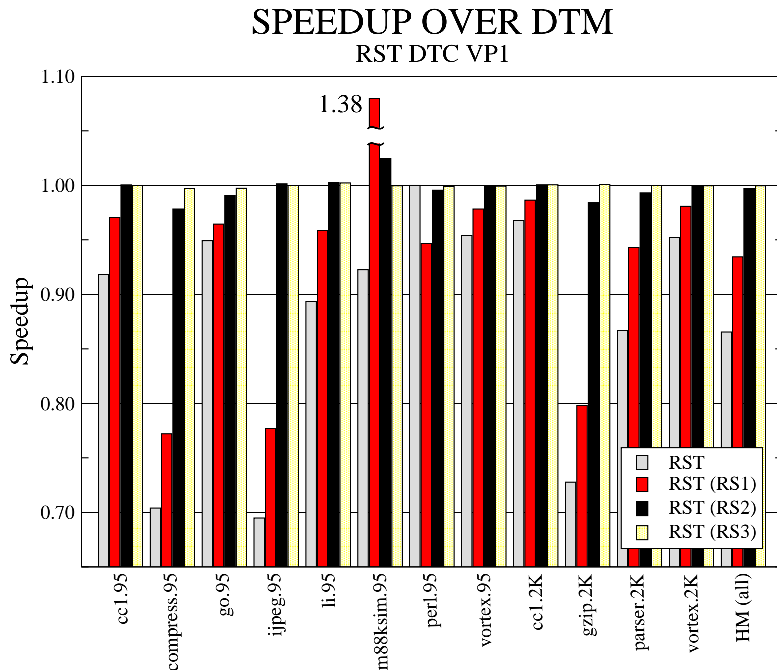

Figure 7.9: Speedup of RST (DTC) over DTM, restricting by number of ready sources

The only benchmark that is able to achieve considerable performance improvements over DTM is *m88ksim.95*, and only in the case of restricting prediction to at least one ready input (RS1). As a matter of fact, this benchmark shows better performance without any confidence mechanism than this confidence scheme, where it obtains a speedup over DTM of 1.96 for the same VP1 configuration against the 1.38 in the current case.

Increasing the number of required input sources decreases the number of predictions, which makes performance tends to the same as non-speculative trace reuse (DTM).

Restricting prediction to the cases where 3 sources are ready reduces predicted and mispredicted traces to less than 1% of all reused, predicted or mispredicted traces (with exception of *m88ksim.95*, where RST can still contribute with 1% of predictions or mispredictions). Therefore, this threshold does not only contributes to reduction of mispredicted but also of many correctly predicted traces.

We conclude that restricting predictions by the number of ready sources is not able to reduce mispredictions while keeping correct predictions to contribute for

performance improvements, and the best results obtained with this technique barely could reach the performance of non-speculative trace reuse.

7.3.5 Remarks for this Section

From the results with simple confidence thresholds, we can conclude that none of the tested mechanisms is able to obtain reduction of mispredictions while keeping correctly predicted traces to further improve performance. Constraining trace speculation by the number of ready sources, by the critical path found in the traces or by the number of alternative traces are not enough to help RST improving performance.

Therefore, we assert that more complex and more powerful mechanisms must be developed in order to unleash at least part of the performance improvements that our experiments with oracle confidence have shown to be possible with RST. In the next sections, we study confidence mechanisms based on saturated counters to choose whether a speculation should be considered or not.

7.4 RST with confidence counter 4096 3 3 3 1 3

For the following set of experiments, we want to test a confidence mechanism based on saturated counters that allows predictions when the counters reach certain thresholds. Correct predictions increase counters, while mispredictions incur in counters being decreased.

The first configuration uses a confidence table with 4096 entries, each one being a saturated counter with 2 bits (Table 7.3). Each counter starts with the maximum value (3), and each correct prediction increases a counter by one. A misprediction decreases a counter by 3. Predictions are only allowed after the counter achieved the value 3, therefore 3 consecutive correct predictions will allow a trace to be speculatively reused.

Table 7.3: Confidence configuration for counter 4096 3 3 3 1 3

Parameter	Value
Entries	4096
Bits per entry	2
Maximum value (saturation)	3
Threshold for prediction	3
Correct prediction increment	1
Misprediction decrement	3
Initial value	3

The confidence table is indexed by the PC counter, and it does not uses tags to ensure that a counter is used only for a certain PC address. Thus, there will be a certain amount of aliasing for each counter.

We selected this configuration to start our analysis because it provides some compromise between allowing predictions when it does not know about previous history but at the same time, it avoids that successive misprediction for the same entry occur. From this initial configuration, we explore the results, trying to *(i)* im-

prove overall performance, and also *(ii)* to reduce the variability among speedups of different benchmarks.

7.4.1 Speedup

For the first measurement for this configuration, we verify how RST performance changes in function of the confidence mechanism. Figure 7.10 shows the speedup for RST over the baseline architecture for the configuration of Table 7.3. For each set, the first bar presents the speedups for DTM for means of comparison; the next bar shows results for RST when only one input value can be predicted per reused trace, while the last bar depicts the speedups when at most two values can be predicted per trace. The last columns show the harmonic mean for the speedup of all benchmarks for each configuration.

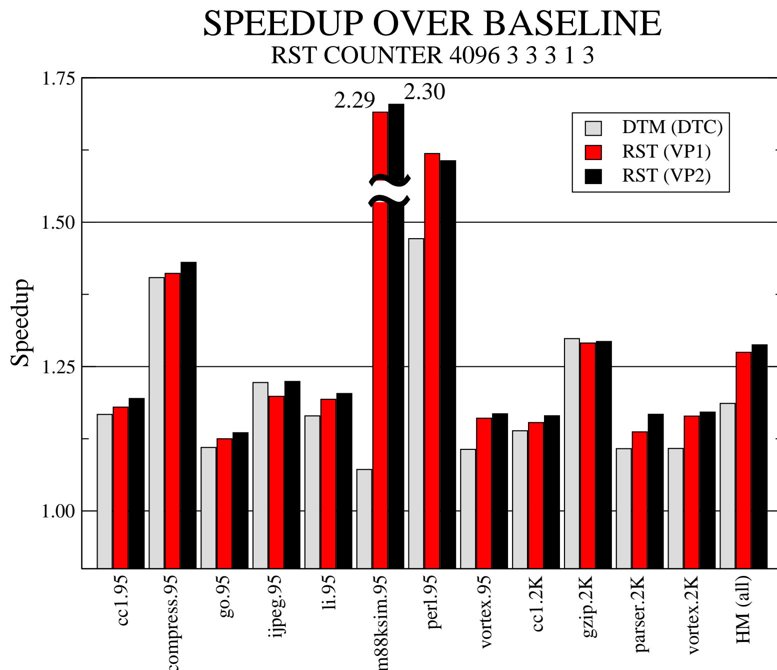

Figure 7.10: Speedup over baseline architecture, counter 4096 3 3 3 1 3

Both RST and DTM present performance improvements for all benchmarks, in the range from 1.07 (*m88ksim.95* running over DTM) to 2.30 (also *m88ksim.95*, but for RST predicting two inputs per trace). From all the simulated benchmarks, *m88ksim.95* from SPEC95int showed to be the most sensitive to different configurations of RST. Predicting only one input provides a speedup of 2.29 for the same benchmark, thus there is no significant improvement for this specific case for predicting two inputs. But for most benchmarks, excepting *perl.95*, there is a small improvement for RST predicting at most two inputs over only one (a speedup of 1.287 for RST VP2 against 1.274 for RST VP1). For both cases, the achieved speedup is greater than the obtained for DTM, of about 1.18 over the baseline architecture.

A better way to compare RST and DTM results is to calculate the speedups for RST over DTM, shown in Figure 7.11 for the same configurations used in Figure 7.10.

RST provides a speedup of 1.07 over DTM when predicting one value, and 1.08 when predicting two values at most (harmonic means) for the small price in hardware

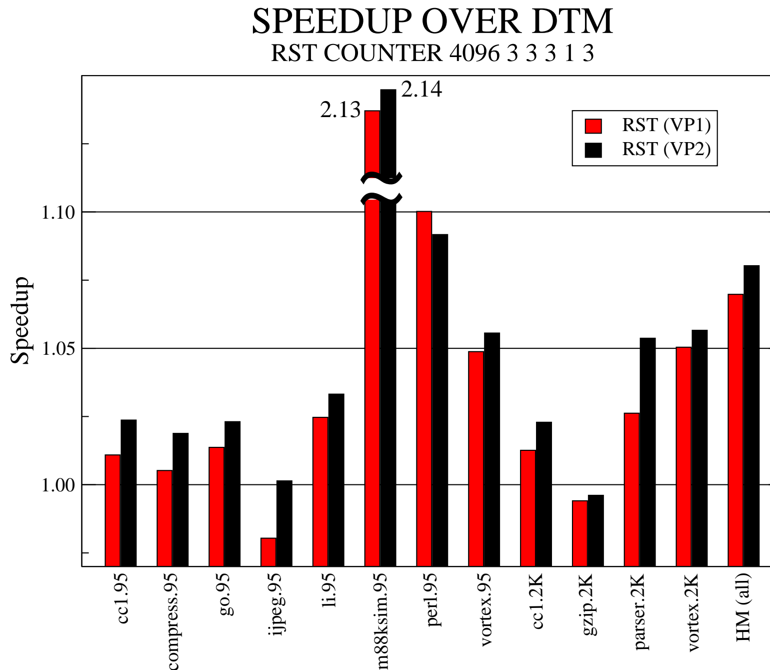

Figure 7.11: Speedup over DTM, counter 4096 3 3 3 1 3

of adding the RS3 stage to verify predictions and the confidence table. For VP2, all benchmarks present better results than DTM but for *gzip.2K*, and for VP1 only the benchmarks *ijpeg.95* and *gzip.2K* show decreases in performance of less than 2% when compared to DTM (these decreases will be analyzed in the next subsections). For both cases, the performance is still much better than in the baseline architecture (a speedup of 1.22 for *ijpeg.95* and 1.16 for *gzip.2K*). The benchmark *perl.95* does not benefit from predicting more than one trace input, as the graph shows.

From these results, we verify that RST with the chosen confidence mechanism provides considerably better performance than non-speculative trace reuse and than the baseline architecture without reuse.

7.4.2 Reuse rate

Only showing the number of reused instructions without classifying them as reused inside or outside traces does not allow the complete realization of how RST increases performance. Reusing traces is usually better than reusing isolated instructions because critical paths may be collapsed, a wider virtual dispatch rate than the pipeline width may be achieved, and traces may also correct several branch predictions in a single cycle. In the next graphs, we first present the total number of instructions skipped by reuse, and after that we show the results for instructions skipped by traces only.

Figure 7.12 presents the rate of instructions that were reused or bypassed by trace reuse for DTM and RST. The total number of skipped instructions grows from 17.19% in DTM to 19.30% in RST (VP1) and to 20.32% to RST (VP2).

Figure 7.13 depicts the rate between instructions bypassed by traces over all simulated instructions, counting only committed instructions. In some cases, like for the benchmarks *compress.95*, *go.95*, and *li.95*, the number of instructions bypassed

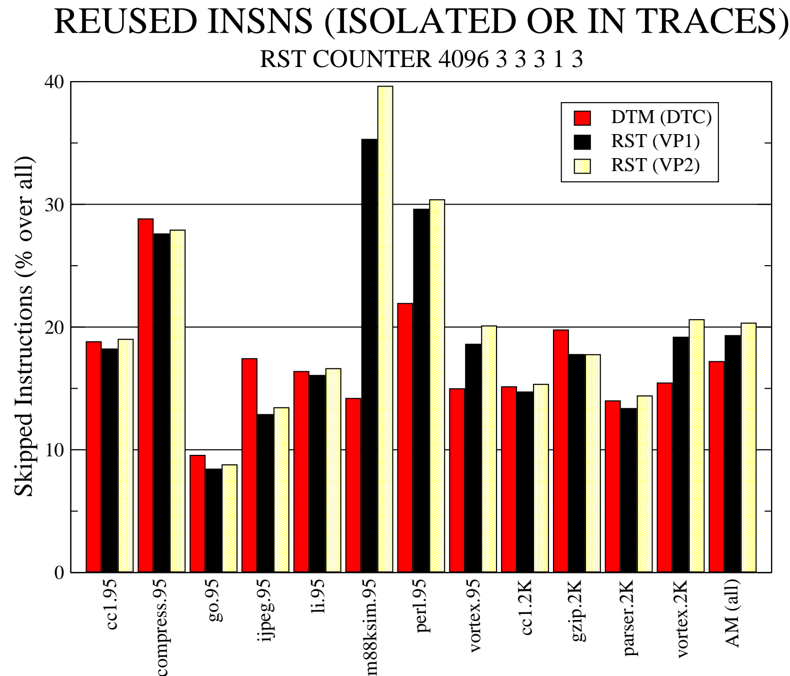

Figure 7.12: Instructions reused (isolated or in traces), counter 4096 3 3 3 1 3

by traces is smaller than for DTM, but performance still experiences improvements, what is more likely related to RST reusing traces in the critical path for these benchmarks and for values being anticipated by value prediction. In other cases, like for *ijpeg.95* and *gzip.2K*, both performance and number of instructions bypassed by trace reuse are reduced. But in average, the number of instructions bypassed by traces increases from 11.68% in DTM to 14.43% in RST (VP1) and 15.61% in RST (VP2) (arithmetic mean). From all benchmarks, *m88ksim.95* obtains again the most benefit from speculative trace reuse, with bypass rates around 47 to 49%. This result explains the outstanding speedups that RST can achieve for this benchmark.

Figure 7.14 presents the contribution of instructions bypassed by traces, speculatively reused traces or by isolated instructions to the number of committed instructions of each benchmark. For each set of three vertical bars, the first one depicts results for DTM, the second one for RST (VP1), and the third bar presents results for RST (VP2).

The benchmarks *ijpeg.95* and *gzip.2K* have less committed instructions originated by reused traces in RST than in DTM, and this explains the performance reductions observed in Figure 7.11 for them. All the other benchmarks present an increase in performance regardless of whether they reuse more or less instructions from the total of committed instructions. An important factor here is that RST uses the RTC policy instead of DTC for these results, hence traces reused in RST are likely to be different from the ones reused in DTM here. In average, RST is able to have more committed instructions coming from trace reuse than DTM, and RST (VP2) is able to reuse more instructions that are later committed than RST (VP1) too.

From the results discussed in this Section, we can say that RST bypasses more instructions than non-speculative trace reuse with the current confidence mechanism in average, and that it can improve performance even by reusing traces in the hot

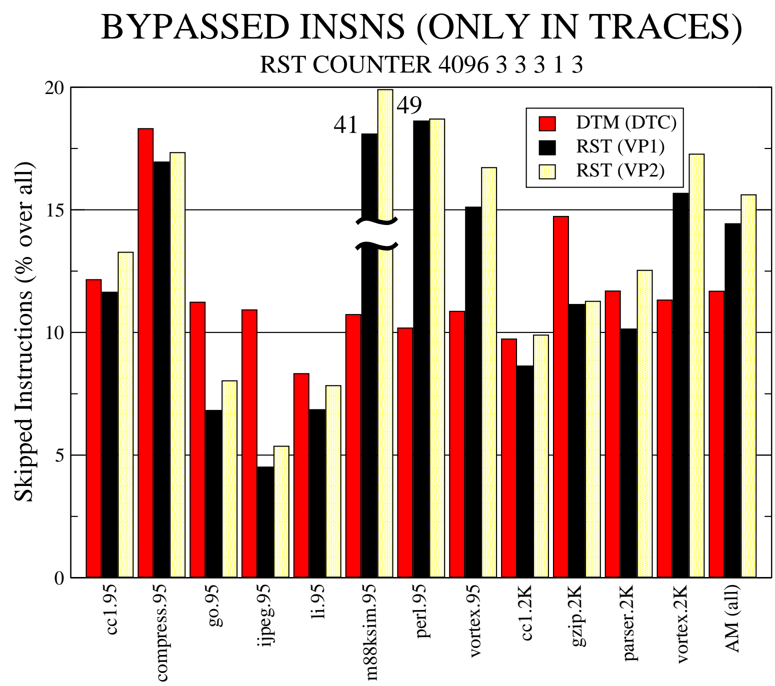

Figure 7.13: Instructions bypassed by traces, counter 4096 3 3 3 1 3

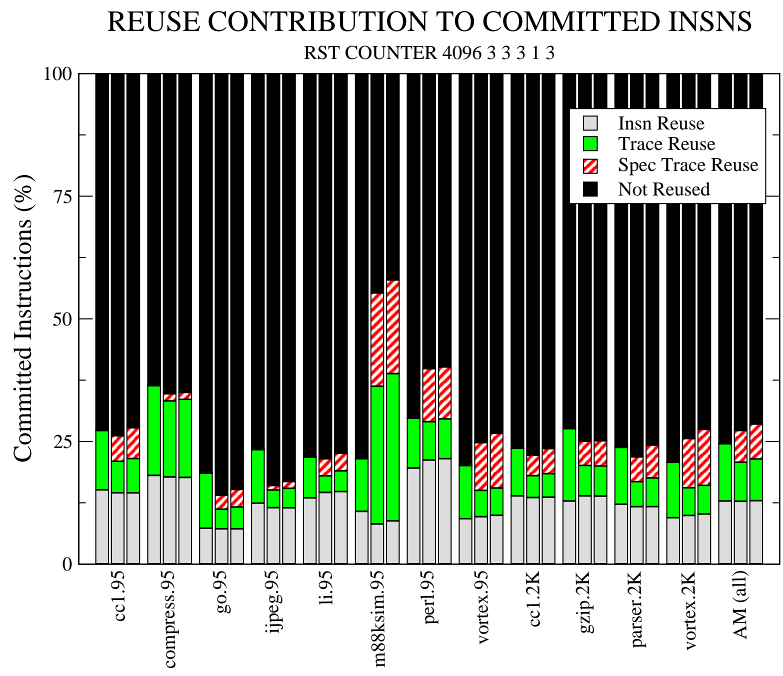

Figure 7.14: Reuse contribution to committed instructions, counter 4096 3 3 3 1 3

paths instead of bypassing more instructions.

7.4.3 Confidence reliability

In this analysis, we want to understand how the chosen confidence mechanism behaves for the selected workload. The rate of correct confidence lookups over all confidence lookups is shown in Figure 7.15. In this measurement, a confidence lookup is considered correct if *(i)* the prediction was correct and the confidence allowed speculative reuse; or *(ii)* it would be a misprediction, but the confidence mechanism did not allow the speculative reuse.

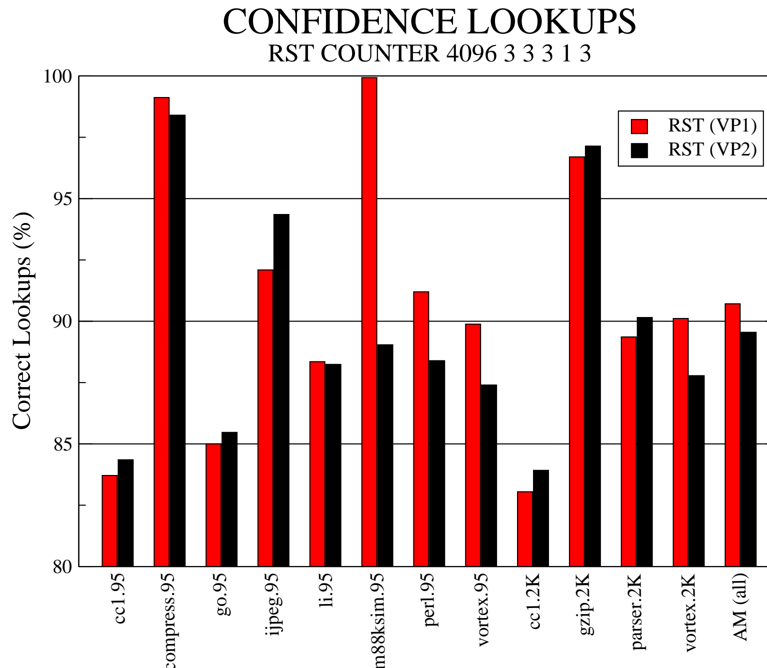

Figure 7.15: Percent of correct confidence lookups, counter 4096 3 3 3 1 3

In most cases, RST (VP1) presents a better confidence hit rate, and this is also the case for the average rate (a small difference of 0.5% only). But as VP2 can predict more traces than VP1, it still can achieve better performance even through a reduced confidence reliability.

7.4.4 Misprediction rates and penalties

For the following measurements, we want to verify how mispredictions cause penalties to RST when the current confidence mechanism is employed.

Table 7.4 shows the misprediction rates for RST VP1 and VP2 for the current confidence mechanism. The misprediction rates are modest when compared to the ones obtained without confidence mechanisms (Table 7.2). The misprediction rates dropped from around 60% to 5%, therefore reducing the penality imposed by mispredictions.

Figure 7.16 shows the cumulative distribution of misprediction penalties for VP1. Most benchmarks show very similar distributions, but the benchmark *gzip.2K* shows a distribution where more mispeculated traces have very large penalties, explaining

Table 7.4: Misprediction rates for RST counter 4096 3 3 3 1 3

BENCHMARK	misprediction rate (%)	
	VP1	*VP2*
cc1 (95)	6.4	8.0
compress95 (95)	0.4	1.5
go (95)	15.7	16.1
ijpeg (95)	10.1	8.7
li (95)	3.6	8.5
m88ksim (95)	0.0	0.0
perl (95)	0.0	0.0
vortex (95)	2.0	3.4
cc1 (2K)	7.1	9.6
gzip (2K)	1.7	2.0
parser (2K)	5.0	7.3
vortex (2K)	1.9	3.1
AM (ALL)	**4.5**	**5.7**

the bad performance observed for it even with the low misprediction rates.

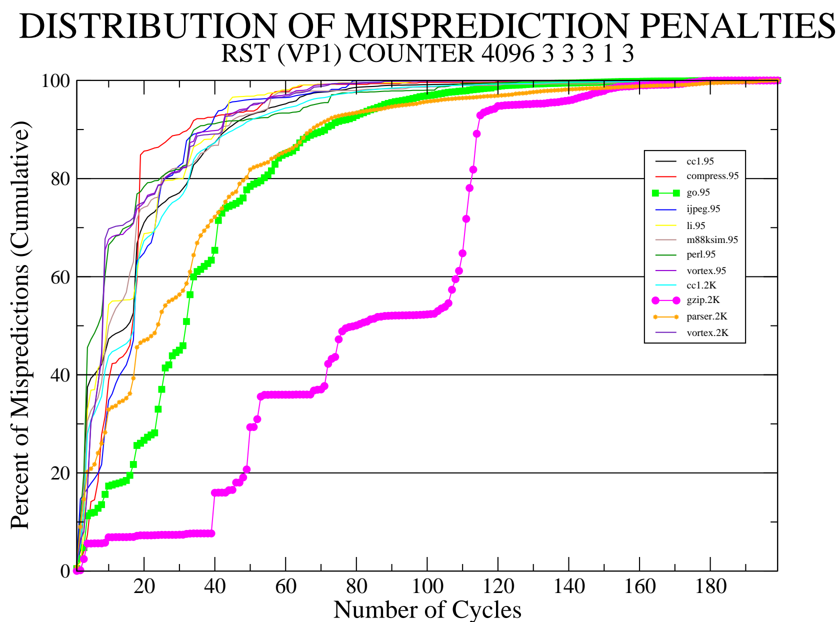

Figure 7.16: Cumulative distribution of misprediction penalties, RST (VP1) confid 4096 3 3 3 1 3

Therefore, we can say that even if the misprediction rates are low, the mispeculation penalties can be higher enough to dissipate the gains that are obtained from correctly speculated traces, which is not the case for most benchmarks as shown in the previous Sections.

7.4.5 Adding path information to confidence

In the previous experiments, the confidence mechanism was accessed using only the current PC address to compute the confidence table index. In the current experiment, we use the outcome of the last 1, 2, and 4 branches xored with the PC address.

Table 7.5 compares the average speedups (HM) over the baseline architecture for the different branch history depths. The results differ only in the third or fourth decimal row.

Table 7.5: Average speedups over baseline varying path depth, RST (VP2) confid 4096 3 3 3 1 3

Branch history depth	Speedup
4	1.2855
2	1.2872
1	1.2843
no path	1.2874

We conclude that adding path information to the confidence mechanism does not increase or improve performance for the current confidence configuration in a significant way.

7.5 RST with confidence counter 4096 7 7 7 1 0

In the previous Section, we used a confidence mechanism with 2 bits per entry and that easily reached the threshold to predict values. For the experiments of this Section, we use a more conservative configuration (Table 7.6), where the saturated counters have a higher threshold, a higher penalty for mispredictions, and a lower initial value. With this parameters, we want to verify the effects of a more restrictive confidence policy to RST. A more restrictive confidence mechanism may avoid mispredictions, but at the same time it may also reduce the number of correct predictions.

Table 7.6: Confidence configuration for counter 4096 7 7 7 1 0

Parameter	Value
Entries	4096
Bits per entry	3
Maximum value (saturation)	7
Threshold for prediction	7
Correct prediction increment	1
Misprediction decrement	7
Initial value	0

7.5.1 Speedup

Figure 7.17 presents speedups for RST over the baseline architecture.

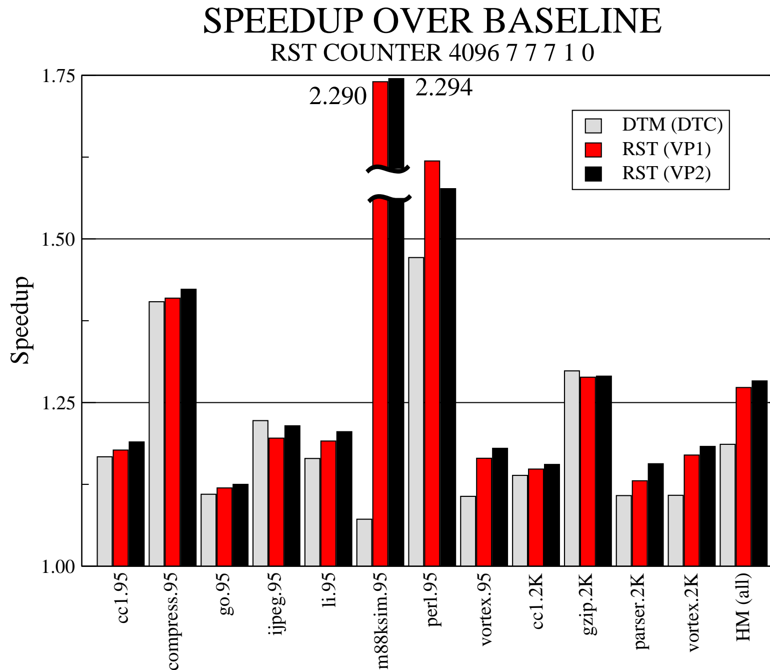

Figure 7.17: Speedup over baseline, counter 4096 7 7 7 1 0

The performance trends are similar as those for the 3 3 3 1 3 confidence configuration, but the overall performance is smaller. While the 3 3 3 1 3 configuration is able to achieve speedups of 1.32 over the baseline architecture, the current configuration can obtain only 1.28. This configuration is very conservative, as the counter must reach the value 7 before a trace can be speculated, and a mispeculation resets the counter to zero. This leads to a situation where in average only 8% of committed instructions come from speculatively reused traces, while in 3 3 3 1 3 they are 14% (for VP2).

As the 3 3 3 1 3 setup requires less bits for each counter and produces better performance, we conclude it is better for the RST architecture with 19 stages.

7.6 Confidence table sizes

The objective of the current experiment is to verify the number of entries in the confidence tables required for good performance in RST. We fixed the number of bits and thresholds for the confidence in *3 3 3 1 3* and varied the number of entries, and then we present the variation in speedup for 8192, 4096, 1024, 512, 256, 128, and 64 entries in the confidence table.

7.6.1 Speedup

Figure 7.18 presents the average speedup (harmonic mean) for RST over the baseline architecture when the number of entries in the confidence table is changed from

8192 to 64 entries. The graph shows results for RST predicting at most one (VP1) and two (VP2) inputs.

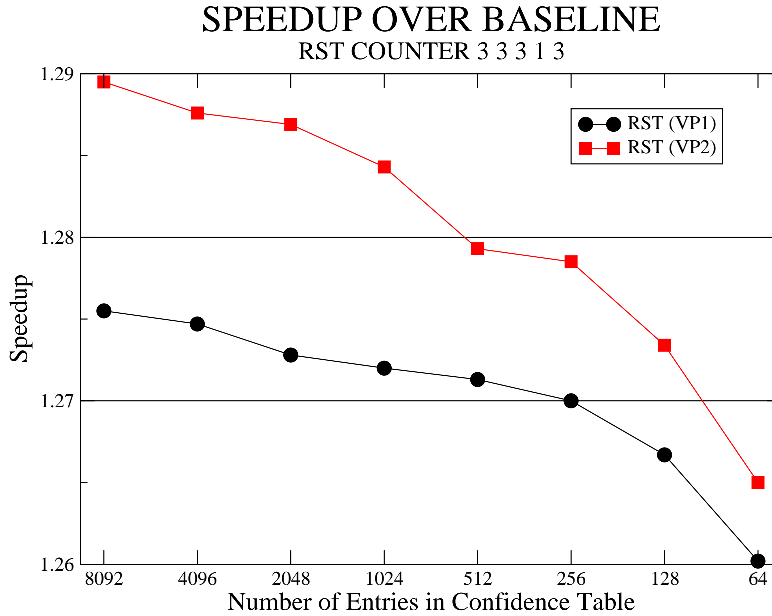

Figure 7.18: Speedup over baseline architecture varying confidence table sizes, RST 3 3 3 1 3

The performance trends are similar for VP1 and VP2, with speedups decreasing when the table sizes are reduced. This behavior for the confidence configuration *3 3 3 1 3* shows that address aliasing deteriorates the reliability of confidence estimation in this case. But in both cases speedups are better than DTM, which could achieve a speedup of 1.18 over the baseline architecture (not shown in the graph).

Another interesting trend here is that speedups for VP2 are more affected by reducing confidence table size than VP1. There is a more significant decrease in performance from 1024 to 512 entries for VP2, and then speedups for RST VP2 are less than 1.01. The gain in speedup obtained from predicting two instead of one input is just 0.003, while for larger tables this difference is around 0.010.

Therefore, reducing the number of entries in the confidence tables tends to reduce the performance gains from predicting more than one input, and then the costs of adding the hardware necessary for the test of an extra predicted value for some traces may not be the best option.

7.6.2 Speedups for a 6-stage pipeline

In this measurement, we verify performance of RST when the pipeline configuration resembles the original *sim-outorder*'s pipeline for the sake of reference and comparison to other works. The remaining configurations are not changed though. Variations on the number of stages of the superpipeline are more discussed in the previous Chapter.

Figure 7.19 presents the speedups over the baseline architecture with six stages for both RST and DTM. In the case of a shorter pipeline, less prune to large mis-peculation penalties, RST is able to outperform or have the same performance as DTM in all simulated benchmarks. The average speedup (HM) over the baseline is 1.21 for DTM and 1.28 for RST, similar to the ones found in deeper pipelines.

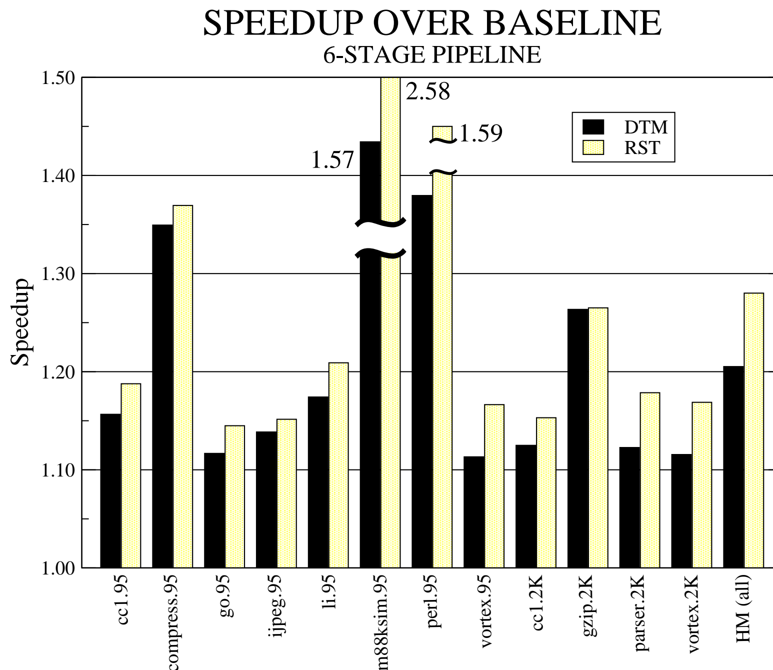

Figure 7.19: Speedup over the baseline with 6-stage pipeline, Memo_Table_T 512 in4 out4 4-assoc VP2, Memo_Table_G 2048 4-assoc

Thus, RST can also provide performance gains for shorter pipelines in a similar way that it increases performance for superpipelined architectures.

7.7 RST with stride prediction and last-value prediction

In this experiment, we verify the effects of stride prediction (SAZEIDES; SMITH, 1997; WANG; FRANKLIN, 1997a) in RST. The mechanism detects strides between consecutive traces and, if it is detected, then only the first trace is kept, but with additional information about the stride. After that, the inputs for that trace may be predicted as a function of the stride.

We simulated a hybrid of stride prediction and last-value prediction with two different confidence mechanisms (Tables 7.3 and 7.6), but we present only results with the *4096 3 3 3 1 3* configuration because they are very similar in all aspects and the chosen configuration presents slightly better performance.

7.7.1 Speedup

Figure 7.20 presents the speedups for stride-aware RST over DTM. The benchmark *m88ksim.95* once again presents the greatest performance improvements,

reaching a speedup of 3.63 over DTM when predicting 2 inputs at most. The benchmarks *ijpeg.95* and *gzip.2K* present performances slightly smaller than DTM (less than 2%), what may be related to the type of workload these benchmarks represent (compression of images or general data). In (WANG; FRANKLIN, 1997a), the *compress* benchmark from SPEC92int presented a large amount of mispredictions when a hybrid of strided prediction and last-value prediction (similar to what is employed in this experiment for RST) was used.

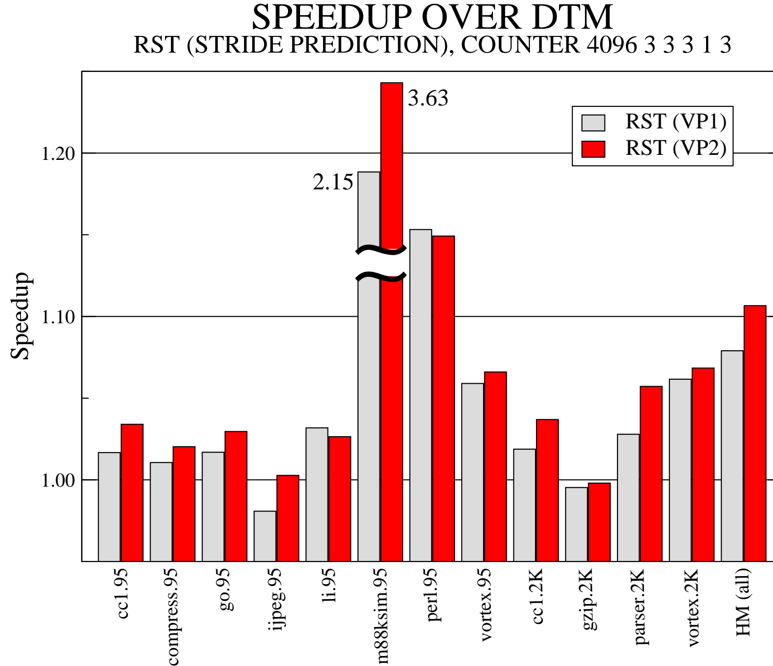

Figure 7.20: Speedups of strided RST 4096 3 3 3 1 3 over DTM

In average (harmonic mean), RST is able to reach speedups of 1.08 (VP1) and 1.10 (VP2) over DTM. Comparing with the baseline architecture, RST achieves speedups of 1.28 (VP1) and 1.32 (VP2) with the same configurations. Comparing to non-strided speculative trace reuse, we have a small average increase in performance, with an average (HM) speedup of 1.32 against 1.29 for RST VP2 over the baseline architecture.

Stride prediction combined with last value prediction can be used to improve performance in RST, but some benchmarks may not adapt well, presenting performance losses. But as we show in these results, the average of all speedups still points to RST for better performance than non-speculative trace reuse.

7.8 RST with confidence by trace

Another possible configuration of confidence is to add confidence bits to the entries in Memo_Table_T. In this setup, there is no separated confidence tables, and the confidence information is obtained together with the entries from Memo_Table_T in the RS1 stage. This reduces the requirements for confidence in terms of chip area to store the bits, as the Memo_Table_T size is usually smaller

than the confidence tables we have simulated until now. On the other hand, it increases the number of accesses to Memo_Table_T for the updates of the confidence counters, and it completely separates the information for the same PC address in different traces.

For the following results, we pick the best results for RST VP1 and RST VP2 among the simulated combinations (DTC or RTC with any of *7 7 7 1 0*, *3 3 3 1 3*, or *3 3 3 1 0*). Most of the other results could not achieve performance better or even equal to DTM.

7.8.1 Speedup

Figure 7.21 displays the speedups for RST (DTC) over DTM for the confidence configured by trace with counters *3 3 3 1 3*.

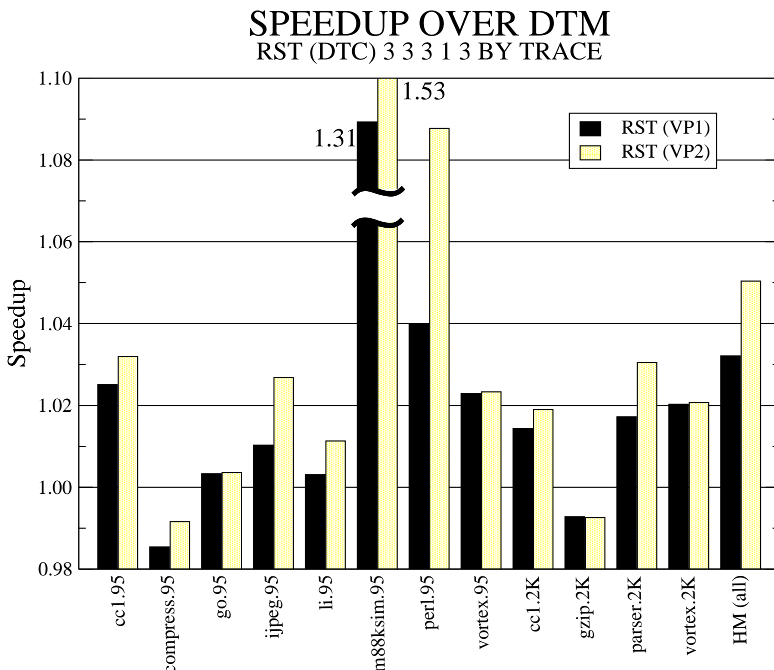

Figure 7.21: Speedup over DTM, RST (DTC) VP1 and VP2, confidence by trace 3 3 3 1 3

The graph shows that, for most benchmarks, this confidence configuration allows RST to achieve better performance than DTM, with an average speedup of 1.03 for VP1 and 1.05 for VP2 (harmonic means). As for other confidence configurations, *gzip.2K* presents worse performance in RST than in DTM; on the other hand, *ijpeg.95* achieves at least small speedups over DTM, differently from other confidence configurations where its performance is not better in RST than in DTM.

The benchmark *compress.95* also presents reduced performance for this configuration, which points us to a trend of reduced performances in the case of programs whose main task consists on some kind of data or image compression.

Even presenting speedups over DTM, this confidence setup produces performance below what was achieved with separated counter tables, thus we do not suggest this mechanism for this architecture configuration.

7.9 Comparison to alternatives

In the following experiments, we directly compare our technique to two alternatives: *(i)* doubling the first-level caches for a similar chip area use; and *(ii)* applying only instruction reuse with twice as much entries in Memo_Table_G.

7.9.1 Speedups over a doubled first-level cache

Figure 7.22 compares speedups over the baseline architecture for DTM, RST VP2 with the counter 4096 3 3 3 1 3 confidence mechanism, and the alternative of using twice as much first level cache as the baseline architecture for the same on-chip storage area spent on memoization tables. As we have dimensioned Memo_Table_T in about 24 KB, and Memo_Table_G in about 32 KB, they are together using roughly the same as the increase in cache sizes (from 32 KB to 64 KB each one).

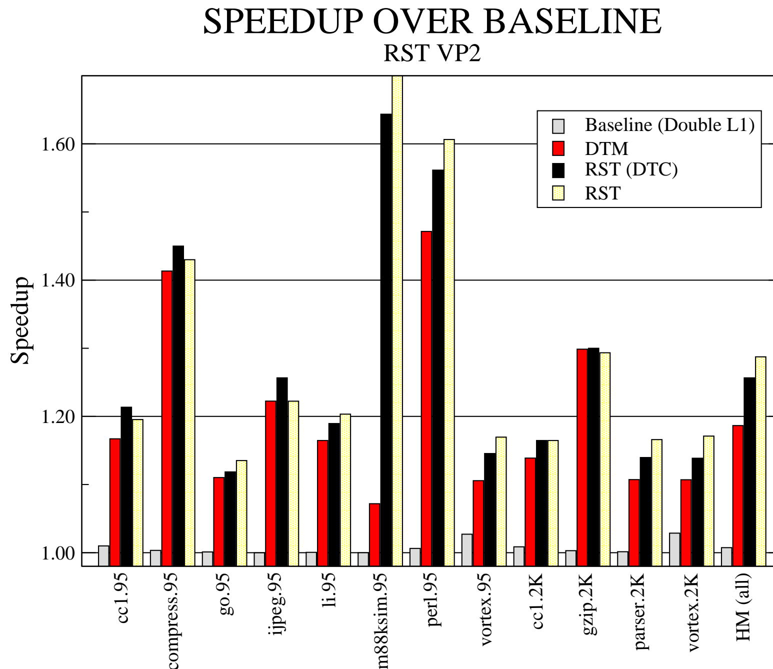

Figure 7.22: Comparison of speedups over baseline for a doubled-l1 cache architecture and DTM/RST architectures

In this case, increasing the first-level caches (both data and instruction caches) does not increase performance in a significant way, with a speedup of less than 1.01 over the baseline architecture (harmonic mean). On the other hand, RST is able to obtain average speedups of 1.26 (DTC) and 1.29 (RTC). These results can be explained by the small miss rates already observed for the benchmarks in the baseline configuration. For the instruction cache, this rate is about 0.5% (arithmetic mean) for baseline and 0.25% for the doubled-cache configuration; for the data cache, the rates are 1.35% for the baseline and 0.95% for the doubled-cache configuration.

Besides the small gains obtained by increasing first-level caches, there is also the related issue of increased access latencies for caches when their sizes are doubled. Thus, RST stands as a very competitive way of expending on-chip resources to increase performance.

7.9.2 Speedups over an instruction reuse technique

The next experiment aims to show how RST compares to approximately the same hardware in reuse tables against an instruction reuse technique. Figure 7.23 shows the speedups for RST and DTM over the instruction reuse technique with twice as much entries in Memo_Table_G.

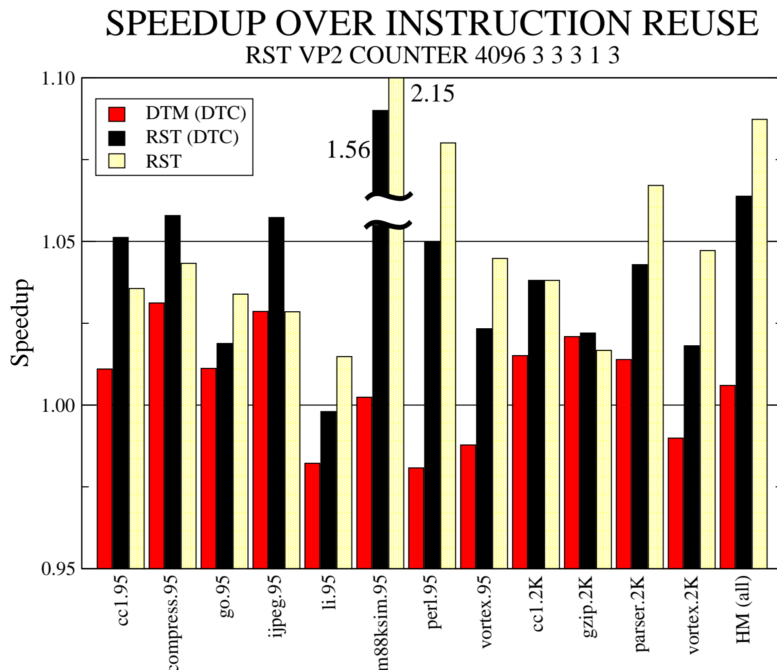

Figure 7.23: Speedup over a instruction reuse architecture, RST VP2

Besides for the benchmark *li.95* and RST (DTC), RST is able to obtain performance improvements over instruction reuse in all cases. DTM provides speedups in some cases and in average (HM) it produces a speedup of 1.005 over the instruction reuse with doubled tables. Thus, for the deep pipeline and current table configurations, DTM does not show much improvement over simple instruction reuse. On the other hand, RST presents speedups of 1.08 over instruction reuse (HM), hence showing a much better performance for a similar implementation cost.

7.9.3 Other comparisons

In this subsection, we briefly compare RST to results obtained by related works where we do not have the ways for simulating the mechanisms for the same pipeline and resource configurations. The reader should bear in mind that the configurations are different, as well as the benchmarks. Therefore, these numbers must be taken in account just as a gross comparison of capabilities for the different mechanisms. The mechanisms are described in Chapter 2.

For the speculative multi-threading scheme described in (WU; CHEN; FANG, 2001), the performance improvements were calculated as a speedup from 1.25 (for only reuse) to 1.40 over an IA-64 baseline architecture without reuse or value prediction, but it has the additional problem of requiring compiler support to mark regions to be reused. It requires also more hardware than RST and the baseline

architecture is very different from our superscalar, superpipeline approach.

Combining instruction reuse and value prediction in a six-stage pipeline (LIAO; SHIEH, 2002) provided a speedup of 5% over the baseline architecture. Another approach implemented in a six-stage pipeline (HUANG; CHOI; LILJA, 1999) combines both SRC and a hybrid value predictor to exploit both value reuse and value prediction achieved speedups of 10% in a 16-wide, 6-stage superscalar architecture and roughly 128 KB of storage for the CDP.

RST is able to achieve performance improvements of 28% over the baseline for a six-stage pipeline configuration, therefore outperforming the former two techniques at a first glance.

7.10 Summary

In the current Chapter, we presented and analyzed results for a feasible RST architecture with different confidence policies to determine which predictions should be pursued. Our results can be summarized as follows:

- Results for a RST architecture without confidence mechanisms, where we showed the need for either confidence mechanisms or other ways to reduce misprediction penalties and achieve speedups;

- Results for an oracle confidence mechanism, determining the speedup upper bounds for RST in the constrained configuration as 1.42 over the baseline (for VP2, RTC);

- Experiments with different confidence policies based on information already available from Memo_Table_T, showing that they were not enough to provide speedups over DTM;

- Results for three confidence mechanisms based on saturated counters, where we determined that RST could achieve speedups of 1.29 over the baseline and 1.08 over DTM;

- A study showing that the inclusion of path information in the confidence mechanism did not help to improve performance;

- Experiments with different sizes of confidence tables, showing that a reduced number of entries reduced performance and also the difference between VP1 and VP2;

- A study of stride-aware RST with realistic confidence, with small performance improvements over RST, obtaining a speedup of 1.32 over the baseline against 1.29 for RST without stride trace creation;

- A comparison to alternatives of expending the same chip area, where RST outperformed both doubling first-level caches and instruction reuse with more than twice as much memoization table than RST.

8 CONCLUSION AND FUTURE WORK

"At any rate I'll never go there again!"

Lewis Carrol, "Alice's Adventure In Wonderland"

In this Chapter, we show our final remarks and future work for this thesis. Section 8.1 presents the thesis summary. Conclusions for the limits study of Chapter 6 are discussed in Section 8.2. Section 8.3 presents the conclusions for the results shown in Chapter 7. Our main contributions are presented in Section 8.4. Finally, future work is discussed in Section 8.5.

8.1 Thesis summary

In this thesis, we presented a new approach to combine redundancy and predictability, called **Reuse through Speculation on Traces** (RST). We first introduced the problem to be solved (traces that were not reused because of inputs that are not ready), then our technique called RST (predicting inputs that are not ready for those traces), an implementation for superscalar processors (over the DTM infrastructure), detailed simulation in a deep pipeline architecture, and finally a comparison of our results against results obtained with alternative mechanisms (mainly non-speculative trace reuse).

RST is a speculative trace reuse framework, and can be used in a variety of processor architectures to increase performance by exploiting both redundancy and predictability – value reuse and prediction – without requiring excessive extra hardware when compared to non-speculative reuse techniques as DTM.

In this work, we proposed an implementation of RST over a superscalar, superpipelined architecture, detailing how the pipeline stages for RST would interact with the instruction pipeline. We studied implementation issues that might arise for the current superscalar architectures when RST is employed, such as extra pressure on the register file and how speculative reuse might affect the semantics of register renaming, presenting possible solutions for these problems.

To validate our work, we implemented a detailed superscalar architecture simulator to compare a set of benchmarks against both a superscalar baseline architecture without reuse and an architecture with trace reuse and similar configuration. We also test a scheme combining last-value prediction and stride prediction in a feasible way to speculatively construct traces to be reused, based on the differences between subsequent traces.

From our simulations, we were able to characterize the behavior of speculative trace reuse when many implementation issues are considered, as table associativity, memory hierarchies, confidence table sizes, and pipeline configurations. As some of those issues have not been studied before for trace reuse in deep pipelines, we also presented them in this thesis.

In our experiments, RST reached average speedups (harmonic means) of 1.29 over a baseline architecture without reuse or prediction, and 1.09 over an architecture with non-speculative trace reuse when realistic configurations and confidence mechanisms are simulated. RST was able to outperform alternative schemes of doubling the first-level caches and only reusing instructions for using the same on-chip area that was necessary to implement the reuse tables of RST's configuration.

8.2 Conclusions from the limits study

From our limits study presented on Chapter 6, we could draw many conclusions about the dynamics and characteristics of speculative trace reuse.

In our simulations, RST presented average speedups (harmonic means) of 1.85 over an architecture without reuse and 1.42 over DTM when perfect confidence and large memoization tables were considered. For this configuration, only the benchmark *gzip.2K* showed speedups of less than 1.15 over DTM, thus we expect that benchmarks with the same characteristics (data compression) may show less redundancy and predictability and will not have the same gains in RST than programs with other characteristics. In other experiments, the benchmark *compress.95* also presented the same trend, confirming our observations.

For most benchmarks, RST could reuse more instructions than DTM, because of its speculative trace reuse. However, we showed that the rate of reused instructions did not directly present an accurate understanding of how performance is increased by RST. In some cases, a smaller percent of reused instructions that were committed resulted in a larger speedup. Therefore, not only the number of instructions contained in the traces but also other characteristics have a weight on the performance gains that can be obtained by reusing traces.

Our results showed that the RTC policy improved the average length of reused traces, and that speculatively reused traces were longer than the regularly reused traces in average. The same comparison for DTM between DTC and RTC policies for trace construction taught us that the RTC policy depends on speculation of the input scopes to provide performance gains.

Variations on the associativity of Memo_Table_T affected performance for RST, but for DTM the difference between fully-associative and direct-mapped tables did not produce much worse results. Aliasing of different entries might cause higher associativities to perform worse for some benchmarks in some configurations. Variations on the number of entries reduced average performance for all architectures, as expected.

RST presented its better average performance (harmonic mean) between 3 and 1 inputs in the input context. For the DTC policy, performance decreased significantly from 3 to 1 inputs, while the more speculative RTC policy allowed RST to achieve similar performance with 1, 2, or 3 inputs. For the output context, best results were achieved for RST (RTC) when 2 outputs were used, and 3 outputs for RST (DTC). As for the number of inputs that might be predicted, our study showed

that 3 or 2 inputs presented the best results while simplifying the hardware for the misprediction tests in RS3.

From our study of different pipeline depths, we discovered that reducing issue or the writeback depth are the most effective way to improve performance for RST for all the tested pipeline depth decreases. We also showed that RTC presented better performance improvements than DTC for RST.

The limitations on performance for wider pipelines were not imposed by the lack of functional units but by all the limitations on memory access, control dependencies and true data dependencies. Increasing the number of functional units showed to be a poor alternative to increase performance in our study.

Stride trace creation has the potential for further improving performance, with speedups about 0.02 above RST without stride identification, but in the current implementation these gains were not as high as expected because of the many cycles between the beginning of a stride sequence, the stride identification, and the stride reuse.

Our results with memory reuse showed that RST without memory reuse has the potential for better speedups than DTM with memory reuse (DTMm). As memory reuse greatly increases hardware complexity, this was another point in favor of RST. If memory reuse is really important for a given architecture and workload, then we suggest to use the less speculative trace creation policy, DTC.

We concluded that variations on cache configuration affect RST in the same way they affect DTM or the baseline architecture without reuse.

Although DTM performed not as well as expected for some benchmarks in comparison to simple instruction reuse in our experiments, we believe that it is possible to obtain more performance from it even in the current baseline architecture by better balancing resources.

8.3 Conclusions from the RST architecture study

From the studies for RST without confidence mechanisms, we learned that RST needs either a mechanism to reduce misprediction rates or a scheme to reduce misspeculation penalties in order to achieve significant speedups. Hence, we studied confidence mechanisms as an easy to implement and tested way to control misprediction rates.

Increasing the number of inputs that can be predicted in traces increased both hardware complexity and misprediction rates. But when confidence mechanisms were used, predicting at most 2 inputs allowed RST to obtain better performance than when only 1 input could be predicted.

The average misprediction penalty for speculatively reused traces in RST without confidence was about 10 cycles for the 19-stages pipeline configuration. For this misprediction penalty and misprediction rates around 60%, the only benchmark in the simulated workload that was able to obtain relevant speedups was *m88ksim.95*.

For the constrained architecture with a perfect confidence mechanism, RST could achieve speedups (harmonic means) of 1.19 over DTM and 1.42 over the baseline architecture without reuse.

Our experiments with simple, stateless confidence mechanism depicted in Section 7.3 did not present speedups over regular trace reuse, thus showing that more complex confidence mechanisms were necessary in order to obtain considerable per-

formance improvements from speculative trace reuse.

With a confidence mechanism based on saturated counters with 4096 entries and only one level, RST provides a speedup of 1.07 over DTM when predicting one value, and 1.09 when predicting two values at most (harmonic means) for the small price in hardware of adding the RS3 stage to verify predictions and the confidence table. For the case of RST VP2, the speedup over the baseline architecture without reuse was 1.29 (harmonic mean). RST was able to have more committed instructions being part of reused traces in many cases, and even in some cases that it reused a smaller part of the committed instructions it still could obtain performance speedups, because of better trace characteristics.

We verified that even though confidence reliability is smaller for VP2 than for VP1, RST still provided better performance improvements when it predicted more inputs, as other speculative trace reuse opportunities existed for this setup.

Although we were able to achieve misprediction rates around 5% when confidence was used, we still had some benchmarks that were not able to obtain speedups over DTM. An example is the benchmark *gzip.2K*, which had misprediction rates of less than 2% for VP1 but whose misprediction penalties were distributed towards larger penalties than the other benchmarks: in 50% of the cases, it was 80 cycles or more, thus each mispeculation having a large impact on performance.

When comparing stride trace creation to speculative trace reuse, we determined that stride-aware RST was able to increase performance from 1.29 to 1.32 over the baseline architecture when predicting 2 inputs at most. This was a small increase in performance, and the extra hardware cost necessary to identify strides may not pay off.

Comparing two different counter-based confidence configurations, we concluded that the 4096 3 3 3 1 3 setup depicted in Table 7.3 was better than the 4096 7 7 7 1 0 described in Table 7.6 both in terms of hardware and performance. We also showed that adding path information to our confidence mechanism did not improve neither decreased performance in a significant way.

Some benchmarks directed at data and image compression had performance issues in RST depending on the confidence configurations, as for *compress.95* in Section 7.8 (confidence counters organized by trace in Memo_Table_T) or for *ijpeg.95* and *gzip.2K* in Section 7.4.

8.4 Contributions

In this thesis, we presented **Reuse through Speculation on Traces** (RST), a speculative trace reuse framework that can be adapted to different processor architectures. RST provides a way for reusing traces with inputs that are not ready during the reuse test, thus addressing the large percent of redundant traces not reused by regular trace reuse.

Subsequent to the definition of RST as a framework for speculative trace reuse, we developed the modifications necessary in a superscalar, superpipelined architecture to accommodate RST and discuss implementation issues.

From the superscalar implementation, we extracted and discussed results for both the performance limits when many configuration aspects are considered, and an actual implementation with non-perfect confidence mechanisms.

Besides the characterization and study of speculative trace reuse, we designed

simulation tools for superscalar, superpipelined architectures that can be also used for other studies that not RST.

We published and presented the following papers in the subject of this thesis:

- PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G. Predicting Trace Inputs with Dynamic Trace Memoization: determining speedup upper bounds. **IEEE TCCA Newsletter**, [S.l.], Oct. 2001 (also presented in the Work in Progress Workshop, PACT'2001).

- PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G.; CHILDERS, B. R.; SOFFA, M. L. Improving Performance through Speculative Trace Reuse. In: CADERNOS DE INFORMÁTICA – PROCESSAMENTO PARALELO E DISTRIBUÍDO NA INFORMÁTICA/UFRGS, 2003, Porto Alegre. **Anais...** Porto Alegre: Instituto de Informática–UFRGS, 2003. v.3, n.1, p.139–144.

- PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G.; COSTA, A. T. da; CHILDERS, B. R.; SOFFA, M. L. The Limits of Speculative Trace Reuse on Deeply Pipelined Processors. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 15., 2003, São Paulo. **Proceedings...** São Paulo: IEEE, 2003. p.36–44.

8.5 Future works

Even with all the studies that we did over RST's characteristics and the detailed simulation of many architectural issues, we still leave many details and improvements for further research.

Compiler and profiler support could be added to help the architecture to identify predictable traces or to better organize instructions to generate more redundant and longer traces.

Many issues with memory reuse and variations of the technique are not approached by this work, thus they can be targeted by future studies. In the current thesis, we only studied prediction of register inputs. Another possibility that can be evaluated later is the prediction of load values when memory reuse is allowed.

More advanced confidence techniques and mechanisms to select predictable traces among the ones present in Memo_Table_T can also be improved, as we only touched the simple confidence mechanisms described by previous works (CALDER; REINMAN; TULLSEN, 1999). Variations of parameters like table sizes, confidence counter thresholds, ways of indexing the confidence tables, and multiple-level tables may be used to tune confidence performance.

Another possible development is to speculatively reuse instructions from Memo_Table_G, instead of only speculatively reusing traces from Memo_Table_T. This approach has the same advantage of having the values to be predicted already available. This feature can also be implemented by allowing traces with only one instruction too.

System calls have proved to be a very important reason for traces being finished. Research on how they could be included in the reuse domain, thus allowing for longer traces, may further improve RST's performance.

There are still many interesting studies to be done for RST. For example, how branch prediction affects RST's performance, or how a Return Address Stack could be integrated into the architecture.

Finally, RST could use a dual core or a multi-threaded design to reduce misspeculation penalties by having a speculative and a non-speculative thread running in parallel like other speculative reuse schemes (WU; CHEN; FANG, 2001).

9 RESUMO DA TESE E SUAS CONTRIBUIÇÕES

"De jeito nenhum voltarei lá!"

Lewis Carrol, "Aventuras de Alice na Terra das Maravilhas"

Neste Capítulo[1], o resumo da tese e suas principais contribuições são apresentados. O resumo da tese é apresentado na Seção 9.1. A análise dos limites do reuso especulativo de *traces* do Capítulo 6 é discutida na Seção 9.2. Por sua vez, a Seção 9.3 apresenta o estudo de uma implementação de RST em uma arquitetura superescalar, a partir dos resultados do Capítulo 7. As principais contribuições desta tese são apresentadas na Seção 9.4. Finalmente, os trabalhos futuros que podem ser derivados desta tese são detalhados na Seção 9.5.

9.1 Resumo da Tese

Nesta tese,uma nova abordagem combinando redundância e previsibilidade, chamada **Reuso através de Especulação de *Traces*** (RST), é apresentada. Inicialmente, foi apresentado o problema (*traces* que não eram reusados por causa de entradas que não haviam sido calculadas ainda), depois a técnica RST (prevendo entradas que não estavam prontas para aqueles *traces*), uma implementação sobre processadores superescalares (utilizando a infra-estrutura de DTM), simulações detalhadas em uma arquitetura com *pipeline* de instruções profundo, e finalmente uma comparação dos resultados com outros obtidos com mecanismos alternativos (especialmente, reuso não-especulativo de *traces*).

RST é uma infra-estrutura para reuso especulativo de *traces* e pode ser usada em uma variedade de arquiteturas de processadores para aumentar desempenho pela exploração simultânea de previsibilidade e redundância – sem necessitar de aumento significativo de *hardware* quando comparado com técnicas de reuso não-especulativo de *traces*, como DTM.

Neste trabalho, propomos uma implementação de RST sobre uma arquitetura superescalar e *superpipeline*, detalhando como os estágios de RST se integrariam com o *pipeline* de instruções. Estudamos as questões de implementação que poderiam surgir para as arquiteturas superescalares da atualidade quando RST é utilizado, como pressão extra no banco de registradores e como o reuso especulativo poderia

[1] This Chapter is written in Portuguese as required by the Graduate Program

afetar a semântica de renomeação de registradores, apresentando possíveis soluções para estes problemas.

Para validar este trabalho, foi implementado um simulador de arquitetura superescalar detalhado para comparar um conjunto de *benchmarks* contra uma arquitetura superescalar base sem reuso e uma arquitetura com reuso de *traces* com configurações similares de recursos. Também foi testado um esquema combinando previsão de valores baseado nos últimos valores vistos e previsão de valores estimados a partir de diferenças entre *traces* consecutivos (*stride prediction*).

A partir das simulações, o comportamento do reuso especulativo de *traces* foi caracterizado quanto a diversas questões de implementação, tais como associatividade de tabelas, hierarquias de memória, tamanho de tabelas de confiança e configurações do *pipeline*. Como algumas dessas questões ainda não haviam sido estudadas anteriormente para reuso de *traces* em *pipelines* profundos, os mesmos também foram apresentados.

Nos experimentos realizados, RST apresentou *speedups* médios de 1.29 sobre a arquitetura base sem reuso ou previsão, e 1.09 sobre uma arquitetura com reuso de *traces* (médias harmônica) em simulações com configuração e mecanismos de confiança realistas. RST também obteve melhor desempenho que alternativas como utilizar o dobro de caches de primeiro nível e reuso de instruções com a mesma área de *chip* usada para a implementação da configuração de RST.

9.2 Análise do estudo de limites

A partir do estudo de limites apresentado no Capítulo 6, foram obtidas as seguintes conclusões a respeito da dinâmica e características do reuso especulativo de *traces*.

Nas simulações, RST apresentou *speedups* médios de 1.85 sobre uma arquitetura sem reuso e 1.42 sobre DTM (médias harmônicas) quando confiança perfeita e grandes tabelas de reuso foram consideradas. Para essa configuração, apenas o *benchmark gzip.2K* apresentou *speedups* de menos que 1.15 sobre DTM, desta forma esperamos que *benchmarks* com as mesmas características (compressão de dados) possam apresentar menos redundância e previsibilidade e desta forma não terão os mesmos ganhos de desempenho que programas com outras características. Em outros experimentos, o *benchmark compress.95* também mostrou a mesma tendência, confirmando nossas observações.

Para muitos *benchmarks*, RST reusou mais instruções que DTM por causa do reuso especulativo de *traces*. No entanto, a taxa de instruções reusadas não se traduz diretamente em incrementos de desempenho devido ao uso de RST. Em alguns casos, uma menor percentagem de instruções reusadas que foram graduadas resultou em um maior aumento de desempenho. Assim, não apenas o número de instruções reusadas que foram graduadas em *traces* mas também as demais características apresentam um peso nos ganhos de desempenho que podem ser obtidos pelo reuso de *traces*.

Os resultados também demostraram que a política RTC melhorou o comprimento médio de *traces* reusados e que *traces* reusados especulativamente apresentaram maior comprimento em média que os *traces* reusados não-especulativamente. A mesma comparação para DTM entre as políticas DTC e RTC para construção de *traces* mostraram que a política RTC depende da especulação nos contextos de

entrada para prover ganhos de desempenho.

Variações na associatividade da Memo_Table_T afetaram o desempenho de RST, mas para DTM a diferença entre tabelas completamente associativas e diretamente mapeadas não produziu resultados muito diferentes. Em certas situações, a colisão de diferentes *traces* para a mesma entrada podem causar piores resultados para maior associatividade para determinados *benchmarks* em algumas configurações. Variações no número de entradas reduziu o desempenho médio para todas as arquiteturas, como esperado.

RST apresentou melhores desempenhos (médias harmônicas) entre 3 e 1 entradas no contexto de entrada. Para a política DTC, o desempenho caiu significantemente de 3 para 1 entradas, enquanto que a política RTC, mais especulativa, permitiu que RST obtivesse desempenhos semelhantes com 1, 2 ou 3 entradas. Quanto aos contextos de saída, os melhores resultados foram obtidos com 2 saídas para RST (RTC) e 3 saídas para RST (DTC). Para o número de valores que podem ser previstos por *trace*, nossos estudos mostraram que 3 ou 2 previsões apresentam os melhores resultados ao mesmo tempo que simplificam o *hardware* para testes de previsões no estágio RS3.

O estudo de diferentes profundidades de *pipelines* mostrou que reduzindo o número de estágios para a delegação ou para a escrita de resultados são as melhores formas de melhorar o desempenho de RST de todas as configurações de *pipeline* testadas. Também mostrou-se que RTC apresenta melhor desempenhos para RST do que a política DTC nestes casos.

As limitações de desempenho para *pipelines* mais largos não foram impostas pela falta de unidades funcionais mas pelo conjunto de limitações, incluindo aí acessos à memória, dependências de controle e dependências de dados verdadeiras. Aumentar apenas o número de unidades funcionais neste caso não resolveria o problema de desempenho, de acordo com nossos estudos.

A criação de *traces* a partir das diferenças entre *traces* criados consecutivamente (*stride trace creation*) possui potencial para aumentar o desempenho, com incrementos nos *speedups* de 0.02 sobre RST sem identificação de *strides*; porém, na implementação atual esses ganhos não são tão altos como esperados devido ao grande número de ciclos entre identificação e efetivo reuso desses *traces*.

Resultados com reuso de acessos à memória demonstraram que RST sem reuso de memória possui potencial para melhores resultados do que DTM com reuso de memória. Como reuso de memória aumenta significantemente a complexidade do *hardware*, esse é outro ponto a favor de RST. Se reuso de memória é importante para uma dada arquitetura e aplicações, então sugerimos o uso da política de criação de *traces* menos especulativa (DTC).

Conclui-se que as variações nas configurações de cache afetam RST da mesma forma que afetam DTM ou a arquitetura base sem reuso.

Embora DTM não obtivesse o desempenho esperado para alguns *benchmarks* em comparação com reuso de instruções em nossos experimentos, acreditamos que seja possível melhorar os resultados através do balanceamento dos recursos da arquitetura.

9.3 Análise do estudo da arquitetura

A partir dos estudos com RST sem mecanismos de confiança, concluiu-se que RST necessita de um mecanismo para reduzir as taxas de erros nas previsões ou algum mecanismo para reduzir as penalidades em casos de previsões incorretas para obter bons resultados. Assim, mecanismos de confiança foram estudados como sendo uma forma simples e testada de controlar as taxas de erros nas previsões.

Aumentar o número de entradas que podem ser previstas em *traces* aumentou tanto a complexidade do *hardware* quanto as taxas de erros nas previsões. Porém, quando mecanismos de confiança foram utilizados, a previsão de até duas entradas permitiu a RST obter melhor desempenho que prevendo apenas uma entrada.

A penalidade média quando ocorreram erros na previsão para *traces* reusados especulativamente em RST sem confiança foi de aproximadamente 10 ciclos para a configuração com 19 estágios. Para essa penalidade e taxas de erros de cerca de 60%, o único *benchmark* que obteve bons *speedups* foi *m88ksim.95*.

Para a arquitetura com recursos limitados mas confiança perfeita, RST obteve *speedups* de 1.19 sobre DTM e 1.42 sobre a arquitetura base sem reuso (médias harmônicas).

Os experimentos com mecanismos simples de confiança sem estado, apresentados na Seção 7.3, não atingiram *speedups* sobre reuso de *traces*, desta forma mostrando que mecanismos mais complexos eram necessários para obter aumentos de desempenho relevantes.

Com um mecanismo de confiança baseado em contadores saturados com 4096 entradas e apenas um nível, RST obteve *speedup* de 1.07 sobre DTM prevendo um valor por *trace*, e 1.09 quando prevendo dois valores no máximo (médias harmônicas) pagando apenas o custo adicional do estágio RS3 para verificar previsões e a tabela de confiança. Para o caso de RST VP2, o *speedup* sobre a arquitetura base sem reuso foi de 1.29 (média harmônica). RST conseguiu ter mais instruções graduadas provenientes de *traces* em muitos casos, e mesmo em casos em que menos instruções graduadas foram reusadas ainda conseguiu melhorar o desempenho, por causa de melhores características dos *traces* reusados.

Também mostrou-se que mesmo que o mecanismo de confiança possua melhores taxas de acerto para VP1 que para VP2, ainda assim VP2 possui melhor desempenho, já que mais oportunidades de especulação existem neste caso.

Apesar de obter-se taxas de erros nas previsões de 5% quando mecanismos de confiança foram usados, alguns *benchmarks* não obtiveram *speedups* sobre DTM. Um exemplo é *gzip.2K*, que teve taxas de erros de menos de 2% para VP1; porém, esse *benchmark* possui maiores penalidades em casos de previsões incorretas, com 80 ciclos ou mais em 50% dos casos. Desta forma, cada previsão incorreta produz um grande impacto no desempenho.

A comparação de criação de *traces* a partir de *strides* com RST produziu uma melhora no *speedup* sobre a arquitetura base de 1.29 para 1.32 quando até dois valores eram previstos, uma pequena melhoria em desempenho que pode não justificar o aumento em complexidade do *hardware* devido ao reconhecimento dos *strides*.

Concluiu-se também que a configuração 4096 3 3 3 1 3 descrita na Tabela 7.3 produziu melhores resultados que a configuração 4096 7 7 7 1 0, descrita na Tabela 7.6, tanto em termos de desempenho quanto em custos estimados do *hardware*. A adição de informação sobre o fluxo de controle no mecanismo de confiança apresentou resultados semelhantes aos mecanismos sem essa informação, sendo portanto

desnecessária.

Alguns *benchmarks* voltados a compressão de dados e imagens apresentaram problemas em termos de desempenho, dependendo de configurações do mecanismo de confiança, como nos casos do *compress.95* na Seção 7.8 (contadores de confiança organizados por *trace* em Memo_Table_T) ou para *ijpeg.95* e *gzip.2K* na Seção 7.4.

9.4 Contribuições

Nesta tese, **Reuso através de Especulação em *Traces*** (RST) foi apresentado, uma infra-estrutura de reuso especulativo de *traces* que pode ser adaptada para diferentes arquiteturas de processadores. RST provê a possibilidade de reusar *traces* cujas entradas não estão prontas para o teste de reuso, desta forma endereçando a grande percentagem de *traces* redundantes que não são reusados por técnicas anteriores de reuso de *traces*.

Após definir RST como uma infra-estrutura para reuso especulativo de *traces*, as modificações necessárias para acomodar RST na implementação em uma arquitetura superescalar e *superpipelined* foram desenvolvidas.

Da implementação superescalar de RST, discutiu-se resultados para o estudo de limites com diversas variações de configuração, bem como para uma implementação com mecanismos de confiança realistas.

Além da caracterização e estudo do reuso especulativo de *traces*, ferramentas de simulação que podem ser usadas para outros estudos foram desenvolvidas.

Publicamos e apresentamos os seguintes artigos no assunto desta tese:

- PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G. Predicting Trace Inputs with Dynamic Trace Memoization: determining speedup upper bounds. **IEEE TCCA Newsletter**, [S.l.], Oct. 2001 (também apresentado no Work in Progress Workshop, PACT'2001).

- PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G.; CHILDERS, B. R.; SOFFA, M. L. Improving Performance through Speculative Trace Reuse. In: CADERNOS DE INFORMÁTICA – PROCESSAMENTO PARALELO E DISTRIBUÍDO NA INFORMÁTICA/UFRGS, 2003, Porto Alegre. **Anais...** Porto Alegre: Instituto de Informática–UFRGS, 2003. v.3, n.1, p.139–144.

- PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G.; COSTA, A. T. da; CHILDERS, B. R.; SOFFA, M. L. The Limits of Speculative Trace Reuse on Deeply Pipelined Processors. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 15., 2003, São Paulo. **Proceedings...** São Paulo: IEEE, 2003. p.36–44.

9.5 Trabalhos futuros

Mesmo com todos os estudos sobre características de RST e a simulação detalhada de diversas questões da arquitetura, ainda deixamos diversos detalhes e melhoramentos para trabalhos posteriores.

Suporte de compiladores e analisadores de perfis pode ser adicionado para ajudar na identificação de *traces* previsíveis ou para melhor organizar as instruções e gerar

traces mais longos e mais redundantes.

Diversas questões com reuso de memória e variações da técnica não foram atacadas nessa tese, podendo serem abordadas em estudos futuros. Neste trabalho, apenas foi estudada a previsão de valores de registradores. Outra possibilidade seria incluir a previsão de valores de acesso à memória quando reuso de memória é utilizado.

Técnicas mais avançadas de confiança e mecanismos para selecionar *traces* previsíveis entre todos os candidatos também podem ser melhorados, já que apenas utilizamos mecanismos simples descritos por trabalhos anteriores (CALDER; REINMAN; TULLSEN, 1999). Variações de parâmetros como tamanhos de tabelas, limites dos contadores, diferentes formas de endereçar a a tabela de confiança e tabelas de múltiplos níveis podem ser usadas para refinar a confiança.

Outro possível desenvolvimento é reusar especulativamente instruções da Memo_Table_G, ao invés de apenas permitir o reuso especulativo de *traces* da Memo_Table_T. Essa alternativa tem a mesma vantagem de já possuir os valores a serem previstos armazenados na tabela de reuso e pode também ser implementada através de *traces* com apenas uma instrução.

Chamadas ao sistema mostraram-se freqüentemente como o motivo para que *traces* fosse terminados. Estudos em como estas chamadas poderiam ser incluídas no domínio de reuso para aumentar os *traces* poderia levar a novas melhorias de desempenho em RST.

Há ainda muitos outros pontos interessantes a serem estudados, por exemplo a forma como a previsão de desvios afeta o desempenho de RST ou como uma pilha de endereços de retorno poderia ser integrada na arquitetura.

Finalmente, RST poderia usar um outro *core* ou um projeto com múltiplas *herdas* de execução para reduzir as penalidades provenientes de erros de previsão, com *herdas* especulativas e não-especulativas executando em paralelo, como outros mecanismos de reuso especulativo (WU; CHEN; FANG, 2001).

REFERENCES

ABELSON, H.; SUSSMAN, G. J. **Structure and Interpretation of Computer Programs**. New York: McGraw Hill, 1985.

ADVANCED MICRO DEVICES. **AMD Athlon Processor Architecture**. Sunnyvale: AMD, 2000. Available at: <http://www.amd.com/products/cpg/athlon/pdf/architecture_wp.pdf>. Acessed in: May 2001.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: an infrastructure for computer system modeling. **IEEE Computer**, Los Alamitos, v.35, n.2, p.59–67, Feb. 2002.

BODIK, R.; GUPTA, R.; SOFFA, M. L. Load-Reuse Analysis: design and evaluation. In: SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 1999. **Proceedings...** New York: ACM, 1999. p.64–76.

BURGER, D. C.; AUSTIN, T. M. **The SimpleScalar Tool Set, Version 2.0**. Madison: University of Wisconsin–Madison, 1997. Technical Report. (CS-TR-1997-1342).

CALDER, B.; GRUNWALD, D. Next Cache Line and Set Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 22., 1995, Santa Margherita Ligure. **Proceedings...** New York: ACM, 1995. p.287–296.

CALDER, B.; REINMAN, G.; TULLSEN, D. M. Selective Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 26., 1999, Atlanta. **Proceedings...** New York: ACM, 1999. p.64–74.

CHONG, F. **Source Distribution for a GCC 2.7.2.3 Compiler Capable of Generating SimpleScalar Binaries**. Davis: UC–Davis, 1999. Available at: <http://arch.cs.ucdavis.edu/RAD/gcc-2.7.2.3.ss.tar.gz>. Acessed in: Dec. 1999.

CITRON, D.; FEITELSON, D.; RUDOLPH, L. Accelerating Multimedia Processing by Implementing Memoing in Multiplication and Division Units. In: ANNUAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 8., 1998, San Jose. **Proceedings...** New York: ACM, 1998. p.252–261.

COSTA, A. T. da. **Exploiting Dynamically the Reuse of Traces in Processor Architecture Level**. 2001. Ph.D. Thesis — COPPE–UFRJ.

COSTA, A. T. da; FRANÇA, F. M. G. **The Reuse Potencial of Trace Memoization**. Rio de Janeiro: COPPE–UFRJ, 1999. Technical Report. (ES–498/99).

COSTA, A. T. da; FRANÇA, F. M. G.; CHAVES FILHO, E. M. Exploiting Reuse with Dynamic Trace Memoization: evaluating architectural issues. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 12., 2000, São Pedro. **Proceedings. . .** São Paulo: SBC, 2000. p.163–172.

COSTA, A. T. da; FRANÇA, F. M. G.; CHAVES FILHO, E. M. The Dynamic Trace Memoization Reuse Technique. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURE AND COMPILATION TECHNIQUES, 9., 2000, Philadelphia. **Proceedings. . .** Los Alamitos: IEEE Computer Society, 2000. p.92–99.

FERNANDES, E. S. T.; SANTOS, A. D. **Arquiteturas Super Escalares**: detecção e exploração do paralelismo de baixo nível. Gramado: Informática UFRGS, 1992.

FLYNN, M. J. **Computer Architecture**: pipelined and parallel processor design. Londres: Jones and Bartlett, 1995.

GABBAY, F.; MENDELSON, A. **Speculative Execution based on Value Prediction**. Israel: Technion–Israel Institute of Technology, 1996. Technical Report. (EE Dept. #1080).

GABBAY, F.; MENDELSON, A. Using Value Prediction to Increase the Power of Speculative Execution Hardware. **ACM Transactions on Computer Systems**, New York, v.16, n.3, p.234–270, 1998.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. **The Performance Potential of Data Value Reuse**. Barcelona: Universitat Politécnica de Catalunya, 1998.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. Trace-Level Reuse. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 28., 1999, Aizu-Wakamatsu. **Proceedings. . .** Los Alamitos: IEEE Computer Society, 1999. p.30–37.

GRUNWALD, D.; KLAUSER, A.; MANNER, S.; PLEZSKUN, A. Confidence Estimation for Speculation Control. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998, Barcelona. **Proceedings. . .** New York: ACM, 1998. p.122–131.

HARBISON, S. P. An Architectural Alternative to Optimizing Compilers. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 1982, Palo Alto. **Proceedings. . .** [S.l.: s.n.], 1982. p.57–65.

HEIL, T. H.; SMITH, J. E. **Selective Dual Path Execution**. Madison: University of Wisconsin, 1996. ECE Technical Report.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 3rd ed. San Francisco: Morgan Kaufmann, 2003.

HENNING, J. L. SPEC CPU2000: measuring CPU performance in the new millenium. **IEEE Computer**, Los Alamitos, v.33, n.7, p.28–35, July 2000.

HOREL, T.; LAUTERBACH, G. UltraSparc III: designing third generation 64-bit performance. **IEEE Micro**, Los Alamitos, v.19, n.2, May/June 1999.

HUANG, J.; CHOI, Y.; LILJA, D. **Improving Value Prediction by Exploiting Both Operand and Output Value Locality**. [S.l.]: Laboratory for Advanced Research in Computing Technology and Compilers, 1999. Technical Report. (Technical Report ARCTic 99-06).

HUANG, J.; LILJA, D. J. Exploiting Basic Block Value Locality with Block Reuse. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 5., 1999, Orlando. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.106–114.

HUANG, J.; LILJA, D. J. Extending Value Reuse to Basic Blocks with Compiler Support. **IEEE Transactions on Computers**, New York, v.49, n.4, p.331–347, Apr. 2000.

HUANG, J.; LILJA, D. J. Exploring Sub-Block Value Reuse for Superscalar Processors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 9., 2000, Philadelphia. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.100–110.

HWANG, K. **Advanced Computer Architecture**: parallelism, scalability, programmability. New York: McGraw-Hill, 1993.

INTEL. **Inside the NetBurst Micro-Architecture of the Intel Pentium 4 Processor**. Available at: <http://download.intel.com/pentium4/download/netburst.pdf>. Accessed in: Jan. 2001.

JOHNSON, M. **Superscalar Microprocessor Design**. Englewood Cliffs: Prentice Hall, 1991.

KESSLER, R. E. The Alpha 21264 Microprocessor. **IEEE Micro**, Los Alamitos, v.19, n.2, Mar./Apr. 1999.

KLAIBER, A. **The Technology Behind Crusoe Processors**. [S.l.: s.n.], 2000. Available at: <http://www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf>. Accessed in: May 2002.

KLAUSER, A.; GRUNWALD, D. Instruction Fetch Mechanisms for Multipath Execution Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 32., 1999, Haifa. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.38–49.

KLAUSER, A.; PAITHANKAR, A.; GRUNWALD, D. Selective Eager Execution on the PolyPath Architecture. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998, Barcelona. **Proceedings...** New York: ACM, 1998.

KOUSHIRO, T.; SATO, T.; ARITA, I. A Trace-level Value Predictor for Contrail Processors. **SIGARCH Comput. Archit. News**, [S.l.], v.31, n.3, p.42–47, 2003.

LAM, M. S.; WILSON, R. P. Limits of Control Flow on Parallelism. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992. **Proceedings...** New York: ACM, 1992. p.46–57.

LEE, J. K.; SMITH, A. Branch Prediction Strategies and Branch Target Buffer Design. **IEEE Computer**, Los Alamitos, v.17, n.1, p.6–22, 1984.

LIAO, C.-H.; SHIEH, J.-J. Exploiting Speculative Value Reuse Using Value Prediction. In: ASIA-PACIFIC CONFERENCE ON COMPUTER SYSTEMS ARCHITECTURE, 7., 2002. **Proceedings...** Melbourne: Australian Computer Society, 2002. p.101–108.

LIPASTI, M. H.; SHEN, J. P. Exceeding the Dataflow Limit via Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 29., 1996, Paris. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p.226–237.

LIPASTI, M. H.; WILKERSON, C. B.; SHEN, J. P. Value Locality and Load Value Prediction. **ACM SIGPLAN Notices**, New York, v.31, n.9, p.138–147, 1996.

MCFARLING, S. **Combining Branch Predictors**. Palo Alto: Western Labs, 1993. Technical Report. (DEC WRL TN–36).

MICHIE, D. Memo Functions and Machine Learning. **Nature**, [S.l.], v.1, n.218, p.19–22, Apr. 1968.

MIPS Technologies Inc. **MIPS R10000 Microprocessor User's Manual**. Mountain View: [s.n.], 1995.

MOLINA, C.; GONZÁLEZ, A.; TUBELLA, J. Dynamic Removal of Redundant Computations. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 13., 1999, Rhodes. **Proceedings...** New York: ACM, 1999. p.474–481.

NAKRA, T.; GUPTA, R.; SOFFA, M. L. Value Prediction in VLIW Machines. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 26., 1999, Atlanta. **Proceedings...** New York: ACM, 1999.

ÖNDER, S.; GUPTA, R. Load and Store Reuse Using Register File Contents. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 15., 2001, Sorrento, Italy. **Proceedings...** New York: ACM, 2001. p.289–302.

PARCERISA, J.-M.; GONZÁLEZ, A. Reducing Wire Delay Penalty through Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 33., 2000, Monterey. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.317–326.

PATEL, S. J.; FRIENDLY, D. H.; PATT, Y. N. **Critical Issues Regarding the Trace Cache Fetch Mechanism**. [S.l.]: University of Michigan, 1997. Tech Report. (CSE-TR-335-97).

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design**: the hardware/software interface. 2nd ed. San Francisco: Morgan Kaufmann, 1997.

PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G. Predicting Trace Inputs with Dynamic Trace Memoization: determining speedup upper bounds. **IEEE TCCA Newsletter**, [S.l.], Oct. 2001. Work presented in the Work in Progress Section of PACT'2001, 2001, Barcelona, Spain.

PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G.; CHILDERS, B. R.; SOFFA, M. L. Improving Performance through Speculative Trace Reuse. **Cadernos de Informática**, Porto Alegre, v.3, n.1, p.139–144, jun. 2003. Work presented in the 1. Workshop do Grupo de Processamento Paralelo e Distribuído, 2003, Porto Alegre, Brazil.

PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G.; COSTA, A. T. da. **Speculative Trace Reuse**. Porto Alegre: UFRGS, 2002. (RP–320).

PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G.; COSTA, A. T. da; CHILDERS, B. R.; SOFFA, M. L. The Limits of Speculative Trace Reuse on Deeply Pipelined Processors. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 15., 2003, São Paulo. **Proceedings...** São Paulo: IEEE, 2003. p.36–44.

POSTIFF, M. A.; GREENE, D. A.; THYSON, G. S.; MUDGE, T. N. The Limits of Instruction Level Parallelism in SPEC95 Applications. **Computer Architecture News**, [S.l.], v.217, n.1, p.31–34, 1999.

REBELLO, V. E. F. **Neurocom Project**. Brasília: CNPq, 1997. (Protem–II CC).

REINMAN, G.; CALDER, B. Predictive Techniques for Aggresive Load Speculation. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 31., 1998. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998.

RICHARDSON, S. E. **Caching Function Results**: faster arithmetic by avoiding unnecessary computation. California: Sun Microsystems Laboratories, 1992. Technical Report. (Tech Report SMLI TR-92-1).

DE ROSE, C. F. D.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre: Sagra Luzzatto, 2003. 152p.

ROTENBERG, E.; BENNETT, S.; SMITH, J. A Trace Cache Microarchitecture and Evaluation. **IEEE Transactions on Computers**, New York, v.48, n.2, p.111–120, Feb. 1999.

ROTENBERG, E.; JACOBSON, Q.; SAZEIDES, Y.; SMITH, J. Trace Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.138–148.

ROTENBERG, E.; SMITH, J. Control Independence in Trace Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 32., 1999, Haifa. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.4–15.

ROTH, A.; SOHI, G. S. Register Integration: a simple and efficient implementation of squash re-use. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 33., 2000, Monterey. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.223–234.

SANTOS, R. R. dos. **Um Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções**. 1997. M.Sc. Dissertation — CPGCC/UFRGS, Porto Alegre.

SANTOS, T. G. S. dos; PILLA, M. L.; SANTOS, R. R. dos; CHAVES FILHO, E. M.; BAMPI, S.; NAVAUX, P. O. A.; NEMIROVSKY, M. D. Resource Tuning in a Multipath Superscalar Architecture. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 11., 1999, Natal. **Proceedings...** Porto Alegre: SBC, 1999. p.27–34.

SATHE, R.; WANG, K.; FRANKLIN, M. Techniques for Performing Highly Accurate Data Value Prediction. **Microprocessors and Microsystems**, [S.l.], v.22, n.6, p.303–313, Nov. 1998.

SAZEIDES, Y.; SMITH, J. E. The Predictability of Data Values. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.248–258.

SHARANGPANI, H.; ARORA, K. Itanium Microprocessor Microarchitecture. **IEEE Micro**, Los Alamitos, v.20, n.5, p.24–43, Sept./Oct. 2000.

SMITH, J. E.; SOHI, G. S. The Microarchitecture of Superscalar Processors. **Proceeding of the IEEE**, New York, v.83, p.1609–1624, Dec. 1995.

SODANI, A. **Dynamic Instruction Reuse**. 2000. Ph.D. Dissertation – University of Wisconsin, Madison.

SODANI, A.; SOHI, G. S. Understanding the Differences Between Value Prediction and Instruction Reuse. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 31., 1998. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p.205–215.

SOHI, G. S.; BREACH, S.; VIJAYKUMAR, T. N. Multiscalar Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 22., 1995, Santa Margherita Ligure. **Proceedings...** New York: ACM, 1995.

SPEC – Standard Performance Evaluation Corporation. **SPEC CPU 95 Technical Manual**. Manassas, Virginia: SPEC Steering Committee, 1995.

STALINGS, W. **Computer Organization and Architecture**: designing for performance. 4th ed. Upper Saddle River: Prentice Hall, 1996.

TULLSEN, D. M.; SENG, J. S. Storageless Value Prediction using Prior Register Values. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 26., 1999, Atlanta. **Proceedings...** New York: ACM, 1999. p.270–281.

TUNE, E.; LIANG, D.; TULLSEN, D. M.; CALDER, B. Dynamic Prediction of Critical Path Instructions. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, 7., 2001, Monterey. **Proceedings...** Los Alamitos: IEEE Computer Society, 2001. p.185–195.

VIANA, L. M. F. A. **Dynamic Trace Memoization with Reuse of Memory Access Instructions**. 2002. M.Sc. Dissertation — COPPE–UFRJ, Rio de Janeiro.

WALL, D. W. **Limits of Instruction-Level Parallelism**. Palo Alto: Western Research Laboratory, 1993. (93/6).

WALLACE, S.; BAGHERZADEH, N. Multiple Branch and Block Prediction. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 3., 1997, San Antonio. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.94–103.

WANG, K.; FRANKLIN, M. Highly Accurate Data Value Prediction Using Hybrid Predictors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.281–290.

WANG, K.; FRANKLIN, M. Highly Accurate Data Value Prediction. In: HIGH-PERFORMANCE COMPUTING CONFERENCE, 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.358–363.

WU, Y.; CHEN, D.-Y.; FANG, J. Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 28., 2001, Göteborg, Sweden. **Proceedings...** New York: ACM, 2001. p.98–108.

YANG, J.; GUPTA, R. Load Redundancy Removal through Instruction Reuse. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 29., 2000, Toronto. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.61–68.

YEH, T.-Y.; PATT, Y. N. Two-Level Adaptive Training Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991, Alburquerque. **Proceedings...** Los Alamitos: IEEE Computer Society, 1991. p.51–61.

YEH, T.-Y.; PATT, Y. N. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 20., 1993, San Diego. **Proceedings...** New York: ACM, 1993. p.257–266.

APPENDIX A PUBLISHED PAPERS