ADDRESS PREDICTION AND RECOVERY MECHANISMS

Enric Morancho Llena

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona (Spain). May, 2002

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor per la Universistat Politècnica de Catalunya

ADDRESS PREDICTION AND RECOVERY MECHANISMS

Author: Enric Morancho Llena
Advisors: José María Llabería and Àngel Olivé

Approved, Thesis Committee:

....

....

....

....

....

ABSTRACT

Mitigating the effect of the large latency of load instructions is one of challenges of micro-processor designers. This thesis analyses one of the alternatives for tackling this problem: address prediction and speculative execution.

Several authors have noticed that the effective addresses computed by the load instructions are quite predictable. First of all, we study why this predictability appears; our study tries to detect the high-level language structures that are compiled into predictable load instructions. We also analyse the conventional address predictors in order to determine which address predictors are most appropriate for the typical applications.

Our study continues by proposing address predictors that use their storage structures more efficiently. Address predictors track history information of the load instructions; however, the requirements of the predictable instructions are different from the requirements of the unpredictable instructions. We then propose an organization of the prediction tables considering the existence of both kinds of instructions. We also show that there is a certain degree of redundancy in the prediction tables of the address predictors. We propose organizing the prediction tables in order to reduce this redundancy. These proposals allow us to reduce the area cost of the address predictors without impacting their performance.

After that, we evaluate the impact of address prediction on processor performance. Our evaluations assume that address prediction is used to start speculatively some memory accesses and to execute speculatively their dependent instructions. On a correct prediction, all the speculative work is considered as correct; on a misprediction, the speculative work must be discarded. Our study is focused on several aspects such as the interaction of address prediction and branch prediction, the implementation of verification mechanisms, the recovery mechanism on address mispredictions, and the influence of several processor parameters (the issue-queue size, the cache latency and the issue width) on the performance impact of address prediction.

Finally, we evaluate several recovery mechanisms for latency mispredictions. Latency prediction is a speculative technique used by the schedulers of some superscalar processors to deal with variable-latency instructions (for instance, load instructions). Our evaluations are focused on a conventional recovery mechanism for latency mispredictions and a new proposal. We also evaluate the proposed recovery mechanism in the scope of address prediction; we conclude that it represents a cost-effective alternative to the conventional recovery mechanisms used for address mispredictions.

AGRAÏMENTS

Aquesta tesi és el fruit de l'esforç desenvolupat durant un seguit d'anys. El dia on es va iniciar aquest treball sembla molt llunyà (realment ho és...) però finalment hem arribat a un punt final. Directa o indirectament, molta gent ha contribuït a la culminació d'aquesta tasca i des d'aquí vull fer palesa la meva gratitut.

Per començar, aquesta tesi no seria el que és sense els seus directors: en Josep Maria Llabería i l'Àngel Olivé. Treballo amb el Josep Maria des de que vaig entrar al departament i durant tot aquest temps he pogut copçar la seva sapiència, minuciositat, entusiasme i dedicació. Més endavant, l'Àngel va incorporar-se a aquest treball i els seus comentaris, esforços i idees han contribuït a millorar aquesta feina. En resum, els he d'agraïr la paciència que han tingut amb mi i tot el temps que m'han dedicat.

Els meus companys de depatx han hagut de soportar-me durant tot aquest temps. Vull explicitar el meu agraïment a tots ells, especialment al Luisma, que ha estat el meu company al llarg d'una gran part de la durada d'aquesta tesi.

Al llarg de tot aquest temps he tingut certa relació, que ens alguns casos ha arribat a amistat, amb moltes persones que estan o van estar relacionades amb la universitat. Esperant no deixar-me a ningú, menciono a l'Agustín, Anna, Cristina, Chema, David, Dolors, Enric, Enrique, Fermín, Jesús, Joan Carles, Joan Manel, Jordi's, Josep, Josep Ramon, Juanjo, Manel, Marisa, Marta, Miguel, Montse, Nacho, Neus, Pau, Pedro, Pepe, Roger, Sergi, Susana,, Toni's i Xavi. A tots ells, gràcies per les mostres de companyerisme que heu tingut amb mi durant tot aquest temps.

No puc deixar de mencionar el bon servei rebut per part de tot el personal d'administració de sistemes del departament i del Cepba; especialment de l'Uri ja que s'haurà atipat de veure les meves simulacions saturant les cues de les seves màquines. Tanmateix, aquest reconeixement és extensiu a tot el personal de secretaria del departament.

Per fer aquest treball he tingut al meu abast una sèrie de recursos. Molts d'aquests recursos són propis de la línia d'investigació "Computació i altes prestacions", dirigida tot aquest temps pel Mateo Valero. Altres recursos són propis del departament que, durant aquest temps ha estat dirigit pel Juanjo Navarro, l'Olga Casals i el Mateo Valero.

Afortunadament, la vida no s'acaba al departament i més enllà d'aquestes parets he pogut comptar en tot moment amb els meus amics. Tot i que sigui un d'aquests manits tòpics, he de reconèixer que han omplert de significat la paraula amistat. Vull mencionar explícitament al Cristhian i al Lluís, amics meus des de fa molt de temps.

La meva família també ha estat al meu costat ja que sempre ha confiant en mi i ha recolçat les meves decisions. Especialment, vull agrair als meus pares i al meu germà haver esperat estoicament l'arribada d'aquest dia sense escatimar la seva estimació i aguantant pacientment les meves manies.

Finalment, vull agrair a la Norma haver il.luminat aquests darrers anys amb la seva fe, energia i entrega. Espero que continuï il.luminant-me per molt de temps.

This work was supported by the Ministry of Education of Spain under contracts TIC-95-0429, TIC-98-0511 and TIC-2001-0995, and by the CEPBA (European Center for Parallelism of Barcelona).

TABLE OF CONTENTS

Abstract i

Agraïments iii

Table of Contents v

List of Figures ix

List of Tables xiv

1 Introduction 1

- 1.1 Motivation ... 1
- 1.2 Related works ... 4
  - 1.2.1 Address Predictors ... 4
  - 1.2.2 Dynamic memory disambiguation ... 5
  - 1.2.3 Hardware prefetching ... 6
  - 1.2.4 Start memory accesses early in the pipeline ... 6
  - 1.2.5 Recovery mechanisms for address mispredictions ... 7
- 1.3 Thesis overview ... 8
- 1.4 Thesis organization ... 9

2 Address Predictability 11

- 2.1 Introduction ... 11
- 2.2 Sources of address predictability ... 12
  - 2.2.1 Global addresses ... 13
  - 2.2.2 Vector accesses ... 14
  - 2.2.3 Switch statements ... 14
  - 2.2.4 Stack accesses ... 15
  - 2.2.5 System constants ... 16
  - 2.2.6 Sequential accesses to vector structures ... 17
  - 2.2.7 Repeated access sequences ... 18
- 2.3 Address predictors ... 18
  - 2.3.1 Last-Address Predictor (LAP) ... 18
  - 2.3.2 Stride Address predictor (SAP) ... 21
  - 2.3.3 Context Address Predictor (CAP) ... 22
  - 2.3.4 Hybrid Address Predictor (HAP) ... 24

2.4 Performance comparison of address predictors 25
2.4.1 Metric description 25
2.4.2 Results 25
2.5 Related works 30
2.6 Chapter summary 32

3 Reducing Prediction-Table Size by Filtering 33

3.1 Introduction 34
3.2 Main idea of the filtering strategies 35
3.3 Benchmark characterizations 36
3.3.1 Address-predictability distributions 36
3.3.2 Working-set size of static load instructions 39
3.3.3 Hit-bursts distributions 40
3.4 Filtering by means of a continuous classification 42
3.4.1 Dynamic classification: <*N, k*> classifying mechanism 43
3.4.2 Filtering by Continuous Classification Last-Address Predictor (CLAP) 44
3.4.3 Evaluation results 46
3.5 Filtering by means of a discrete classification 51
3.5.1 Discrete classification 51
3.5.2 Filtering by Discrete Classification Last-Address Predictor (DLAP) 54
3.5.3 Performance evaluation 57
3.5.3.1 Captured Predictability 57
3.5.3.2 Comparison of filtering strategies 60
3.5.3.3 Influence of partial tagging on performance 62
3.5.3.4 Accuracy 63
3.5.3.5 Influence of associativity on performance 64
3.6 Related Works 66
3.7 Conclusions 67
3.8 Detailed results 67
3.8.1 Similarity 67
3.8.2 Predictability captured by the CLAP 68
3.8.3 Accuracy of the CLAP 69
3.8.4 Predictability captured by the DLAP 70
3.8.5 Accuracy of the DLAP 70

4 Reducing Prediction-Table Size by Narrowing 73

4.1 Introduction 74
4.2 Locality analysis of the effective addresses 74
4.3 Two-Level Last-Address Predictor (TLAP) 77
4.3.1 TLAP design 77
4.3.2 HAT implementation issues 80
4.4 Evaluation of the TLAP 84
4.4.1 Area cost of the predictors 84
4.4.2 Captured address predictability 85
4.4.3 Accuracy 86
4.5 Filtering by Discrete Classification Two-Level Last-Address Predictor (DTLAP) 87
4.6 Related works 89
4.7 Conclusions 90

4.8 Detailed results ... 91
4.8.1 Predictability captured by the TLAP ... 91
4.8.2 Predictability captured by the DTLAP ... 91

5 Evaluation of Address Prediction **93**
5.1 Introduction ... 94
5.2 Processor model ... 96
5.2.1 Baseline processors ... 96
5.2.2 Address-speculative processors ... 99
5.3 Implicit verification on commit ... 106
5.4 Verification through the verification-flow graph ... 108
5.4.1 Information flow for verifying/invalidating instructions ... 109
5.4.2 Issue Queue ... 110
5.4.3 Verification Issue Queue ... 112
5.4.4 Pipeline timing considerations on the communication between IQ, VIQ and check devices ... 116
5.5 Serial verification through the verification-flow graph ... 119
5.5.1 Non-speculative address check ... 119
5.5.2 Speculative address check ... 121
5.5.2.1 Speculative address check and concurrent execution of several instances of a dynamic load instruction ... 123
5.5.3 Processor Performance ... 125
5.6 Enhanced verification-flow graph ... 127
5.6.1 Processor Performance ... 131
5.7 Processors with the issue queue decoupled from the reorder buffer ... 132
5.7.1 Influence of issue-queue size on the performance of baseline processors ... 133
5.7.2 Serial verification ... 134
5.7.3 Enhanced verification ... 135
5.7.4 Influence of increasing address-predictor latency on the performance of address-speculative processors ... 137
5.8 Delayed speculative issue ... 138
5.8.1 Delayed speculative issue versus baseline processors ... 139
5.9 Non-delayed versus delayed speculative issue ... 140
5.9.1 Effect of address mispredictions on performance ... 142
5.10 Related works ... 143
5.11 Conclusions ... 145
5.12 Detailed results ... 146
5.12.1 4-way processors ... 146
5.12.2 8-way processors ... 147

6 Recovery Mechanisms for Latency Mispredictions **149**
6.1 Introduction ... 150
6.2 Background ... 151
6.2.1 Recovery on a mispredicted latency ... 153
6.2.2 Base Pipeline and Issue Queue ... 155
6.3 Keeping issued instructions in the issue queue ... 155
6.3.1 Non-selective nullification ... 156
6.3.2 Selective nullification ... 157

6.4 Keeping issued instructions in the recovery buffer 158
6.4.1 Recovery-Buffer organization 160
6.4.2 Recovery buffer with selective nullification 163
6.4.2.1 Re-issue optimization 164
6.4.3 Recovery buffer with non-selective nullification 166
6.4.4 Effect of wrong-path instructions on recovery-buffer structures 166
6.5 Evaluation 167
6.5.1 Evaluation environment 167
6.5.2 Results 168
6.5.2.1 Integer benchmarks 168
6.5.2.2 Floating-point benchmarks 170
6.6 Conclusions 171
6.7 Detailed results 173
6.7.1 Integer benchmarks 173
6.7.2 Floating-point benchmarks 176

7 Conclusions **179**
7.1 Summary 179
7.2 Future directions 181

A Evaluation Environment **183**
A.1 Evaluation tools 183
A.1.1 Instrumented executables 184
A.1.2 Processor simulators 184
A.2 Benchmarks 186
A.2.1 Benchmark description 186
A.2.2 Simulation intervals for cycle-by-cycle simulations 187

References **193**

LIST OF FIGURES

Chapter 1

Figure 1.1 Execution of a load instruction and a dependent instruction (without address prediction and with address prediction and speculative execution) 3

Chapter 2

Figure 2.1 Address predictability accessing global variables... 13
Figure 2.2 Address predictability accessing global variables (optimized GAT accesses) 13
Figure 2.3 Address predictability accessing vectors.. 14
Figure 2.4 Address predictability in switch statements 15
Figure 2.5 Address predictability accessing the stack 16
Figure 2.6 Address predictability produced by system constants 16
Figure 2.7 Address predictability accessing vector structures sequentially 17
Figure 2.8 Address predictability accessing repeatedly a linked list... 18
Figure 2.9 State diagram of the 2-bit confidence estimator 19
Figure 2.10 Diagram of the Last-Address Predictor.. 20
Figure 2.11 Pseudo-code of the Last-Address Predictor 20
Figure 2.12 Diagram of the Stride Address Predictor 21
Figure 2.13 Pseudo-code of the Stride Address Predictor... 22
Figure 2.14 Diagram of a Context Address Predictor 23
Figure 2.15 Pseudo-code of the Context Address Predictor.. 24
Figure 2.16 Area cost versus captured predictability of the LAP, SAP and CAP in SPEC95-INT benchmarks 28
Figure 2.17 Area cost versus captured predictability of the LAP, SAP and CAP in SPEC95-FP benchmarks 29

Chapter 3

Figure 3.1 Main idea of the filtering strategies 36
Figure 3.2 Static (a) and dynamic (b) load-instruction distribution according to their address predictability by a Last-Address Predictor 37
Figure 3.3 Predictability captured by the LAP without confidence estimation (left bar) and by the LAP with confidence estimation (right bar) distributed in address-predictability ranges.. 38
Figure 3.4 Miss rates in the LAP (using direct-mapped and fully associative AT's) 39

Figure 3.5 Cumulative dynamic-load-instruction distributions according to hit-burst lengths ... 41
Figure 3.6 Average similarity of <*N, 0*> and <*N, 3*> classifying mechanisms ... 44
Figure 3.7 Diagram of the Filtering by Continuous Classification Last-Address Predictor (CLAP) ... 45
Figure 3.8 Decision tables applied by the replacement algorithms of the LAP and by the CLAP ... 45
Figure 3.9 Pseudo-code of the Filtering by Continuous Classification Last-Address Predictor (CLAP) ... 46
Figure 3.10 Miss rate in the LAP and the CLAP in SPEC95-INT benchmarks (AT's are fully associative with LRU replacement policy, CT's are unbounded) ... 47
Figure 3.11 Miss rate in the LAP and the CLAP in all benchmarks (direct-mapped AT and CT) ... 48
Figure 3.12 Predictability captured by the LAP and by the CLAP in large and extra-large benchmarks ... 50
Figure 3.13 Accuracy achieved by the LAP and by the CLAP ... 51
Figure 3.14 Diagram of the Filtering by Discrete Classification Last-Address Predictor (DLAP) ... 54
Figure 3.15 Decision tables applied by the replacement algorithms of the CLAP and the DLAP ... 55
Figure 3.16 Pseudo-code of the Filtering by Discrete Classification Last-Address Predictor (DLAP) ... 56
Figure 3.17 Predictability captured by the LAP, the LAP+CT and the DLAP in large and extra-large benchmarks ... 58
Figure 3.18 Decision tables applied by the replacement algorithms of the DLAP and by the Conservative Predictor ... 60
Figure 3.19 Predictability captured by the LAP, the Conservative Predictor and DLAP in benchmarks m88ksim and go using direct-mapped AT's ... 61
Figure 3.20 Accuracy of the LAP versus the number of tagbits in benchmark gcc (circles show the saturation point for every number of AT entries) ... 63
Figure 3.21 Accuracy of the LAP, the Conservative Predictor, and the DLAP in large and extra-large benchmarks using direct-mapped AT's. ... 64
Figure 3.22 Average predictability captured in Class-A benchmarks (m88ksim and perl) and in Class-B benchmarks (go, gcc and vortex) for several associative mappings ... 65
Figure 3.23 Predictability captured by the LAP, the Conservative Predictor and the DLAP in benchmarks m88ksim and go using associative tables ... 65
Figure 3.24 Similarity in each benchmark of <*N, 0*> and <*N, 3*> classifying mechanisms ... 68
Figure 3.25 Predictability captured by the LAP and by the CLAP in small and medium benchmarks ... 69
Figure 3.26 Accuracy of the LAP and the CLAP in small and medium benchmarks ... 69
Figure 3.27 Predictability captured by the DLAP in small and medium benchmarks ... 70
Figure 3.28 Accuracy of the LAP, the Conservative Predictor, and the DLAP in small and medium benchmarks using direct-mapped AT's ... 71

Chapter 4

Figure 4.1 Dynamic-load-instruction distributions according to their match depth; the dynamic load instructions have been filtered out according to their confidence information........ 76
Figure 4.2 Diagram of the Two-Level Address Storage (TLAS) organization........ 77
Figure 4.3 Diagram of the Two-Level Last-Address Predictor (TLAP)........ 78
Figure 4.4 Pseudo-code of the Two-Level Last-Address Predictor (TLAP)........ 80
Figure 4.5 Effect on captured predictability of filtering-out HAT allocations without managing empty HAT entries (without) and managing empty HAT entries (with) in *gcc* and *compress*........ *82*
Figure 4.6 Effect on captured predictability of filtering-out HAT allocations by managing empty HAT entries........ 82
Figure 4.7 Influence of dynamic chunk selection on the accuracy of the TLAP........ 83
Figure 4.8 Influence of the algorithm used by the replacement policy of HAT on the predictability captured by the TLAP........ 84
Figure 4.9 Predictability captured by the LAP and the TLAP in large and extralarge benchmarks........ 85
Figure 4.10 Accuracy of the LAP and the TLAP........ 87
Figure 4.11 Diagram of the Filtering by Discrete Classification Two-Level Last-Address Predictor (DTLAP)........ 88
Figure 4.12 Predictability captured by the DTLAP and the LAP in large and extra-large benchmarks........ 89
Figure 4.13 Predictability captured by the LAP and the TLAP in small and medium benchmarks........ 91
Figure 4.14 Predictability captured by the LAP and the DTLAP in small and medium benchmarks........ 92

Chapter 5

Figure 5.1 Block organization of the baseline processor (hollow boxes and solid lines) and the address-speculative processor (all boxes and lines)........ 96
Figure 5.2 Processor pipeline assumed in this work........ 97
Figure 5.3 Execution of a load instruction........ 98
Figure 5.4 Execution of a load instruction assuming: a) correctly predicted load instruction and b) incorrectly predicted load instruction........ 100
Figure 5.5 Execution of a predicted load instruction and the speculative issue of a dependent instruction: a) non delaying the issuing of the dependent instruction and b) delaying it........ 101
Figure 5.6 IPC versus data-cache latency in baseline and in address-speculative processors with implicit verification mechanism (for benchmarks *go* and *perl*)........ 106
Figure 5.7 IPC versus data-cache latency in baseline and in address-speculative processors with implicit verification mechanism........ 107
Figure 5.8 External inputs and communication between the Issue Queue and the Verification Issue Queue........ 108
Figure 5.9 Information flow for verifying/invalidating instructions after the address check of a predicted load instruction........ 110

Figure 5.10 Dependence-matrix structure of the Issue Queue ... 111
Figure 5.11 Issue-queue structure ... 112
Figure 5.12 Components of the Verification Issue Queue ... 113
Figure 5.13 Matrix structure of the Verification Issue Queue (VIQ) ... 114
Figure 5.14 Structure of the Verification Issue-Queue ... 116
Figure 5.15 Communication between VIQ and IQ devices ... 117
Figure 5.16 Qualitative cycle-time distribution of IQ and VIQ components ... 117
Figure 5.17 Qualitative cycle-time distribution of address checking and VIQ components ... 119
Figure 5.18 Example of a non-speculative address check ... 120
Figure 5.19 Validation of dependent predicted load instructions ... 121
Figure 5.20 Example with one/several concurrent instances of a load instruction: a) one instance, b) several instances ... 124
Figure 5.21 Example of the serial verification ... 126
Figure 5.22 IPC versus data-cache latency in baseline and in address-speculative processors with implicit and serial verification mechanism ... 127
Figure 5.23 Example of the serial verification of large chains of dependent instructions ... 128
Figure 5.24 Computation of the Collapsed Graph Table ... 130
Figure 5.25 Computation of the Matrix structure of the VIQ ... 130
Figure 5.26 Example of the enhanced verification ... 131
Figure 5.27 IPC versus data-cache latency on baseline processors and address-speculative processors with serial and enhanced verification ... 132
Figure 5.28 IPC versus issue-queue size on baseline processors ... 134
Figure 5.29 IPC versus issue-queue size on baseline processors and address-speculative processors with serial verification ... 135
Figure 5.30 IPC versus cache latency on baseline processors and address-speculative processors with serial verification ... 135
Figure 5.31 IPC versus issue-queue size on address-speculative processors with serial and enhanced verification ... 136
Figure 5.32 IPC versus address-predictor latency and data-cache latency on address-speculative processors with non-delayed speculative issue ... 137
Figure 5.33 Execution of a predicted load instruction and the speculative issue of a dependent instruction: a) non delaying the issuing of the dependent instruction and b) delaying it ... 139
Figure 5.34 IPC versus issue-queue size in baseline processors and address-speculative processors with delayed speculative issue ... 140
Figure 5.35 Performance of baseline and address-speculative processors ... 141
Figure 5.36 IPC versus data-cache latency on address-speculative processors with real and oracle address predictors ... 143
Figure 5.37 Performance of 4-way baseline and address-speculative processors on each SPEC95-INT benchmark ... 147
Figure 5.38 Performance of 8-way baseline and address-speculative processors on each SPEC95-INT benchmark ... 148

Chapter 6

Figure 6.1 Pipeline designs without latency prediction ... 152
Figure 6.2 Instruction flow after issuing a load instruction (LD) with predicted hit latency ... 153

Figure 6.3 Example of an instruction flow with a latency-mispredicted instruction 154
Figure 6.4 Base processor pipeline 155
Figure 6.5 Issue-Queue structure. 156
Figure 6.6 Placement of the recovery buffer in the processor pipeline.................................... 158
Figure 6.7 Interface between the issue queue and the recovery buffer.................................. 159
Figure 6.8 Recovery-Buffer organization 160
Figure 6.9 Re-issue logic of the recovery buffer 162
Figure 6.10 Recovery buffer with selective nullification 164
Figure 6.11 Instruction-flow example using the recovery buffer with selective nullification and the re-issue optimization 165
Figure 6.12 Instruction-flow example of the recovery buffer with non-selective nullification.. 166
Figure 6.13 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks...................... 169
Figure 6.14 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks...................... 169
Figure 6.15 Influence of both the recovery mechanism and the verification delay on the performance of processors executing floating-point benchmarks............ 170
Figure 6.16 Influence of both the recovery mechanism and the issue-queue size on the performance of processors executing floating-point benchmarks............ 171
Figure 6.17 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks *go*, *m88ksim*, *gcc* and *compress*.. *174*
Figure 6.18 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks *li*, *ijpeg*, *perl* and *vortex* *175*
Figure 6.19 Influence of both the recovery mechanism and the verification delay on the performance of processors executing floating-point benchmarks *tomcatv*, *swim*, *hydro2d* and *mgrid*... *176*
Figure 6.20 Influence of both the recovery mechanism and the verification delay on the performance of processors executing floating-point benchmarks *applu*, *turb3d*, *fpppp* and *wave5*... *177*

Appendix A

Figure A.1 Evolution of the predictability captured by a last-address predictor during the execution of the SPEC95-INT benchmarks.. 189
Figure A.2 Evolution of the predictability captured by a stride predictor during the execution of the SPEC95-FP benchmarks 191

LIST OF TABLES

Chapter 2

Table 2.1 Predictability captured by the address predictors with unbounded prediction tables and no confidence mechanism in SPEC95 benchmarks 26
Table 2.2 Captured predictability and accuracy of the address predictors with unbounded prediction tables and confidence mechanism in SPEC95 benchmarks 27

Chapter 3

Table 3.1 Benchmark classification according to their working-set sizes of static load instructions. 40
Table 3.2 Percentage of dynamic load instructions related to hit-burst of length zero, up to four and up to ten executions, and miss rates in some LAP configurations and benchmarks 42

Chapter 5

Table 5.1 Baseline processor configurations used in this thesis 98
Table 5.2 Captured predictability and accuracy obtained by the address predictor on SPEC95-INT benchmarks 101
Table 5.3 Decision logic outputs for instructions that non perform address check 115
Table 5.4 Decision-logic results using non-speculative address check 121
Table 5.5 Outputs of the Decision logic with speculative address checks 123
Table 5.6 Outputs of the Decision logic with concurrent executions of several instances of a load instruction 125

Chapter 6

Table 6.1 Miss rates for the SPEC95 benchmarks in a direct-mapped 64K first-level data cache. 168

Appendix A

Table A.1 SPEC95 benchmark description 186
Table A.2 SPEC95 benchmark characterization 187
Table A.3 Selected simulation intervals for SPEC95-INT benchmarks 190
Table A.4 Selected simulation intervals for SPEC95-FP benchmarks 192

1 INTRODUCTION

This chapter introduces this thesis. First, we outline the motivation of this work. After that, we describe some works related to address prediction. Next, we overview the work developed in this thesis. Finally, we detail the organization of the whole document.

1.1 Motivation

High-performance processors try to extract parallelism from sequential programs in order to execute several instructions at the same time; this ability allows a reduction in the execution time of the programs. To extract parallelism, processors exploit the instruction-level parallelism (ILP) available in a program. Unfortunately, the amount of ILP is limited because the dependences between the instructions of a program determine an execution order.

Dependences between instructions are classified into three types: data dependences, name dependences and control-flow dependences. Instruction *j* data-depends on instruction *i* if the result produced by instruction *i* is consumed (directly or indirectly) by instruction *j*. A name

dependence appears between two instructions that update the same storage element (a register or a memory location). Control-flow dependences are produced by instructions that explicitly determine the next instruction to be executed.

Computer architects use the average number of instructions committed per cycle (IPC) to measure the performance of a processor. The IPC is one of the factors that determine the execution time of the program; the execution time depends directly on both the number of committed instructions and the clock cycle time, and also depends in the inverse proportion on the IPC obtained by the processor.

Several hardware techniques have been proposed to increase the IPC obtained by the processors. For instance, register renaming, prediction techniques and speculative execution. Register renaming [Sima00] eliminates name dependences related to register locations. Prediction techniques are used to advance some processor actions in order to reduce the latency of some instructions. However, these actions do not update the processor state (registers, memory), so when a misprediction occurs, no recovery action is needed. An example of prediction techniques is prefetching the next cache line on a cache miss [Smit82].

Speculative execution can be considered as a further step in techniques that advance some processor actions. It is supported by some kind of prediction (for instance, a branch outcome, the independence of a load instruction on a previous store instruction); after that, some instructions dependent on the prediction are executed speculatively. Later, the prediction is verified. On a correct prediction, the speculatively executed instructions can commit their results into the architectural state of the processor; on a misprediction, a recovery mechanism must discard the effects of the misprediction.

Currently, high-performance processors use prediction and speculative execution to increase the IPC in several scenarios: executing control-flow dependent instructions (all processors), disambiguating the memory reference stream (PA-8000 [Hunt95], Alpha 21264 [Kess99]), scheduling instructions dependent on a load instruction before knowing its latency (Alpha 21264 [Kess99], Sparc64 V [Dief99]).

However, to the best of our knowledge, no processor applies speculative execution to a more generic case of data dependences. To apply speculative execution on data dependences, a value predictor must predict the result of an operation. While a branch-direction predictor performs a binary prediction for every conditional branch instruction (taken branch or not taken branch), in the scope of data dependences a value predictor must perform 32-bit (or 64-bit) predictions. In spite of the huge prediction space, several works [LWS96][Gabb96] have shown that most results are highly predictable. For instance, the results of the arithmetic instructions, memory accesses and even the effective addresses computed by load and store instructions. This highly predictability can be explained by the fact that some instructions present a repeatable behaviour that can be learnt by a finite-state machine.

Although all register-writing instructions generate data dependences, the dependences produced by load instructions are specially critical due to: a) the tendency of load instructions to start time-critical chains of dependent instructions, b) the large frequency of load instructions (around 25% of committed instructions) and c) their large latency (latency on data-cache hits is two or three cycles on current processors). Moreover, load latency is expected to grow in next-generation processors. Our evaluations show that this latency increment will significantly degrade processor performance. For instance, assuming a 4-way processor with a 3-cycle load-use latency (Section 5.2.1), executing integer benchmarks, increasing load-latency one cycle degrades processor performance about 6%; larger increments (two and three cycles) produce larger degradations (around 11% and 16% respectively).

As the continually increasing load latency is one of the key challenges on high-performance processors, lots of researchers have proposed techniques that try to reduce or tolerate load latency. The highly predictability of the effective addresses computed by the load instructions offers an opportunity to face the challenge: address prediction can be used to speculatively start memory accesses early in the pipeline and to speculatively execute their dependent instructions. That is, the application of speculative execution to execute instructions dependent on a load instruction before computing its effective address.

Figure 1.1 depicts the execution of a load instruction and a dependent instruction fetched on the same cycle. It compares an execution without address prediction (Figure 1.1-a) versus an execution with address prediction and speculative execution (Figure 1.1-b). The non shaded pipeline stages are related to the non-speculative execution of the instructions: fetch (F), decode and register renaming (D / Ren), issue queue (IQ), register read (R), execution (Exe), effective-address computation (@) and memory access (m). The shaded stages are related to the speculative execution: address prediction (Pred), speculative issue of the memory access (LAQ), memory access (m) and prediction checking (check). We have assumed that first-level data-cache latency is two cycles and an address-prediction latency is one cycle.

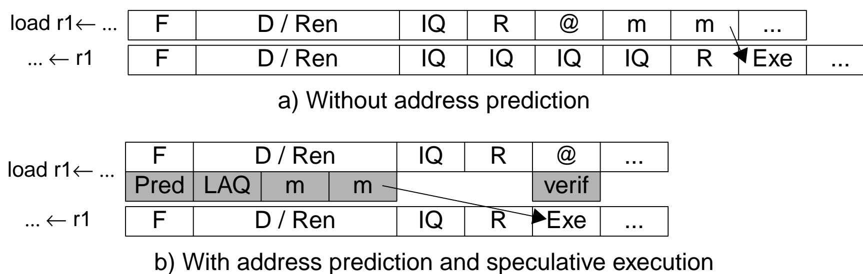


Figure 1.1 Execution of a load instruction and a dependent instruction (without address prediction and with address prediction and speculative execution)

In the first case, the instruction dependent on the load instruction must wait in the issue-queue stage until the load instruction is issued and data is available. In the second case, if the speculative memory access has finished when the dependent instruction is inserted in the

issue queue, the dependent instruction can be issued on the next cycle. Assuming a correct prediction, address prediction and speculative execution allows us to obtain the result of the dependent instruction several cycles in advance with respect to the non-speculative execution.

One of the purposes of this thesis is to explore the ability of address prediction and speculative execution to mitigate the effect of the load latency.

1.2 Related works

Some researchers have suggested the use of hardware mechanisms that predict the effective addresses that will be computed by load and store instructions. This section presents a brief description of works related to address prediction. In addition, in each chapter of this document, the works most closely related to that chapter are discussed.

- First, we describe works that introduce address predictors. Most of these predictors can also be used in the scope of value prediction; that is, predicting the result of register-writing instructions. This thesis, however, is focused on address prediction.

- After that, we relate works that make use of address prediction. We have classified these works into the following types: dynamic disambiguation of memory references, hardware prefetching and starting memory accesses early in the pipeline.

- Finally, we report recovery mechanisms for address mispredictions and speculative execution.

1.2.1 Address Predictors

In this subsection we describe the main classes of address predictors: Last-Address Predictor, Stride Address Predictor, Context Address Predictor and Hybrid Address Predictor.

These address predictors track the effective addresses computed by each memory-access instruction in a data structure named *prediction table*. Using this information and a prediction model, the address predictor can predict the effective address that will be computed on the next execution of the memory-access instruction. In order to reduce the amount of wrong predictions, all the address predictors use confidence estimators that prevent some predictions that will probably be wrong.

The most simple address predictor assumes that a memory instruction will compute the same effective address that it did in its previous instance. This predictor is named Last-Address Predictor and has been used in the scope of starting the memory accesses early ([EiVa93]).

The Stride Address Predictor assumes that every memory instruction generates effective addresses in arithmetic progression. This predictor has been used in the scopes of hardware

prefetching ([ChBa95]), dynamic disambiguation of memory references ([GoGo97a]) and early starting of the memory accesses ([EiVa93]).

The Context Address Predictor assumes that memory instructions compute repeatedly some sequences of effective addresses. Several works have proposed Context Address Predictors in the scope of hardware prefetching ([ChPu97]) and in the scope of early starting the memory accesses ([BJR+99]).

The Hybrid Address Predictor combines several of the previous predictors and tries to use the address predictor best suited to each memory instruction. This predictor has been used in the scope of early starting the memory accesses ([BMP+98], [BJR+99]).

A deeper analysis of address predictors is provided in Chapter 2.

1.2.2 Dynamic memory disambiguation

Dynamic memory disambiguation consists in determining dynamically whether two instructions are memory dependent.

When instruction *j* (consumer) data-depends on instruction *i* (producer), a storage element (register o memory location) is used to communicate a value from the producer instruction to the consumer instruction. This storage element is identified by a register number or by a memory address. In the first case, data dependences can be detected as soon as the instructions have been decoded. In the second case, the producer instruction is a store instruction and the consumer instruction is a load instruction, so dependence detection is not resolved until the computation of the effective addresses of both memory instructions. This delay restricts the out-of-order execution of a load instruction independent on a previous store instruction.

Some processors assume conservatively that each load instruction data-depends on its previous store instructions. Consequently, a load instruction cannot be out-of-order executed respect its previous store instruction. This assumption produces a performance decrease because most load instructions are independent on its previous store instruction.

However, some works propose the use of address prediction to predict memory dependences between load and previous store instructions. González and González [GoGo97a] proposed to predict the effective addresses of load and store instructions, and this information is used to detect the dependence of a load instruction on a previous store instruction. On a predicted dependence, data is speculatively forwarded from the store to the load instruction; on a predicted independence, the load instruction accesses data cache speculatively. In both cases, the instructions dependent on the load instruction may be executed speculatively. Other work [Sato98] uses address prediction with the same goal.

1.2.3 Hardware prefetching

Hardware prefetching makes use of address prediction to bring data into a memory-hierarchy level closer to the processor. Thus, on a future reference to the predicted address, load-latency is expected to be reduced.

In the simplest hardware-prefetching mechanism, when a line is referenced, a prefetch is started for the next-sequential line [Smit82]. After this proposal, several works propose hardware-prefetch mechanisms with more complex address predictors: Stride Address Predictors ([ChBa95]), Context Address Predictors ([ChPu97]) and Markov-based Address Predictors ([JoGr97]). None of the previous hardware-prefetching mechanisms need a recovery mechanism on address mispredictions because no instructions are executed speculatively; the only negative impacts of address mispredictions are cache pollution and wasted memory bandwidth.

A more complex mechanism proposed in [GoGo97b] combines hardware prefetching and speculative execution of dependent instructions. A stride predictor is used to predict the effective address that will be computed on the next instance of every load instruction. This address prediction is used to start a prefetch and the obtained data is recorded in a structure indexed by load-instruction PC. When a load instruction is fetched, this structure is checked in order to detect a prefetch for this instruction; if so, the dependent instructions can be executed speculatively using the prefetched value. An additional structure is used to invalidate some prefetchings due to data dependencies between store and load instructions. A recovery mechanism must recover the processor state on address mispredictions.

1.2.4 Start memory accesses early in the pipeline

Other works use address prediction to allow load instructions to access memory early in the pipeline instead of waiting until the computation of the effective address. In some cases, these works also allow the speculative execution of the instructions dependent on the predicted load instruction.

The authors of [EiVa93] presented two mechanisms to obtain the effective addresses of the load instructions early in the pipeline: by using the current value of the base registers of the load instructions and by using a stride predictor. The obtained addresses were used to access memory early in the pipeline. On a correct prediction, load latency is reduced by one cycle; on a misprediction no recovery mechanism is needed due to the pipeline organization and the in-order scheduler used by the processor.

Austin et al. [APS95] proposed the use of a *fast adder* to predict the portion of the effective addresses needed to speculatively access data cache. The fast adder performs the prediction by OR'ing the set index portion of the base register and the offset. On a correct prediction, load latency is reduced by one cycle; on a misprediction, no recovery mechanism is needed due to

the pipeline organization. Note that the address predictor used in this work is a state-less predictor; that is, it does not need a prediction table.

Austin and Sohi [AuSo95] extended the previous work to reduce even more load-instruction latency. They use a small cache to record base addresses; this cache is accessed concurrently to instruction cache (in pipeline organizations with one decode stage) or in the first decode stage (in pipeline organizations with several decode stages). After that, the fast adder predicts the portion of the effective address and the speculative data-cache access is started. No instructions are executed speculatively because the speculative value is only used after verifying the correctness of the prediction.

An exhaustive evaluation of load speculation techniques (dependence prediction, address prediction, value prediction and memory renaming) is presented in [ReCa98]. The authors applied address prediction to access memory speculatively and to execute speculatively the instructions dependent on the predicted load. They evaluated several address predictors (last-address, stride, context and hybrid) and they used two possible recovery mechanisms (squash and selective re-execution), but they did not present practical implementations for the recovery mechanisms.

1.2.5 Recovery mechanisms for address mispredictions

When address prediction is combined with speculative execution of dependent instructions, a recovery mechanism must be provided to deal with address mispredictions. This is due to the speculative execution of some instructions with incorrect input data; these instructions must not update the architectural state of the processor and must be re-executed using the correct input data.

Although all high-performance processors implement a recovery mechanism to deal with branch mispredictions, this mechanism is too rough to be applied to address mispredictions. On branch mispredictions, the instructions fetched after the mispredicted branch violate the semantic correctness of the program. Consequently, they must be flushed-out from the pipeline and the instruction-fetch address must be corrected. However, on address mispredictions, the instructions fetched after the mispredicted load instruction are semantically correct (unless a branch misprediction is involved). Consequently, to avoid re-fetching them, they may remain in a buffer. Moreover, on address mispredictions, only the issued instructions dependent on the misprediction must be re-issued.

Sato [Sato98] proposed the implementation of a recovery mechanism for address mispredictions. This mechanism assumes a processor where the issue queue (structure that contains instructions waiting for operands) and the reorder buffer (structure that contains all in-fly instructions) are combined into a structure named Register Update Unit (RUU, [SoVa87]).

Sato extended the RUU structure to deal with mispredictions. He made the most of two characteristics of the RUU: a) the instructions allocated in the RUU snoop the results computed

by the other instructions allocated in the RUU, b) the instructions are inserted and extracted from the RUU in program order. Sato's proposal consists in: a) marking that an issued instruction must be re-issued when a new value for one of his operands has been detected, b) exploiting the fact that a completed instruction in the head entry of the RUU has non-speculative operands. Other works [Rote99][Saze99] have used similar recovery mechanisms in the scope of value prediction.

This solution imposes a restriction on the number of reorder-buffer entries because the issue queue is less scalable than the reorder buffer. Thus, actual processors decouple the issue queue from the reorder buffer. However, in order not to increase the performance impact of misprediction recovery, the recovery mechanism must include a device which removes issued instructions from the issue queue when they are non-speculative.

Other works describe recovery mechanisms in a more restrictive speculative scope: latency prediction (for instance, some processors predict if a cache reference will hit data cache [Kess99], [Dief99]). The restriction appears because, in the scope of latency prediction, the speculative instructions are issued after issuing the predicted instruction. However, in the scope of address prediction, the speculative instructions may be issued before issuing the effective-address computation of the predicted load instruction. As the recovery mechanisms for latency mispredictions are more cost-effective than the recovery mechanisms for address mispredictions, we will evaluate the convenience of restricting the speculative issue of the instructions in the address-prediction scenario in order to allow the use of recovery mechanisms specific for latency mispredictions.

A deeper analysis of recovery mechanisms is provided in Chapters 5 and 6.

1.3 Thesis overview

The purpose of this thesis is to explore the ability of address prediction and speculative execution to mitigate the effect of the load latency. This work focuses on the following issues:

- **Reducing prediction-table sizes:** The amount of information recorded in the prediction-tables of the address predictors is very large; it is comparable or even larger than current first-level-cache capacities. We have designed and evaluated several strategies to reduce the amount of information recorded in the prediction tables while maintaining the performance of the address predictors. We have obtained up to 60% capacity reductions.

- **Recovery mechanisms in address-prediction scope:** Address mispredictions may produce the speculative execution of some instructions with incorrect input data. To obtain a correct execution result, these instructions must be re-executed with the correct input data. The mechanism responsible for re-executing these instructions and for guaranteeing

a correct architectural state of the processor is named *recovery mechanism*. We have designed and evaluated, using a detailed simulator, recovery mechanisms that enable the issue queue to be decoupled from the reorder buffer.

- **Latency prediction:** We designed and evaluated a recovery mechanism which reduces the pressure on the issue queue. The mechanism has been used to evaluate processor performance in the scopes of load-latency prediction and address prediction.

1.4 Thesis organization

This document is organized as follows:

- Chapter 2 describes the phenomenon of the address predictability. First, we present typical high-level code fragments that are compiled to instruction sequences with predictable load instructions. After that, we introduce the typical address prediction models (last-address, stride, context and hybrid). Finally, we compare the maximum performance of these prediction models.

- Chapter 3 presents two techniques for reducing the amount of information recorded in the prediction table of a Last-Address Predictor. The strategy used by both techniques is to filter out the allocation of unpredictable instructions in the prediction table. The motivation is the highly biased distribution of the address predictability, that is, some load instructions are highly predictable and some are highly unpredictable. Moreover, the replacement algorithm of the prediction tables does not consider predictability information. Consequently, unpredictable instructions can evict predictable ones.

- Chapter 4 also presents a technique for reducing the amount of information recorded in the prediction table of a Last-Address Predictor. In this case, the strategy consists in reducing the redundancy of the information recorded in the prediction table. Redundancy appears due to the spatial-locality property of the memory references; redundancy produces the replication of the high-order portions of some effective addresses in the prediction table.

  The techniques presented in Chapter 3 and in Chapter 4 have been evaluated in terms of predictor performance (captured predictability and accuracy) and area-cost reductions. We do not present results in terms of committed instructions per cycle (IPC), because in these chapters we focus on reducing prediction-table capacity maintaining predictor performance. We focus on the impact of address prediction on processor performance in the following chapter.

- Chapter 5 evaluates the performance impact of address prediction in an out-of-order processor. The evaluated processor configurations use an issue queue decoupled from the reorder buffer, and verification mechanisms which are designed for removing the

speculatively issued instructions from the issue queue as soon as they are known to be non-speculative. The processor configurations evaluated include 4-way and 8-way issue-width processors with several first-level-cache latencies. Moreover, it is evaluated the effect of delaying the speculative issue of instructions until issuing the predicted load instruction; this delay allows the use a recovery mechanism designed for latency prediction.

- Chapter 6 presents the design and the evaluation of two recovery mechanisms for latency prediction. The first mechanism keeps the speculatively issued instructions in the issue queue; the second mechanisms keeps them in a new structure named Recovery Buffer. Moreover, both mechanisms are designed with selective and non-selective nullification policies. The recovery mechanisms are evaluated in the scope of predicting the latency of the load instructions; the evaluations compare the IPC results for integer and floating-point benchmarks. Our results suggest the use of different recovery mechanisms for the integer and the floating-point execution cores.

- Chapter 7 concludes with a summary of this thesis, its conclusions, and a discussion of future work.

- Appendix A describes the simulation environment (simulators, processor model, processor configurations and benchmarks) used in the evaluations presented in this thesis.

2 ADDRESS PREDICTABILITY

This chapter describes the phenomenon of the address predictability. After a brief introduction, in Section 2.2 some high-level language scenarios that produce address-predictable memory instructions are presented. In Section 2.3 the main classes of address predictors proposed in the literature are given. In Section 2.4 some performance results of the address predictors are shown. In Section 2.5 some related works are reviewed. Finally, in Section 2.6 the chapter is summarized.

2.1 Introduction

Temporal locality and spatial locality are two well known properties exhibited by the memory references of a program. Both properties are exploited by caches in order to reduce the latency

of most memory references. However, the impact of load latency (even assuming cache hit) on processor performance is still significant.

Data caches exploit the locality of a program as a whole; that is, without considering which instructions exhibit locality. On the other hand, some works have also proposed exploiting the locality of each memory-access instruction in order to reduce the impact of load latency. To exploit the locality, these proposals rely on hardware mechanisms (named *address predictors*) that find a repeatable pattern; other address predictors recognize address patterns that follow an arithmetic progression.

After that, they predict the effective address that will be computed on the next execution of this memory-access instruction. This prediction can be used to start the memory access early in the pipeline; that is, before computing the effective address of the memory reference. Consequently, on a correct prediction, memory access may be anticipated by several processor cycles.

Address predictors keep track of several instances of each memory instruction in order to identify if the effective addresses computed by a memory instruction are exhibiting a repeatable pattern, or a pattern that follows an arithmetic progression. On a positive identification, the address predictor can predict the effective address that will be computed on the next instance of the memory instruction. A memory instruction is predictable by an address predictor if the instruction is exhibiting a pattern that can be identified by the address predictor.

Address predictors can identify several classes of sequences of effective addressees. We detail the main classes:

- *Constant*: a memory instruction that always computes the same effective address; for instance: 100, 100, 100,... This sequence of addresses is exhibiting temporal locality.

- *Stride*: a memory instruction that computes effective addresses in arithmetic progression; for instance: 100, 101, 102, 103,...

- *Repeated*: a memory instruction that repeatedly computes a sequence (strided or not strided) of effective addresses; for instance: 100, 112, 104, 100, 112, 104, 100,... This sequence of addresses is exhibiting temporal locality.

To understand when and why address predictability appears, in the following section some high-level code fragments are presented where their generated assembly codes have predictable load instructions. That is, these code fragments are sources of address predictability.

2.2 Sources of address predictability

In the next subsections, we present high-level code fragments related to the most predictable load instructions of the benchmark suite (SPEC95 benchmarks, described in Appendix A.2) used in this thesis.

2.2.1 Global addresses

The RISC architecture used in this work (Alpha) uses 32-bit instructions and 64-bit logical addresses. In this environment, the compiler must generate load instructions for loading 64-bit addresses of global objects into registers. However, these loadings cannot be performed directly because a 64-bit address cannot be coded as an immediate operand of the instruction.

Alpha AXP compilers solve this problem by recording the global addresses in a *Global-Address Table* (GAT). In fact, the compiler creates one (or several) GAT for every module, and the linker gathers the GATs into one GAT section. GAT is accessed using a register named *global pointer* (*gp*) and a 16-bit immediate displacement. Potentially, every procedure can use a different GAT; thus, the first instructions of every procedure update the value of the *gp* to point to the GAT related to the procedure.

To access a global object, two load instructions are involved. First, the address of the object is obtained accessing the GAT; second, the object is accessed. That is, global objects are accessed indirectly via the GAT.

Figure 2.1 shows the code generated by the compiler to access a global variable (*n*). Load instruction (1) obtains the address of the global variable from GAT; register *gp* contains the base address of the GAT and the displacement points to the related GAT entry. Load instruction (2) accesses the obtained address. Both load instructions will always compute the same effective address.

```
int n;                               /* 540(gp) contains global-variable address (&n) */
         ───────►(1) ldq a1, 540(gp) /* a1 = global-variable address (&n)*/
... = n;         (2) ldq a2, 0(a1)   /* a2 = global-variable value (n)*/
```

Figure 2.1 Address predictability accessing global variables

However, the linker can apply some optimizations. For instance, the GAT can record global-variable values instead of recording global-variable addresses. More optimizations are proposed in [SrWa94].

```
int n;                        /* 724(gp) contains the global-variable address (&n)*/
        ──────► ldq a1, 724(gp) /* a1 = variable value (n)*/
... = n;
```

a) scalar reference

```
int *c;                        /* 750(gp) contains the global-variable address (&c) */
         ──────► ldq a1, 750(gp) /* a1 = global-variable value (c) */
... = *c;        ldq a2, 0(a1)   /* a2 = pointer contents (*c) */
```

b) pointer reference

Figure 2.2 Address predictability accessing global variables (optimized GAT accesses)

Figure 2.2 shows how global variables are accessed if GAT records global-variable values. In case a), accessing a global scalar variable, the optimization eliminates one load instruction, but

the remaining load instruction will always compute the same effective address. Case b), traversing a global pointer, is analogue to the example of Figure 2.1.

2.2.2 Vector accesses

The address of a vector element is computed by adding the base address of the vector and the element-vector size times the element index. The base addresses of the vectors are also recorded in the GAT. Thus, in order to access a vector element, the compiler generates code for:

- Obtaining the vector base address by accessing GAT
- Computing the address of the element
- Accessing the element

An example of a vector access is shown in Figure 2.3. Load instruction (1) obtains the vector base address and always computes the same effective address. Load instruction (2) can exhibit address predictability depending on the predictability of the values of the vector index.

```
                                     /* 110(gp) contains vector base address */
                                     /* a0 = i */
int v[N];
          ───────▶ (1) ldq    at, 110(gp) /* at = vector base address (&v[0]) */
... = v[i];            s4addq a0, at, a0  /* a0 = element address (&v[i]) */
                   (2) ldq    at, 0(a0)   /* at = element value (v[i]) */
```

Figure 2.3 Address predictability accessing vectors

2.2.3 Switch statements

There are several possibilities for generating the assembly code for a switch statement.

- The easiest way consists in performing a cascaded comparison; that is, comparing the key of the switch statement with the first switch alternative. If they do not match, the key is compared with the second alternative, and so on until a match or the end of the alternatives is found.
- Another possibility consists in the use of a branch table and an indirect branch. The table is filled with the instruction addresses related to the statement alternatives, and the compiler generates code for branching indirectly through this table. The advantage of this alternative is that the number of branches executed for every execution of the switch statement is independent of the value of the key.

Alpha-AXP compilers use the second alternative, and they record the base address of the branch table in the GAT. The compiler must generate code for:

- Initializing a branch table with the addresses related to every alternative

- Transforming the key into an index to the branch table
- Testing bounds of the transformed key
- Accessing the branch table
- Branching indirectly

Figure 2.4 shows the assembly output of a switch statement. Load instruction (1) obtains the base address of the branch table; this load always computes the same effective address. The predictability of load instruction (2) depends on the predictability of the values of the key.

```
                        /* Table[0] = _lab1;  Table[1] = _def;  */
                        /* Table[2] = _lab34; Table[3] = _lab34; */
                        /* a0 = key */

                        subl    a0, 0x1, t0 /* value transformation a0 = key-1 */
                        cmpult  t0, 0x4, t1 /* compare to maximum value */
switch (key) {          beq     t1, _def    /* branch to _def if key > 4 */
  case 1:           (1) ldq     at, 100(gp) /* at = Table base address (&Table[0])*/
    ...                 s4addq  t0, at, t0  /* t0 = Table[t0] address (&Table[t0])*/
  case 3:           (2) ldl     t0, 0(t0)   /* t0 = Table[t0] value */
  case 4:    ---->      addq    t0, gp, t0  /* gp-relative value */
    ...                 jmp     zero, (t0)  /* indirect branch */
  default:
    ...             _lab1:
}                     ...

                    _lab34:
                      ...

                    _def:
                      ...
```

Figure 2.4 Address predictability in switch statements

2.2.4 Stack accesses

The stack is a run-time data structure used to record the activation blocks of the procedures that are being executed. The stack is implemented in memory: its base address is fixed and it grows towards consecutive memory locations; a processor register named *stack pointer* (*sp*) points to the element most recently inserted. The stack is accessed by load and store instructions relative to the *sp* register.

The value of register *sp* defines the stack depth. The amount of different effective address computed by each stack-based load instruction of a procedure is equal to the number of different stack depths seen at the executions of the call instructions to this procedure. However, some procedures are always called from the same stack depth; in this case, each stack-based load instruction of the procedure will always compute the same effective address.

```
                                  proc:
                                       addq  sp, -64, sp
                                       stq   ra, 0(sp)
                                       stq   s0, 8(sp)
while (cond)                           stq   s1, 16(sp)
  proc(a,b);                           ...
  ...                 ------>          ldq   v0, 56(sp)
}                                      ...
                                       ldq   s1, 16(sp)
                                       lqq   s0, 8(sp)
                                       ldq   ra, 0(sp)     /* ra = return address */
                                       addq  sp, 64, sp
                                       ret   zero, (ra), 1
```

Figure 2.5 Address predictability accessing the stack

For instance, in Figure 2.5, every loop iteration calls the procedure *proc*. As stack depth must be the same at every *proc* call, the effective addresses computed by each memory-access instruction related to the *sp* register will be the same at every call.

2.2.5 System constants

System information that is always recorded at the same effective address may be accessed by some user procedures, library procedures and macro-instructions. Consequently, the related load instructions will always compute the same effective address.

```
                                 if (isalpha(c)) n++;
                                           | Preprocessing
                                           v
     if ((((*(__lc_ctype->core.iswctype)) == 0L) ?
         (int) (__lc_ctype->_mask[c] & (0x001)) :
         (*(__lc_ctype->core.iswctype)) (c,0x001,__lc_ctype))) n++;
                                           | Compilation
                                           v
(1)  ldq    a2, -3278(gp) /* a2 = __lc_ctype address */
(2)  ldq    a2, 0(a2)     /* a2 = __lc_ctype value */
(3)  ldq    v0, 48(a2)    /* v0 = __lc_ctype->core.iswctype value */
     bne    v0, _else     /* if v0!=0 branch _else (never branchs) */
     ldq    t0, 24(sp)    /* t0 = c */
(4)  ldq    t1, 104(a2)   /* t1 = __lc_type->mask base address */
     extqh  t0, 0x1, t0
     sra    t0, 0x38, t0
     s4addq t0, t1, t2    /* t2 = __lc_ctype->_mask[c] address */
(5)  ldl    a0, 0(t2)     /* a0 = __lc_ctype->_mask[c] value */
     and    a0, 0x1, a0
     br zero, _out
  _else:
     ldq    a0, 24(sp)    /* dead code */
     addq   zero, 0x1, a1
     bis    v0, v0, t12
     extqh  a0, 0x1, a0
     sra    a0, 0x38, a0
     jsr ra, (v0)
     ...
  _out:
```

Figure 2.6 Address predictability produced by system constants

Figure 2.6 shows an example. Macro-instruction *isalpha* checks if a character is alphabetic. *isalpha* consults system-configuration values to answer. Load instructions (1), (2), (3) and (4) always compute the same effective address. The predictability of load instruction (5) depends on the predictability of the parameter of the macro-instruction.

2.2.6 Sequential accesses to vector structures

Some programs access vectors sequentially using a loop statement. The effective addresses computed by the load instruction related to these accesses are in arithmetic progression; that is an arithmetic-progression pattern.

For instance, Figure 2.7 shows both a high-level code that accesses sequentially a vector and its assembly output. The effective addresses computed by load instruction (2) are in arithmetic progression because the address computed by an instance of the instruction is equal to the address computed by the previous instance plus a constant (4, the size of every vector element). Applying loop unrolling to this loop will generate a code that performs several vector accesses at every iteration. However, the effective addresses produced by each one of these load instructions will also be in arithmetic progression, but with a larger common difference. In addition, load instruction (1) will always compute the same effective address (Section 2.2.2).

```
int v[N];
int i;
...

for (i=0; i<N; i++)
{

   ...= v[i];
}
```

```
(1)  ldq   t1, -32736(gp)  /* t1 = vector base address */
     bis   zero, zero, t0  /* t0 = 0 */
           ...
   loop:
(2)  ldl   t3, 0(t1)       /* t3 = v[t0] */
     addl  t0, 0x1, t0     /* t0 = t0 + 1 */
     cmplt t0, N, t4       /* test loop bound */
     lda   t1, 4(t1)       /* t1 = t1 + 4 */
     ...
     bne   t4, loop        /* branch if t0 < N */
```

Figure 2.7 Address predictability accessing vector structures sequentially

Arithmetic-progression patterns may also be exhibited by accesses to dynamically allocated data structures. For instance, let us assume a dynamically allocated list structure. When the list is being allocated, the program requests memory storage for each list element. Typically, the memory allocator allocates a request next to the previous one. As the requests are equally sized (*s* bytes), the base addresses of the list elements are in arithmetic progression (*a*, *a+s*, *a+2s*,...). If this list is traversed in the same order as the list elements have been allocated, the load instructions that access the list will compute addresses in arithmetic progression.

The stack accesses performed by recursive invocations of a procedure may also exhibit an arithmetic-progression pattern. Assume a stack-based memory-access instruction that is executed only once at every procedure invocation. On recursive invocations, the memory-access instruction will compute effective addresses in arithmetic progression with a stride equal to the size of the activation block of the procedure.

2.2.7 Repeated access sequences

In some scenarios, a load instruction can repeatedly compute a sequence of effective addresses; that is, the sequence of effective addresses computed by the load instruction is exhibiting temporal locality.

A typical scenario where this situation arises is traversing a linked list. Figure 2.8 shows both the high-level code that traverses a linked list searching for an element, and its assembly code. Although the list may be updated during program execution, some portions may remain unchanged during several traverses. In this case, load instructions (1) and (2) will repeatedly compute the same sequence of effective addresses. For instance, if the depicted portion of the list remains unchanged during some traverses, load instruction (2) will repeatedly compute the address sequence 400, 160, 320 and 720; moreover, load instruction (1) will repeatedly compute the address sequence 408, 168, 328 and 728.

```
struct elem                                              /* v0 contains p */
{ struct elem *next;                                     /* t2 contaits key */
  int         key;                    beq v0, out        /* branch if v0 is null */
} *start, *p;                      loop:
                                   (1) ldl t1, 8(v0)     /* t1 = p->key */
...                          --->      xor t1, t2, t1    /* t1 = p->key XOR key */
p = start;                             beq t1, out       /* branch if equal */
while (p && p->key != key)         (2) ldq v0, 0(v0)     /* v0 = p->next */
  p = p -> next;                       bne v0, loop      /* branch if v0 not null */
...                                out:
```


Figure 2.8 Address predictability accessing repeatedly a linked list

Another scenario for this situation is produced by repeated sequential accesses to vectors. The related load instruction will repeatedly compute the same sequence of vector-element addresses.

2.3 Address predictors

This section reports the main classes of address predictors proposed in the literature. Each subsection is related to an address-prediction class and details its functionality, its block diagram, its pseudo-code and the amount of information that it records.

2.3.1 Last-Address Predictor (LAP)

The Last-Address Predictor (LAP) is designed to predict effective addresses computed by load instructions that exhibit temporal locality. The LAP assumes that a load instruction will compute the same effective address as in its previous execution.

This predictor predicts correctly the addresses computed by load instructions that always generate the same address (such as accesses to global variables, some stack accesses).

Moreover, load instructions presenting bursts of executions that compute the same effective address will also be predicted correctly.

The *learning time* of an address predictor is defined as the number of executions of a load instruction that must be tracked by the address predictor in order to be able to predict the next effective address computed by the load instruction. The learning time of the LAP is one execution.

The simplest implementation of an LAP consists in a direct-mapped table indexed by using some bits of the instruction PC. Each table entry contains an effective address: the last effective address computed by the load instructions mapped to the table entry. Consequently, the allocation policy of the table is the *always allocate* policy; that is, after the execution of a load instruction, its effective address is recorded in its related table entry. This table is named the *Prediction Table* of the predictor; in the scope of address prediction it will be named the *Address Table* (*AT*).

To reduce the number of wrong predictions, two basic mechanisms can prevent possible wrong predictions from being performed:

- Tagging table entries. Tagging is useful to identify which load instruction the information recorded in a table entry is related to. Tagging avoids predicting a load instruction using information related to another load instruction. Table entries can be fully tagged or partially tagged.

- Using a confidence estimator. This estimates how likely a prediction is of being correct. The most used confidence estimator is the bimodal estimator [Smit81] which was proposed in the scope of branch prediction. This estimator (Figure 2.9 shows its state diagram) consists of a 2-bit saturating counter related to every prediction-table entry. The counter is increased by one on correct predictions and decreased by one on wrong predictions. If the counter value is over a threshold value (for instance, the counter-value 01), the confidence estimator allows predicting the next instance of the instruction. Note that the use of confidence estimation can increase the learning time of a predictor.

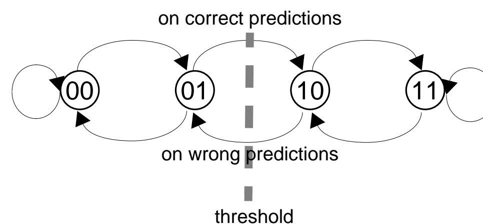


Figure 2.9 State diagram of the 2-bit confidence estimator

Figure 2.10 shows the scheme of an LAP with tagged entries and bimodal estimation.

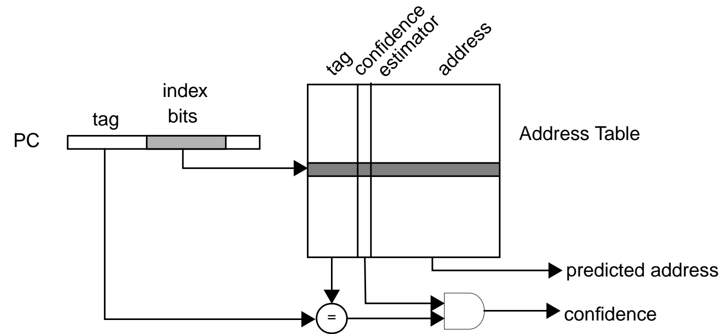


Figure 2.10 Diagram of the Last-Address Predictor

Figure 2.11 presents the pseudo-code that predicts and updates the Address Table (AT) of a LAP with tagged entries and bimodal confidence estimators. Operators ++ and -- update the counter of the confidence estimator in a saturated way.

```
/* Predicts and effective address
   Output variable:
    predict: predicted address
*/

Prediction(PC)
{
  idx = index_bits(PC);
  tag = tag_bits(PC);

  if (AT[idx].tag == tag) {
    if (AT[idx].conf > threshold)
      predict = AT[idx].addr;
  }
}
```

```
/* Updates the Address Table */

Update(PC, addr)
{
  idx = index_bits(PC);
  tag = tag_bits(PC);

  if (AT[idx].tag == tag) {
    if (AT[idx].addr == addr)
      AT[idx].conf++;
    else
      AT[idx].conf--;
  }
  else {
    AT[idx].tag = tag;
    AT[idx].conf = INI_CONF;
  }
  AT[idx].addr = addr;
}
```

Figure 2.11 Pseudo-code of the Last-Address Predictor

The next formula gives the amount of information recorded by an LAP. For instance, an LAP with a 4K-entry AT, 64-bit effective addresses, 4-bit tags and 2-bit confidence estimators records 286.720 bits; that is, 35 Kbytes.

$$AreaCost_{LAP} = \text{AT_entries} \times (\text{addr_bits} + \text{tag_bits} + \text{conf_bits})$$

Several works have used predictors equivalent to the LAP to predict effective addresses or instruction results: [EiVa93], [LWS96], [ReCa98].

2.3.2 Stride Address predictor (SAP)

The Stride Address Predictor (SAP) is designed to predict effective addresses that exhibit arithmetic-progression pattern. The SAP predicts that a load instruction will compute an effective address equal to the sum of the last computed effective address and a stride. The stride is defined as the difference of the effective addresses computed in two consecutive instances of the load instruction.

This predictor predicts correctly load instructions that compute effective addresses in arithmetic progression. Consequently, the SAP is able to generate addresses never seen by the processor (the LAP only generates addresses that have been previously computed by the processor). A load instruction correctly predicted by the LAP can also be predicted correctly by the SAP because a constant sequence of addresses presents a particular case of arithmetic progressions. However, the learning time of the SAP is larger than the learning time of the LAP (two executions versus one execution).

The most simple implementation of an SAP consists of a direct-mapped table indexed by using some bits of the instruction PC. Each table entry contains an effective address, the last effective address computed by the load instructions mapped to the table entry, and the stride. To reduce the number of wrong predictions, tags and confidence estimators can be attached to the prediction-table entries. Potentially, the stride field of this table should be 64-bit wide. However, some studies have shown that the high-order bits of the stride are not significant, and most strides are 8-bit or 16-bit wide [GVD01].

Figure 2.12 shows the scheme of an SAP with tagged entries and with bimodal confidence estimators.

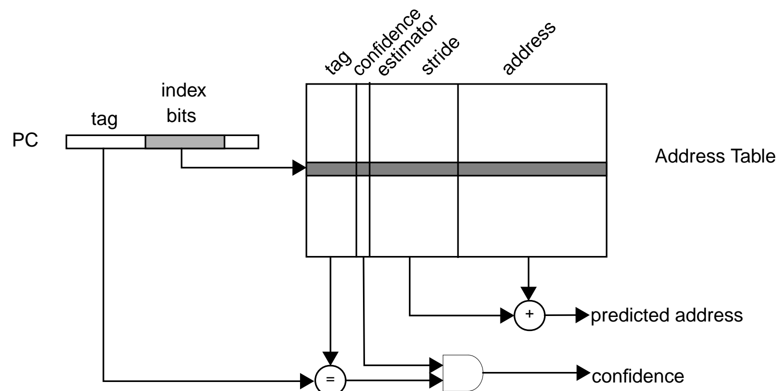


Figure 2.12 Diagram of the Stride Address Predictor

Figure 2.13 presents the pseudo-code that predicts an effective address and updates the Address Table of an SAP.

```
/* Predicts an effective address
   Output variable:
    predict: predicted address
*/

Prediction(PC)
{
  idx = index_bits(PC);
  tag = tag_bits(PC);

  if (AT[idx].tag == tag) {
    if (AT[idx].conf > threshold)
      predict = AT[idx].addr
           + AT[idx].stride;
  }
}
```

```
/* Updates the Address Table */

Update(PC, addr)
{
  idx = index_bits(PC);
  tag = tag_bits(PC);

  if (AT[idx].tag == tag) {
    if (AT[idx].addr == addr)
      AT[idx].conf++;
    else
      AT[idx].conf--;
    AT[idx].stride = addr - AT[idx].addr;
  }
  else {
    AT[idx].tag = tag;
    AT[idx].conf = INI_CONF;
    AT[idx].stride = 0;
  }
  AT[idx].addr = addr;
}
```

Figure 2.13 Pseudo-code of the Stride Address Predictor

The next formula gives the amount of information recorded by an SAP. For instance, an SAP with a 4K-entry AT, 64-bit effective addresses, 4-bit tags, 2-bit confidence estimators and 8-bit strides records 319.488 bits, that is, 39 Kbytes.

$$AreaCost_{SAP} = \text{AT_entries} \times (\text{addr_bits} + \text{tag_bits} + \text{conf_bits} + \text{stride_bits})$$

Several works have used address predictors similar to the SAP. For instance: [EiVa93], [ChBa95], [GoGo97a].

2.3.3 Context Address Predictor (CAP)

A Context Address Predictor (CAP) is designed to predict effective addresses computed by load instructions that exhibit temporal locality. CAP tries to record the sequence of effective addresses computed by the instances of each load instruction. If the predictor detects that a load instruction is generating a recorded sequence again, the predictor predicts that the next effective address will be the next effective address in the recorded sequence.

The basic implementation of this predictor consists in a two-level structure.

- The first-level table is named the *Value History Table* (VHT). The VHT is indexed by the PC of the load instruction and contains the context; that is, the sequence of effective addresses being generated by the load instruction. The *order* of a context-address predictor is defined as the length of the sequence of addresses recorded in the context.

- The second-level table is named the *Value Prediction Table* (VPT). The VPT is indexed by the hashed context and contains the predicted address that follows the context.

The learning time of the CAP is longer than that of the previous predictors. It is equal to the order of the predictor plus the length of the repeated sequence minus one. That is, some executions are needed to fill the context of the instruction. After that, the sequence can be learnt, and finally the predictor can predict effective addresses for this instruction.

A load instruction correctly predicted by the LAP can also be predicted by the CAP. Moreover, some load instructions correctly predicted by the SAP can also be predicted by the CAP. These load instructions must compute repeatedly the same strided sequence of effective addresses (for instance, 1, 3, 5, 1, 3, 5, 1,...). The remaining load instructions predicted by the SAP cannot be predicted by the CAP because the CAP generates only effective addresses previously seen by the processor.

As in the previous predictors, prediction tables can be tagged and can contain confidence estimators. The design space of CAP predictors is very large because there are a lot of new design parameters such as: the order of the predictor, context representation, the indexing function of VPT,... [Saze99] analyses some of these parameters. A typical indexing function is obtained by shifting each address of the context some bits to the left (for instance, 0 bits the most recent address, 3 bits the second most recent address, 6 bits the third most recent address,...), xor-ing these values, and selecting the low-order bits of the result needed to index the VPT.

Figure 2.14 shows the scheme of a CAP with tagged VHT entries and with bimodal confidence estimators in the VPT entries.

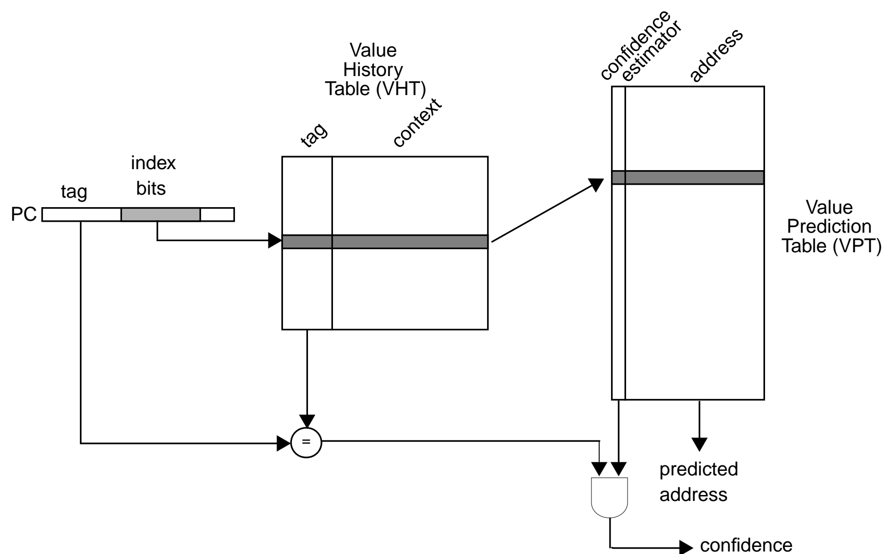


Figure 2.14 Diagram of a Context Address Predictor

Figure 2.15 presents the pseudo-code that predicts an effective address and updates the prediction tables of a CAP.

```
/* Predicts an effective address
   Output variable:
    predict: predicted address
*/

Prediction(PC)
{
  idx = index_bits(PC);
  tag = tag_bits(PC);

  if (VHT[idx].tag == tag) {
    idx2 = HASH(VHT[idx].context);
    if (VPT[idx2].conf > trh)
      predict = VPT[idx2].addr;
  }
}
```

```
/* Updates VHT and APT */

Update(PC, addr)
{
  idx = index_bits(PC);
  tag = tag_bits(PC);

  if (table[idx].tag == tag) {
    idx2 = HASH(VHT[idx].context);
    if (VPT[idx2].addr == addr)
      VPT[idx].conf++;
    else {
      VPT[idx].conf--;
      VPT[idx2].addr = addr;
    }
  }
  else {
    VHT[idx].tag = tag;
    VHT[idx].conf = INI_CONF;
    VHT[idx].context = NULL_CONTEXT;
  }
  insert(addr, VHT[idx].context);
}
```

Figure 2.15 Pseudo-code of the Context Address Predictor

The next formula gives the amount of information recorded by a CAP. For instance, a CAP with a 4K-entry VHT, 16K-entry VPT, 64-bit effective addresses, 14-bit contexts, 4-bit tags and 2-bit confidence estimators records 1.155.072 bits; that is, 141 Kbytes.

$$AreaCost_{CAP} = VHT\text{_entries} \times (\text{tag_bits} + \text{context_bits}) + VPT\text{_entries} \times (\text{conf_bits} + \text{addr_bits})$$

Other works ([SaSm97], [ReCa98]) have used this predictor; several variants of this predictor are also presented in Section 2.5. Moreover, this predictor was previously used in branch prediction [YePa92].

2.3.4 Hybrid Address Predictor (HAP)

Every address predictor is specialized in identifying a certain kind of address pattern. A hybrid predictor combines several existent predictors to obtain a new predictor that tries to use the address predictor best suited to each instruction.

A hybrid predictor needs a metapredictor (or selector) that selects the predictor that will predict every instruction. The metapredictor can consider the operation code of the predicted instruction or the past behaviour of the predictors with the instruction. A simple metapredictor to choose between two predictors consists of a table of 2-bit saturating counters indexed by predicted-instruction PC. Every counter tracks which predictor is more accurate for the instructions that share that counter.

Several works have used hybrid predictors to predict effective addresses or instruction results; for instance [WaFr97], [BMP+98] and [PMT99]. Hybrid prediction has also been used in the context of branch prediction [McFa93].

2.4 Performance comparison of address predictors

In this section a simple evaluation of the basic address predictors reported in the previous section is presented. We have used them to predict the effective addresses computed only by the load instructions.

2.4.1 Metric description

In this work, we will use three metrics to measure the performance of the address predictors:

- *Captured predictability*, defined as the percent of correctly predicted effective addresses out of the total number of executed load instructions.
- *Accuracy*, defined as the percentage of correct predictions out of the number of predictions.
- *Area cost*, defined as the amount of information recorded in the prediction tables of a predictor.

2.4.2 Results

In order to evaluate the address predictors, we have instrumented (Appendix A.1) the load instructions of the benchmark suite (Appendix A.2). Benchmarks have been run until completion.

Maximum captured predictability

First, we have computed the potential performance of the predictors. Table 2.1 shows the maximum predictability captured by the Last-Address Predictor, the Stride Address Predictor and the Context Address Predictor. In these evaluations we have used LAP's and SAP's with unbounded AT's and no confidence mechanisms; also, we have used CAP's with order 4, unbounded VHT's and extremely large VPT's (2^{20} entries).

We can observe that LAP predicts correctly around 56% of the effective addresses in SPEC95-INT benchmarks, but only around 25% of effective addresses in SPEC95-FP benchmarks. In most integer benchmarks, SAP increments slightly the predictability captured by the LAP; only in *m88ksim, compress* and *ijpeg* is the increment significant. On the other hand, the SAP is very effective in floating-point benchmarks due to the presence of strided vector accesses. Finally, the CAP is more effective than SAP in integer benchmarks (79% versus 66%), but not in floating-point benchmarks (76% versus 87%).

In benchmarks *go* and *perl*, the maximum predictability captured by the LAP is bigger than the maximum predictability captured by the SAP. This is due to the fact that the learning time of

SAP (two instructions) is longer than the learning time of LAP (one instruction), and most instructions predictable by the SAP in these programs are also predictable by the LAP.

SPEC95-INT Benchmark	LAP	SAP	CAP
go	52.66	50.21	69.44
m88ksim	71.89	84.22	95.75
gcc	64.86	65.96	81.96
compress	57.66	76.23	65.39
li	35.55	38.18	82.27
ijpeg	21.71	73.18	45.44
perl	79.22	78.95	99.92
vortex	62.65	61.17	93.15
Average	55.77	66.01	79.16

SPEC95-FP Benchmark	LAP	SAP	CAP
tomcatv	2.78	97.78	69.75
swim	0.24	99.30	52.90
su2cor	23.89	83.20	59.74
hydro2d	7.72	96.77	68.10
mgrid	0.29	92.92	87.33
applu	44.02	77.32	65.46
turb3d	31.10	61.31	86.99
apsi	26.81	83.06	89.97
fpppp	97.29	97.73	99.14
wave5	19.38	80.82	77.88
Average	25.35	87.02	76.17

Table 2.1 Predictability captured by the address predictors with unbounded prediction tables and no confidence mechanism in SPEC95 benchmarks

Maximum captured predictability using bimodal confidence mechanisms

With confidence mechanisms, some predictions that will probably be wrong can be avoided. However, some correct predictions may also be avoided. Table 2.2 shows the captured predictability and the accuracy of the predictors obtained by adding a confidence estimator to the predictors used in the evaluations presented in Table 2.1. Note that the accuracy of the predictors presented in Table 2.1 is equal to their captured predictability.

Comparing Table 2.1 and Table 2.2, there is a reduction in the maximum predictability captured by the address predictors. Confidence mechanisms increase the learning time of the predictors and prevents some predictions that may be correct.

Benchmark	LAP		SAP		CAP	
	Predict.	Accur.	Predict.	Accur.	Predict.	Accur.
go	47.81	92.40	44.95	92.32	62.65	93.90
m88ksim	69.53	96.89	79.86	94.18	94.16	97.89
gcc	60.11	95.03	61.47	96.11	75.92	95.43
compress	57.27	99.72	71.73	96.17	61.95	99.18
li	30.20	94.11	31.06	94.16	76.27	96.07
ijpeg	19.11	93.14	68.67	93.05	38.78	96.71
perl	79.19	97.61	76.36	99.54	99.90	99.97
vortex	56.28	93.90	57.73	96.85	90.70	97.63
SPEC95-INT average	52.43	95.35	61.47	95.29	75.04	97.09
tomcatv	2.64	98.09	97.12	98.89	54.71	76.04
swim	0.24	99.94	98.95	99.30	42.39	75.15
su2cor	23.58	99.71	82.28	98.84	51.92	96.34
hydro2d	7.40	99.95	95.76	97.95	50.99	91.58
mgrid	0.29	99.94	89.53	93.30	80.95	96.73
applu	43.05	95.88	67.56	84.73	59.33	99.08
turb3d	28.89	99.06	53.11	87.24	84.55	98.30
apsi	26.19	96.71	74.99	90.05	86.96	99.42
fpppp	97.19	99.91	97.19	99.84	98.99	99.84
wave5	17.83	95.34	78.61	97.77	73.38	98.41
SPEC95-FP average	24.73	98.45	83.51	94.79	68.42	93.09

Table 2.2 Captured predictability and accuracy of the address predictors with unbounded prediction tables and confidence mechanism in SPEC95 benchmarks

Area cost versus captured predictability

We will compare the area cost of several predictor configurations versus their captured predictability. We present the results of LAP and SAP configurations from 256-entry to 16Kbyte-entry AT's. Furthermore, we present the results of CAP configurations with order four, from 256-entry 16Kbyte-entry VHT's, and the number of VPT entries is four times the number of VHT entries. Figure 2.16 shows the results for SPEC95-INT benchmarks and Figure 2.17 for SPEC95-FP benchmarks. Each graph is related to a benchmark; the vertical axis stands for the captured predictability, the horizontal axis stands for the area cost of the predictors (note the logarithmic scale) and a line connects the results for each address predictor.

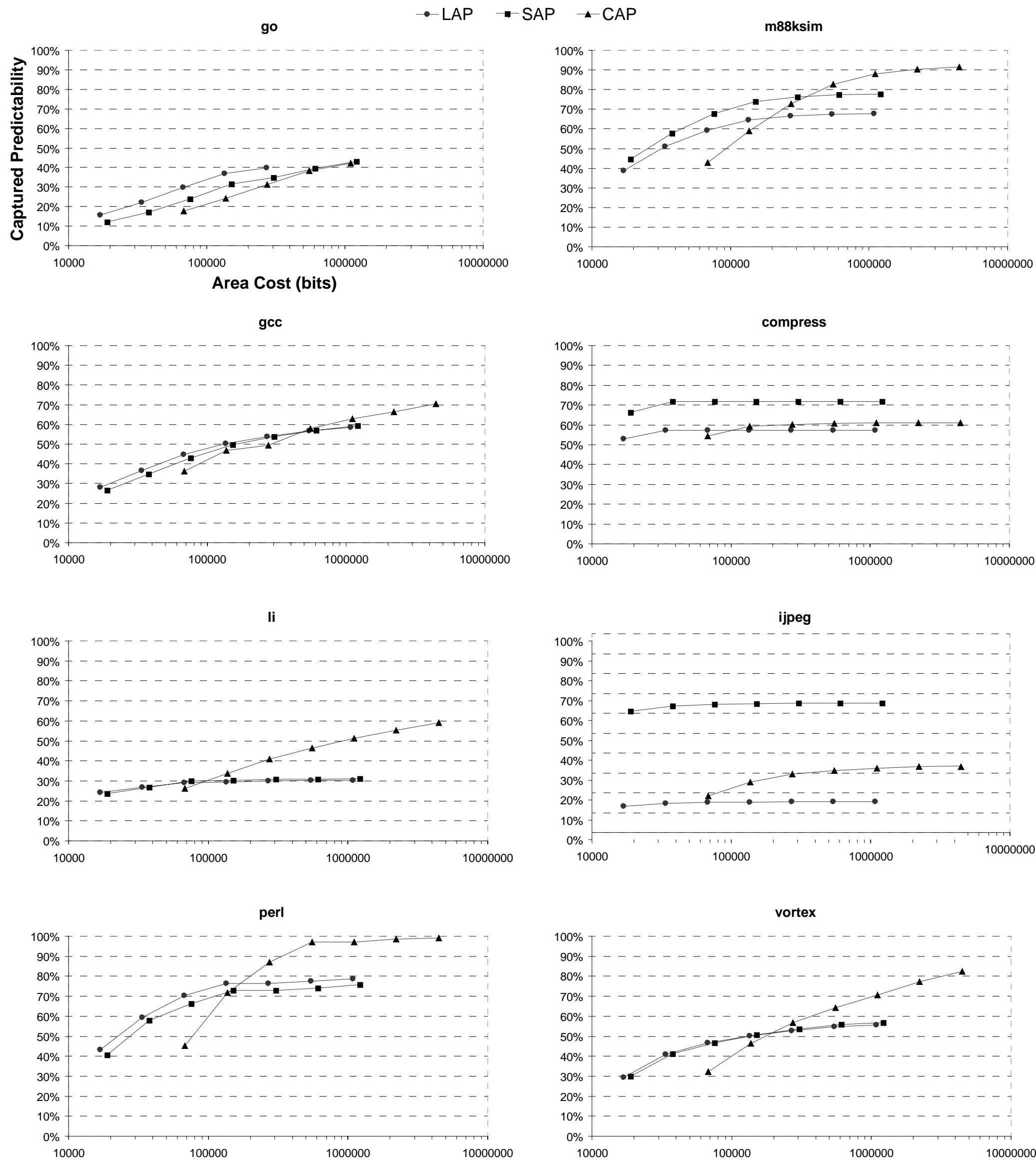


Figure 2.16 Area cost versus captured predictability of the LAP, SAP and CAP in SPEC95-INT benchmarks

In all integer benchmarks except *ijpeg* and *compress*, the CAP achieves the best performance but it needs a large area cost to outperform the LAP and the SAP. However, with a small area-cost budget, LAP and SAP offer better performance than CAP. In most benchmarks, the LAP and the SAP offer a similar cost-performance relationship.

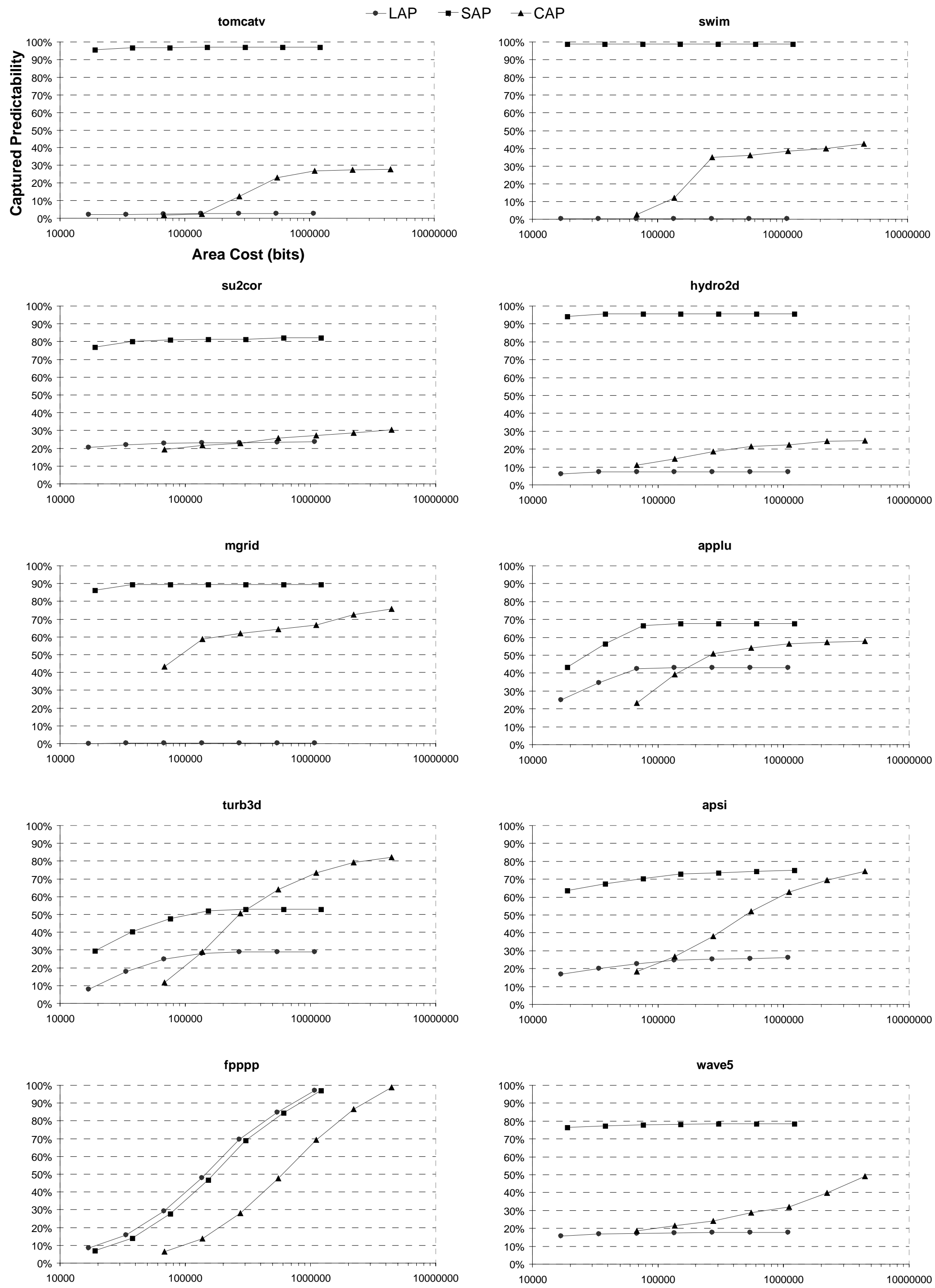


Figure 2.17 Area cost versus captured predictability of the LAP, SAP and CAP in SPEC95-FP benchmarks

In most floating-point benchmarks, the SAP offers the best cost-performance relationship. In benchmarks *tomcatv*, *su2cor* and *hydro2d*, CAP performance is much smaller than the value presented in Table 2.2, because the mapping function to the VPT is not the best suited for these benchmarks.

2.5 Related works

Lipasti et al. [LWS96] introduced the concept of *value locality* when they observed that the sequence of results produced by each instruction exhibits temporal locality; that is, each instruction tends to produce results that have been produced in previous executions of the instruction. Their evaluations observed this property in arithmetic instructions and in memory access instructions (both in the values loaded from memory and in the effective addresses). After this observation, they proposed to extract more parallelism from programs by using a predictor that exploits this property.

In order to perform a prediction, the predictor must select a value from the value history of the instruction. The easiest implementation consists in selecting the last result produced by the instruction; that is, a predictor like the Last-Address Predictor. However, although the sequence of effective addresses computed by a load instruction exhibits temporal locality, they may not be predictable by a Last-Address Predictor. For instance, a load instruction that randomly accesses two memory locations exhibits temporal locality (only produces two different effective addresses) but no predictor can learn its access pattern. Furthermore, a sequence of addresses that are in arithmetic progression does not exhibit temporal locality, but it can be predicted by a Stride Address Predictor. Consequently, although a sequence of addresses exhibit temporal locality, it might not be predictable by a predictor. The sequence is predictable if it follows a pattern that can be learnt by the predictor.

We have reported the basic implementations of the main address predictions, but several works have presented slight variations to these schemes.

Eickemeyer and Vassiliadis [EiVa93] proposed a variation to the SAP that does not update the stride field after every prediction in order to avoid mispredictions in repeated stride sequences. This variation is useful for load instructions that produce strided sequences and that are placed inside a loop nest, because it avoids the learning time of the predictor on the first loop iterations. Chen and Baer [ChBa95] proposed the usage of branch-outcome information to avoid mispredictions related to strided load instructions placed inside loop nests. This variation was applied on prefetching; it avoids starting more prefetchings than loop iterations.

Several variants to the CAP reported in Section 2.3.3 have also been proposed. The main difference between them is the context information recorded in the VHT.

- The *Global-correlated context-address predictor* ([BJR+99]) builds the context from the base addresses of the memory accesses instead of using the effective addresses. The VPT records base addresses instead of effective addresses. Consequently, the memory-access instructions that use the same base addresses will share the same VPT entries. For instance, the load instructions that access several fields of a structure element (load instructions (1) and (2) in Figure 2.8). To obtain the predicted effective address, the base address obtained from the VPT must be added to the offset of the load instruction.

- The *Differential Finite Context Method* ([GVD01]) was proposed in the scope of value prediction but can also be applied to address prediction. The context is built from the sequence of differences between the last results of an instruction. Thus, all the instructions that generate the same sequence of differences will share the same VPT entries. The VPT records the differences related to each context. To obtain the predicted result, the last instruction result must be added to the predicted difference. This predictor also allows several instructions to share VPT entries. For instance, if we apply it to address prediction, and some load instructions are accessing several fields of a linked list load instructions (1) and (2) in Figure 2.8), the sequence of differences produced by these load instructions will be unique. Moreover, the sequence of differences of all the load instructions that are in arithmetic progression with stride one will be the same. Consequently, these load instructions will share a VPT entry.

Another variant to the CAP was proposed by [WaFr97] in the scope of value prediction. This proposal takes advantage of the temporal locality property of the instructions results. The authors evaluated the number of different results produced by an instruction and found that in most cases it was limited by four. Thus, they organize the prediction tables with this restriction. Their VHT records the four most recent unique results of each instruction and their VPT selects one of these four results. Their VPT is indexed by a *history pattern* built from the location identifiers (00, 01, 10 and 11) of the results recorded in VHT.

The reported address predictors predict a load instruction by considering only the previous instances of this load instruction. Other address predictors predict a load instruction by considering the behaviour of other load instructions. For instance, the *Dependence-based Address Predictor* (DEAP), proposed in [AEK+01], relies on the data dependences between two load instructions. The DEAP tries to detect if a load instruction (producer) loads a value that is used by another load instruction (consumer) as the base address of its memory access. Thus, the effective address of the consumer load is predicted by considering the value loaded by the producer load instruction. These kinds of predictors can predict load instructions that compute a sequence of effective addresses exhibiting neither temporal locality nor arithmetic-progression pattern.

2.6 Chapter summary

In this chapter we have shown that some memory-access instructions generate effective addresses exhibiting temporal locality or following an arithmetic-progression pattern. In the case of an Alpha architecture, we have presented high-level code scenarios that produce these load instructions.

Then, we have described address predictors. An address predictor is a hardware mechanism designed both to identify load instructions that exhibit a repeatable pattern, and to predict the effective addresses computed by the next instances of these load instructions. We have reported the main classes of address predictors proposed in the literature: Last-Address Predictor, Stride Address Predictor and Context Address Predictor.

After that we have evaluated the address predictors. The evaluations show that address predictors can correctly predict a significant percentage of effective addresses computed by load instructions. In SPEC95-INT benchmarks, the simplest address predictor (the Last-Address Predictor) correctly predicts about 52% of effective addresses computed during their execution; the Context Address Predictor can capture even more predictability (75%). In the case of SPEC95-FP benchmarks, the Stride Address Predictor correctly predicts about 83% of the effective addresses. Moreover, the accuracy of the predictors is around 95%.

The high predictability of the effective addresses suggests a new opportunity to tolerate the latency of the load instructions. An address predictor can predict the effective addresses early in the pipeline (note that prediction tables are indexed with the instruction PC) and memory can be accessed before computing the effective address. On data availability, the instructions dependent on the load instruction may be executed speculatively.

Despite the high predictability of the effective addresses, the performance impact of address prediction and speculative execution on processor performance must also take other issues into account such as area-cost of the address predictors, the integration of address prediction in a processor pipeline, address-misprediction handling. These issues will be analysed in the following chapters.

3 REDUCING PREDICTION-TABLE SIZE BY FILTERING

In this chapter two techniques for reducing the amount of information recorded in the prediction table of an address predictor are presented. Both techniques use the same strategy: filtering-out the allocation of unpredictable load instructions in the prediction table. This chapter is organized as follows. In Section 3.1, the chapter is introduced. In Section 3.2, the main idea of the filtering techniques are presented. In Section 3.3, the benchmarks used in our evaluations are characterized; this characterization will be used to design the proposed filtering techniques. In Section 3.4 an address predictor that uses the first filtering technique is presented and evaluated. In Section 3.5, an address predictor that

uses the second filtering technique is presented and evaluated. In Section 3.6, some related works are reviewed. Finally, in Section 3.7 this chapter is summarized and in Section 3.8 the results of our proposals for all SPEC95-INT benchmarks are presented.

3.1 Introduction

The amount of information recorded in the prediction tables of the address predictors turns out to be very large. For instance, assuming 64-bit effective addresses, a Last-Address Predictor with a 4K-entry Address Table manages around 32 Kbytes of information. A Context Address Predictor with a 4.096-entry Value History Table and a 16K-entry Value Prediction Table manages around 140 Kbytes of information. That is, an amount of information similar of ever greater than that of current first-level caches.

Potentially, the address predictors reported in Section 2.3 predict correctly a significant amount of effective addresses. However, the use of bounded prediction tables can produce conflicts between the load instructions, and these conflicts can reduce the performance of the address predictor. We present two scenarios where conflicts in the prediction tables reduce predictor performance:

- Predictor resources are efficiently used when they are assigned to predictable instructions. However, the policy used by the reported address predictors to allocate load instructions in the prediction tables does not consider the predictability of the load instructions. These predictors use the *always allocate* allocation policy; that is, if a prediction-table probe detects a miss in the prediction table, the prediction table will be updated by inserting the load instruction that produced the miss. Consequently, an unpredictable load instruction can evict a predictable one. The use of a more sophisticated allocation policy can filter-out some allocations in the prediction tables to avoid the allocation of unpredictable load instructions. Filtering is valuable to increase the predictability captured by the predictor maintaining its area cost, or to capture the same predictability reducing its area cost.

- Confidence estimators are useful for reducing the amount of address mispredictions; that is, an address prediction only takes place when there is a high confidence in the correctness of the prediction. The reported predictors predict a load instruction only when the load instruction has shown during some executions that it is predictable. Consequently, some executions that would be correctly predicted are not predicted; that is, confidence counters represent a trade-off between the number of right predictions and the number of mispredictions. Unfortunately, when a load instruction is evicted from the prediction table, its confidence information is lost. Then, when this instruction is allocated again in the

prediction table, the confidence estimator must start the estimation of the load instruction from scratch. The use of other confidence mechanisms can reduce the confidence-estimator start-up produced every time a load instruction is allocated in the prediction table.

In this chapter we propose two enhancements to the Last-Address Predictor (LAP), which focus on reducing prediction-table conflicts. Our proposals try to filter-out unpredictable load instructions and to reduce the loss of predictability produced by the confidence-estimator start-up. The first variation is named Filtering by Continuous Classification Last-Address Predictor (CLAP); the second variation is named Filtering by Discrete Classification Last-Address Predictor (DLAP). We have focused on the LAP because it offers a significant performance on integer benchmarks (Section 2.4.2), and address prediction is expected to have a larger impact in integer benchmarks than in floating-point benchmarks (they are less sensitive to load latency than integer benchmarks [APS95]).

We have evaluated the performance of the proposed address predictors in terms on captured predictability, accuracy and area cost, and we have compared them against that of the LAP. To perform these evaluations, we have instrumented the benchmarks (Section A.1.1) and run them until completion.

The evaluations show that our proposals need less area-cost than the LAP to capture the same predictability; CLAP saves around 19% and DLAP saves around 40%. As the DLAP offers better performance than the CLAP, we compare the DLAP against filtering predictors proposed in the literature, assuming direct-mapped and associative prediction tables.

In this chapter we present results in terms of captured predictability, accuracy and area cost. In Chapter 5 we will present IPC evaluations of address prediction.

3.2 Main idea of the filtering strategies

Using a LAP, the confidence information of all the load instructions that conflict at the same AT entry is merged into one confidence counter. This fact can degrade the confidence information; consequently, the number of correct predictions may be reduced and the number of wrong predictions may be increased. To solve this problem, we will try to maintain the confidence information of each load instruction without interference, even for the load instructions that conflict in AT.

This goal is achieved by using a new prediction table, the Classification Table (CT), which records the confidence information of the executed load instructions. The number of CT entries will be larger than the number of AT entries in order to keep the confidence information of load instructions that conflict in AT. To simplify the implementation of the filtering predictor, both the number of AT entries and CT entries should be a two-power number. We will name the *table-size ratio* of the filtering address predictor to the relation between the number of CT entries and the

number of AT entries. Figure 3.1 shows an example where the table-size ratio is four; that is, CT records the classification of up to four load instructions mapped at the same AT entry.

We can use the information recorded in CT to avoid the placement of unpredictable load instructions in AT. Applying this filtering to direct-mapped AT's, conflicts between predictable and unpredictable load instructions in AT are avoided. However, the predictability captured by the predictor is increased only when, in an execution-program context, several unpredictable load instructions and only one predictable load instruction are mapped at the same AT entry. Moreover, this information will be used to initialize the confidence estimator of the predictor. Consequently, the use of the CT will reduce the miss rate related to predictable load instructions.

Updating the information recorded in CT is a critical aspect of this proposal. The main difference between the two filtering techniques proposed in this chapter is how CT is updated.

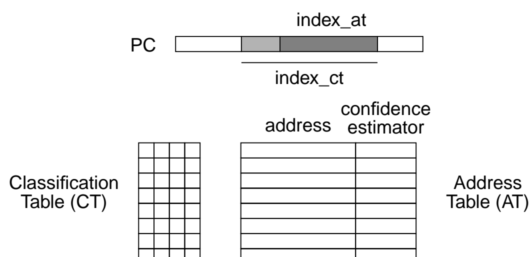


Figure 3.1 Main idea of the filtering strategies

3.3 Benchmark characterizations

This section presents some characterizations of the SPEC95-INT benchmarks. These characterizations will be used to design the filtering techniques presented in this chapter, and to decide the evaluated range of prediction-table capacities. First, in Section 3.3.1 an analysis of how the address predictability is distributed among the instructions of the benchmarks is presented. Then, in Section 3.3.2 the working-set size of load instructions of each benchmark is evaluated. Finally, in Section 3.3.3 the bursts of accesses that are hits in direct-mapped prediction tables are examined.

3.3.1 Address-predictability distributions

We present an analysis of the contribution of the load instructions to the overall address predictability captured by a LAP with an unbounded Address Table and no confidence mechanism. For each load instruction, we have evaluated its predictability (defined as the percent of correctly predicted executions out of the number of executions). After that, we have grouped load instructions in ranges of predictability. Figure 3.2-a presents the distribution of the static load instructions according to their predictability. It fades out from the highly predictable load instructions (90%-100% range, at the bottom of each bar), to the highly unpredictable ones (0%-10% range, at the top of each bar). We can observe that between 25% (*perl*) and 60%

(*ijpeg*) of static load instructions can be classified as highly predictable, and between 20% (*go*) and 45% (*perl*) as highly unpredictable.

To show the contribution of each predictability range on the number of executed load instructions, every static load instruction has been weighted with its execution frequency; Figure 3.2-b shows this distribution. In some benchmarks, highly predictable and highly unpredictable load instructions represent almost all executed loads (*compress*, *perl*). In other benchmarks, load instructions with a medium predictability account for a significant proportion of the number of executed load instructions (*go*, *vortex*). The static and the dynamic load-instruction distributions can differ significantly (for instance, in benchmark *ijpeg*, 60% of static load instructions are highly unpredictable, but they only represent 16% of executed load instructions).

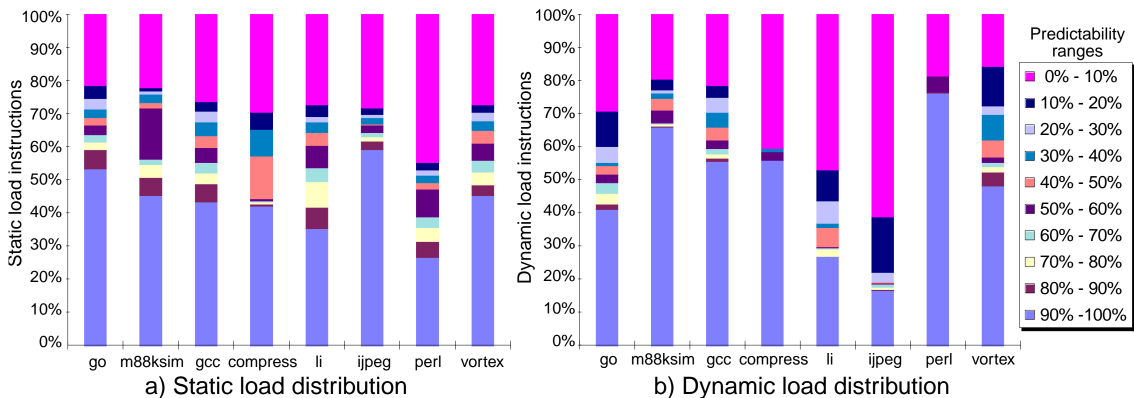


Figure 3.2 Static (a) and dynamic (b) load-instruction distribution according to their address predictability by a Last-Address Predictor

The absolute contribution of every predictability range to the captured predictability show us the significance of medium-predictable load instructions to the predictability. Left bars related to benchmarks in Figure 3.3 show the contribution of every load-instruction range to the predictability captured by the Last-Address Predictor. The main contribution is made by static load instructions in range 90%-100%. They account for between 82% and 97% of the overall predictability. The static load instructions in range 70%-90% represent a small contribution. They account, at most, for 8% of the overall predictability. Remaining overall predictability is produced by static load instructions that belong to lower predictability ranges; for instance, the contribution of load instructions in the 10%-60% range to the overall predictability varies from 6.5% to 2%.

Influence of load-instruction classification on predictability distributions

We use two-bit saturated counters as a dynamic classifying mechanism of load instructions. A counter is assigned to each prediction-table entry. When a load instruction computes the same effective address as in its previous execution, the counter will be increased by one, otherwise it will be decreased by one. We classify a load instruction as predictable if its saturated-counter value is greater than one; otherwise, it is classified as unpredictable. This classification can be

used as a confidence estimator, that is, to predict a load instruction it must be classified as predictable.

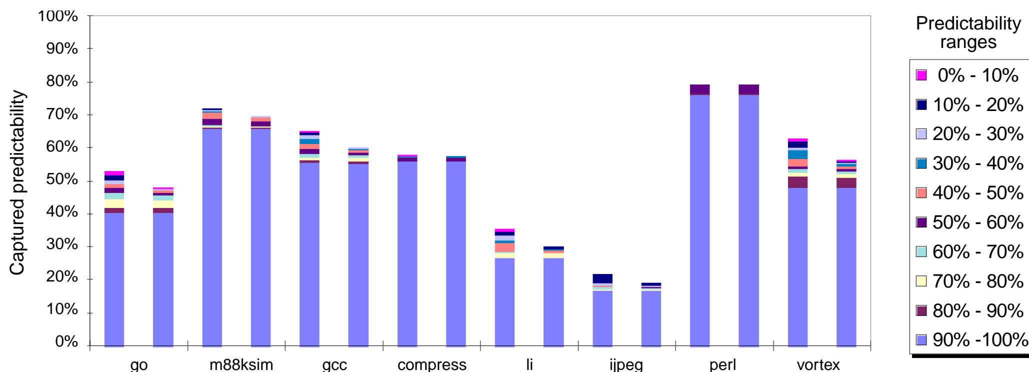


Figure 3.3 Predictability captured by the LAP without confidence estimation (left bar) and by the LAP with confidence estimation (right bar) distributed in address-predictability ranges.

Classifying counters detect load instructions that, on a burst consecutive executions, compute the same address; these bursts will be called predictable bursts. When a predictable burst is detected, the load instruction is classified as predictable and, on subsequent executions, it will be predicted. When the predictable burst finishes, the instruction is classified as unpredictable until a new predictable burst is detected. As the classifying predictor needs some executions of the load instruction to detect a predictable burst, some predictability of the load instruction is not captured. Moreover, when a short predictable burst is detected, it can be over or almost finished. Short predictable bursts are mainly produced by load instructions with medium or low predictability.

Two-bit saturated counters classify correctly highly predictable and highly unpredictable load instructions, but the ones with medium or low predictability may be classified wrongly. Then, the predictability captured by the address predictor may be reduced. Using unbounded tables, Figure 3.3 shows the predictability captured by the LAP without confidence estimation (left bar) and by the LAP with confidence estimation (right bar) distributed according to the load-instruction predictability ranges. The decrease in the captured predictability attributable to the classifying mechanism is related to the contribution of load instructions with medium or low predictability to overall predictability, because the classifying predictor is not able to fully capture their predictability. This reduction can account for up to 15% of the captured predictability.

In summary, we have shown that the predictability of load instructions do not spread uniformly among them; that is, a significant number of static load instructions are highly predictable or highly unpredictable. This non-uniform distribution is not considered by address predictors that use the always allocate policy because unpredictable load instructions can be allocated in AT. These allocations do not contribute to the predictability captured by a predictor and, moreover, can damage some predictability if these allocations evict predictable load instructions. Consequently, filtering the allocation of unpredictable load instructions in AT can be profitable to

increase the predictability captured by the predictor or to reduce its area cost, because we can capture as much predictability using less AT entries; that is, capacity misses are reduced. In this chapter, we will present a mechanism that records confidence information of the evicted load instructions as well as guiding the replacement algorithm in AT.

3.3.2 Working-set size of static load instructions

In order to dimension properly the amount of prediction-table entries used in the evaluations performed in this chapter, we have computed the working-set size of static load instructions of each benchmark.

First, we have evaluated the miss rate of the LAP using both direct-mapped and fully associative mapping policies (the latter with the Least Recently Used (LRU) replacement policy). The miss rate of a LAP is defined as the percentage of executed load instructions which tags are not present in AT. Figure 3.4 shows the results obtained in the SPEC95-INT benchmarks; the vertical axis stands for the miss rate and the horizontal axis stands for the number of AT entries.

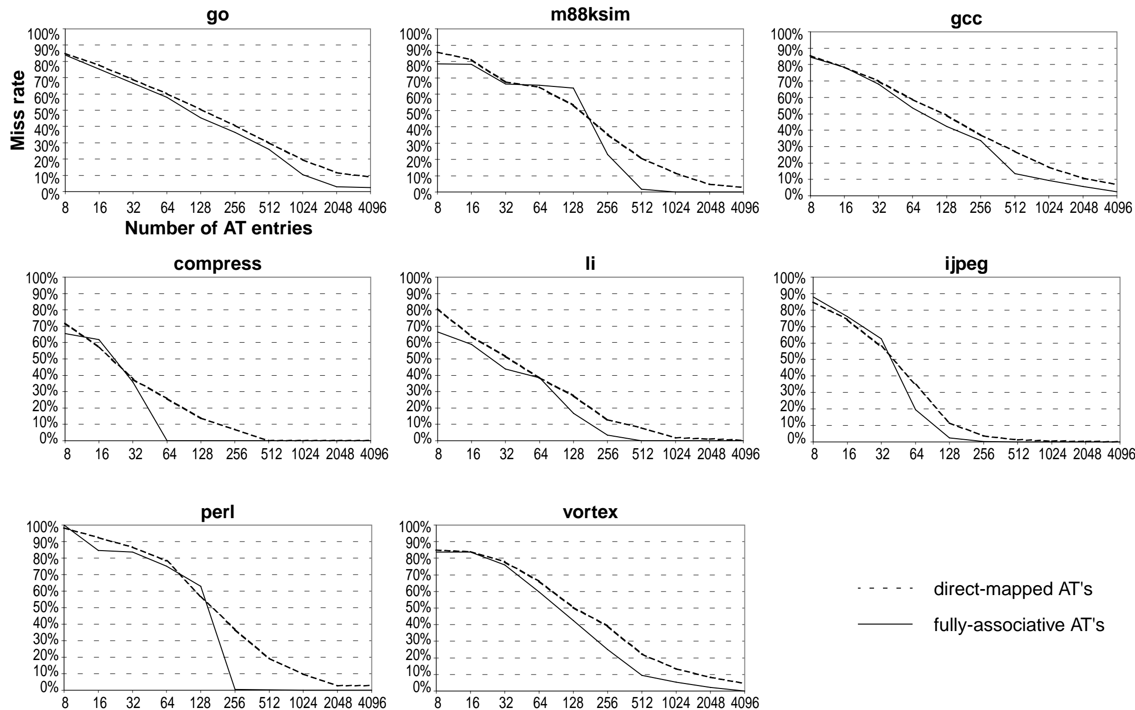


Figure 3.4 Miss rates in the LAP (using direct-mapped and fully associative AT's)

From these results, we made the following observations:

- The presence of benchmarks that need a very large AT to achieve a low miss rate. For instance, in benchmark *go,* a 1.024-entry fully associative AT or a 2.048-entry direct-mapped AT are needed to achieve a 10% miss rate.

- The existence of a significant amount of conflicts due to direct mapping. For instance, in benchmark *m88ksim*, a 512-entry fully associative AT exhibits a 3% miss rate while the direct-mapped AT exhibits a 20% miss rate.

After that, we define the *working-set size of static load instructions of a benchmark* as the minimum two-power number of AT entries needed to achieve a 1% miss rate, at most, in a fully associative AT with LRU replacement policy. Table 3.1 shows a classification of all the analysed benchmarks according to their working-set sizes.

Class	Benchmark	Working-set size of static load instructions
Small	*compress*	≤ 128
Medium	*li, ijpeg, perl*	256 - 512
Large	*m88ksim*	1.024
Extra-Large	*go, gcc, vortex*	≥ 2.048

Table 3.1 Benchmark classification according to their working-set sizes of static load instructions.

The working-set size of a benchmark is related to the number of AT entries needed by a LAP configuration to achieve a performance similar to the one obtained using an unbounded AT. In previous evaluations ([BMP+98][CRT99][GoGo97b][LWS96][RFKS98]), typical prediction-table sizes ranges from 1.024 to 4.096 entries and, in some cases, mapping is four associative. These sizes are big enough to capture the whole working set of load instructions of most SPEC95-INT benchmarks. Moreover, associativity reduces conflict misses in prediction tables but their access time can be a restriction. In this chapter, we also evaluate smaller table sizes to know the behaviour of our proposed predictors when they are pressured by capacity and conflict misses, because the goal is filtering the allocation of unpredictable load instructions in the prediction table.

3.3.3 Hit-bursts distributions

We define the *hit-burst length* as the number of consecutive executions of a static load instruction that are hits in AT. A hit burst of length N involves $N+1$ consecutive executions of the load instruction: the first one produces the allocation of the load instruction in AT (a miss in AT), and the remaining N executions are hits in AT.

On a replacement in an AT entry, its confidence-counter value is initialized to one. Consequently, the next execution of the allocated load instruction will not be predicted, and will be only used to modify its confidence-counter value. The loss of predictability due to this initialization can be large if the eviction of a load instruction from AT after few executions of this load instruction is usual. For instance, consider a 100%-predictable load instruction, and let its hit-burst length be two executions. The first execution of the load instruction will be a miss in AT,

so the LAP will allocate the load instruction in AT and will set to one its confidence-counter value. The first hit of the burst will not be predicted, but will increase the confidence-counter value, and the second (last) hit of the burst will be predicted. Consequently, the predictor will be able to predict only one of the two executions of the hit burst; that is, the loss of predictability will be 50%. On the other hand, consider the same load instruction with a hit-burst length of 10 executions. In this case, the loss of predictability will be 10%.

We have evaluated the number of occurrences of every hit-burst length, and they have been weighted by their number of involved executions. Figure 3.5 shows cumulative distributions of the hit-burst lengths in SPEC95-INT benchmarks using direct-mapped AT's. The vertical axes stand for the percent of dynamic load instructions and the horizontal axes stand for the hit-burst length, and every graph is related to a number of AT entries. The horizontal axis is cut at hit-burst length 18, but all graphs saturate at 100%. As the number of AT entries increases, the number of misses decreases, and the hit-burst lengths become larger.

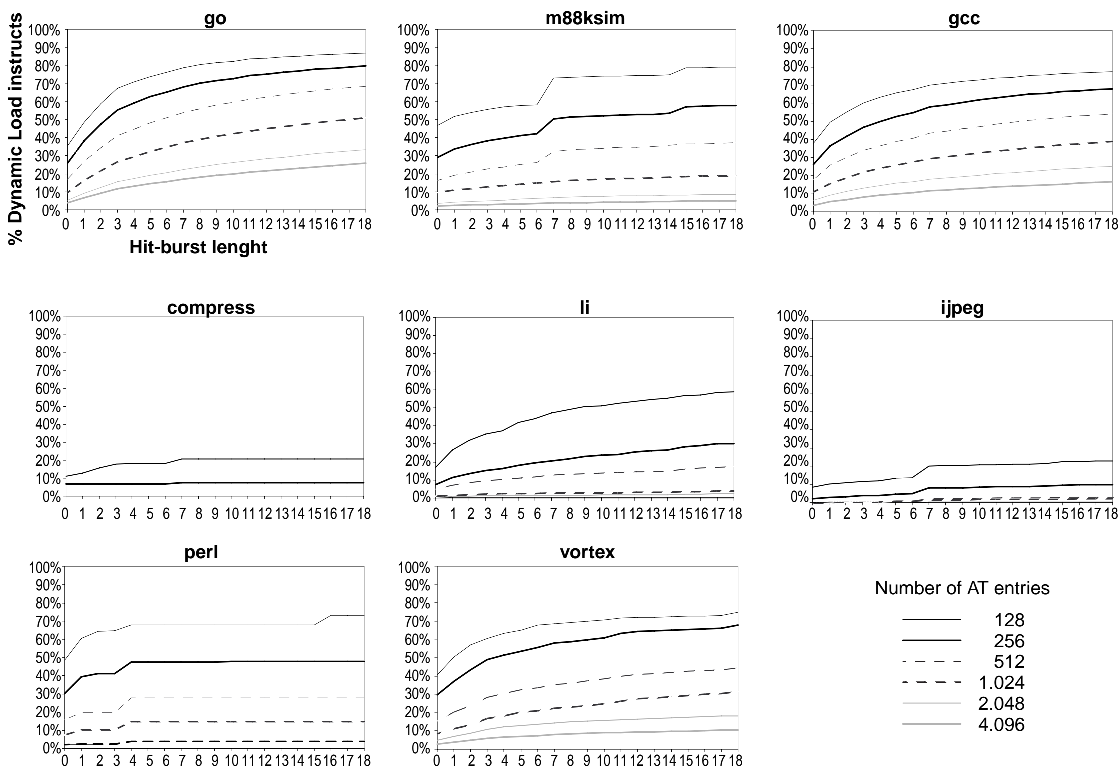


Figure 3.5 Cumulative dynamic-load-instruction distributions according to hit-burst lengths

We have observed that a significant amount of dynamic load instructions is related to short hit bursts. For instance, in benchmark *go*, for a 1.024-entry direct-mapped AT, a 29% of dynamic load instructions are related to hit bursts of, at most, length four. To capture the potential

predictability of these short hit bursts, an address predictor must be able to predict the load instructions as soon as they are allocated in AT.

Moreover, we have noticed that the percentage of dynamic load instructions related to hit bursts of length zero is significant in some benchmarks and configurations. For instance, in benchmarks *m88ksim* and *go* using a 1.024-entry AT, they represent about 10% of the dynamic load instructions. Zero-length hit bursts are generated by load instructions that are allocated in AT and, before their next execution, they are evicted from AT.

Although several benchmarks exhibit a similar miss rate, their hit-burst length distributions can present significant variations. For instance using direct-mapped AT's, the 256-entry AT in benchmark *li*, the 1.024-entry AT in *m88ksim* and the 2.048-entry AT in *go* exhibit around a 12% miss rate, but their hit-burst distributions differ significantly. Table 3.2 shows the percent of dynamic load instructions related to hit bursts of length zero, up to four and up to ten executions in these benchmarks. For the same miss rate, hit-burst lengths tend to be shorter in benchmark *go* than in benchmark *m88ksim*. For instance, the percentage of dynamic load instructions that are related to hit-burst lengths shorter than or equal to four instructions in benchmark *go* is 17.21%, while in benchmark *m88ksim* it is 13.39%.

		256-entry *li*	1.024-entry *m88ksim*	2.048-entry *go*
hit-burst length	0	7.54	9.4	5.18
	≤ 4	16.41	13.39	17.21
	≤ 10	23.65	16.89	26.16
Miss rate (direct-mapped AT)		12.69	11.63	11.72

Table 3.2 Percentage of dynamic load instructions related to hit-burst of length zero, up to four and up to ten executions, and miss rates in some LAP configurations and benchmarks

To exploit fully the predictability available in short hit bursts of predictable load instructions, we propose recording confidence information of the evicted load instructions. This information will be used to initialize the confidence-counter value of the AT entries when the load instructions are re-allocated in AT.

3.4 Filtering by means of a continuous classification

This section develops the first technique proposed to reduce prediction-table size of a LAP. First, in Section 3.4.1, a mechanism for classifying continuously load instructions according to their predictability is proposed. In Section 3.4.2, the Filtering by Continuous Classification Last-Address Predictor (CLAP) is introduced, a predictor that uses the previous classifier to avoid the allocation of unpredictable load instructions in the Address Table. Finally, in Section 3.4.3 the evaluation of this predictor is presented.

3.4.1 Dynamic classification: *<N, k>* classifying mechanism

All bits of the effective addresses are needed to perform a memory access, but they are not needed to classify a load instruction. We propose the use of few bits of the effective addresses as input of the load classifier.

To classify a load instruction as non-predictable by a LAP, it is sufficient to detect that one bit of the effective addresses computed in two consecutive executions of a load instruction differs. Consequently, a few bits of the computed addresses (*N* bits, for instance the least-significant bits) can be enough to classify load instructions. If one of these bits is different, the classification will be correct (equal to the classification performed comparing all bits of the computed addresses). Wrong classifications will be produced, for instance, if addresses are in arithmetic progression and the analysed bits are not modified (a load instruction with a large stride); this load instruction will be classified as predictable but it is not.

Furthermore, the mapping of language data types into architectural data types performed by the compiler can cause some low-order bits of the computed addresses to be non-significant for classifying most load instructions. For instance, paddings performed by the compiler to align fields of data structures.

Then, we propose a classifying mechanism named <*N*, *k*>, where *N* is the number of bits used to classify load instructions, and *k* is the number of low-order bits discarded from the computed addresses. That is, the classifier skips the *k* low-order bits of the computed addresses and then selects the *N* low-order bits.

To evaluate the <*N*, *k*> classifying mechanism, we compared it against the classifying mechanism that uses all bits of the effective address (*<64, 0>*). We define the *similarity of the <N, k> classifying mechanism* as the percentage of coincidences of the classifying counters of *<N, k>* and *<64, 0>* (every time a load is executed, the classifying counter values related to this load are checked) out of the number of executed loads.

In this work we have used load-instruction traces taken from an 21264 Alpha-AXP processor. Since the fundamental unit of data of Alpha architecture is 8 bytes [Bhan96], we have also evaluated the discarding of the 3 low-order bits of the effective addresses for classifying load instructions.

Figure 3.6 shows the average similarity in the SPEC95-INT benchmarks of two *N*-bit mechanisms: no-skipping (*<N, 0>*) and 3-bit skipping (*<N, 3>*). Mechanism *<N, 3>* does not take advantage of selecting more than five bits due to eight-byte un-aligned computed addresses. Its similarity graph is saturated at 90%. To achieve a similarity greater than 90%, the three low-order bits must be selected as shown in the graph of *<N, 0>*.

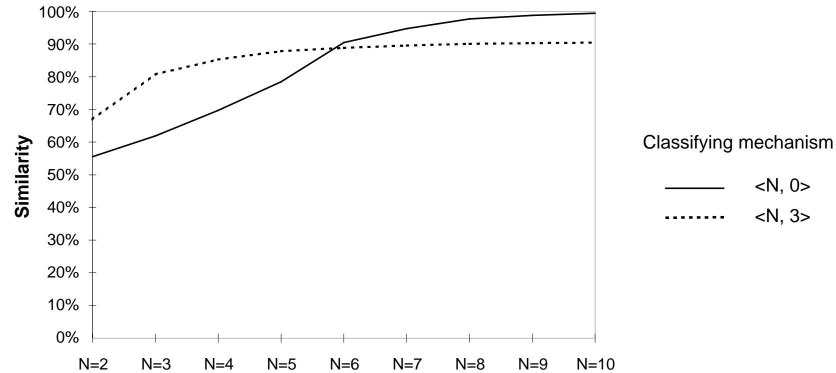


Figure 3.6 Average similarity of *<N, 0>* and *<N, 3>* classifying mechanisms

Cases *<3, 3>* and *<4, 3>* obtain a high similarity (over *80%)*. Moreover, *<4, 3>* achieve almost the same similarity as *<6, 0>*. We will use the *<3, 3>* classifier mechanism in our proposed predictor; the similarity of this classifier is about 80%. To improve the similarity of the proposed classifier, the operation code of load instructions can be used to decide dynamically the number of skipped bits. However, we do not use this improvement in this work.

Figure 3.24 in Section 3.8.1 presents the similarity for each SPEC95-INT benchmark.

3.4.2 Filtering by Continuous Classification Last-Address Predictor (CLAP)

This section describes a predictor mechanism with run-time classification. It takes advantage of two considerations:

- As we have shown in the previous section, a few bits of the effective addresses are enough to classify load instructions precisely.

- Only when a load is predictable is it necessary to record the whole effective address accessed by the load instruction

Following these considerations, we propose to split the Address Table of the LAP into two tables: the Classification Table (CT) and the Address Table (AT). CT is used to classify dynamically load instructions according to their predictability. AT stores the last effective addresses computed by predictable load instructions. Figure 3.7 shows a diagram of the mechanism. It will be named *Filtering by Continuous Classification Last-Address Predictor* (*CLAP*).

To classify load instructions dynamically, the predictor uses the <*N, k*> mechanism described in Section 3.4.1. CT is direct mapped and each entry contains two fields: a two-bit saturated counter, and *N* bits of the effective address. The counter is used to classify load instructions continuously; that is, each executed load updates the CT.

AT is also direct mapped and each entry contains a complete effective address and a *ct_tag*; this tag identifies the CT entry related to an AT entry.

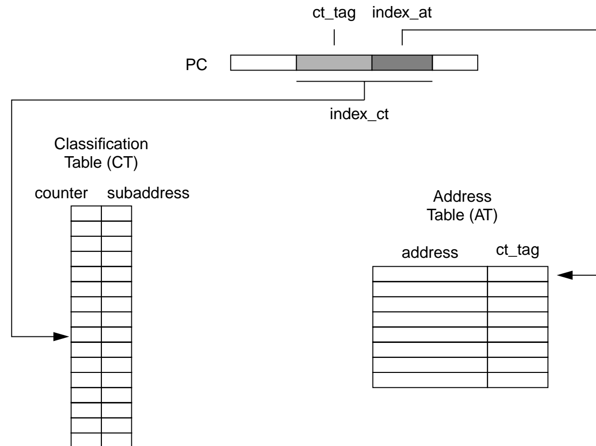


Figure 3.7 Diagram of the Filtering by Continuous Classification Last-Address Predictor (CLAP)

The proposed predictor avoids the placement of highly unpredictable load instructions in the AT using the information recorded in CT. The placement of load instructions in AT are filtered using CT: their saturated counter must be greater than 1. This filtering allows predictable load instructions to continue being placed in AT, and provides more chances to exploit their predictability. Figure 3.8 compares the decision tables applied by the LAP and by the CLAP to decide if a load instruction that misses in the AT must be allocated in AT. The LAP applies the always allocate replacement policy, and the CLAP filters some allocations by considering the information obtained from the CT.

		Allocated instruction	
		unpredictable	predictable
Missing instruction	unpredictable	Replace	
	predictable		

a) LAP

		Allocated instruction	
		unpredictable	predictable
Missing instruction	unpredictable	Don't replace	
	predictable	Replace	

b) CLAP

Figure 3.8 Decision tables applied by the replacement algorithms of the LAP and by the CLAP

The predictor works as follows: When a load instruction is fetched, the appropriate CT and AT entries are selected, and the *ct_tag* field is checked to determine if the AT entry is related to this CT entry. If so, the counter value in CT is used to decide if the load is predicted, otherwise it is not predicted. The procedure *Prediction* in Figure 3.9 depicts the pseudo-code related to these actions.

The prediction tables are updated after the address stage of the pipeline (procedure *Update* in Figure 3.9 shows its pseudo-code). On an AT hit, the selected entry is updated using the computed effective address. On an AT miss, the counter value in CT entry is checked to decide if the current AT entry is to be replaced.

```
/* Predicts an effective address
   Output variable:
      -pred: predicted address
*/

void Prediction(PC) {
   index_at = INDEX_AT(PC);
   index_ct = INDEX_CT(PC);
   ct_tag = CT_TAG(PC);
   if ((AT[index_at].ct_tag == ct_tag) &&
       (CT[index_ct].counter > 1))
      pred_addr = AT[index_at].address;
}
```

```
/* Updates CT and AT */

void Update(PC, address) {
   index_at = INDEX_AT(PC);
   index_ct = INDEX_CT(PC);
   ct_tag = CT_TAG(PC);
   subaddress = SELECT_N_BITS(address);
   if (AT[index_at].ct_tag == ct_tag) {
      if (AT[index_at].address == address)
            CT[index_ct].counter ++;
      else  CT[index_ct].counter --;
      AT[index_at].address = address;
   }
   else  {
      if (CT[index_ct].subaddress == subaddress)
            CT[index_ct].counter ++;
      else  CT[index_ct].counter --;
      if (CT[index_ct].counter > 1) {
            AT[index_at].ct_tag = ct_tag;
            AT[index_at].address = address;
      }
   }
   CT[index_ct].subaddress = subaddress;
}
```

Figure 3.9 Pseudo-code of the Filtering by Continuous Classification Last-Address Predictor (CLAP)

The following expression shows the area cost of the CLAP as a function of the number of table entries. We have assumed 64-bit effective addresses, 2-bit confidence counters and the <3, 3> classifying mechanism.

$$AreaCost_{CLAP} = \text{CT_entries} \times (2+3) + \text{AT_entries} \times \left(64 + \log_2\left(\frac{\text{CT_entries}}{\text{AT_entries}}\right)\right)$$

3.4.3 Evaluation results

In this section an evaluation of the CLAP compared versus the LAP is presented. Their miss rate, area-cost, captured predictability and accuracy have been compared.

Miss rate in the prediction tables

Using the LAP, every load instruction is allocated in the AT. However, using the CLAP, only a portion of all load instructions are allocated in AT, because CT filters-out the allocation of some load instructions. Thus, we expect this filtering to reduce the amount of capacity misses in AT comparing a LAP and a CLAP with the same AT size.

To measure this reduction, we evaluated the miss rate of both predictors. The miss rate of the LAP is the percentage of load instructions that miss in AT out of the number of executed load instructions. On the other hand, two cases produce a miss in the CLAP: a) a load instruction that misses in CT, and b) a load instruction that hits in CT, which is classified as predictable by CT, and which misses in AT. The miss rate of the *CLAP* is the percentage of misses in its tables out of the number of executed load instructions.

First, we have evaluated the miss rate of both predictors when they are implemented using fully associative AT's with LRU replacement policy, unbounded CT's, and the *<64, 0>* classifying mechanism. To compute both miss rates we have employed fully tagged AT tables. These miss rates can be assumed as capacity miss rates. Figure 3.10 shows the miss rate of some configurations of the LAP and the CLAP in the SPEC95-INT benchmarks; the horizontal axis represents the number of AT entries, and the vertical axis shows the miss rate.

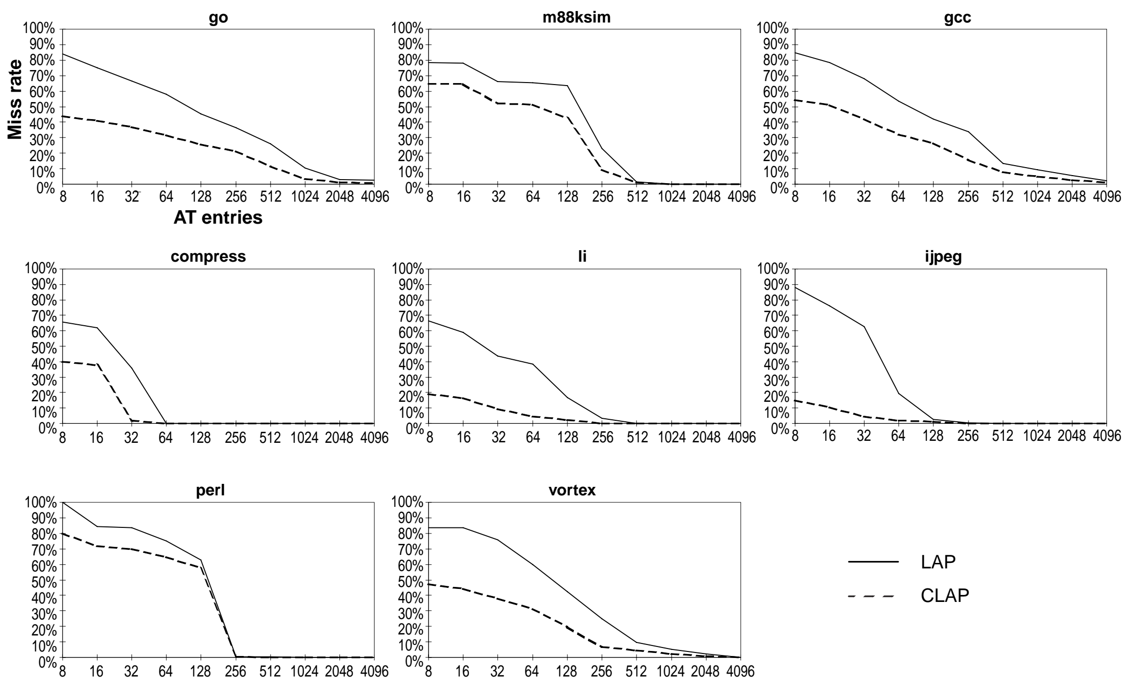


Figure 3.10 Miss rate in the LAP and the CLAP in SPEC95-INT benchmarks (AT's are fully associative with LRU replacement policy, CT's are unbounded)

From Figure 3.10, when the same number of effective addresses are stored in AT, the CLAP has a lower miss rate than the LAP because CT filters the load instructions that can be placed in AT. As AT's become larger, the miss-rate reduction is less significant due to the increment of capacity in both tables. Similar miss rates are achieved using a CLAP with half AT-entries than an LAP. For instance, in benchmark *go*, the CLAP with AT-entries=512 and a LAP with AT-entries=1.024 achieve about a 10% miss rate.

Next, we have evaluated the influence of the mapping policy and the number of CT entries on the miss rate; AT's and CT's will be direct mapped. Bounded and direct-mapped CT's increase the miss rate of the CLAP due to capacity and conflict misses in CT.

Figure 3.11 displays the miss rate of some configurations of the LAP and the CLAP in SPEC95-INT benchmarks for several table-size ratios (CT entries/AT entries). The horizontal axis represents the number of AT entries (we focus on AT's with at least 128 entries); the vertical axis shows the miss rate. A configuration with table-size ratio equal to 1 is a degenerated CLAP configuration; then the minimum analysed ratio is 2 and the maximum is 16. In addition, in Figure 3.11 the miss rate for unbounded CT tables is shown.

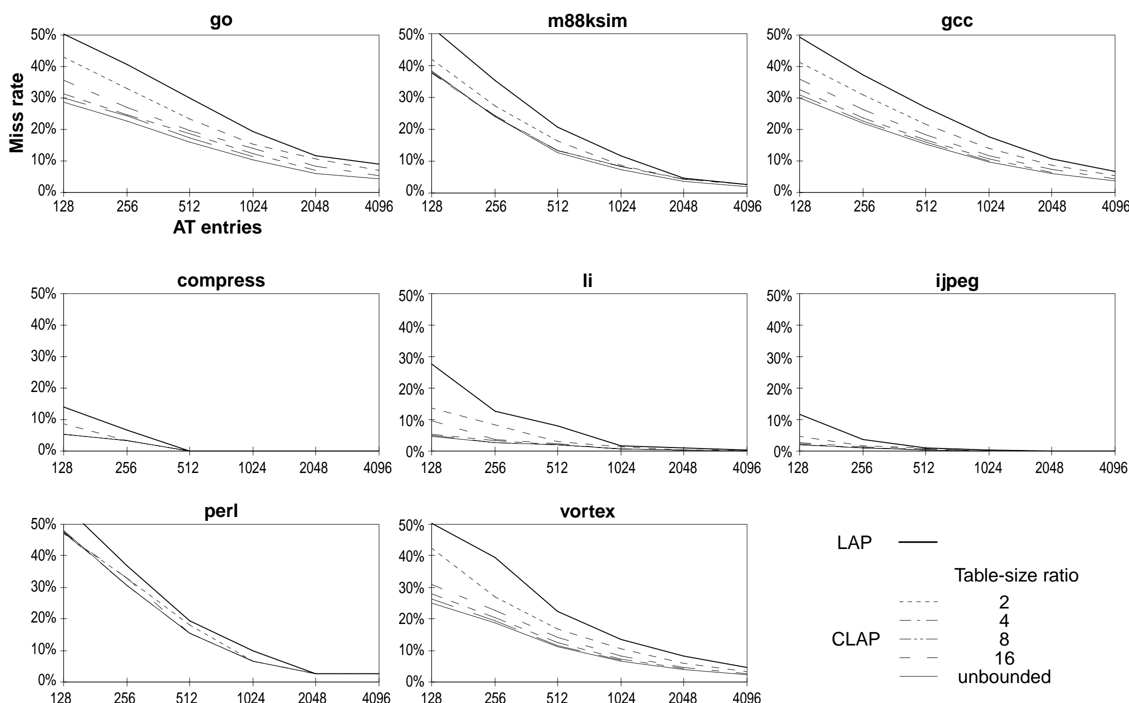


Figure 3.11 Miss rate in the LAP and the CLAP in all benchmarks (direct-mapped AT and CT)

For benchmark *go*, the CLAP with a table-size ratio equal to 2 shows a miss reduction of about 20% in the whole range of evaluated configurations. Conflicts in CT between unpredictable load instructions do not influence classification, but conflicts between predictable load instructions classify CT entry as unpredictable. If the ratio is increased, a finer classification and an additional miss-rate decrease can be achieved. That is, the classification will be improved when at least one of two load instructions that previously collide in CT entry is predictable. However, it improves the miss rate when only one of the load instructions that collide in AT entry is predictable. Furthermore, for large ratios, the reduction in misses is gradually fewer. There are a few conflicts in CT but remain conflicts in AT between predictable load instructions.

Area Cost

Area distribution between the LAP and the CLAP is very different. CLAP saves area for storing addresses compared with LAP. This area saving is used to classify the load instructions more finely. In that case, the classification improves the utilization of the AT entries.

We will use CLAP configurations with an area cost smaller than the area cost of the LAP configuration with twice AT entries. From among these configurations, we select the configuration with bigger CT size to obtain the finest possible classification. From area-cost expressions, we obtain a table-size ratio equal to 8 (CT entries/AT entries=8). This CLAP configuration represents an area-cost saving of 19% compared with the LAP.

Captured Predictability

We will compare the predictability of the CLAP and the LAP for several configurations. Unbounded CT's obtain the maximum miss-rate reduction compared with the LAP. For benchmark *go* (Figure 3.11), proposed configurations of the CLAP achieve about 80% of the maximum miss reduction. For the other benchmarks, this reduction is similar considering the working-set size of load instructions. Furthermore, since the biggest working-set size of load instructions in analysed benchmarks is about 8K load instructions, we limit the CT size in our evaluation to 8K entries.

We will name the CLAP configurations as {AT size, CT size}. From previous observations we select the following configurations: {128, 1.024}, {256, 2.048}, {512, 4.096}, {1.024, 8.192} and {2.048, 8.192}. From Figure 3.6 we select the *<3, 3>* classifying mechanism; this mechanism achieves a 80% similarity.

Figure 3.12 shows the predictability captured by every predictor configuration. The horizontal axes represent area cost (using a logarithmic scale), while the vertical axes show captured predictability. In this section we present results only for large and extra-large benchmarks (Table 3.1). The other benchmarks show similar behaviour when working-set size of load instruction is considered (Figure 3.25 in Section 3.8).

First, we can compare a CLAP configuration versus the LAP configuration with twice AT entries (they are represented by a cross and the first plus sign with higher area cost). We can observe that, except for *m88ksim*, the CLAP configuration captures more predictability than the LAP; that is, CT allows a more efficient use of the AT entries because only predictable instructions are allocated in AT.

The behaviour of *m88ksim* benchmark is due to the fact that most of its load instructions are highly predictable. Consequently, the CT will classify them as predictable, and no filtering will be performed.

From Figure 3.11, the absolute miss-rate difference between the CLAP with unbounded CT and the LAP is bigger for lower AT sizes. This potential increase in performance is observed in

Figure 3.12, which shows more significant increases of predictability for small AT's. Furthermore, for large AT's, the miss rate is lower; consequently, the miss-rate reduction obtained by the CLAP represents a small captured-predictability increment.

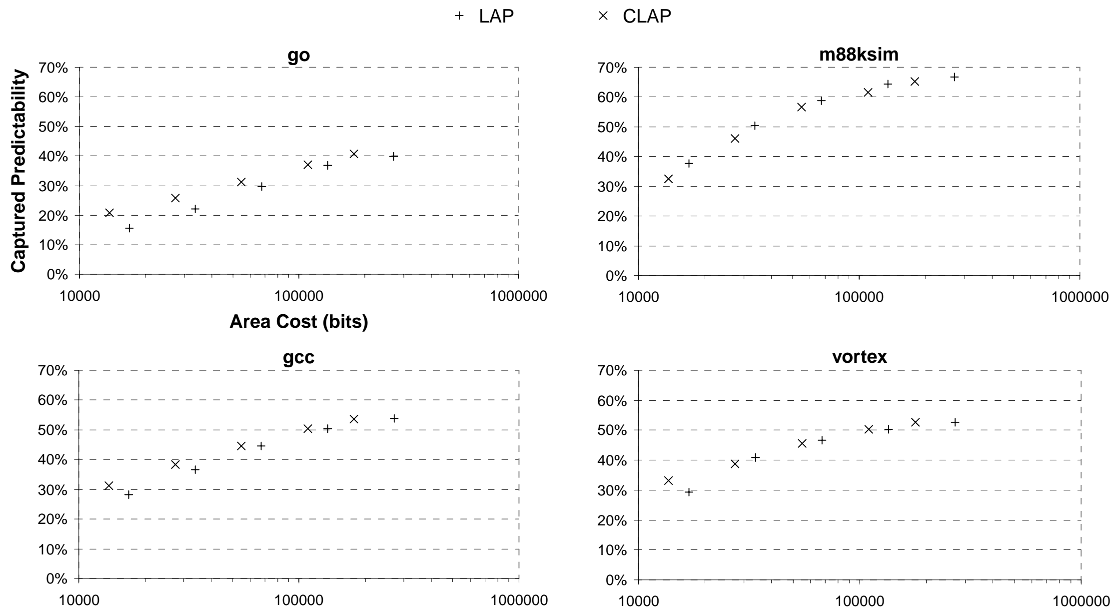


Figure 3.12 Predictability captured by the LAP and by the CLAP in large and extra-large benchmarks

Differences in predictability smaller than 3% in almost all the benchmarks are observed between CLAP configurations and a LAP configuration with twice AT entries. Moreover, the CLAP configuration needs only 81% of the area-cost needed by the LAP configuration.

Accuracy

Another advantage of the CLAP concerns the amount of mispredictions. Address predictors are designed to capture as much address predictability as possible, and they should also mistake the minimum number of predictions because every misprediction could have a penalty of some processor cycles.

To compare the correctness of the predictors, we evaluated their accuracy (Section 2.4.2). Figure 3.13 shows the accuracy achieved by the LAP and by the CLAP in selected benchmarks.

In almost every benchmark, any CLAP configuration achieves a higher accuracy than the most accurate LAP configuration. This is due to several factors: a) the classification performed by the CLAP is more precise than the one performed by the LAP due to the bigger number of classifying counters, and b) as AT is partially tagged, the CLAP can detect some conflicts in AT between CT entries, preventing probable mispredictions.

Moreover, the accuracy of the LAP is sensitive to AT size, while the accuracy of the CLAP is almost independent of AT size. Comparing the {256, 2.048} CLAP to the 256 AT entries LAP, the accuracy is increased from 8% (*vortex*) to 16% (*go*); for bigger AT's, the difference between their accuracy decreases.

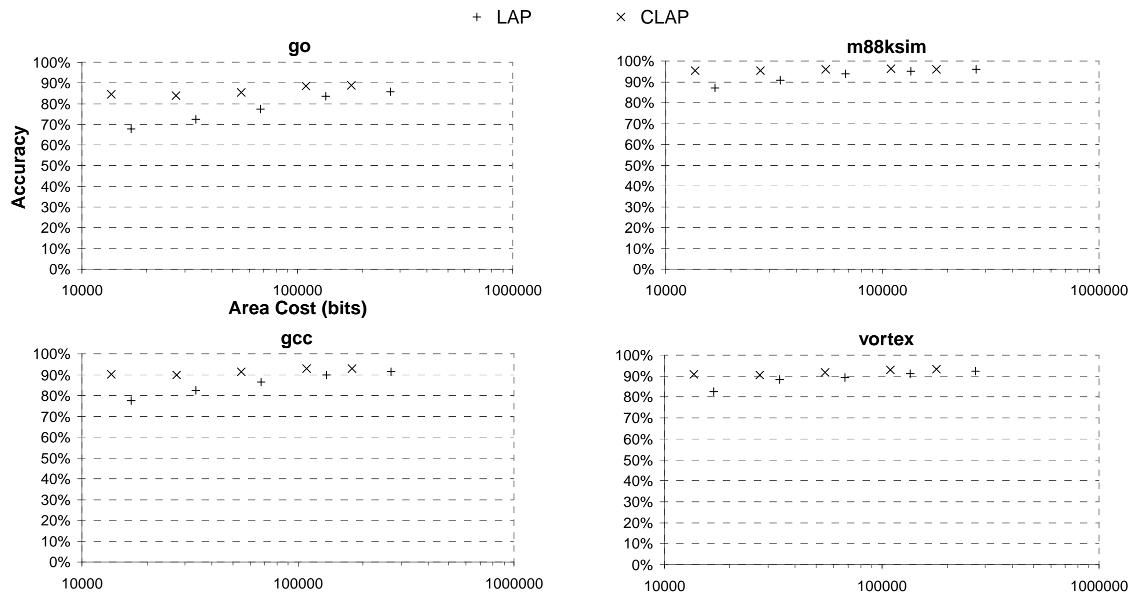


Figure 3.13 Accuracy achieved by the LAP and by the CLAP

3.5 Filtering by means of a discrete classification

This section presents the second technique for reducing prediction-table size by filtering-out some allocations in the prediction table. The main difference from the previous technique consists in the classification performed by the CT. This difference will also affect the replacement algorithm in AT.

This section is organized as follows. First, in Section 3.5.1 the discrete classification is described. Next, in Section 3.5.2 the Filtering by Discrete Classification Last-Address Predictor (DLAP) is introduced. Finally, in Section 3.5.3 a performance evaluation of the predictor is presented.

3.5.1 Discrete classification

In this subsection, we present a mechanism that classifies load instructions discretely, that is, the classification of a load instruction will not be updated by all the executions of this load instruction. We will detail the main characteristics of the classifier.

Recording classification information of the evicted load instructions

From hit-burst length evaluations (Section 3.3.3), we suggest the addition of a Classification Table (CT) to the LAP to record information of the evicted load instructions. The number of CT entries will be larger than the number of AT entries. CT will be indexed using the PC of the load instruction.

When a load instruction is evicted from AT, we classify it as unpredictable if its confidence-counter value is zero or one, otherwise it is classified as predictable. This classification is recorded in the CT. When a load instruction is re-allocated in AT, its confidence-counter value is set to one if the load instruction is classified as unpredictable by CT; otherwise, the confidence-counter value is set to two. These initializations represent a trade-off between captured predictability and accuracy, and allow the predictor to exploit the predictability available in the short hit-burst lengths.

Filtering the allocation in AT by means of the CT

Predictability analysis shows that some load instructions are highly unpredictable. The allocation of an unpredictable load instruction in AT can evict a predictable one, producing a decrease in the captured predictability. Consequently, we propose avoiding the allocation in AT of unpredictable load instructions. Our goal is to reduce the capacity misses in AT related to predictable load instructions.

Applying this filtering to direct-mapped AT's, conflicts between predictable and unpredictable load instructions in AT are avoided. However, the predictability captured by the predictor is increased only when, in an execution-program context, several unpredictable load instructions and only one predictable load instruction are mapped at the same AT entry.

We will classify dynamically the static load instructions as predictable or unpredictable. For the load instructions allocated in AT, we will use the current confidence-counter value to classify them. For the load instructions not allocated in AT, we will use the information recorded in CT. Note that this classification is discrete and dependent on the number of AT entries. Since the classification of a load instruction will be updated only when it is allocated in AT, a hit-burst length larger than zero is required to update the classification. On a zero-length hit burst, the load instruction will maintain its previous classification.

First, we have evaluated the same filtering as that of the CLAP; that is, a load instruction classified as unpredictable by CT will not replace a load instruction classified as predictable by AT. In any other case, the replacement will be performed.

Using unbounded CT's and initializing all the CT entries as predictable, our evaluations show that the discrete classification and the previous replacement algorithm is rough for medium-predictable load instructions. Reclassification is obstructed by predictable load

instructions allocated in AT that have not been executed for a long time, and the predictor captures less predictability than the LAP.

In addition, to give all the load instructions a chance to be classified, they must be allocated at least once in AT. Using a bounded CT, in different program-execution contexts, load instructions with a different predictability may be mapped at the same CT entry. Consequently, we need a mechanism to reclassify a load instruction classified by CT as unpredictable, since this classification may be due to a collision at the same CT entry.

Re-evaluating the classification of load instructions classified as unpredictable

To re-evaluate the classification of a load instruction, it must be allocated in AT. The previous replacement algorithm only allocates an unpredictable load instruction when it collides with another load instruction classified as unpredictable, but this is not enough.

An opportunity to reclassify load instructions classified as unpredictable by CT is provided when they collide in AT with a predictable load instruction that has not been executed for a long time; that is, in a change of program-execution context.

We then use a saturated collision counter at every AT entry. These counters reflect the execution ratio between the load instructions. The collision-counter value is decreased on AT hits, and increased when unpredictable load instructions miss in AT. When the counter achieves a threshold value, even the unpredictable load instructions can be allocated in AT.

A small threshold value for the collision counter is valuable: a) to detect a change in the program-execution context, and b) to filter conflicts in AT between unpredictable load instructions. In the latter case, the goal is to cause the hit-burst lengths of the unpredictable load instructions allocated in AT to be larger than zero. A short hit-burst length is enough to reclassify a load instruction. On the other hand, a small threshold value can favour the allocation of unpredictable load instructions if the execution ratio is dominated by unpredictable load instructions. In this case, filtering is reduced and can vanish, and the behaviour of our proposal would be close to the behaviour of the LAP.

We will use two-bit collision counters and three as a threshold value for these counters; that is, an unpredictable load instruction is allocated in AT when the collision-counter value is three. The use of larger collision counters is not valuable because it decreases the results obtained in some benchmarks.

CT initialization

Initializing all the CT entries as unpredictable is valuable because it filters load instructions with zero hit-burst lengths. Note that the allocation of these load instructions does not increase the predictability captured by the predictor, although some of the predictability can be damaged. Our evaluations show that a predictor that initializes CT entries as unpredictable captures more predictability than initializing them as predictable.

3.5.2 Filtering by Discrete Classification Last-Address Predictor (DLAP)

In this subsection we propose an address predictor that uses the discrete classification presented in the previous subsection. This predictor will be named *Filtering by Discrete Classification Last-Address Predictor* (DLAP).

Figure 3.14 shows a diagram of the DLAP. It uses two prediction tables: the Address Table (AT) and the Classification Table (CT); both tables will be direct mapped. Each CT entry contains one bit that records the classification of a load instruction.

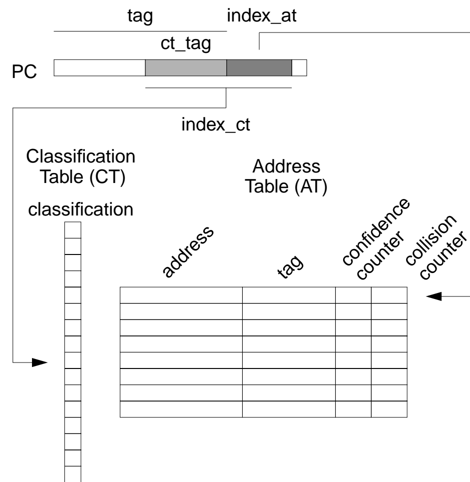


Figure 3.14 Diagram of the Filtering by Discrete Classification Last-Address Predictor (DLAP)

Each AT entry contains four fields: an effective address, a tag, a confidence-counter value and a collision-counter value. The effective address will be used to predict addresses, the tag contains the bits of the PC that do not index AT (the lower bits of the tag identify the CT entry related to the AT entry), the confidence-counter value is used to decide if the load instruction allocated in the AT entry must be predicted, and the collision-counter value reflects the execution ratio between the allocated load instruction and the unpredictable ones that collide at the same AT entry.

It was decided to record only one bit per CT entry, because recording the whole confidence-counter value was not a cost-effective alternative. Results were similar, but for a 1,024-entry AT and a table-size ratio equal to 8, recording two bits per CT entry represents a cost increment of around 10% (Section 3.5.3.1 presents an area-cost evaluation of our proposal).

Figure 3.15 compares the decision tables applied by the CLAP and by the DLAP to decide if a load instruction that misses in the AT must be allocated in AT. We can observe the difference between both decision tables: the DLAP can allocate load instructions classified as unpredictable in AT.

a) CLAP

		Allocated instruction	
		unpredictable	predictable
Missing instruction	unpredictable	Don't replace	
	predictable	Replace	

b) DLAP

		Allocated instruction	
		unpredictable	predictable
Missing instruction	unpredictable	Delay	
	predictable	Replace	

Figure 3.15 Decision tables applied by the replacement algorithms of the CLAP and the DLAP

The predictor works as follows. When a load instruction is fetched, the appropriate AT entry is selected, and the tag field is compared with some bits of the PC to determine if the AT entry is related to the executed load instruction (AT hit). If so, the confidence-counter value is used to decide if the load instruction must be predicted; otherwise, it is not predicted. The procedure *Prediction* in Figure 3.16 depicts the pseudo-code related to these actions.

The prediction tables are updated after the address stage of the pipeline (procedure *Update* in Figure 3.16 depicts its pseudo-code). On an AT hit, the confidence-counter value is increased as in the LAP. Moreover, the computed address is recorded in the address field of the selected AT entry, and the collision-counter value of the AT entry is decreased by one. Note that in an implementation of the procedure most prediction-table accesses can be performed in parallel.

On an AT miss, we will use a simple replacement mechanism in AT. A load instruction classified by CT as predictable will immediately replace the load instruction allocated in the AT entry. However, if the load instruction is classified as unpredictable by CT, the replacement is only performed if the collision-counter value is greater than 2; otherwise, the collision-counter value of the AT entry is increased by one.

On a replacement in AT, AT fields are updated according to the executed load instruction, the collision-counter value is reset and the classification related to the evicted load instruction is recorded in CT. Values 0 and 1 classify the load instruction as unpredictable, values 2 and 3 classify it as predictable.

On an AT miss produced by a load instruction that also collides in CT with the load instruction allocated in AT, the replacement mechanism only considers the collision counter value to decide the allocation of the load instruction. In this case, the confidence counter is set to one on a replacement.

We will evaluate several table-size ratios: 2, 4, 8 and 16. Increasing the table-size ratio produces a finer classification. That is, the classification will be improved when a predictable load instruction does not collide in CT using the larger ratio. The increase in captured predictability

depends on the conflicts in AT; that is, for an AT size and large table-size ratios, the captured predictability saturates.

```
/* Predicts an effective address
Output variables:
-predict: predicted address
*/

Prediction(PC) {
  index_at = INDEX_AT(PC);
  tag = TAG(PC);
  if ((AT[index_at].tag == tag) {
    /* AT hit */
    if (AT[index_at].conf > 1) {
      predict = AT[index_at].addr;
    }
  }
}
```

```
/* Updates CT and AT */

Update(PC, addr) {
  index_at = INDEX_AT(PC);
  index_ct = INDEX_CT(PC);
  tag = TAG(PC);
  if (AT[index_at].tag == tag) { /* AT hit */
    if (AT[index_at].addr == addr)
      AT[index_at].conf ++;
    else AT[index_at].conf --;
    AT[index_at].addr = addr;
    AT[index_at].colls --;
  }
  else { /* AT miss */
    if(CT_TAG(PC)!=CT_TAG(AT[index_at].tag)) {
      curr_conf = AT[index_at].conf;
      exec_conf = CT[index_ct];
      index_ct_curr =
        COMBINE(AT[index_at].tag, index_at);
      if ((exec_conf == 1) OR
          (AT[index_at].colls > 2)) {
        AT[index_at].tag = tag;
        AT[index_at].addr = addr;
        AT[index_at].colls = 0;
        AT[index_at].conf = exec_conf + 1;
        CT[index_ct_curr] = HIGH_BIT(curr_conf);
      }
      else AT[index_at].colls ++;
    }
    else { /* AT miss and CT conflict */
      if (AT[index_at].colls > 2) {
        AT[index_at].tag = tag;
        AT[index_at].conf = 1;
        AT[index_at].addr = addr;
        AT[index_at].colls = 0;
      }
      else AT[index_at].colls++;
    }
  }
}
```

Figure 3.16 Pseudo-code of the Filtering by Discrete Classification Last-Address Predictor (DLAP)

The discrete classification is obtained from confidence counters of the AT. As conflicts in CT are also conflicts in AT, the discrete classification can be erroneous. Only conflicts in CT between unpredictable load instructions do not influence the result of the discrete classification: it will be unpredictable. In conflicts between other kinds of load instructions, the discrete classification will also be unpredictable. The last case increases the filtering effect and, in some cases, could increase the captured predictability if a conflict in AT exists with a predictable load instruction mapped in another CT entry.

Our predictor makes the most of the classification recorded in CT in two ways: a) to initialize the confidence-counter of the AT entry on a replacement, and b) as an input of the replacement mechanism of AT. As a result, filtering will increase the hit-burst lengths related to the predictable load instructions, as well as reducing the overall loss of predictability produced by the learning phase of the confidence mechanism.

The following expression gives the area cost of the DLAP as a function of the number of table entries. To determine entry lengths, we are assuming 64-bit virtual addresses, *tagbits* of tag in the AT, 2-bit confidence counters and 2-bit collision counters; every CT entry records only one bit.

$$\text{AreaCost}_{DLAP} = \text{CT_entries} \times 1 + \text{AT_entries} \times (64 + 2 + 2 + tagbits)$$

3.5.3 Performance evaluation

This section presents an evaluation of our proposal, comparing the area cost, the predictability and the accuracy of the DLAP versus the LAP. Moreover, we will compare our proposal with other replacement algorithms proposed to filter the allocation of instructions in the prediction tables [CRT99][RFKS98].

3.5.3.1 Captured Predictability

We present a comparison between the predictability captured by the *LAP* and the DLAP. The evaluated configurations of both predictors have an amount of tagbits equal to 17-$\log_2$(AT entries); this decision will be justified in Section 3.5.3.3

To show the influence of recording the classification of the evicted load instructions and filtering the allocation of unpredictable load instructions on the performance, we present results for three cases: (1) the *LAP*, (2) an *LAP* with a CT that is only used to initialize the confidence counter on allocations in AT, and (3) the *DLAP*. Case (2) will be named LAP+CT.

Figure 3.17 presents results for the large and the extra-large benchmarks; Figure 3.27 in Section 3.8, presents results for the remaining benchmarks. In these graphs, the vertical axes stand for the captured predictability and the horizontal axes stand for the area cost of the predictors. A line connects the predictability captured by the LAP for several numbers of AT entries. Other lines connect results for LAP+CT (black points) and DLAP (white points) for an AT size and several table-size ratios.

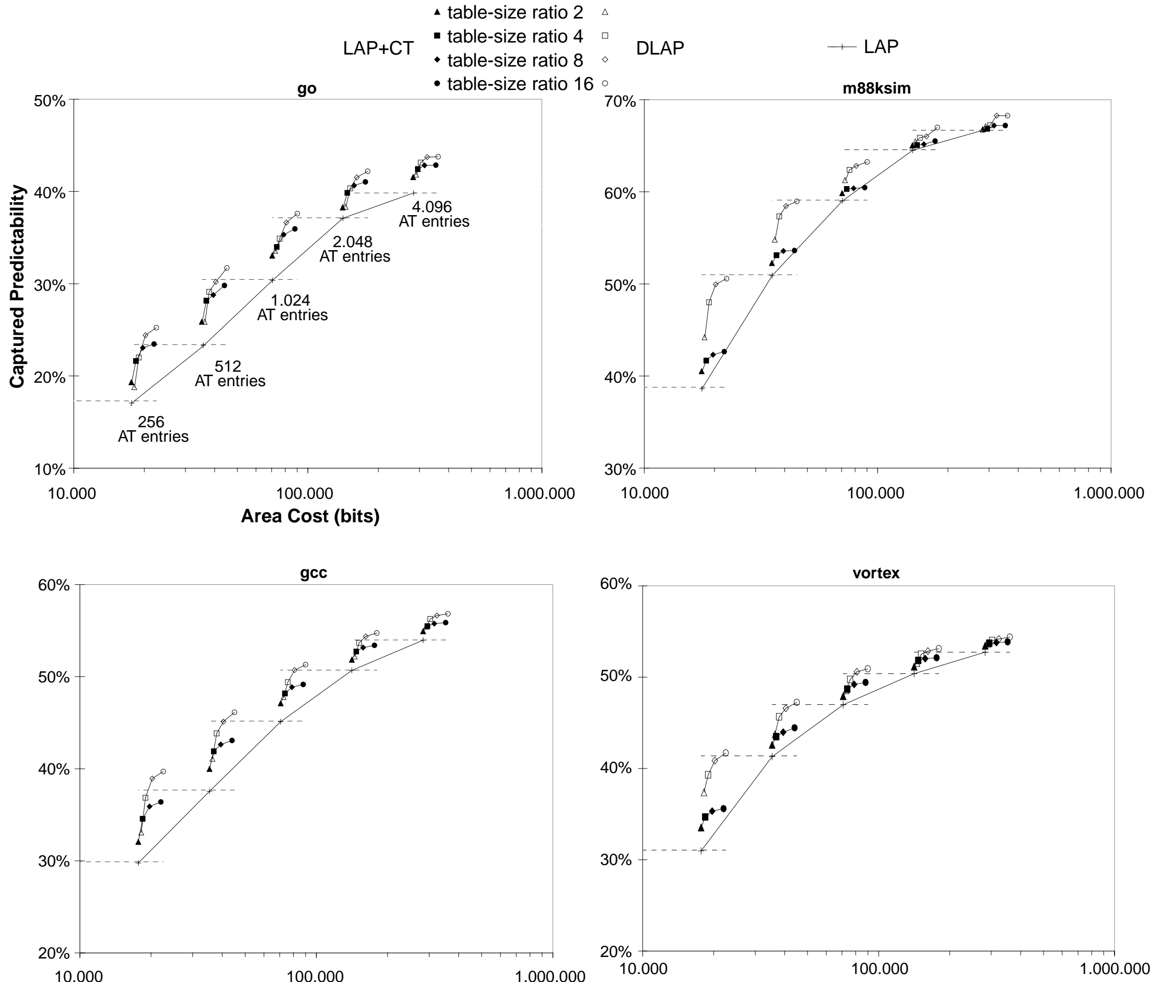


Figure 3.17 Predictability captured by the LAP, the LAP+CT and the DLAP in large and extra-large benchmarks

Comparisons between predictor configurations with the same number of AT entries
LAP+CT always outperforms LAP because it is able to exploit the hit bursts of the predictable load instructions fully. As the number of AT entries increases, hit-burst lengths become larger due to the reduction on the AT miss rate. Then, the difference between the predictability captured by the LAP and LAP+CT decreases. Note that in LAP and LAP+CT predictors, the predictability available in hit bursts of length zero cannot be exploited because the load instructions are evicted from AT before their next execution.

The benefits of the LAP+CT are significant in configurations with a great amount of dynamic load instructions related to short hit bursts (Table 3.2). For instance, benchmark *go* with a 2.048-entry AT, and benchmark *m88ksim* with a 1.024-entry AT exhibit a miss rate of about 12%, but the increase in predictability produced by recording the classification is bigger in *go* than in *m88ksim*.

The difference between LAP+CT and the DLAP is due to the filtering performed by means of the CT. We can appreciate that filtering increases the predictability in almost all benchmarks. In some cases, the filtering effect is remarkable for few table-size ratios: benchmarks *m88ksim* and *gcc*, because there is a significant proportion of hit-bursts of length zero.

For every number of AT entries, the predictability captured by the DLAP increases as the table-size ratio is increased. The captured predictability saturates for large table-size ratios because it is limited by the number of AT entries and the mapping policy; that is, increasing excessively the number of CT entries produce a marginal increment in the captured predictability.

Comparisons between predictor configurations with a different number of AT entries
Filtering the allocation of the unpredictable load instructions in AT reduces the number of AT entries needed by a predictor to achieve a performance level. Now, we will compare DLAP configurations against LAP configurations with twice AT entries.

In Figure 3.17, discontinuous lines show the predictability captured by several LAP configurations. These lines allow us to compare the predictability captured by a DLAP configuration versus an LAP configuration with twice AT entries.

From the results, we see that the DLAP can capture more predictability than the LAP with twice AT entries. This is possible because doubling the number of AT entries of the LAP increases the predictability captured by the LAP only if doubling causes a predictable load instruction not to conflict with other load instructions of the same program-execution context. But in some cases, doubling the AT does not remove all the conflicts in an AT entry. The DLAP tries to eliminate conflicts in the AT entries by filtering the allocation of unpredictable load instructions in AT; that is, filtering is successful if there is only one predictable load instruction colliding at the same AT entry. When two predictable load instructions collide at the same AT entry, the DLAP cannot capture their predictability, but doubling the AT may eliminate this conflict. In this case, the LAP can outperform a DLAP with half AT entries, for instance benchmark *m88ksim* with 2.048 AT entries.

Most DLAP configurations with table-size ratios 8 or 16 capture as much predictability as the LAP with twice AT entries. Furthermore, in some cases (*gcc*, *go*, *vortex*), the DLAP outperforms the LAP with twice AT entries. These results show that there is a significant proportion of conflicts between predictable and unpredictable load instructions, so doubling the AT of the LAP is not a cost-effective solution.

Using as a reference the area cost of an LAP, the area cost of a DLAP with half AT entries and table-size ratio 8 is around 43% smaller. For a table-size ratio 16, the area-cost reduction is around 36%.

We can conclude that the use of the proposed mechanism to filter the allocation of the unpredictable load instructions can increase the performance of the *LAP* as much as doubling its

number of AT entries. Moreover, the area-cost increment is smaller than the area cost of doubling the AT. Finally, our proposal gives to the designer a wide range of configurations for obtaining the best fit to the available area.

3.5.3.2 Comparison of filtering strategies

We will compare our filtering strategy with the ones proposed in [CRT99][RFKS98]. In these works, their replacement algorithm uses confidence information of the allocated load instructions, and mapping-conflict counters. They add a replacement counter to every AT entry of the LAP. This counter is decreased on a correct prediction and increased on a misprediction or when a miss is detected. Replacement takes place on saturation of the replacement counter; after that, the counter is set to zero. We will name a predictor with this replacement algorithm a *Conservative Predictor* because it prioritizes the load instructions allocated in the prediction table versus a colliding load instruction that misses in the prediction table (even if it is predictable). That is, the *Conservative Predictor* does not use classification information of the instructions that collide with the entry as our replacement algorithm does. For this comparison, we will employ 3-bit replacement counters because we have observed that they saturate the predictability captured by a *Conservative Predictor*.

Figure 3.18 compares the decision tables applied by the DLAP and by the Conservative Predictor to decide if a load instruction that misses in the AT must be allocated in AT. We can observe that the decision table of the Conservative Predictor does not use information of the instructions that are not allocated in AT.

		Allocated instruction	
		unpredictable	predictable
Missing instruction	unpredictable	Delay	
	predictable	Replace	

a) DLAP

		Allocated instruction	
		unpredictable	predictable
Missing instruction	unpredictable	Delay	
	predictable		

b) Conservative Predictor

Figure 3.18 Decision tables applied by the replacement algorithms of the DLAP and by the Conservative Predictor

We can classify the SPEC95-INT benchmarks into two classes: (A) benchmarks with a high amount of dynamic load instructions that are highly predictable (*m88ksim* and *perl*) and, (B) the remaining benchmarks. Now, we will comment on the behaviour of the filtering strategies in both benchmark classes. To show the differences, we choose as representative of each class the benchmark with the largest working set of static load instructions; that is, *m88ksim* and *go*.

Figure 3.19 shows the predictability captured by the LAP, the *Conservative Predictor* and the DLAP for different numbers of AT entries using direct-mapped tables in benchmarks *m88ksim* and *go*. To evaluate our proposal, we have selected a table-size ratio equal to 16.

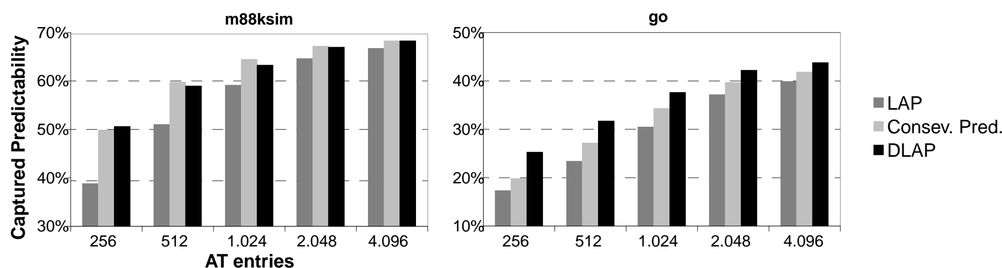


Figure 3.19 Predictability captured by the LAP, the *Conservative Predictor* and DLAP in benchmarks *m88ksim* and *go* using direct-mapped AT's

In benchmark *m88ksim*, most conflicts are between predictable load instructions. Both the *Conservative Predictor* and our proposal can capture more predictability than the LAP because they are able to filter conflicts between predictable load instructions. However, for some AT sizes, our proposal shows a slight decrease (2%) versus the *Conservative Predictor*.

Both replacement algorithms filter the predictable load instructions in a different way. The *Conservative Predictor* favours the allocated load instructions; then, a highly predictable load instruction allocated in the prediction table will only be evicted on saturation of the replacement counter.

In contrast, the DLAP is filtering the allocation of predictable load instructions as a side effect of initializing CT entries as unpredictable: a potentially predictable load instruction is classified in CT as unpredictable. Then, potentially predictable load instructions that have a few conflicts with an allocated predictable load instruction are not reclassified as predictable. If they have a bigger number of conflicts, their reclassification will be only delayed.

However, in the case of two load instructions classified as predictable in different execution contexts, and which in a third execution context collide in AT, using the DLAP can produce a ping-pong effect and not capture predictability of either instruction.

In both replacement algorithms, the allocation of predictable or potentially predictable load instructions is favoured by collisions produced by unpredictable load instructions. However, in our proposed algorithm, a load instruction classified as predictable will be allocated in AT without delay.

In benchmark *go*, both the *Conservative Predictor* and the DLAP capture more predictability than the LAP. However, the DLAP is able to capture more predictability than the *Conservative Predictor* because it prioritizes the predictable load instructions; that is, the allocation of load

instructions classified as predictable is not delayed. However, the *Conservative Predictor* can delay the allocation of these load instructions in an execution-context change or in the same execution context. It is not delayed only if it collides with an unpredictable load instruction whose replacement counter saturated. In the same execution context, unpredictable load instructions can produce a delay chain that makes the allocation of a predictable load instruction difficult.

We can conclude that our replacement algorithm will perform better than that of the *Conservative Predictor* on benchmarks with a significant amount of conflicts between predictable and unpredictable load instructions. On benchmarks with a significant amount of conflicts between predictable load instructions, the loss of predictability of the DLAP with respect to the *Conservative Predictor* is small.

Comparing the area cost of the *Conservative Predictor* and that of the DLAP for the same number of AT entries, our proposal represents an area-cost increment of around 20%. However, our proposal is able to obtain the performance of an *LAP* with twice AT entries; which represents an area-cost decrease of around 40%.

In Section 3.5.3.5 we will comment on the influence of associative mapping on the performance of both filtering strategies.

3.5.3.3 Influence of partial tagging on performance

In this subsection an attempt will be made to reduce the area cost of the LAP and the *DLAP*. We will evaluate if a predictor with a partial-tagged AT can emulate the performance of a fully tagged AT. Partial-tagging in prediction tables has also been evaluated in other contexts [Fagi95].

In the *LAP* and the *DLAP*, the tag bits and the index bits identify the load instruction allocated in an AT entry. This identification is used by the replacement algorithm to detect collisions in an AT entry, and, on a replacement, to initialize the confidence counter of the AT entry. Using partial-tagged AT's instead of fully tagged AT's, some identifications can be erroneous and produce mispredictions. Nevertheless, as the processor checks the predictions, and a recovery mechanism restores the correct architectural state on mispredictions, an exact identification of the load instructions allocated in AT is not needed.

We have evaluated the effect of partial-tagging AT on the captured predictability and on the accuracy of the predictors. The number of tagbits needed to saturate the accuracy of the predictor in a benchmark depends on the number of static load instructions of the benchmark, the temporal distribution of their references, and their placement in the program code. We have performed simulations over a wide range of tagbits for every AT size to detect the number of tagbits needed to saturate the accuracy and the predictability of the predictors.

From our results, we may conclude that in the evaluated benchmarks, both predictors should use up to 17 bits to tag and index AT to saturate their accuracy. The use of a lower number of bits

can reduce the accuracy up to 2%. As an example, we show results for benchmark *gcc* in Figure 3.20. The horizontal axis shows the accuracy of some configurations of the LAP versus the number of tagbits in benchmark *gcc*. We may observe that the accuracy of an LAP with a number of AT entries saturates when the number of index bits plus tag bits is 17.

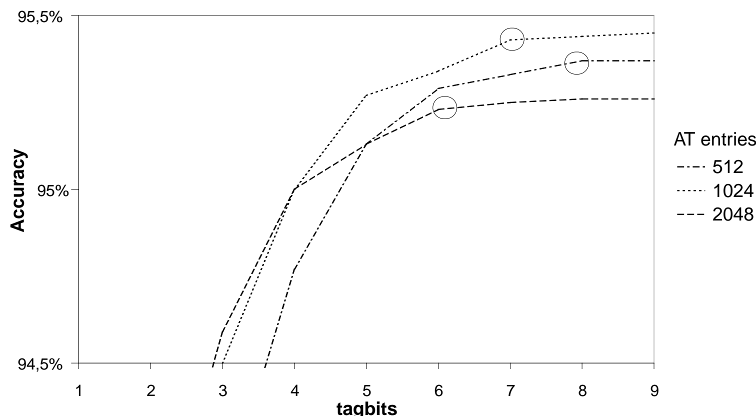


Figure 3.20 Accuracy of the *LAP* versus the number of tagbits in benchmark *gcc* (circles show the saturation point for every number of AT entries)

In addition, we have observed that the predictability is less sensitive to the number of tagbits than the accuracy. Using 11 bits to tag and index AT, the captured predictability saturates for both predictors.

3.5.3.4 Accuracy

Figure 3.21 shows the accuracy of the LAP, the DLAP and the Conservative Predictor in the large and extra-large benchmarks (Figure 3.28 shows results for the remaining benchmarks). The vertical axis stands for the accuracy of the predictor and the horizontal axis stands for the number of AT entries. Every graph is related to a predictor and a benchmark.

One may observe that, for the same number of AT entries, the DLAP is more accurate than the LAP. The only exception is benchmark *m88ksim* for 256 and 1.024 AT entries. This is due to the large difference between the number of predictions performed by both predictors (Figure 3.17). Furthermore, in most cases, a *DLAP* configuration is as accurate as an *LAP* configuration with twice AT entries. The accuracy of the *Conservative Predictor* is between that of the LAP and that of our proposal; the only exception is benchmark *m88ksim*, where the three accuracies are very close.

When load instructions are allocated in the AT, the confidence counters guide the prediction. Then, the accuracy of the DLAP can be improved using other types of confidence estimators. This additional increase in accuracy will be added to the performance obtained by the use of the classification.

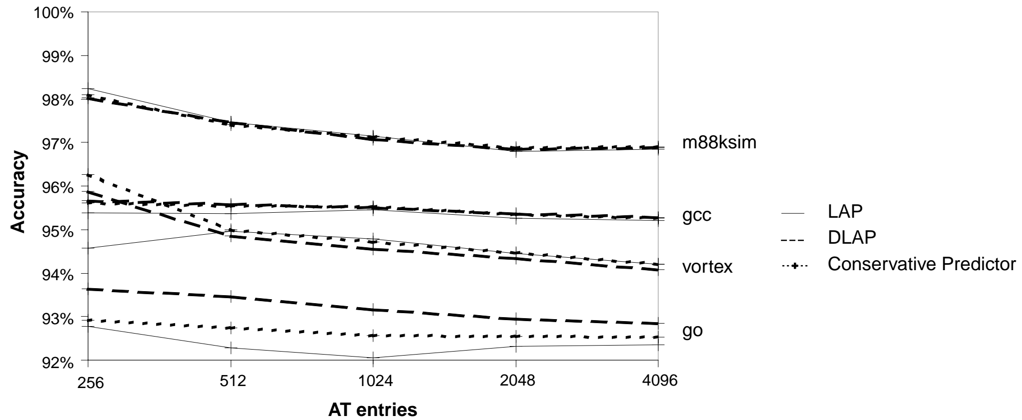


Figure 3.21 Accuracy of the LAP, the *Conservative Predictor*, and the *DLAP* in large and extra-large benchmarks using direct-mapped AT's.

3.5.3.5 Influence of associativity on performance

Temporal reuse is related to capacity and conflict misses in AT. When AT size or its associativity is increased, the hit-burst lengths become larger and the predictability captured by the predictor can be increased. However, the access time of the associative AT's can be a design restriction. Consequently, it is valuable to evaluate the performance increase of an associative AT compared with a direct mapped AT. In this section the gain obtained by the filtering strategy here proposed is evaluated, in the context of associative tables, and compared with that of the *Conservative Predictor*.

The CT of the *DLAP* will be direct mapped and the table-size ratio will be 16. The replacement mechanism will use a collision counter per AT entry: on an AT hit, the collision counter is decreased; on an AT miss, the collision counter of the LRU entry of the set is increased. The LRU entry is replaced by an unpredictable load instruction on saturation of the collision counter.

Figure 3.22 compares the *DLAP* and the *LAP*, both using direct-mapped, 2-way and 4-way AT's. Each bar is related to a number of AT entries and an associativity. The lower section of each bar stands for the predictability captured by a *LAP* configuration, and the upper section of each bar stands for the increment in captured predictability achieved by using a *DLAP*. We show the weighted average predictability in every benchmark class (Section 3.5.3.2). We do not present results for benchmarks *li*, *compress* and *ijpeg* since, in the evaluated sizes, they are close to saturation.

For Class-A benchmarks (*m88ksim* and *perl*), our proposed filtering mechanism increments the predictability captured using associative AT's up to 1.024 AT entries. For larger AT's, the captured predictability is almost saturated.

For Class-B benchmarks (*go*, *gcc* and *vortex*), the increment is noticeable in all the evaluated range of AT sizes and all the associativities. Moreover, *DLAP* configurations using up to a

2.048-entry direct-mapped AT's capture more predictability than the *LAP* with a 2-way AT of the same size. A similar relation holds, up to 1.024-entry AT's, for direct-mapped *DLAP* configurations versus 4-way *LAP* configurations.

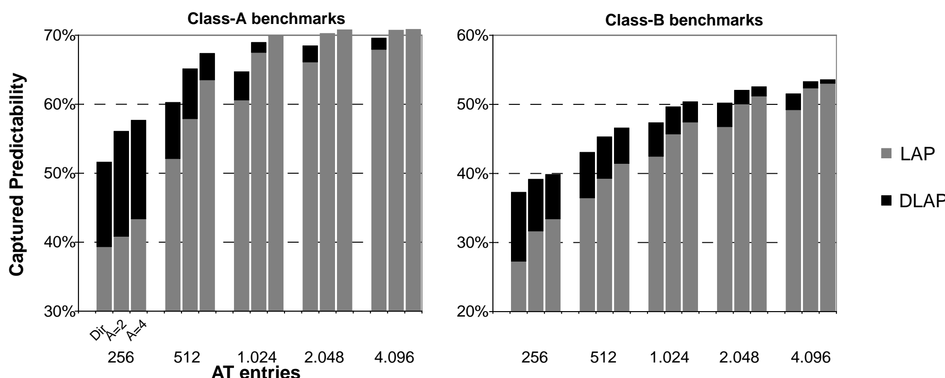


Figure 3.22 Average predictability captured in Class-A benchmarks (*m88ksim* and *perl*) and in Class-B benchmarks (*go*, *gcc* and *vortex*) for several associative mappings

For 2-way AT's, the *DLAP* can achieve a performance close to the 2-way *LAP* with twice AT entries and with a significant area-cost reduction (around 40%). The maximum predictability decrease is 2%. For 4-way AT's, the maximum predictability decrease is also 2%.

Figure 3.23 compares the predictability captured by the *LAP*, the *Conservative Predictor* and the DLAP in the context of associative AT's. The replacement algorithm in the *Conservative Predictor* is an extension similar to that of the *DLAP* and to that presented in [CRT99]. We made similar conclusions as in Section 3.5.3.2, when two benchmark classes were analysed. It is worth noting that differences between both predictors are reduced.

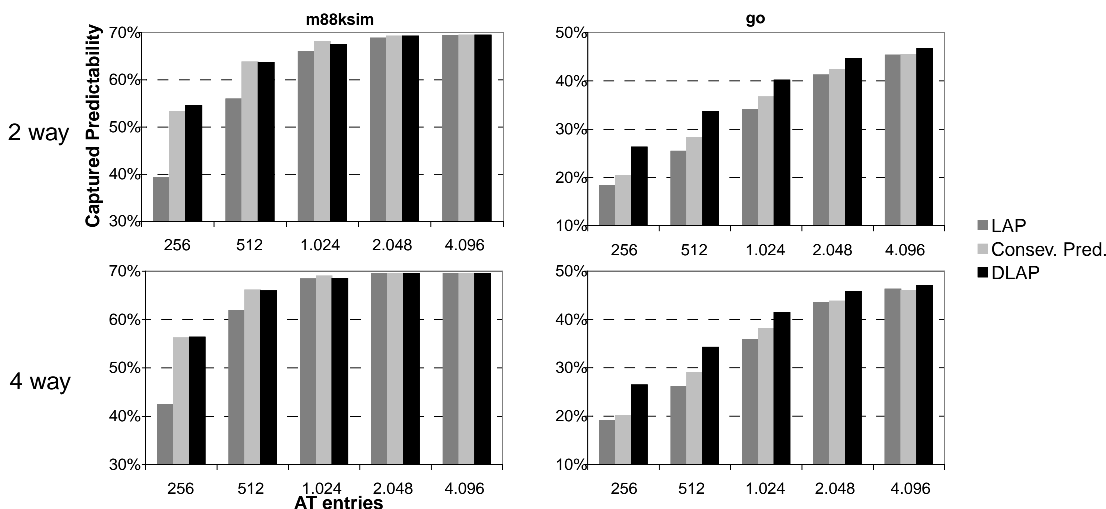


Figure 3.23 Predictability captured by the *LAP*, the *Conservative Predictor* and the DLAP in benchmarks *m88ksim* and *go* using associative tables

3.6 Related Works

Lipasti et al. [LiSh96] have proposed a value predictor that decouples the classification information from the value information. They use two tables: the Classification Table (CT) and the Value Prediction Table (VPT). The classification information recorded in CT is used to decide which load instructions can be predicted, and to introduce hysteresis in the replacement of the values recorded in the VPT. However, it is not used to filter the load instructions that can be allocated in the VPT. The authors evaluate different configurations of CT and VPT, but they focus on configurations with a bigger number of VPT entries than CT entries. Moreover, they do not analyse the influence of the miss rates and the hit-burst lengths on the performance of the predictor.

Berkerman et al. [BJR+99] applied the idea of the classification mechanism proposed in Section 3.4.1 to a Context Address Predictor. Their goal was to prevent random accesses from polluting the Value Prediction Table (VPT). They updated the VPT only when a load instruction computed the same effective-address portion twice in a row.

Gabbay et al. [GaMe97] have proposed an alternative method to prevent the placement of unpredictable load instructions in the AT. They propose classifying statically the load instructions according to their predictability. This classification is performed using program profiling. The compiler insert hints that are used by the predictor to determine if a load instruction should be allocated in a prediction-table entry. The threshold value used by the compiler to classify load instructions is a program-dependent value, and it is normally supplied by the user. Static classification has some drawbacks: a) it needs a profile execution, b) static classification is not binary compatible, c) it does not adapt to changes in the predictability of the load instructions.

Mechanisms that detect the usefulness of the predictions to reduce the execution time have been proposed in [CRT99][RFKS98]. These mechanisms are used to select the instructions that must be inserted in the prediction tables. Our proposal can improve their performance because we filter the allocation of unpredictable load instructions in the prediction table.

Filtering the allocation of information in the prediction tables has been used in the context of branch predictors [DrHo98][EdMu98]. These works propose hybrid predictors that predict some branch instructions with a simple predictor (last value or bimodal), and the remaining ones with a more complex predictor (context based). As some branch instructions can be correctly predicted with the simple predictor, the authors propose filtering these branches out of the complex predictor. However, when no predictor is predicting a branch instruction with strong confidence, all predictors in the hybrid predictor are updated simultaneously, and resources are allocated for a branch instruction in all prediction tables. Our proposal can be included and used to filter the allocation of information in some prediction tables.

3.7 Conclusions

The effective addresses computed by load instructions in the integer benchmarks exhibit a significant tendency to be predictable. Using a conventional predictor (LAP) with an unbounded prediction table, an average 50% of the computed addresses can be correctly predicted, but the tendency of load instructions to be predictable does not spread uniformly among them.

Capacity and conflicts misses in the prediction table reduce the performance of the predictor for several reasons. First, on a miss, the executed load instruction cannot be predicted. Second, as some confidence measures used by the predictors have a learning phase of some executions, when a load instruction is allocated in the prediction table some of its following executions can not be predicted. Third, unpredictable load instructions can replace predictable load instructions in the prediction table because some predictors use the *always allocate* allocation police.

Our characterizations show that the amount of executions of a load instruction from its allocation in the prediction table until its eviction is small. Thus, the penalty imposed by the learning phase of the confidence measures can be significant. Moreover, most load instructions are highly predictable or highly unpredictable.

We have proposed two address predictors that record classification information of the instructions evicted from the AT. The recorded information in used both to guide the replacement algorithm of the AT and to initialize the confidence estimators. The evaluations show that our proposals present the same predictor performance as the LAP, but they require a smaller area cost (19% and 40% respectively).

Filtering techniques can also be applied to stride-address predictors and to value predictors to reduce their area cost.

3.8 Detailed results

In some figures of this chapter, we have presented partial results (average results or results for selected benchmarks) of our proposals. In this section we complement these figures by presenting respectively the individual results and the results for the remaining benchmarks.

3.8.1 Similarity

Figure 3.24 presents the similarity of both the *<N, 0>* and the *<N, 3>* classification mechanisms in each SPEC95-INT benchmark. We can observe that the *<N, 3>* mechanism does not reach 100% similarity in benchmarks *gcc*, *compress* and *ijpeg*. This is due to the existence of load instructions that generate effective addresses where the three low-order bits are significant for classifying the instructions.

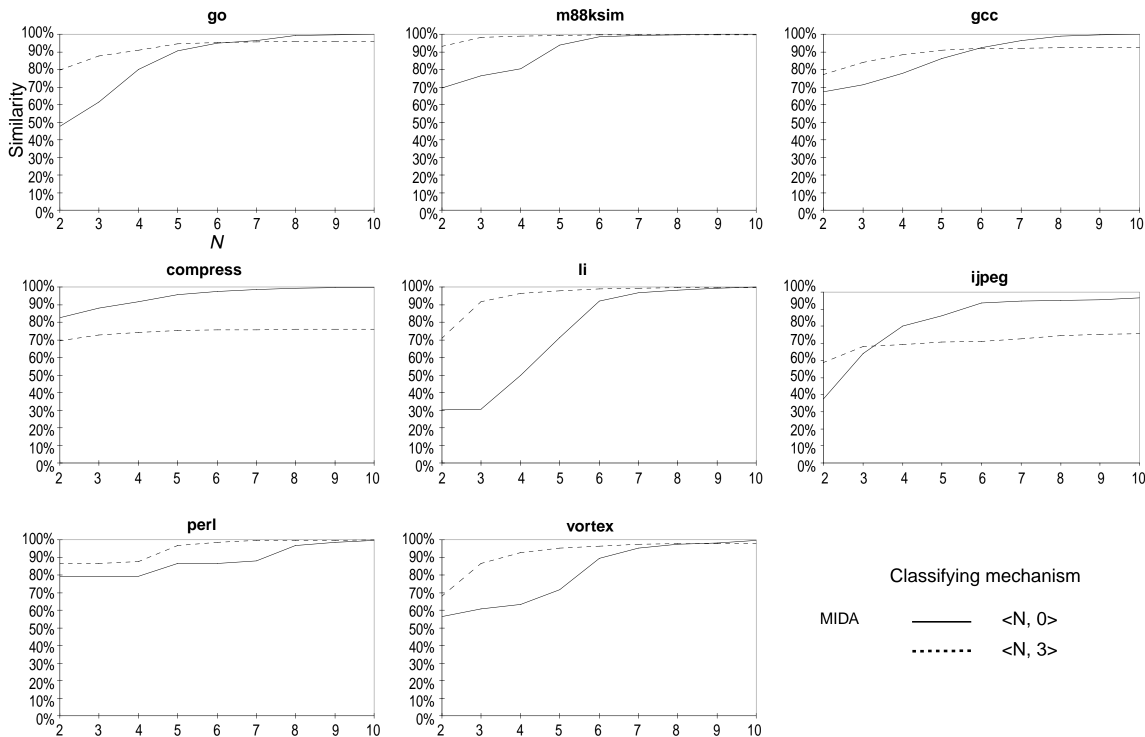


Figure 3.24 Similarity in each benchmark of *<N, 0>* and *<N, 3>* classifying mechanisms

3.8.2 Predictability captured by the CLAP

Figure 3.25 shows the predictability captured by the LAP and the CLAP in small and medium benchmarks. In benchmarks *li* and *ijpeg*, the CLAP needs less area cost to capture the same predictability as the LAP. The behaviour of benchmark *perl* is due to the high amount of conflicts between predictable load instructions; as the CLAP does not filter the allocation in AT of predictable instructions, doubling the number of AT entries is more effective than adding a CT. Benchmark *compress* present a behaviour similar to benchmark *perl*.

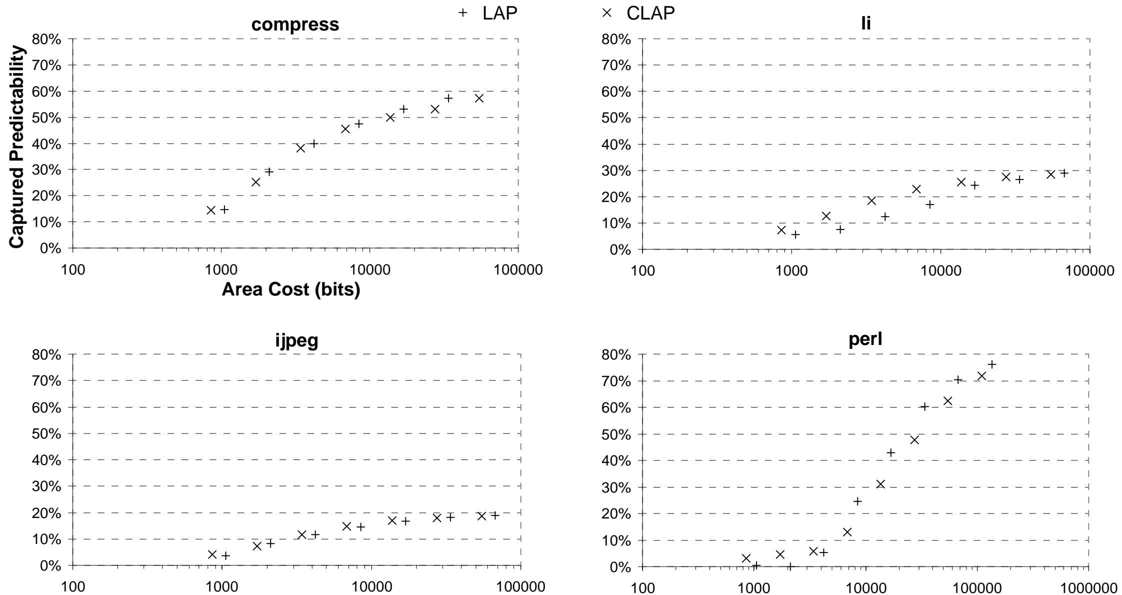


Figure 3.25 Predictability captured by the LAP and by the CLAP in small and medium benchmarks

3.8.3 Accuracy of the CLAP

Figure 3.26 shows the accuracy of the CLAP and the LAP in small and medium benchmarks. We can observe that in almost all presented results, the CLAP is more accurate than the LAP with the closest area cost.

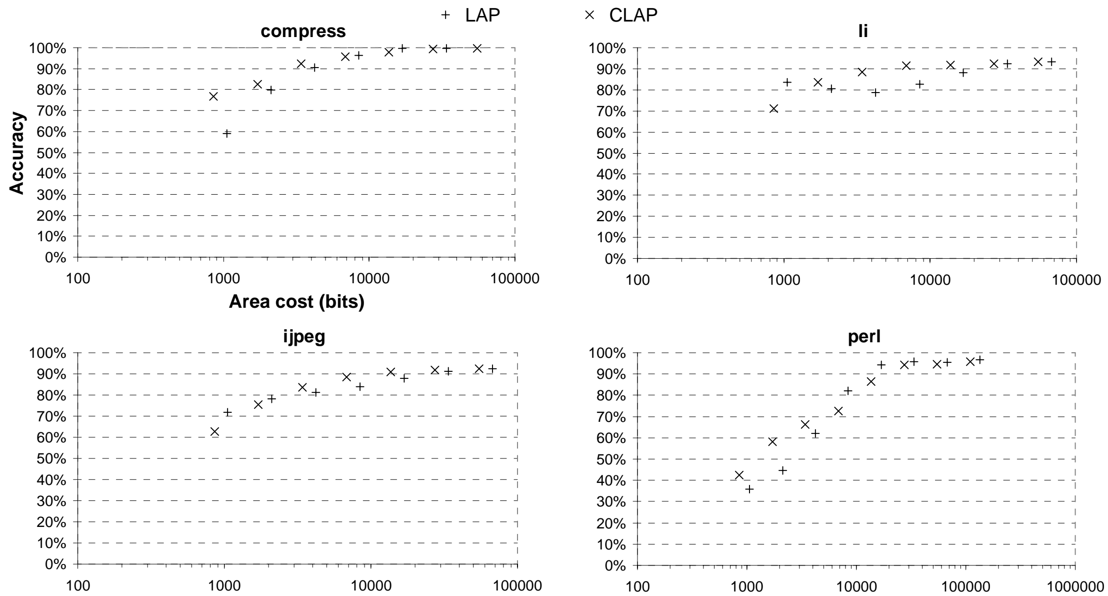


Figure 3.26 Accuracy of the LAP and the CLAP in small and medium benchmarks

3.8.4 Predictability captured by the DLAP

Figure 3.27 shows the predictability captured by the DLAP in small and medium benchmarks. Benchmark *perl* is similar to *m88ksim* due to its large amount of conflicts in AT, and its large predictability. Benchmarks *compress* and *ijpeg* are similar to *li*, their results almost saturate in the presented configurations due to their small working set of load instructions.

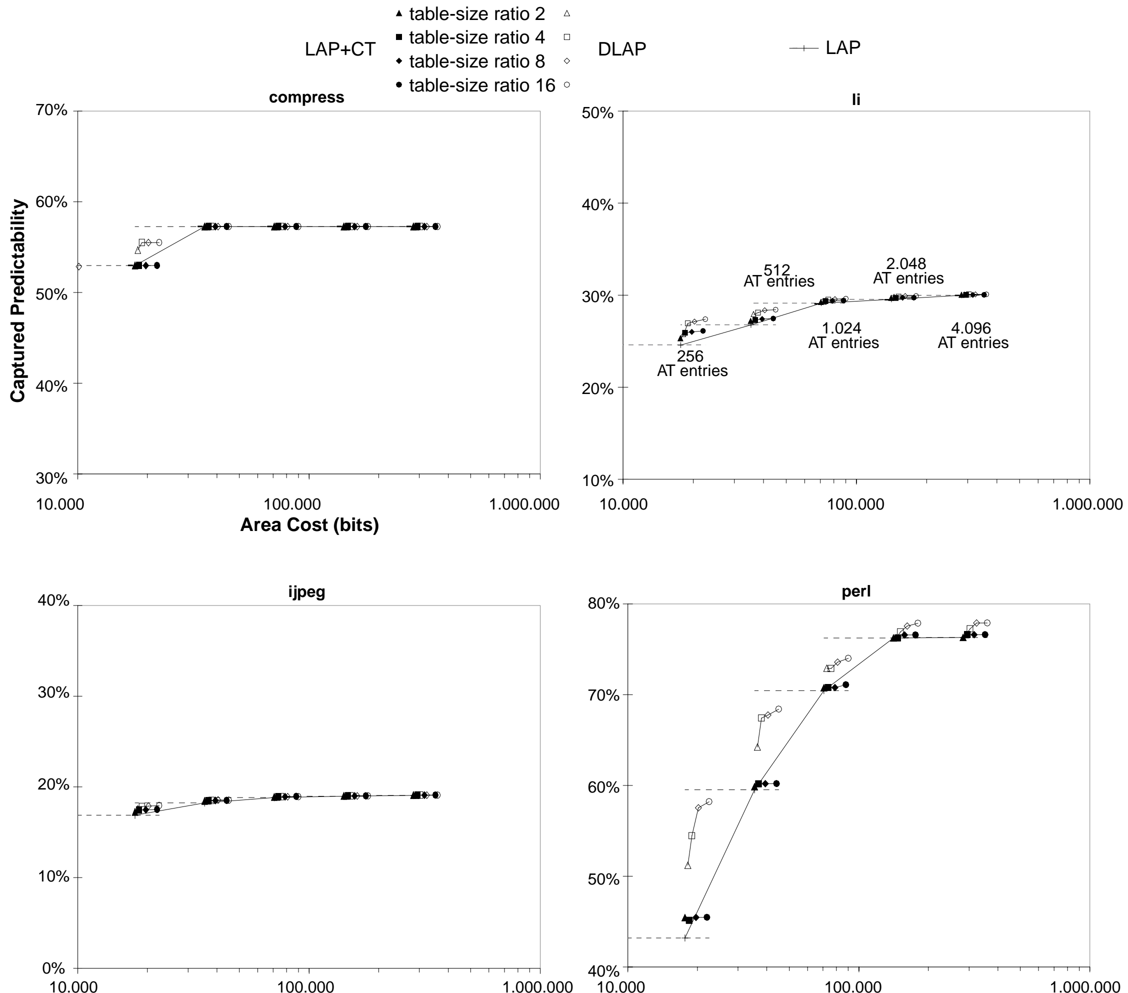


Figure 3.27 Predictability captured by the DLAP in small and medium benchmarks

3.8.5 Accuracy of the DLAP

Figure 3.28 shows the accuracy of the LAP, the DLAP and the Conservative Predictor in small and medium benchmarks. As in the remaining benchmarks, the accuracy of the DLAP is higher than that of the Conservative Predictor and LAP.

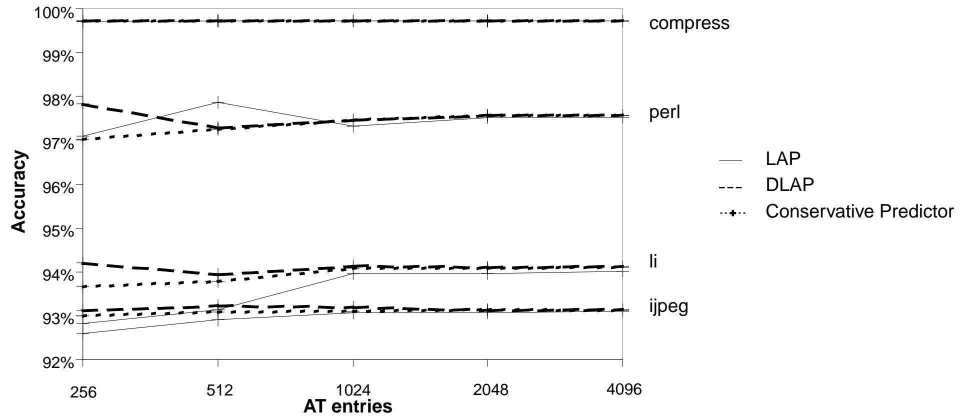


Figure 3.28 Accuracy of the LAP, the *Conservative Predictor*, and the *DLAP* in small and medium benchmarks using direct-mapped AT's

4 REDUCING PREDICTION-TABLE SIZE BY NARROWING

In this chapter, a technique for reducing the amount of information recorded in the prediction table of an address predictor is presented. Our motivation is the presence of a certain degree of redundancy in the effective addresses recorded in the prediction table; this redundancy is produced by both the temporal-locality and the spatial-locality properties of the memory references. We propose a prediction-table organization that reduces the redundancy present in the prediction table. This chapter is organized as follows. Section 4.1 is the introduction. In Section 4.2 an evaluation of the locality of the addresses recorded by the prediction-table of a Last-Address Predictor (LAP) is presented.

Section 4.3 introduces an organization of the prediction tables named Two-Level Address Storage (TLAS) and applies it to the LAP. Section 4.4 shows that by applying TLAS to the LAP, the area-cost of the LAP can be reduced without affecting its performance. In Section 4.5, the inclusion of TLAS in an enhanced last-address predictor presented in Chapter 3 is evaluated. In Section 4.6, some related works are reviewed. Finally, the conclusions of this work are summarized in Section 4.7 and in Section 4.8, the results of our proposals for all SPEC95-INT benchmarks are given.

4.1 Introduction

As we pointed in the previous chapter, the capacity of the prediction tables of the address predictors turns out to be very large. Prediction-table capacity is proportional both to the number of prediction-table entries and to the entry width. Consequently, a reduction in any of these parameters will produce a reduction in the capacity of the prediction table.

In the previous chapter we proposed two techniques for reducing the number of prediction-table entries; both techniques filter-out the allocation of the unpredictable load instructions. In this chapter, we will propose a technique for reducing the entry width of the prediction table; that is, to narrow the prediction-table entries. We make use of two well known characteristics of the memory-reference stream of the programs: the temporal-locality and the spatial-locality properties.

Both reference properties produce a certain degree of redundancy in the contents of the prediction tables. For instance, due to the temporal locality, several prediction-table entries may record the same effective address; due to the spatial locality, several prediction-table entries may record the same high-order bits of the effective addresses.

In this chapter we propose an organization of the prediction tables named *Two-Level Address Storage* (TLAS). The design of this organization takes into account the temporal properties of the memory-reference stream and reduces the redundancy in the prediction table. Consequently, the organization reduces the capacity of the prediction tables and the area cost of the predictor.

We will apply the TLAS organization to the Last-Address Predictor (LAP) and to a filtering address predictor proposed in the previous chapter, the Filtering by Discrete Classification Last-Address Predictor (DLAP). We have evaluated the proposals by instrumenting the SPEC95-INT benchmarks; we will show that the performance of both original predictors is not affected and, moreover, a significant area-cost reduction if around 28% and 60% is obtained.

4.2 Locality analysis of the effective addresses

Many works have analysed a property of memory references: the locality. It has been defined as the program's tendency to reference memory in non-uniform, highly localized patterns [Bela66]. Temporal locality and spatial locality are the main classes of locality present in memory references. Temporal locality is the tendency to reference a memory location that has previously

been referenced; spatial locality is the tendency to reference a memory location that is close to another memory location referenced in the past.

The addresses computed by the load instructions exhibit both kinds of locality. This can produce a certain degree of redundancy in the content of the Address Table (AT) of a Last-Address Predictor (LAP):

- Temporal redundancy: As the LAP assigns an AT entry to every static load instruction, load instructions that compute the same address record the address redundantly at several AT entries. For instance, this effect is produced by the accesses to the same global variable from different routines, and the stack accesses performed by routines called from the same stack depth.

- Spatial redundancy: Some addresses recorded in AT point to close memory locations. Then, in most cases, the addresses will only differ in their low-order bits. This produces redundancy because some addresses recorded in the AT have the same value in their high-order bits. For instance, global variables stored in consecutive addresses and stack accesses produce this effect.

Locality analysis

We will present an analysis of the locality of the addresses recorded in AT. To perform this analysis, the last reference time of every address is used to maintain a temporal ordering on a stack of slots; every slot records an address and an address is recorded in only one slot. If an address recorded in the stack is not the last address computed by any load instruction, its slot becomes an empty slot. Then, if an address is not found in the stack, the address is recorded in the most recent empty slot and it is moved to the top of the stack.

We define as *one* the *match depth* of the Most Recently Used (MRU) address, the match depth of the second MRU address as *two*, and so on. Then, for every dynamic load instruction, we evaluate the match depth of its computed address. If it is not found in the stack, its match depth is infinite.

In many works (Section 3.3.1, [GaMe97], [SaSm97]) it has been observed that some static load instructions are highly unpredictable. Unpredictable load instructions are then filtered-out in order not to update the stack of slots. To decide if a dynamic load instruction is unpredictable, we will use the confidence-counter value of an unbounded LAP: the execution of a load instruction will be classified as predictable if its confidence-counter value is greater than one. We consider that the match depth of the addresses computed by unpredictable load instructions is zero.

To analyse the temporal locality of the effective addresses, we must discard zero low-order bits of the addresses, and to evaluate the spatial locality in AT we must discard some low-order bits of the addresses. In Figure 4.1, the vertical axes stand for the match depth, the horizontal axes stand for the number of low-order bits discarded, and every graph is related to a percentage

of dynamic load instructions. The meaning of every graph is that the addresses referenced by this percentage of dynamic load instructions have a match depth smaller than or equal to the vertical height. For instance, in benchmark *compress*, when 6 low-order bits of the addresses are discarded, the addresses referenced by the 99% of dynamic load instructions have a match depth ≤13. This value reflects the number of different high-order portions recently referenced by the predictable load instructions.

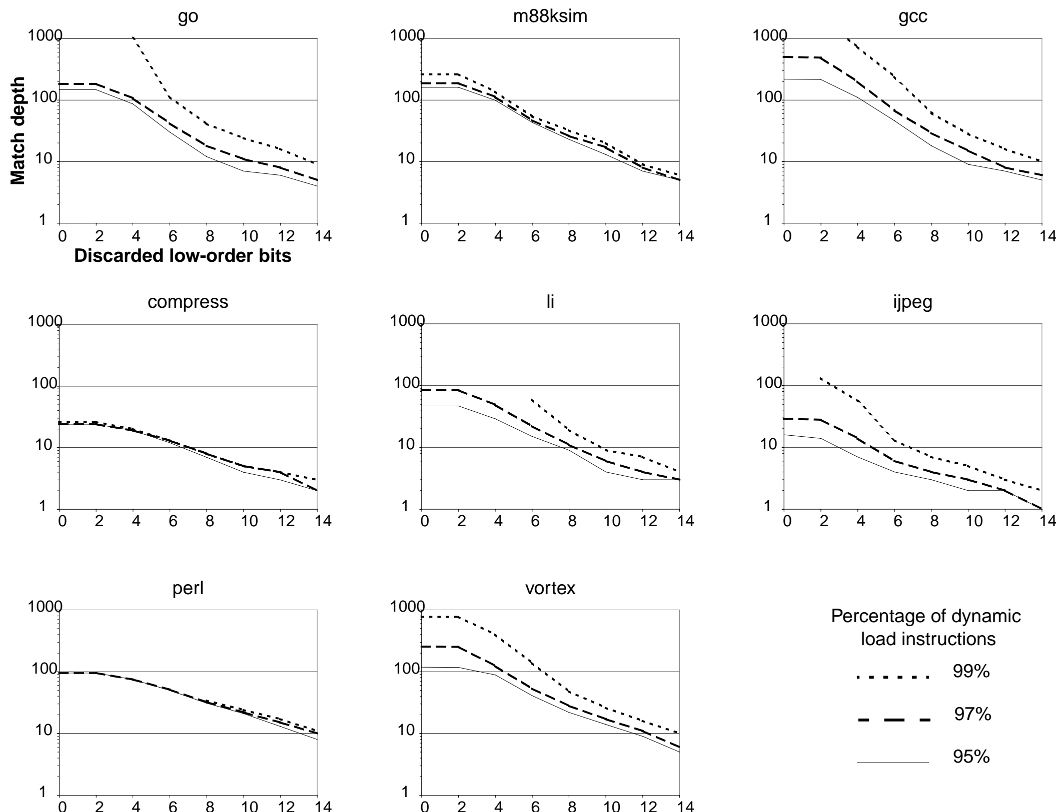


Figure 4.1 Dynamic-load-instruction distributions according to their match depth; the dynamic load instructions have been filtered out according to their confidence information

We may observe that the lowest order bits of the addresses computed by predictable load instructions are normally not significant as a consequence of the Alpha architecture, because most load-instruction operation codes produce eight-byte aligned addresses and load instructions that are not aligned are normally unpredictable by a LAP. As the number of low-order bits discarded grows, the match depth related to the percentage of load instructions decreases.

The working-set size of static load instructions (Section 3.3.2) is much bigger than the maximum match depth of 99% of all dynamic load instructions. For instance, in benchmark *go*, the working-set size of static load instructions is larger than 2.048 load instructions, but when 12

low-order bits of the addresses are discarded, 99% of dynamic load instructions have a match depth ≤16.

The results suggest an organization of the predictor where high-order bit portions of addresses are shared between several low-order bit portions; that is, in a many-to-one mapping. Thus, exploiting the spatial locality in address portions can be valuable in reducing the area cost of predictor. This organization will be named *Two-Level Address Storage* (TLAS). Figure 4.2 shows a diagram of this organization.

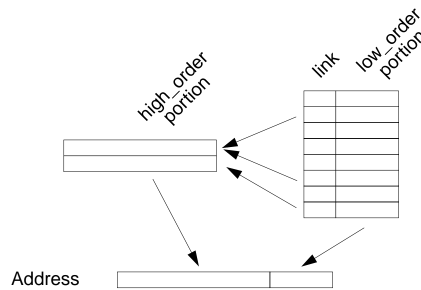


Figure 4.2 Diagram of the Two-Level Address Storage (TLAS) organization

4.3 Two-Level Last-Address Predictor (TLAP)

In this section, the Address Table of the LAP is replaced by the suggested *Two-Level Address Storage* (TLAS) organization. The new predictor will be named *Two-Level Last-Address Predictor* (TLAP). We describe the TLAP, and we perform evaluations on the management of High-Address-Table entries and the filtering-out of some allocations. Two replacement algorithms are also evaluated.

4.3.1 TLAP design

We have shown that there is a certain degree of redundancy in the information recorded in the AT of the LAP. To reduce the redundancy in the prediction table of an address predictor, the TLAS organization is used. The AT is split into two parts (Figure 4.3): the Low-Address Table (LAT) and the High-Address Table (HAT). The low-order bits of the effective addresses will be recorded in the LAT and the high-order bits in the HAT. Moreover, each LAT entry has a link to reference one HAT entry. Then, a HAT entry can be shared by several LAT entries: a one-to-many relationship.

To predict a load instruction, the TLAP indexes LAT using the PC of the load instruction. This access retrieves the low-order bits of the predicted address and a link to HAT. Subsequently, the predictor accesses HAT using the link; this access obtains the high-order bits of the predicted address. The predicted address is formed by the concatenation of the low-order bits and the high-order bits obtained from both tables.

The TLAP must access both tables sequentially to predict a load instruction. This does not imply an implementation restriction, since the LAT can be accessed very early in the pipeline

and, in current processors, the number of pipeline stages before issuing a instruction is so large that both prediction tables can be accessed before issuing the dependent operations. For instance, in the Alpha 21264 processor, an instruction passes through 5 pipeline stages before be issued [Alph99]. Moreover, we could reduce the critical path of the predictor could be reduced by recording enough low-order bits in the LAT for indexing the cache.

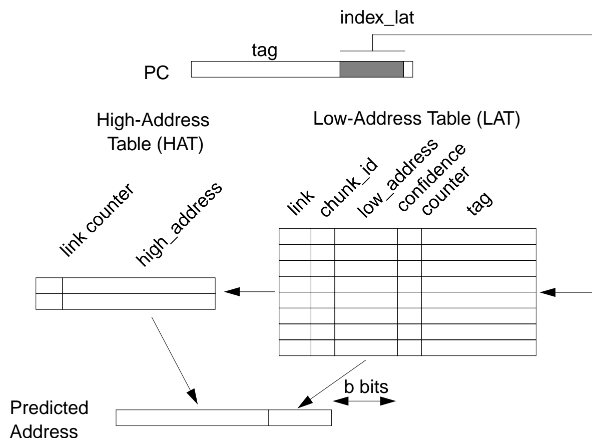


Figure 4.3 Diagram of the Two-Level Last-Address Predictor (TLAP)

LAT and HAT management

The TLAP updates the LAT in the same way as the LAP updates the AT; that is, using the *always-allocate* policy.

To link an LAT to an HAT entry, the TLAP applies a hash function to the high-order bits of the computed address and searches for them in the HAT. The hash function used depends on the number of HAT entries. Fully associative lookups are possible using a HAT with a small number of entries; as a reference, current microprocessors perform fully associative lookups in up to 96-entry TLB's [JaMu98].

To reduce the eviction of useful information on a HAT miss, the TLAP searches for HAT entries that are related to zero LAT entries (empty HAT entries). In order to detect the empty HAT entries, we will associate to every HAT entry a link counter that reflects the number of LAT entries linked to it. If no empty HAT entry is found, the TLAP picks a HAT entry randomly except the MRU one (non-MRU algorithm). These operations can be performed in parallel before selecting the replaced entry.

When an LAT entry is replaced, or an update due to a change in the high-order portion, the link counter of the previously related HAT entry is decreased by one (this operation can produce an empty HAT entry). When the LAT entry is linked to a HAT entry, its link counter is updated: it is increased if the high-order portion was already allocated (HAT hit), or otherwise set to one.

When a HAT entry is replaced, the LAT entries that were pointing to the evicted HAT entry are not invalidated because the cost is considerable and requires more complex logic. Note that these incoherencies do not break the program correctness because the predictor belongs to an speculative mechanism. These incoherencies, however, could decrease the performance of the predictor.

In a later section we will show the performance decrease if the detection of empty HAT entries is not considered on HAT replacements. The differences between the non-MRU and the LRU policies will be described, and the link-counter width discussed.

Filtering out HAT allocations

Storing high-order bits related to unpredictable load instructions is not useful because these load instructions will not be predicted. The TLAP allocates these loads in LAT but avoids the allocation of high-order bits computed by these loads in HAT; that is, their LAT entries will not be linked to any HAT entry.

To classify the load instructions, the TLAP will use a small chunk of the effective addresses (it will be recorded in LAT). This idea was proposed in Chapter 3 using the low-order bits of the addresses.

Most load instructions can be properly classified by checking the low-order bits of their effective addresses, but for some load instructions this is not sufficient. For instance, for loads with a stride multiple of 2^b, where b is the number of low-order bits analysed, the classification will be wrong and can produce mispredictions. However, a proper classification can be performed by analysing other bits of the addresses.

For that reason, we propose in this chapter to select dynamically the effective-address chunk that must be recorded in the LAT entry to classify each load instruction. As LAT entries record b bits of the addresses, we divide the addresses in several non-overlapped b-bit chunks. The TLAP will assume that two addresses are the same if the address-chunk contents are equal, and it will update the confidence counter accordingly.

When a load instruction is allocated in LAT, its classification is initialized as unpredictable and the TLAP uses the low-order bits of its addresses to classify it.

For load instructions classified as predictable, all bits of the address are used to update their confidence counter. However, when the classification of a load instruction changes from predictable to unpredictable, the predictor breaks the link with the HAT entry, decreases its link counter, selects the lower chunk of the computed address that is not equal to that of the predicted address, and records it in the LAT entry. In subsequent executions, classification is performed by checking the selected address chunk. To this end, every LAT entry contains a field that identifies the address chunk recorded in it. When the classification changes to predictable (the confidence-counter value of the load instruction goes from 1 to 2) the link is established.

In next subsection, we will show the performance decrease when filtering is not used, and the effect of the chunk selection on the accuracy of the TLAP.

Figure 4.4 shows the pseudo-code of the TLAP.

```
void Update(PC, addr, pred_addr,
            index_lat, index_hat, tag)
{
  if (LAT[index_lat].tag == tag) {
    if (LAT[index_lat].conf > 1) {
      if (addr == pred_addr)
        LAT[index_lat].conf ++;
      else {
        LAT[index_lat].conf --;
        if (LAT[index_lat].conf == 1) {
          /* Transition 2 -> 1 */
          LAT[index_lat].chunk_id =
           INDEX_DIF_CHUNK(addr,pred_addr);
          HAT[index_hat].links--;
        }
      }
    }
    else {
      chunk = CHUNK(addr,
                LAT[index_lat].chunk_id);
      if (chunk == LAT[index_lat].low_addr)
        LAT[index_lat].conf ++;
      else LAT[index_lat].conf --;
      if (LAT[index_lat].conf == 2)
        /* Transition 1 -> 2 */
        LAT[index_lat].chunk_id = 0;
    }
  }
  else {
    if (LAT[index_lat].conf > 1)
      HAT[index_hat].links--;
    LAT[index_lat].tag = tag;
    LAT[index_lat].conf = 1;
    /* stands for the [0, b-1] chunk */
    LAT[index_lat].chunk_id = 0;
  }
  LAT[index_lat].low_addr =
   CHUNK(addr, LAT[index_lat].chunk_id);
  if (LAT[index_lat].conf > 1) {
    index_hat= INSERT(HAT,HIGH_ADDR(addr));
    LAT[index_lat].link = index_hat;
  }
}

void Prediction(PC) {
 index_lat = INDEX_LAT(PC);
 tag = TAG(PC);
 if (LAT[index_lat].tag == tag) {
  if (LAT[index_lat].conf > 1) {
   index_hat =
      LAT[index_lat].link;
   pred_addr = CONCATENATE(
      HAT[index_hat].high_addr,
      LAT[index_lat].low_addr);
  }
 }
}
```

Figure 4.4 Pseudo-code of the Two-Level Last-Address Predictor (TLAP)

4.3.2 HAT implementation issues

In this subsection we evaluate different policies in HAT replacement algorithm. All the evaluations presented in this section assume unbounded LAT's.

To perform a prediction, HAT is accessed after accessing LAT, close to the fetch stage; in a later stage HAT is looked-up and updated. In this work we take into account the fact that access time of large HAT's can be a restriction. Then we evaluate HAT's with an access time close to that of the register file. Consequently, we use 16, 32 and 64-entry HAT's. Moreover, associative lookups can also be performed for these sizes. Figure 4.1 suggests the number of low-order bits that should be recorded in the LAT entries (*b*): 10, 12 and 14. Without restricting HAT access-time, other designs can be considered for bigger HAT's as n-way associative mapping and sequential lookups [ZZY97].

First we perform evaluations using the LRU replacement algorithm in HAT. Later, we present performance differences of the LRU versus the non-MRU algorithm. We also show accuracy differences between classifying load instructions checking the low-order bits against allowing a dynamic chunk selection. Captured-predictability differences between both classifiers are, however, negligible.

Filtering HAT allocations and managing empty HAT entries

To evaluate the effect of managing empty HAT entries and the effect of filtering out the allocation in HAT of address portions computed by unpredictable load instructions, we compare four TLAP designs. We evaluate the influence of managing empty HAT entries when HAT allocations are not filtered out. The influence of managing empty HAT entries when HAT allocations are filtered out is also evaluated.

Our results show that the management of empty HAT entries increases the predictability captured by a TLAP. Moreover, without the use of filtering, the performance increase is bigger; that is, HAT allocations are more sensitive to the management of empty HAT entries. This effect is specially noticeable in benchmark *compress*. In *compress*, a load instruction accesses a large hash table (up to 2^{19} bytes). This load instruction is unpredictable and pollutes the HAT with different values. By managing empty HAT entries, the HAT entry related to this load instruction becomes an empty entry, and it is reused to record the new address portion.

Figure 4.5 shows the influence of both policies on the predictability captured by the TLAP in benchmark *compress*. The vertical axis stands for the predictability captured by the predictors, and the horizontal axis shows different predictor configurations and benchmarks. Results for configurations with the same number of low-order bits in LAT entries are grouped; the difference between them is the number of HAT entries. The lower section of each bar stands for the predictability captured by the TLAP configuration that does not filter, and the upper section stands for the predictability increment due to filtering. One may observe that, without managing empty HAT entries, the influence of filtering for b=10 and 16-entry HAT is significant. However, when managing empty HAT entries, its influence is almost insignificant.

Filtering is valuable to reduce the working set of different high-order address bits. Predictable load instructions can use HAT entries that could have been related to unpredictable load instructions. Medium-predictable load instructions can also unlink HAT entries on unpredictable bursts. The increase in performance is noticeable in benchmarks with a large number of unpredictable static load instructions. For instance *gcc* (Figure 4.5). Observe that filtering increases the predictability captured by the TLAP independently of the management of empty HAT entries; but when both characteristics are combined, the TLAP captures more predictability.

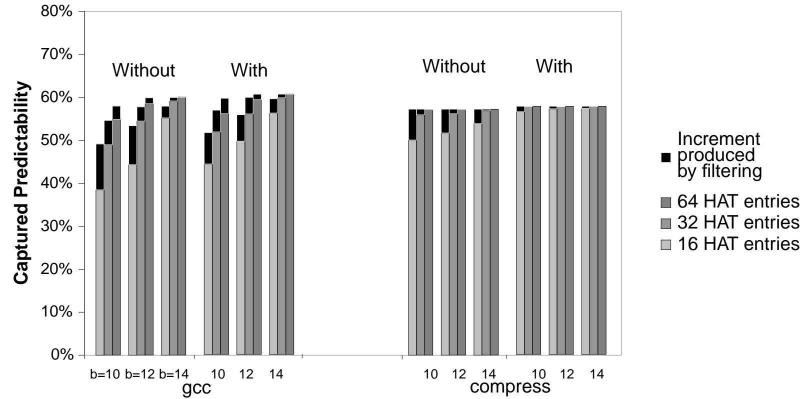


Figure 4.5 Effect on captured predictability of filtering-out HAT allocations without managing empty HAT entries (without) and managing empty HAT entries (with) in *gcc* and *compress*

From previous observations we see that the predictability increase is obtained by managing empty HAT entries and by filtering-out HAT allocations. Each characteristic is useful for some benchmarks. In the case of TLAP configurations that manage empty HAT entries, Figure 4.6 shows the captured predictability without filtering and filtering-out HAT allocations for all the evaluated benchmarks.

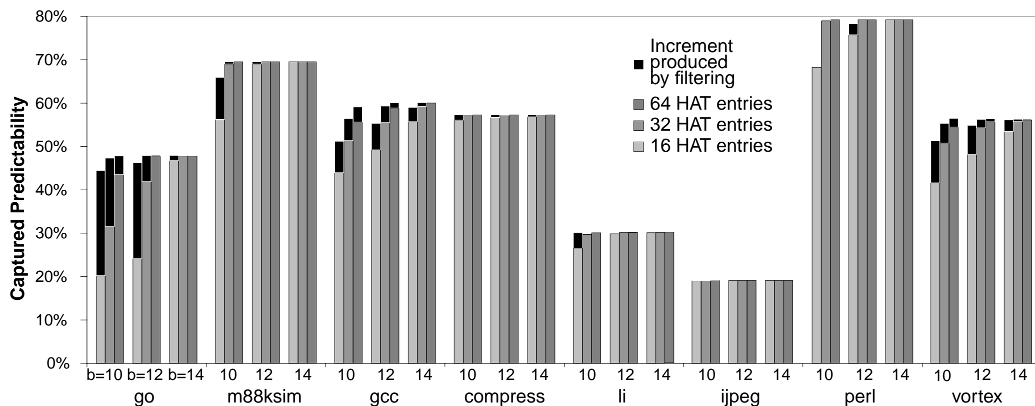


Figure 4.6 Effect on captured predictability of filtering-out HAT allocations by managing empty HAT entries

Configurations with *b*=14 and 32 HAT entries or 64 HAT entries, and *b*=12 and 64 HAT entries capture as much predictability as the LAP with an unbounded AT (except in *gcc*). In the remaining configurations, we can appreciate a captured-predictability decrease of the TLAP with respect to the LAP. The decrease is significant in benchmarks *go*, *gcc* and *vortex*. These results are coherent with the match-depth evaluations (Figure 4.1), because these benchmarks give the largest match depths.

Influence of chunk selection on the accuracy

Chunk selection hardly has any influence on the predictability captured by the TLAP, but the accuracy is sensitive to it. Figure 4.7 presents an evaluation of the accuracy of the TLAP in the two SPEC95 benchmarks that compute addresses with large strides: *ijpeg* (up to 2^{11} bytes) and the SPEC95-FP benchmark *turb3d* (up to 2^{18} bytes). The lower section of each bar shows the accuracy of a TLAP that always selects the low-order bits of the addresses; the upper section of each bar reflects the increment on the accuracy when dynamic selection of the proper chunk is performed. With chunk selection, the accuracy of the predictor saturates for all the analysed configurations.

In the remaining SPEC95-INT benchmarks, chunk selection hardly has any influence on the accuracy of the TLAP, because these benchmarks present strided addresses with a short stride.

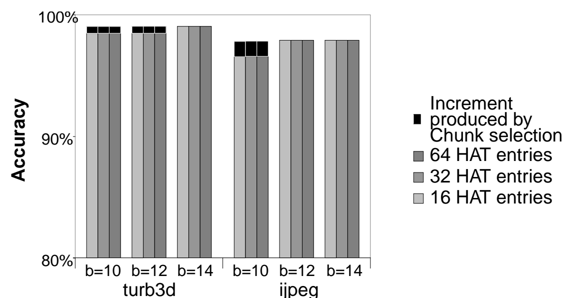


Figure 4.7 Influence of dynamic chunk selection on the accuracy of the TLAP

Non-MRU versus LRU algorithm

When no empty HAT entry is found, we propose a replacement policy that selects a HAT entry randomly except the MRU one (non-MRU algorithm) because the implementation of the LRU algorithm is complex and expensive for large tables. Figure 4.8 shows the performance difference between both replacement policies. The lower portion of each bar stands for the predictability captured using the non-MRU algorithm, and the upper portion of each bar stands for the increment due to the LRU algorithm.

The performance of the TLAP is not saturated in some configurations with the non-MRU replacement algorithm because there is a significant amount of capacity misses in HAT. In these cases, the LRU algorithm is better than non-MRU algorithm and the predictability decrease is limited by 2.4%. For configurations with b=12 and 64 HAT entries, and with b=14 and 32 or 64 HAT entries, both the LRU and the non-MRU algorithms exhibit the same performance.

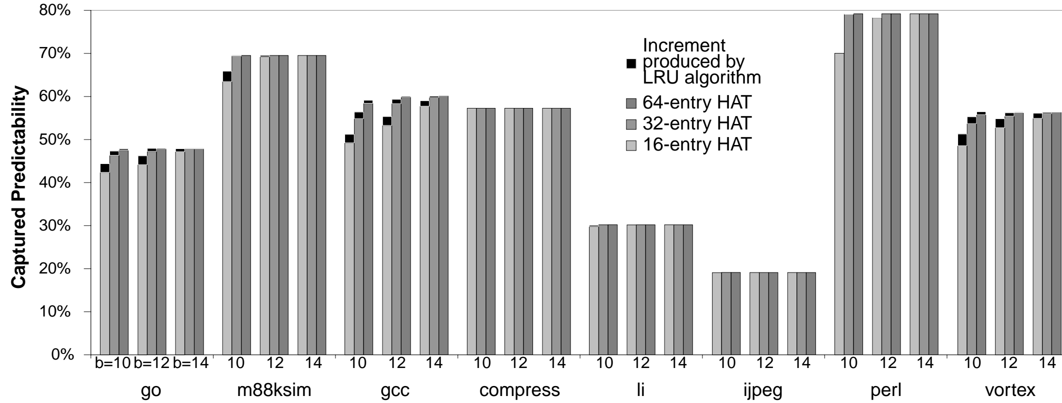


Figure 4.8 Influence of the algorithm used by the replacement policy of HAT on the predictability captured by the TLAP

Link-counter width

Our experiments show that when three-bit link counters are used, the performance of the TLAP is almost saturated. Note that the goal of these counters is to detect empty HAT entries, and three-bit counters estimate the empty HAT entries with a high degree of correctness.

4.4 Evaluation of the TLAP

This section presents an evaluation of three characteristics of the TLAP using bounded prediction tables: area cost, captured predictability and accuracy, and compares them with those of the LAP.

Bounded LAT's can reduce the pressure over HAT, decreasing the amount of capacity misses. However, even using 64 HAT entries, the accuracy of some TLAP configurations decreases with respect to that of the LAP. Consequently, the results showed in this chapter are focused on 64-entry HAT's. Moreover, we use TLAP configurations that manage empty HAT entries, 3-bit link counters, non-MRU replacement algorithm in HAT and b=10, 12 or 14. Working-set size of static load instructions of the benchmarks (Section 3.3.2) justifies the selected LAT-size range being from 256 to 4.096 entries.

4.4.1 Area cost of the predictors

We evaluate the area cost of an address predictor as the number of bits of information that it records. The following expression shows the area cost of the TLAP with the non-MRU replacement policy as a function of the number of prediction-table entries. We have assumed the use of 64-bit logical addresses, t-bit tags, 3-bit link counters in the HAT entries, and b-bit address chunks in each LAT entry. The last component of the TLAP area cost is needed to record the index of the MRU HAT entry.

$$AreaCost_{TLAP} = \text{HAT_entries} \times (3 + (64 - b)) + \text{LAT_entries} \times \left(\log_2 \text{HAT_entries} + \left\lceil \log_2 \frac{64}{b} \right\rceil + b + 2 + t\right) + \log_2 \text{HAT_entries}$$

The number of tag bits influences the accuracy of the predictor. In the analysed benchmarks, we showed (Section 3.5.3.3) that the accuracy of the LAP saturates when the number of index bits plus tag bits is 17. We will use TLAP configurations with this number of tag bits.

The area-cost reduction from an LAP configuration to a TLAP configuration with the same number of AT and LAT entries depends on the number of LAT entries, the number of HAT entries and *b*. In the evaluated configurations, the reduction ranges from 37% (256-entry LAT, *b*=14) up to 60% (4.096-entry LAT, *b*=10).

4.4.2 Captured address predictability

Figure 4.9 shows the predictability captured by the TLAP and the LAP in the benchmarks with the largest working-set size of static load instructions and match depths. The horizontal axes stand for the area-cost of the predictor and the vertical axes stand for the predictability captured by the predictor. The graph in the top left-hand corner is labelled with the number of AT and LAT entries of the predictor configurations.

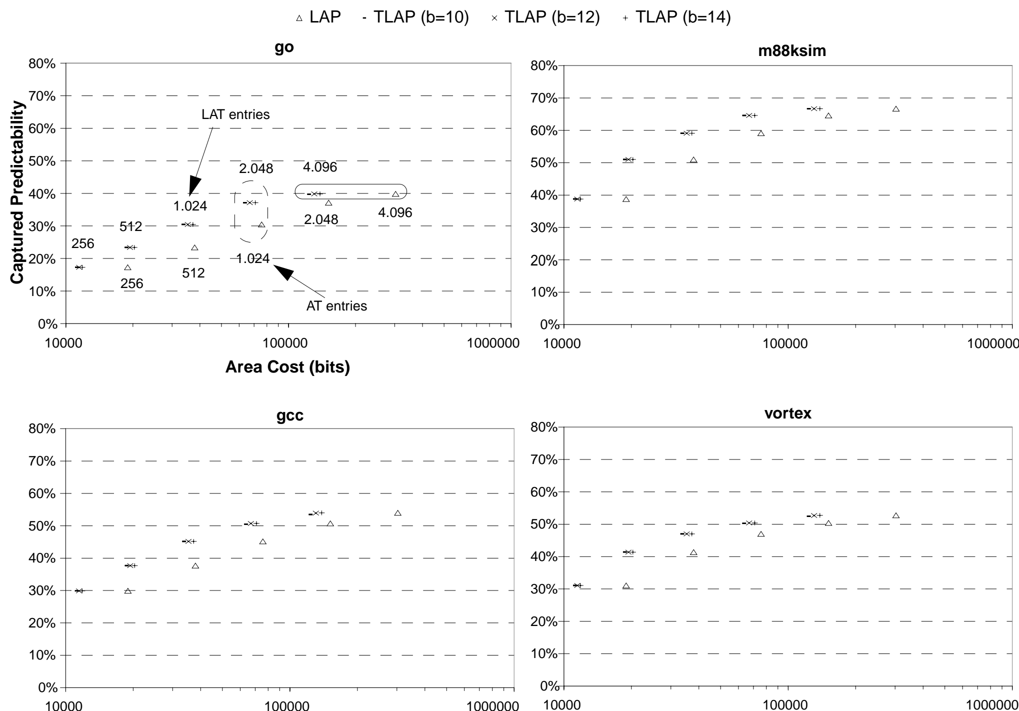


Figure 4.9 Predictability captured by the LAP and the TLAP in large and extralarge benchmarks

The area-cost reduction from a TLAP to a LAP configuration with the same number of LAT and AT entries does not represent a performance loss. In benchmark *go*, a continuous oval surrounds configurations with AT entries=LAT entries=4.096. Note that in these cases, the load instructions allocated in LAT are also allocated in AT.

Similar area-cost for LAP and TLAP configurations are obtained when the number of LAT entries doubles the number of AT entries. In benchmark *go* (these configurations are surrounded by a dashed oval), for LAT entries=2×AT entries=2.048. In this case, the TLAP configurations outperform the LAP configuration because LAT has fewer capacity misses than AT.

The remaining benchmarks present a similar behaviour but in a different AT/LAT-size range. Figure 4.13 in Section 4.8 shows the results for these benchmarks.

4.4.3 Accuracy

Although predictability grows as the number of AT entries increases, accuracy may not behave in the same way. Conflicts in AT can increase accuracy because fewer predictions are performed. When there are few conflicts in AT, accuracy depends on the ability of the confidence counter to characterize the load behaviour and to prevent some predictions. In this work, we use two-bit saturating counters as a confidence mechanism. More conservative confidence mechanisms can increase accuracy but they also reduce the captured predictability. Moreover, they can decrease the pressure over the HAT; that is, our results for HAT-entry requirement are an upper limit for the HAT-entry requirement of these confidence mechanisms. However, the evaluation of other confidence estimators is beyond the scope of this work.

Figure 4.10 compares the accuracy of both the LAP and the TLAP in several benchmarks. The vertical axes stand for the accuracy of the predictors and the horizontal axes for the same predictor configurations as in Figure 4.10 (without showing area cost); configurations with the same number of AT and LAT entries are grouped.

In TLAP, there is another characteristic that introduces mispredictions. When an HAT entry is replaced, the TLAP does not invalidate the LAT entries linked previously to this HAT entry. Then, the next execution of a load instruction allocated at these LAT entries may be mispredicted.

For small LAT's, some of these mispredictions are removed because an LAT replacement invalidates the link. In this case, a slightly accuracy decrease is observed. For large LAT's, the working-set size of values for the high-order bits of the addresses can be larger than the number of HAT entries. In this case, an accuracy decrease is observed. For instance, for TLAP configurations with *b*=10 the difference is limited by 0.7% (*gcc*).

Accuracy increases as *b* grows, since the pressure over HAT decreases, so more replacements are performed using empty HAT entries. For TLAP configurations with *b*=14, TLAP configurations are as accurate as the LAP.

Benchmark *ijpeg* presents a sharp decrease for 256 LAT entries and *b*=10. This behaviour is due to load instructions with large stride. The allocation of one of these load instructions in HAT produces a misprediction before classifying it as unpredictable. Its eviction from LAT by conflicts and subsequent reallocation reproduces the previous behaviour.

When LAT size doubles the number of AT entries, configurations with a similar area cost are obtained. In these cases, the captured predictability is bigger in TLAP, but accuracy is usually lower, due to the accuracy of the TLAP (*b*=14) being similar to that of the LAP with same AT size, and also to the irregular behaviour of the accuracy of LAP.

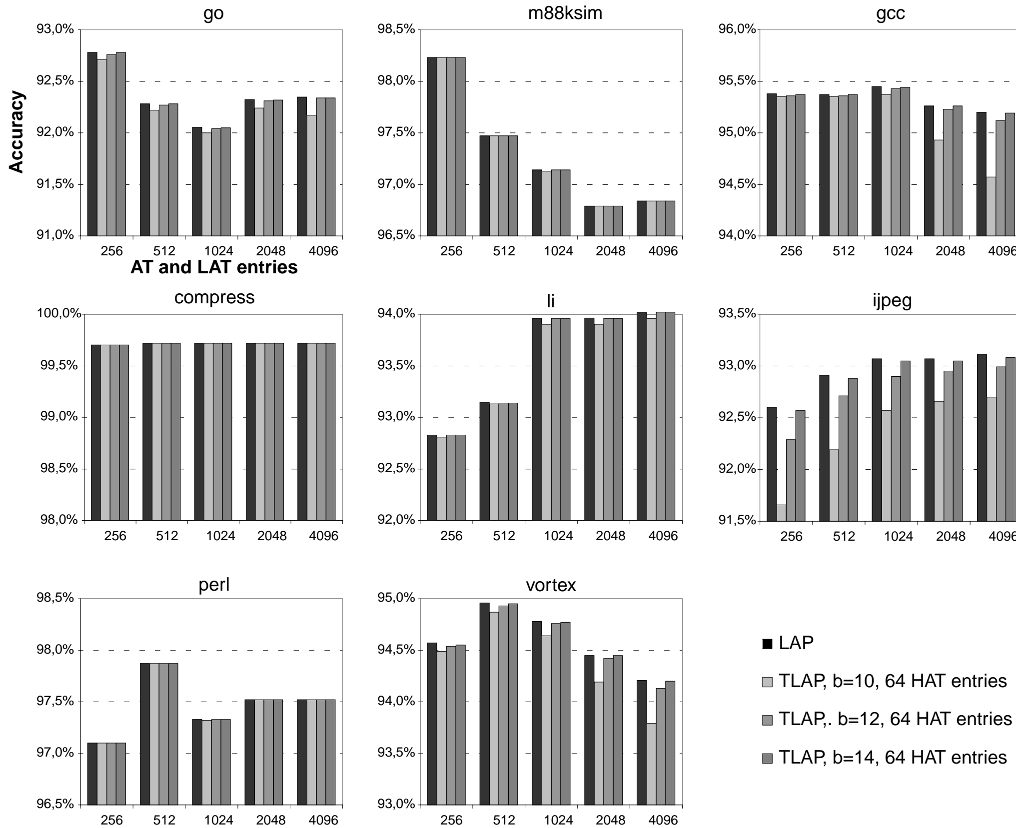


Figure 4.10 Accuracy of the LAP and the TLAP

4.5 Filtering by Discrete Classification Two-Level Last-Address Predictor (DTLAP)

In some works, predictors with filtering capacity have been proposed. The purpose of these predictors is to avoid the allocation of unpredictable instructions in the prediction table. To do this, they delay the allocation of an instruction when it collides a few times with an allocated predictable instruction that is being executed. Some proposals delay the allocation of any instruction [CRT99][RFKS98]. Others proposals delay the allocation only of the unpredictable ones (Section 3.4 and Section 3.5) by means of a dynamic classification.

We will evaluate the influence of TLAS organization on the performance of a Last-Address Predictor with filtering capacity. The evaluation is performed on the DLAP (Section 3.5.2); DLAP improves the performance of the LAP with twice AT entries, and reduces its area cost around

40%. We will name the resulting predictor *Filtering by Discrete Classification Two-Level Last-Address* Predictor (DTLAP). Figure 4.11 shows the diagram of the DTLAP.

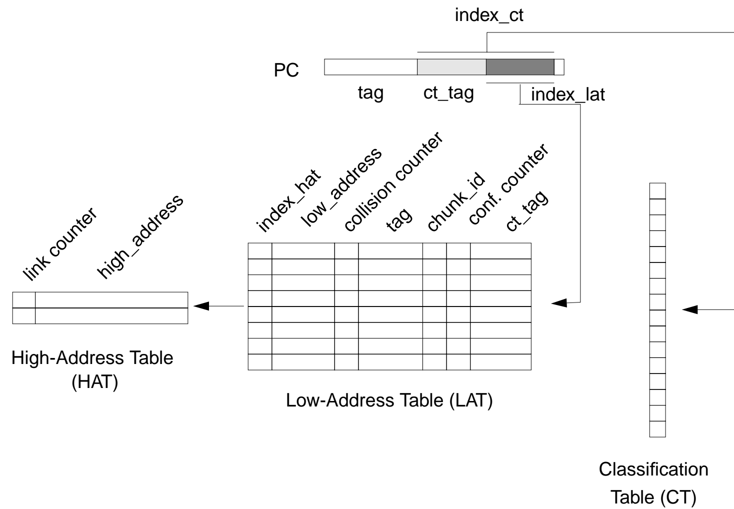


Figure 4.11 Diagram of the Filtering by Discrete Classification Two-Level Last-Address Predictor (DTLAP)

First, Figure 4.12 shows the predictability results of the DTLAP versus the LAP. Results show that the DTLAP needs as many HAT entries as the TLAP to achieve the performance of the DLAP. Although the DLAP is filtering the allocation of unpredictable load instruction in LAT, the TLAP is also filtering the allocation in HAT of high-order bits computed by unpredictable load instructions. Moreover, the chained filtering of the DTLAP does not increase sufficiently the performance of the predictor with fewer HAT entries to equalize the performance using 64 HAT entries.

Furthermore, filtering HAT allocations using LAT information cannot be suppressed because the predictors with filtering capacity do not guarantee the exclusion of unpredictable load instructions from LAT. Moreover, the information used to filter HAT allocations is more accurate than that used to filter LAT allocations.

The area cost reduction for a DTLAP when TLAS organization is used ranges from 29% (256 AT/LAT entries) to 40% (4.096 AT/LAT entries).

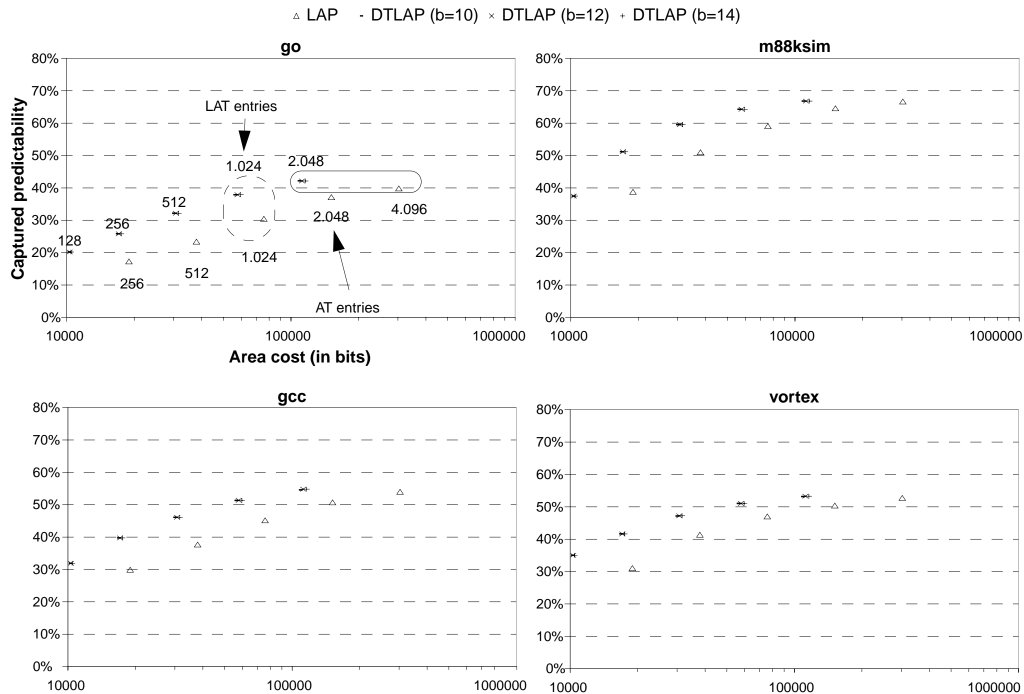


Figure 4.12 Predictability captured by the DTLAP and the LAP in large and extra-large benchmarks

4.6 Related works

The spatial-locality property of the memory references have been used to reduce the address-bus width [FaPa91]. More recently, it has also been used to reduce the energy consumed by the address bus [MLC97].

Brooks and Martoniosi [BrMa99] noticed that half of the integer operations of SPEC95-INT benchmarks require 16 bits or less. They propose to exploiting this fact in order to reduce the power consumed by integer execution units by disabling the upper portion of the functional unit. They also combine multiple narrow operations to execute them in parallel in the same functional unit.

Works closer to the scope of this chapter propose an organization that reduces the area cost of address tags in caches [Sezn94][WSY97]. These organizations are similar to TLAS, but our proposal has a simpler control logic. TLAS does not need to maintain the exact correspondence between both tables because it will be applied to a predictor, then a recovery mechanism on mispredictions is already supplied. [Fagi95] proposed narrowing the entries of the Branch Target Buffers by using partial-tagged entries instead of fully-tagged ones.

An organization similar to TLAS is proposed in [Sezn96] for sharing page-number information between several processor devices. Replacement in shared tables is then guided for devices that share it.

Rychlick et al. [RFK+98] proposed the Popular Last-Value Predictor which exploits the temporal locality in the results produced by register-writing instructions. They report results for 4.096-entry LAT's. Differences between our work and Rychlick's work arise in LAT mapping and management of the HAT entries. We take an explicit trace of unused entries in HAT to reduce capacity misses. Another difference is that we add a filtering ability to reduce capacity misses in HAT: we review a filtering idea presented in Section 3.4.1, and extend it to reduce pollution in HAT by filtering out unpredictable load instructions. In this work we have also evaluated the temporal and the spatial locality of the addresses computed by predictable load instructions for a large range of high-order bits.

Sato et Arita [SaAr00] applied a narrowing technique to value prediction. They noticed that the results of a significant amount (about 50%) of register-writing instructions are *short*; that is, the high-order bits of the results are zero. The authors proposed the use of a hybrid predictor with two components: one component for instructions that produce *long* results and the other component for instructions that produce *short* results. The evaluations show that the area-cost of the conventional predictor can be reduced between 25% and 50%, without affecting the performance impact of value prediction.

4.7 Conclusions

We have shown that the spatial-locality property of the memory references produces redundancy in the address field of the prediction tables of the address predictors, because the number of different values for the high-order bits of the addresses recorded in the prediction tables is relatively small.

We have used this property to reduce the amount of information recorded in the prediction tables. Our proposed organization splits the addresses computed by the load instructions in two parts: the high-order bits and the low-order bits. Addresses with the same high-order bits share the only copy of these bits.

We also show that: a) management of empty entries in the table that records the high-order bits (HAT), and b) filtering-out the allocations of high-order bits related to unpredictable load instruction reduces capacity misses in HAT and improves the performance of the organization.

The inclusion of this organization and control in a typical last-address predictor, or in an enhanced address predictor with filtering capacity, reduces the area-cost of the predictor without performance loss.

For *b*=14 and 64 HAT entries, both the LAP and the TLAP are predicting correctly and mispredicting the same dynamic load instructions. A processor that implements the TLAP will thus obtain the same IPC improvement as that obtained by implementing the LAP with a significant area-cost reduction.

Other prediction models (stride-based, context-based and hybrid) can also make the most of the spatial-locality of the addresses to reduce their area cost.

4.8 Detailed results

4.8.1 Predictability captured by the TLAP

Figure 4.13 presents the predictability captured by the TLAP on small and medium benchmarks.

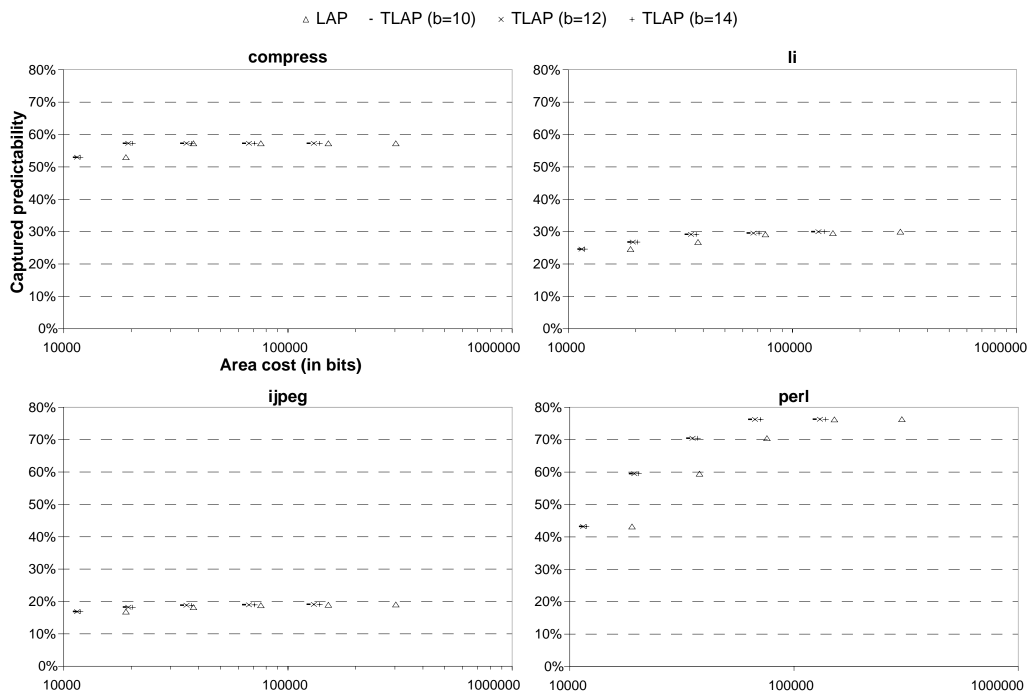


Figure 4.13 Predictability captured by the LAP and the TLAP in small and medium benchmarks

4.8.2 Predictability captured by the DTLAP

Figure 4.14 shows the predictability captured by the DTLAP in small and medium benchmarks.

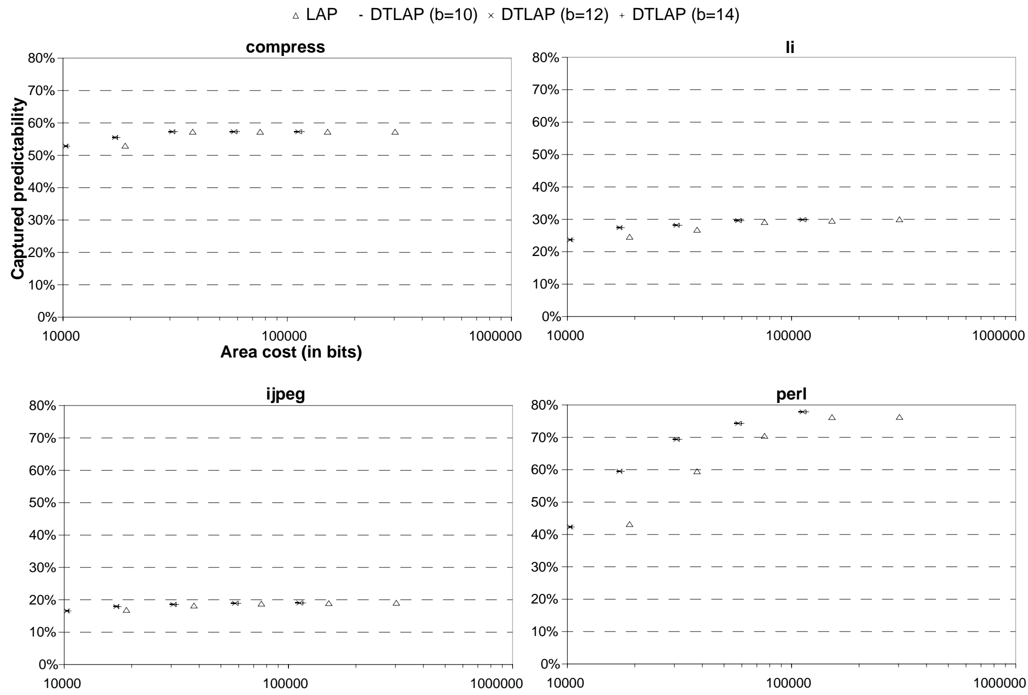


Figure 4.14 Predictability captured by the LAP and the DTLAP in small and medium benchmarks

5 EVALUATION OF ADDRESS PREDICTION

In this chapter, an evaluation of the impact of address prediction and speculative execution on processor performance is presented. To perform this evaluation, it is necessary to overcome a practical problem: the design space to be explored is huge because the impact depends on a plethora of aspects (issue-queue size, first-level-cache latency, issue width, verification mechanism of address predictions, recovery mechanism on address mispredictions,...). This chapter is focused on the influence of some of these aspects on the impact of address prediction: the verification mechanism (and its relationship with branch-instruction resolutions), the issue-queue size (assuming an issue queue

decoupled from the reorder buffer) and the speculative-issue police (and its relationship with the recovery mechanism). This chapter is organized as follows. In Section 5.1, this chapter is introduced. In Section 5.2, the processor model used in the evaluations is described. In Section 5.3, a basic verification mechanism is reported, and it is also evaluated in the scope of address-speculative processors with non-speculative branch resolutions. In Section 5.4, the verification through the verification-flow graph is introduced; in Section 5.5 and Section 5.6, two verification mechanisms based on the verification-flow graph are detailed, and they are also evaluated in the same scope as Section 5.3. In Section 5.7, the previous verification mechanisms are evaluated in the scope of removing the instructions from the issue queue in processors that decouple the issue queue from the reorder buffer. In Section 5.8, the delayed speculative issue policy is presented. In Section 5.9, the address-speculative processors presented in this chapter are compared. Finally, in Section 5.10, some related works are reported, in Section 5.11, the chapter is concluded, and in Section 5.12, detailed performance results are presented.

5.1 Introduction

Branch prediction and speculative execution are techniques used by all superscalar processors to deal with control-flow dependences. As both techniques have been used since early processors, many works have evaluated the impact of these techniques taking into account a wide range of parameters [SAMC98]. Some works have recently proposed the use of prediction and speculative execution to deal with true-data dependences. These works propose the prediction of the results of the register-writing instructions (value prediction) or the effective addresses computed by the load instructions (address prediction) in order to execute speculatively their dependent instructions. As the increasing load latency is one of the key challenges on processor design, we have focused on address prediction.

Several works ([Sato98][ReCa98][AEK+01]) have presented evaluations of address prediction and speculative execution. Although these evaluations yield promising results, they focus on limited ranges of prediction environments. For instance, they use large issue queues and, with respect to the recovery mechanism, either do not propose its practical implementation or use a non-scalable one (they assume that the number of issue-queue entries is equal to the number of reorder-buffer entries). Further analysis are needed to observe the influence of several design parameters: a) in which moment a prediction-check result is used for driving actions (after performing the check, on commit time), b) the propagation of the verification/invalidation of the instructions (serially, in an enhanced way), c) the use of issue queues with a number of entries smaller than that of the reorder buffer. Also, one may consider the pipeline timing of the communication signals between the issue-queue device, which dynamically schedules instructions, and the device that takes actions after checking the predictions.

In the following paragraphs, the main aspects analysed in this chapter are described.

Using address prediction, the result of the speculative execution of a branch instruction may differ from the branch-prediction result; that is, the processor may have to deal with a branch misprediction detected speculatively. In the scope of value prediction, Sodani and Sohi [SoSo98] evaluated two approaches to deal with these branch mispredictions: a) by allowing speculative branch resolutions and b) by delaying branch resolutions until the branch-instruction operands are known to be non-speculative. Their average results show that the speculative branch resolution presents about a 3% performance degradation with respect to the non-speculative branch resolution. In this work we employ the non-speculative branch-resolution approach (like Sazeides [Saze99], Rotenberg [Rote99],...), and in this chapter we evaluate several alternatives for verifying the operands of the branch instructions: implicit verification on commit (*implicit verification*), verification through the serial verification-flow graph (*serial verification*), and verification through the enhanced verification-flow graph (*enhanced verification*).

In most existing processors, the number of issue-queue entries is smaller than the number of reorder-buffer entries, since the use of larger issue queues may increase cycle time [PJS97]. Although some previous evaluations [Sato98][ReCa98] have assumed processors where the number of entries of both structures is the same (such as AMD K6 [Half96], HP PA-8000 [Hunt95]), our evaluations assume processors where the issue queue has a number of entries smaller than that of the reorder buffer (such as Alpha 21264 [Kess99], Sparc V [Dief99], Pentium 4 [Carm00]) because the issue queue is less scalable than the reorder buffer. In this chapter we evaluate the influence of the number of issue-queue entries on the impact of address prediction and speculative execution.

After performing a speculative memory access, its dependent instructions may be issued. We consider two speculative issue policies for the dependent instructions. First, an instruction dependent on a speculative memory access may be issued as soon as the speculatively memory access has been performed (*non-delayed speculative issue*). Second, an instruction dependent on a speculative memory access must be issued after issuing the effective-address computation for the predicted load instruction (*delayed speculative issue*). Although the non-delayed alternative has more performance potential than the delayed alternative, we are interested in the delayed speculative issue because it can be considered as a particular case of a speculative technique named latency prediction. Then, the delayed speculative issue can use a specific recovery mechanism that reduces the pressure on the issue queue with respect to a more generic recovery mechanism. In this chapter we evaluate the influence of the speculative-issue police on the impact of address prediction, and its sensitivity to the number of issue-queue entries.

The evaluated processor configurations consider existing, 4-way processors and next-generation, 8-way processors. Also, as first-level-cache access latency is expected to increase on next generation processors, we evaluate first-level-cache access latencies that range from two to four cycles.

The evaluations will be performed using a detailed cycle-by-cycle simulator (Section A.1.2). Unless noted, we will present the harmonic average of the performance results for the SPEC95-INT benchmarks (Section A.2) in 4-way and in 8-way processors; we will plot the harmonic average performance versus the cache latency or versus the issue-queue size.

5.2 Processor model

This section details the main characteristics of the processor models used in the evaluations performed in this chapter. The first subsection describes the baseline processors (without address prediction) and next subsection describes the address-speculative processors.

5.2.1 Baseline processors

Our baseline processor model is similar to an existing superscalar processor. Its block organization is depicted in Figure 5.1 (solid lines and hollow boxes). The fetch unit generates the effective addresses of the instructions that will be fetched from the instruction cache. To achieve high instruction-fetch bandwidths, the fetch unit also predicts the behaviour of the branch instructions. The outcomes of the conditional branches are predicted using an hybrid predictor that combines a local predictor and a global predictor (*gshare*); branch targets are predicted using a Branch Table Buffer (BTB) and a Return Address Stack (RAS).

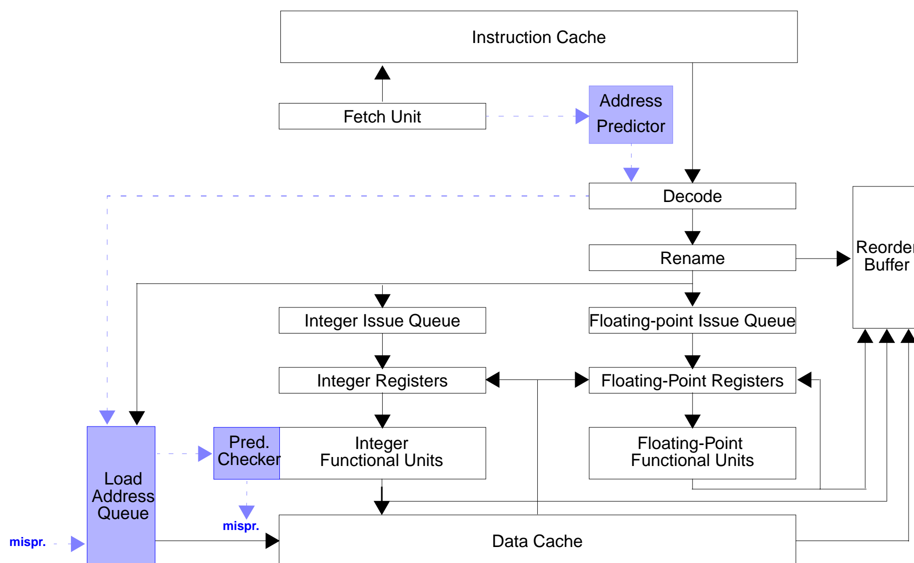


Figure 5.1 Block organization of the baseline processor (hollow boxes and solid lines) and the address-speculative processor (all boxes and lines)

After fetching the instructions, they are decoded and their architectural-register operands are renamed into physical registers. Then, each renamed instruction is inserted into the issue queue and into the reorder buffer.

Each renamed instruction waits in the issue queue for the availability of its source operands

and its functional unit. When these requirement are satisfied, the instruction is ready to be issued. Then, a component of the issue queue named *select logic*, selects the oldest instructions for issuing them.

After that, the source operands of the instruction are obtained from the physical-register file or from a bypass network (not shown in the figure). Next, the instruction is executed in the appropriate functional unit. When the result is available, it is bypassed to its data-dependent instructions and it is written in the register file. Moreover, all the functional units notify the identifiers of the executed instructions to the Reorder Buffer.

Finally, the instructions whose execution has been completed are retired from the Reorder Buffer in program order. When an instruction is retired, the physical register assigned to the previous mapping of the architectural register is freed.

Figure 5.2 presents the processor pipeline related to this organization. However, some stages may take several processor cycles.

Figure 5.2 Processor pipeline assumed in this work

The processor pipeline can be divided into two parts:

- Front end. This is responsible for the initial in-order processing of the instructions; that is, fetching, decoding, renaming and inserting them into the issue queue.
- Back-end. This is responsible for the out-of-order execution of the instructions and the in-order instruction commitment.

The first component of the front-end pipeline is the fetch engine. It is in charge of providing the instructions to be decoded. In detail, the fetch engine accesses the instruction cache to obtain some cache lines, selects instructions from the accessed lines, aligns the selected instructions, drives them to the decoders, and performs branch prediction. In back-end pipeline, the execution of a load instruction requires computing its effective address, a memory access, and driving data to the bypass network and to the register file.

The clock frequencies of future processors is expected to be higher than those of existing processors, and wire delays are expected to increment the effective access latency of the memory operations. These expected tendencies may increase both the fetch-engine latency and the data-cache access latency. To reflect this trend, in this work we evaluate fetch engines and data caches whose latencies range from two to four cycles; at each configuration we suppose that both latencies are equal.

Alpha 21264 processor is an example of a two-cycle-latency fetch engine. Its instruction cache is accessed during the first cycle, and the instructions are aligned and driven to the decoders during the second cycle. A four-cycle-latency fetch engine may reflect a two-cycle-latency instruction cache and an additional two-cycle latency for aligning the instructions and driving them to the decoders. In case that the instruction cache is non-pipelined, or the instruction cache is accessed on alternate cycles because branch prediction takes two cycles, instruction fetches must be double-width in order to maintain the instruction-fetch bandwidth seen by the decoders.

We have supposed that the remaining components of the front-end pipeline (decoding, renaming and inserting into the issue queue) have a fixed latency of four cycles. Figure 5.3 shows the processor stages for executing a load instruction when fetch-engine and cache-access latencies are two cycles.

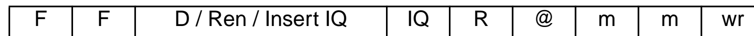


Figure 5.3 Execution of a load instruction

Table 5.1 details the characteristics of the baseline processors used in this work: a 4-way processor (resembles the Alpha 21264 processor) and a 8-way processor.

	4 way	8 way
Fetch width	4 instructions	8 instructions
Fetch-engine latency	2 cycles	
Branch Prediction	2^{15}-entry local predictor, 2^{15}-entry global predictor, 32-entry RAS 1024-entry, 4-way BTB	
	1 taken branch per cycle	2 taken branches per cycle
Reorder Buffer	128 entries	256 entries
Memory dependences	Oracle Prediction	
Decode width	4 instructions	8 instructions
Branch Misprediction Penalty	6 cycles + fetch-engine latency	
Issue width	4 Integer + 2 Floating-point	8 Integer + 4 Floating-point
Functional units	4 Integer ALUs (2 ALU's can compute effective addresses) 1 Integer Mult./Div. 1 FP ALU, 1 FP Mult./Div. 2 memory ports (any combination of loads/stores)	8 Integer ALU's (4 ALU's can compute effective addresses) 2 Integer Mult./Div. 2 FP ALU's, 2 FP Mult./Div. 4 memory ports (any combination of loads/stores)
Latencies	1-cycle ALU, 3/20-cycle integer Mult./Div. 4-cycle FP ALU, 4/12-cycle FP Mult./Div. 2-cycle data cache	
First-level caches	Separated caches, 64Kbyte, direct-mapped, write back, write allocate	
Second-level cache	Unified, 1 Megabyte direct-mapped, 12-cycle latency	
Main memory	80-cycle latency	
Commit width	8 instructions	16 instructions

Table 5.1 Baseline processor configurations used in this thesis

5.2.2 Address-speculative processors

To perform address prediction and speculative execution, the baseline processor organization must be modified. The following subsections detail some characteristics of the processors with address prediction and speculative execution. In this chapter, we will refer to these processors as *address-speculative processors*.

Differences with respect to the baseline processor

Figure 5.1 depicts the processor organization that performs address prediction and speculative execution. Its main differences with respect to the baseline organization are detailed in the following list:

- Concurrently with instruction fetch, the address predictor accesses prediction tables to obtain address predictions for the load instructions being fetched. We assume an optimistic address predictor with unbounded bandwidth, the same latency as the fetch engine and immediate update of the address tables.

- Each predicted load instruction is inserted in the Load-Address Queue (LAQ) during the decode stage. Moreover, the predicted address and the mapping for the destination register must be recorded in the related LAQ entry. After that, the speculative memory access can be issued from the LAQ. Unless indicated, we assume that predicted effective addresses are inserted into the LAQ during the first cycle of the decode stage.

- The number of cache accesses that can be performed every cycle remains unchanged with respect to the baseline processor. An arbiter prioritizes memory access initiated from the issue queue respect memory accesses initiated from the LAQ.

- A functional unit must check the correctness of each prediction. The functional unit that performs this checking obtains the predicted effective address from the LAQ.

- A verification mechanism must detect when a register value is known to be non-speculative. As there are several alternatives to implement this mechanism (they are discussed in Section 5.3, Section 5.4, Section 5.5 and Section 5.6), it is not depicted in Figure 5.1.

The processor pipeline associated to the previous organization is the same as that of the baseline processor. Figure 5.4 shows the execution of predicted load instructions, we have assumed that both the fetch-engine and the cache-access latencies are two cycles, and the availability of a free data-cache port for the predicted memory access. Figure 5.4-a depicts the execution of a correctly predicted load instruction. Figure 5.4-b describes the execution of a incorrectly predicted load instruction.

In our evaluations, we have varied both the fetch-engine latency and the first-level data-cache latency in the same way as in the baseline processors. Section 5.7.4 presents an evaluation of

the sensitivity of the processor performance to the moment when a speculative memory access that uses a predicted address can be issued. That is, the predicted load is inserted in the LAQ structure later than in Figure 5.4.

F	F	D / Ren / Insert IQ				IQ	R	@
pred	pred	LAQ	m	m	wr			check

a) Execution of a correctly predicted load instruction
(actions related to the speculative execution have been shaded)

F	F	D / Ren / Insert IQ				IQ	R	@	m	m	wr
pred	pred	LAQ	m	m	wr			check			

b) Execution of a incorrectly predicted load instruction
(actions related to the speculative execution have been shaded)

Figure 5.4 Execution of a load instruction assuming: a) correctly predicted load instruction and b) incorrectly predicted load instruction

Finally, the address-speculative processors use the non-speculative branch resolution; that is, the operands of a branch instruction must be known to be non-speculative before resolving the branch instruction.

Address predictor

All the evaluations of address-speculative processors performed in this chapter use an Hybrid Address Predictor (Section 2.3.4) made up of a conventional Stride-Address Predictor, a Context-Address Predictor [BJR+99] and a bimodal selector.

Unless indicated, the evaluations presented in this chapter use a configuration of the previous address predictor with large prediction tables: the Stride-Address Predictor has a 16K-entry Address Table and the Context-Address Predictor has 16K-entry Value History Table and Value Prediction Table. Using this configuration, we stress all the mechanisms specific to address prediction.

We have considered a *real* and an *oracle* version of the address predictor; both versions correctly predict the same load instructions. However, the oracle version performs no mispredictions. Comparing the results for both versions, we are able to measure the influence of address mispredictions on processor performance. Unless noted, we present results for the *real* version of the address predictor.

Table 5.2 presents the captured predictability and the accuracy obtained by the real version of the address predictor on SPEC95-INT benchmarks. In the case of the oracle version, its

captured predictability is the same as that of the real predictor, and its accuracy is 100%.

Benchmark	Captured predictability	Accuracy
go	59.16	91.65
m88ksim	96.27	97.81
gcc	83.26	95.02
compress	76.16	96.25
li	84.98	93.78
ijpeg	75.20	95.96
perl	98.99	99.33
vortex	90.66	95.47
average	83.08	96.00

Table 5.2 Captured predictability and accuracy obtained by the address predictor on SPEC95-INT benchmarks

Speculative issue policies

We consider two alternatives for performing the speculative issue of the instructions dependent on a predicted load instruction: *non-delayed issue* and *delayed issue*. In the first case, an instruction can be issued speculatively as soon as it enters into the issue queue. In the second case, the speculative issue is delayed until issuing the effective-address computation of the predicted load instruction it depends on. Figure 5.5 shows the execution of a predicted load instruction and a dependent instruction using both alternatives. In this example, the delayed-issue alternative delays the issuing of the dependent instructions one cycle with respect to the non-delayed alternative. The first instruction dependent on the predicted load instruction must be issued on the cycle next to issuing the effective-address computation of the load instruction.

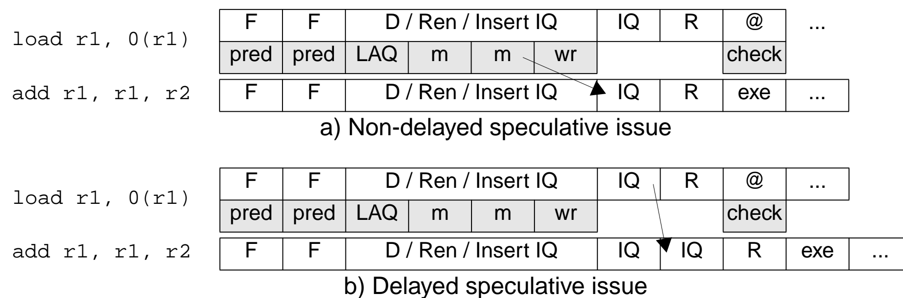


Figure 5.5 Execution of a predicted load instruction and the speculative issue of a dependent instruction: a) non delaying the issuing of the dependent instruction and b) delaying it

The non-delayed alternative is more aggressive than the delayed one because the computation of the effective address of the predicted load instruction does not delay the speculative issue of the dependent instructions. When the address computation is issued, all its dependent instructions may have been executed speculatively. However, the non-delayed alternative can stress the issue queue because the propagation process for verifying instructions

requires the speculatively issued instructions to be maintained in the issue queue until they are verified.

The delayed alternative is a particular case of a speculative technique named latency prediction, which is applied by several superscalar processors (for instance, Alpha 21264 processor performs load hit/miss prediction[Kess99]). Our evaluations of the delayed alternative assume the use of a recovery mechanism named *Recovery Buffer*. This mechanism is specific for latency mispredictions, and allows removing the speculatively issued instructions from the issue queue as soon as they are issued. We describe the Recovery Buffer in Chapter 6.

In this chapter, we evaluate both alternatives in order to decide if the delayed one may represent a cost-effective alternative to the non-delayed one.

Address-prediction checkings

As address predictions may be incorrect, the processor must check the address predictions to determine if the speculative work dependent on the predictions can be committed. Consequently, for each predicted load instruction, its predicted effective-address must be compared to its non-speculative effective address. There are several alternatives to perform this check.

- Serially. After computing the non-speculative effective address, it is compared to the predicted effective address.

- Concurrently to the non-speculative effective-address computation (without waiting for the carry propagation of the effective-address computation). We describe two alternatives to perform it:
  a) Cortadella and Llabería [CoLl92] proposed a circuit that evaluates *A+B=K* conditions without carry propagation. Using this circuit, the prediction is checked by setting *A* to the non-speculative base-register value, *B* to the load-instruction displacement and *K* to the predicted effective address.
  b) After performing an effective-address prediction, the base address related to this prediction can be computed by substracting the load-instruction displacement from the predicted address. The address-prediction check then consists in comparing the base address of the prediction to the non-speculative base-register value.
  As the latency of both concurrent alternatives is smaller than the latency of the adder, the address-misprediction signal can be generated during the computation of the non-speculative effective address.

In this work we use the concurrent alternative since it allows a faster detection of the mispredictions. Moreover, correct predictions are notified to the verification circuit in order to be used on the next cycle.

Finally, we assume that the address-prediction checkings are performed only by the ALU's that compute effective addresses. Enabling all the ALU's to perform address-prediction

checkings may be an interesting alternative because produces an increment of the checking bandwidth, and then the dependent instructions can be issued earlier; however, after a misprediction, the load instruction must be re-issued.

Recovery mechanism on address mispredictions

As some address predictions may be wrong, some instructions executed speculatively may have used incorrect input data. To maintain program correctness, these instructions must be re-executed. The mechanism responsible for maintaining program correctness despite address mispredictions is the recovery mechanism. There are several alternatives to implement this recovery mechanism:

- *re-fetch* versus *re-issue*: On *re-fetch* mechanisms, the instructions to be re-executed must be read again from memory. This alternative is also named squashing, and it is similar to the recovery mechanism on branch mispredictions because all the instructions younger than the mispredicted load instruction are flushed-out from the pipeline and the fetch unit is redirected to the next instruction after the mispredicted load instruction.
  On *re-issue* mechanisms, the instructions are not read again from memory because the processor keeps enough information to re-execute them. *Re-issue* mechanisms can be classified by considering which storage structure (the issue queue or an additional structure) is used for keeping this information.

- *selective* versus *non-selective*: The *re-issue* mechanisms can be classified into *selective* and *non-selective*. On *selective* mechanisms, only the speculatively issued instructions dependent on the misprediction are re-issued; on *non-selective* mechanisms, some independent instructions may also be re-issued.

In this chapter we will use recovery mechanisms that re-issue instructions selectively.

Re-issue mechanisms

As previously pointed out, the re-issue mechanisms may differ on the storage structure used for keeping the speculatively issued instructions. We describe some alternatives which have been applied in several speculative scopes (address prediction, value prediction and latency prediction).

On one hand, several authors (Sato [Sato98], Sazeides [Saze99]) have proposed using the issue queue. They assumed a processor where the reorder buffer and the issue queue are combined into a structure named Register Update Unit (RUU) [SoVa87]; that is, both the number of issue-queue entries and the number of reorder-buffer entries are the same. Consequently, the RUU keeps all the issued instructions until they are committed (or until they are flushed-out due to a branch misprediction). Every RUU entry monitorizes the results produced by the remaining instructions allocated in the RUU. When all the operands of an instruction are available, the instruction can be selected for issue. After issuing an instruction, it is marked as non-visible to the issue logic; that is, the instruction is prevented from being selected again for issue.

The same authors extended the RUU to deal with address or value mispredictions. First, to identify predicted instructions, they add a new field at every RUU entry. Second, to re-issue the instructions dependent on a misprediction, they noted that its dependent instructions issued speculatively are still allocated in the RUU; that is, although these instructions are non-visible to the issue logic, they are still monitorizing the results produced by other instructions. Consequently, when a non visible instruction detects a new value for one of its operands, the instruction will be made visible again to allow its re-issue.

On the other hand, Morancho et al. [MLO01] proposed the use of a new storage structure to record the speculatively issued instructions. They focus on the scope of latency prediction; in this scope, the instructions issued speculatively are issued after issuing the predicted instruction, and the interval from issuing a predicted instruction until checking its prediction is fixed. Taking advantage of both facts, a recovery mechanism specific for latency prediction can be designed; this mechanism is named *Recovery Buffer,* and it allows removing the speculatively issued instructions from the issue queue as soon as they are issued. As the delayed speculative issue is a particular case of latency prediction (the latency of some load instructions is predicted to be one cycle), our evaluations of the address-speculative processors with delayed speculative issue assume the use of the recovery buffer.

Invalidation mechanism

An address misprediction requires a recovery action because the speculatively issued dependent instructions have produced wrong results due to the use of misspeculated source operands. After detecting the address misprediction, there are several alternatives to invalidate these wrong results: *serial invalidation* and *parallel invalidation*.

- *Serial invalidation*. After detecting the misprediction, the mechanism invalidates the destination registers of the instructions as soon as they are re-issued. Consequently, after detecting the address misprediction, a wrong result not directly dependent on the misprediction may still be used and may also produce new wrong results. Note that the program correctness is guaranteed because all these instructions will be re-issued. Although this process wastes issue slots, it represents a performance degradation only if younger instructions with correct operands are not selected for issue. Sato [Sato98] used this alternative.

- *Parallel invalidation*. After detecting the misprediction, the mechanism invalidates concurrently the destination registers of all the issued instructions dependent on the misprediction. Consequently, no instruction will use these wrong results. Sazeides [Saze99] used this alternative in the scope of value prediction.

In this work, we combine the non-delayed speculative issue with two invalidation mechanisms: serial invalidation and enhanced invalidation; we combine the delayed speculative issue with the parallel invalidation. Note that our schedulers select the oldest ready instructions for issue. Consequently, we expect that the serial invalidation mechanism will produce only a

small performance degradation.

Verification process

The verification process is responsible for propagating that an operand has become non-speculative to its consumer instructions. This process influences branch-resolution delay because we use non-speculative branch resolution; that is, to resolve a branch instruction, its operands must be verified. Section 5.4, Section 5.5, Section 5.6 and Section 5.7 discuss several strategies to perform the verification process.

Influence of the verification process on processor performance

In the literature, there are several alternatives to perform the verification process. On one hand, Sato [Sato98] supposed an RRU-based processor[1], and used a mechanism where no specific hardware is devoted to propagating the verifications; we name it *implicit verification on commit*. On the other hand, several works ([Dief99], [Saze99]) described the use of additional hardware that allows a faster verification process. a) Diefendorff [Dief99] depicts how the Sparc64 V processor speculates on the ordering of the memory accesses; however, the author does not detail the implementation of the mechanism and only points out some ideas (for instance, relating new state information to the registers). b) Sazeides ([Saze99]) proposed a combined parallel verification/invalidation mechanism in the scope of value prediction. His implementation assumes that, after detecting a correct prediction, the verification can be propagated serially to all the instructions dependent on the prediction on a single cycle.

In next sections we evaluate the influence on processor performance of when the actions (verification process) driven by address-check results are initiated. These evaluations are first performed assuming that both the issue-queue and the reorder-buffer devices have the same number of entries; consequently, these evaluations focus on the impact of the verification process just on branch-instruction resolution.

We evaluate three verification mechanisms: the implicit verification, the serial verification and the enhanced verification. First, we present results for the implicit verification. Then, we present a design of a verification unit (the Verification Issue Queue) which is used for implementing the serial verification and the enhanced verification. Next, we show processor performance using these verification mechanisms.

Finally we evaluate processor performance for processors where the number of issue-queue entries is smaller than the number of reorder-buffer entries. Our evaluations compare the baseline processors (without address prediction) versus the address-speculative processors.

---

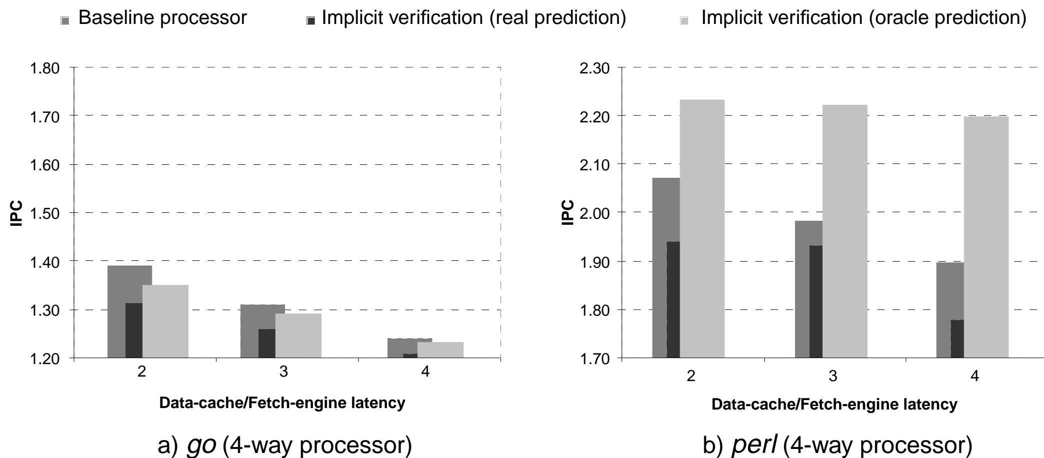
1. RUU-based processors have the reorder buffer and the issue queue grouped into the same structure: the RUU. Our evaluations assume 128 RUU entries for the 4-way processors, and 256 RUU entries for the 8-way processors.

5.3 Implicit verification on commit

This mechanism takes advantage of the fact that the instruction in the head entry of the RUU has non-speculative operands because no previous instruction can modify them. Then, an instruction gets verified when it reaches the head entry of the RUU and its execution has been completed.

Using this strategy, the instructions are verified in program order. However, in the same way that several instructions can be committed on the same cycle, several instructions can be verified on the same cycle; that is, a parallel verification of up to commit-width instructions ([Saze99]).

Although this mechanism is the simplest one (no specific hardware is devoted to propagate verifications), it has a side effect that can affect processor performance. As we decided that branch instructions will not be resolved until their operands are verified, the verification based on retirement increases branch penalty because branch resolution is delayed until commit stage. This overpenalty decreases the gain due to address prediction and, in some cases, may produce net performances smaller than that of the baseline processors.



a) *go* (4-way processor) b) *perl* (4-way processor)

Figure 5.6 IPC versus data-cache latency in baseline and in address-speculative processors with implicit verification mechanism (for benchmarks *go* and *perl*)

Figure 5.6 shows the performance of baseline processors and address-speculative processors with the implicit verification mechanism executing benchmarks *go* (a) and *perl* (b). In the case of *go* benchmark, the penalty due to delaying branch resolutions is bigger than the benefits of address prediction; we can affirm it because the performance of the address-speculative processor that uses the oracle predictor is smaller than that of the baseline processor.

Using the real predictor, the address-misprediction penalty delays branch resolutions, including the resolutions of the branch instructions independent on address mispredictions. On branch mispredictions, the recovery action for the branch misprediction is delayed until commit

stage and the performance degradation may be significant. In the case of *perl* benchmark, considering the penalties produced by address mispredictions degrades significantly the potential performance of address prediction.

Note that, on an address misprediction, the speculatively issued instructions have used resources that could have been used to execute younger, prediction independent instructions. Then, after detecting the address misprediction, these younger instructions must be executed and the speculatively issued instructions must be re-executed. This produces a performance loss that depends on a) the length of the chain of dependent instructions executed speculatively, b) the amount of prediction independent instructions, c) the chains of dependent instructions which speculative execution depends on several address predictions and d) address predictions dependent on other address predictions. In the last case, the effective addresses of a load instruction and that of a dependent load instruction may be predicted; then, the younger load instruction and its chain of dependent instructions may be executed up to three times until verifying its results. Performance loss is more significant on programs with high ILP because the execution of some instructions independent on the misprediction is delayed.

We also present the average results in Figure 5.7. On average, address mispredictions degrade the potential performance of address prediction around 2% (4-way processors) and from 9% to 6%. In some cases, address mispredictions produce that the average performance of address-speculative processors is smaller than that of baseline processors.

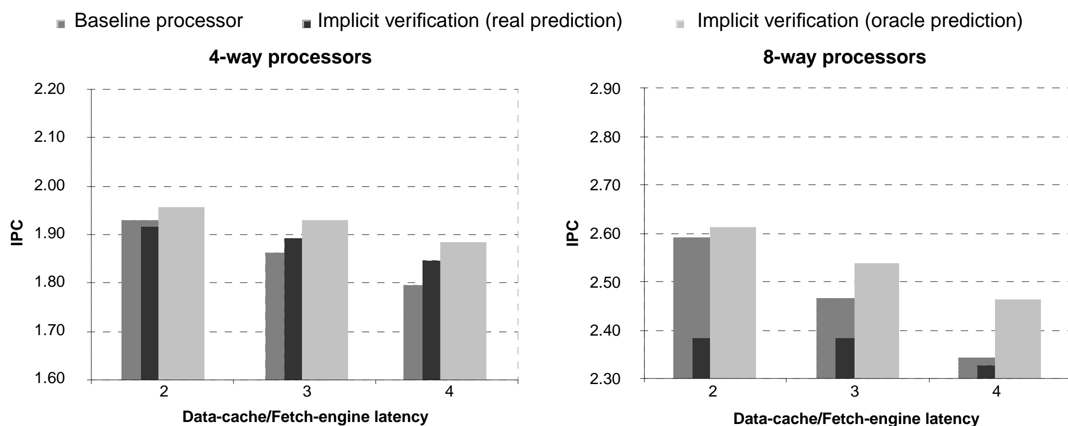


Figure 5.7 IPC versus data-cache latency in baseline and in address-speculative processors with implicit verification mechanism

These results suggest that faster verification mechanisms must be evaluated, because delaying the resolution of mispredicted branch instructions does not allow a full exploitation of the potentiality of address prediction.

5.4 Verification through the verification-flow graph

We first describe the basic idea of the mechanism. The processor executes concurrently two flow graphs: a data-flow graph and an associated verification-flow graph. The data-flow graph is traversed at operation latency. The verification-flow graph may be serial or enhanced. The serial verification-flow graph is identical to the data-flow graph, but the latency of all instructions is one cycle. In Section 5.6, we describe an Enhanced verification-flow graph that allows a faster traverse of the dependence chains.

Once an address-prediction has been checked, the verification-flow graph is traversed. The traverse starts at the predicted load instruction, and advances at an uniform single-cycle rate by level of the verification-flow graph. As this work assumes a selective invalidation mechanism, the verification-flow graph is also traversed to propagate invalidations in case of misspeculations.

The data-flow graph is managed by the Issue Queue (IQ) and the verification-flow graph is managed by the Verification Issue Queue (VIQ). The physical-register state information is composed by two components: a) data value is *available*/*non-available* and b) data value is *speculative*/*valid*. Each component is handled by a different device. The IQ tracks if data value is available/non-available and the VIQ tracks the other component. In this work, we refer to each component as *register state*; context will usually disambiguate its meaning, otherwise, it will be explicited.

Figure 5.8 shows the external inputs and the communication between the IQ and the VIQ devices. Address-predicted load instructions are issued from the LAQ and, when data is available, their destination-register state (available, non-available) is updated in the IQ for waking-up speculatively their dependent instructions. The woken-up instructions will compete to be selected for issue and, after issuing them, they will wake-up speculatively their dependent instructions. To perform a fast recovery action in a misspeculation, the issued instructions are kept in the IQ while they are speculative.

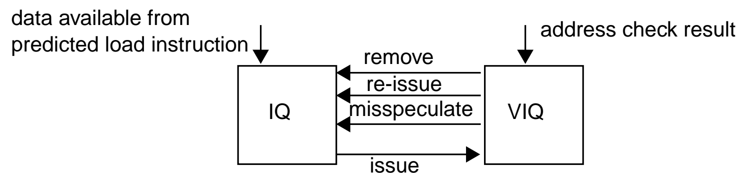


Figure 5.8 External inputs and communication between the Issue Queue and the Verification Issue Queue

Instructions are verified/invalidated by tracking source-register states (speculative, valid) in the Verification Issue Queue (VIQ). An instruction (other than an address-predicted load) is verified when all its source operands are valid, and it is invalidated when some of its operands are misspeculated. Verified/invalidated information is communicated to the IQ for removing (verified), or re-issuing (invalidated) speculatively issued dependent instructions.

When the source operands of an address-predicted load instruction are valid, and the address-check result (correct/incorrect)[2] is known, the verification/invalidation process of a chain of speculatively issued dependent instructions is initiated.

This section is organized as follows. Firstly, we describe the information flow for verifying/invalidating instructions. Secondly, we detail the Issue Queue and the Verification Issue Queue. Finally, we present some timing considerations on the communication between the Issue Queue and the Verification Issue Queue.

5.4.1 Information flow for verifying/invalidating instructions

After the Rename stage, instruction information is inserted in both the IQ and the VIQ. Each cycle, the IQ communicates to the VIQ which instructions are issued. When the source operand of an address-predicted load instruction is valid, the load instruction is issued from the IQ and performs address check.

In Figure 5.9 is showed the information flow for verificating/invalidating the instructions after address check; we suppose that the serial verification-flow graph is used. After issuing an address-predicted load instruction, its computed address is compared for equality with the predicted address in the check stage, and address-check result (correct/incorrect) is notified to the VIQ device. When the cycle is finished it is know if the address-predicted load instruction is verified, or if the predicted address is mispredicted and a new memory access has been initiated. In the later case, the speculatively issued dependent instructions must be invalidated because they have used a misspeculated value.

Verify/invalidate information of an address-predicted load instruction is known by both the IQ and the VIQ devices at the beginning of the next cycle. The load instruction is removed from the IQ device because the load instruction has been issued with a valid source operand; note that in case of an incorrect address-check result, the correct memory access is being issued. However, when predicted address is mispredicted, the destination-register state in the IQ is set to non-available. Thus, the non-issued instructions which directly depend on the load instruction will wait in the IQ for the availability of the new data value.

Concurrently, in the VIQ device, verify/invalidate information of an address-predicted load instruction is used to verify/invalidate the speculatively issued instructions directly dependent on the load instruction (first level of the verification-flow graph).

In the next cycles, verified/invalidated-instruction information is used to remove/re-issue instructions in the IQ and to verify/invalidate speculatively issued instructions of the next level of the verification-flow graph in the VIQ. Thus, verify/invalidate information is propagated to the following levels of the verification-flow graph in a rate of one level by cycle, and misspeculated

2. Result of the comparison of the address stored in LAQ with the computed address: correct prediction or incorrect prediction

instructions are re-issued.

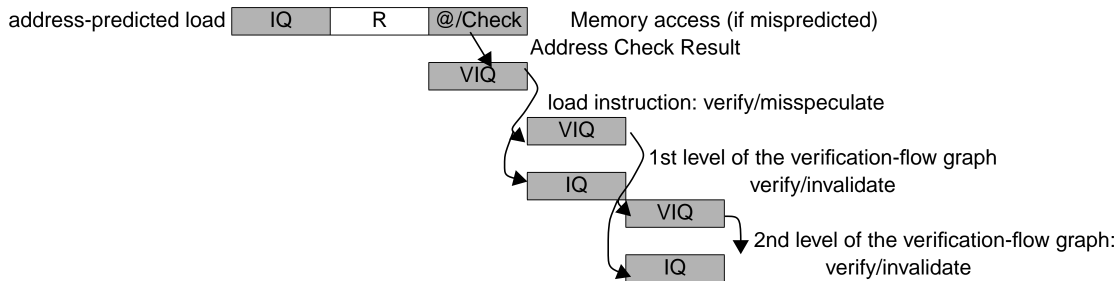


Figure 5.9 Information flow for verifying/invalidating instructions after the address check of a predicted load instruction

5.4.2 Issue Queue

In dynamically-scheduled superscalar processors, instructions wait in the issue queue for the availability of operands and functional units. To issue instructions out-of-order to the functional units, the issue queue has two components: a) *wakeup logic* and b) *select logic*. The *wakeup logic* keeps monitoring the dependencies among the instructions in the issue queue and, when the operands of a queued instruction become available, this logic will mark the instruction as ready. The *select logic* selects which ready instructions will be issued to the functional units on the next cycle. When latency of the selected instructions is elapsed, dependent instructions are woken up.

Dependence matrix

The Issue Queue includes a dependence matrix (Figure 5.10) for tracking dependencies among instructions. The matrix has as many rows as the number of instructions analysed simultaneously for scheduling, and as many columns as the number of physical registers (registers for short).

The columns are wires that cross all rows, and each row includes a bit for each column. Each column marks the data availability of a register, and it is set by a count-down latency counter or by a shift register connected to the column.

When an instruction is inserted in an issue-queue entry, in the associated row of the dependence matrix, the bits of the registers used as source operands of the instruction are set. Also, the latency counter related to the destination register is initialised to the instruction latency[3].

Each crosspoint of the dependence matrix contains a logical circuit that determines if the required source operand is available. For each row, the outputs of these logical circuits are used to compute a *ready bit* that indicates if the instruction is ready to be selected by the *select logic*.

3. In this chapter we suppose an oracle predictor for the load-instruction latency.

Ready bits are evaluated every cycle.

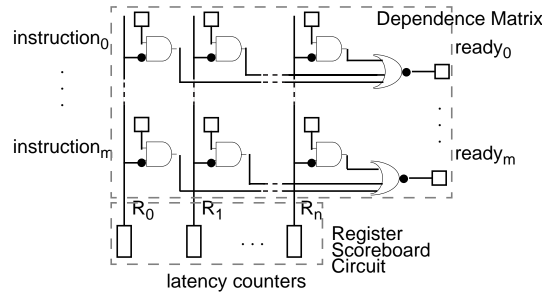


Figure 5.10 Dependence-matrix structure of the Issue Queue

When an instruction is issued, the latency counter related to its destination register is decreased on every cycle. Then, when latency lapses, the column will be set to mark the availability of the result.

Re-issue mechanism

Issued instructions may use speculative source operands. Thus, when a data value is misspeculated, its issued consumer instructions must be re-issued in order to be executed with the new data value. To perform a fast recovery, the issued instructions are kept in the IQ until they are verified. While an issued instruction waits in the IQ, it is made non-visible to the *select logic.* This is handled by using a *no-request bit* in each dependence-matrix row; this bit it is set when the instruction is issued.

When an instruction must be reissued because it uses a misspeculated operand, the instruction is made visible again to the *select logic*. Moreover, its destination-register state is set to non-available; this delays the issuing of its directly dependent instructions until the instruction is re-issued and a new data value is computed.

Two control circuits perform previous operations: the *removal circuit* and the *register-scoreboard circuit*. Figure 5.11 shows the interface between them and other issue-queue elements.

Every cycle, the *removal circuit* is notified by the VIQ of the instructions that must be removed/re-issued, and the *register-scoreboard circuit* is notified of the misspeculated instructions. The *removal circuit* removes verified instructions from the IQ, and makes visible again the instructions that must be re-issued. The *register-scoreboard circuit* activates latency counters (for each issued instruction, and when data of the address-predicted load instruction becomes available) or unsets columns (for each misspeculated instruction).

As *ready bits* are re-evaluated every cycle, misspeculated instructions will re-evaluate their ready bits, waiting for operand availability; also, all instructions directly dependent on the misspeculated instructions will be slept.

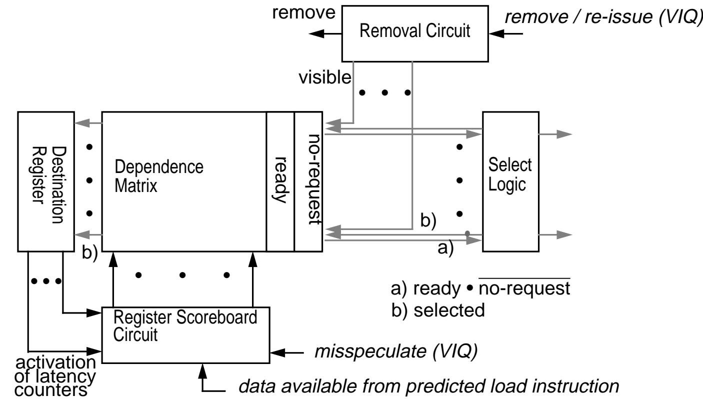


Figure 5.11 Issue-queue structure

Address-predicted load instructions

Address-predicted load instructions are issued from the LAQ in order to perform a speculative memory access; when it is finished, its destination-register state (available, non-available) is set to available in order to wake-up speculatively its directly dependent instructions. However, if an address-predicted load instruction is issued from the issue queue before initiating the speculative memory access, the speculative access is not performed and address-check result is considered as correct in the VIQ. This fact is detected when, in order to perform the address check, the predicted address is read from LAQ.

The IQ is notified of the incorrect address-check results using the misspeculate signal; this signal is used for updating, if it is needed, the destination-register state of the load instructions. Register state is set to non-available when following two conditions are satisfied: a) the address-check result is incorrect (compared addresses do not match) and b) the remaining latency of the load instruction is larger than the latency from the issue stage to the first execution stage.

5.4.3 Verification Issue Queue

In this section we describe the Verification Issue Queue, which implements the verification/invalidation through the verification-flow graph.

All load instructions perform address check in order to update the prediction tables of the address predictor, but address-check result of non-address-predicted load instructions is always correct. Then, the statement “load instructions that perform address check” refers to the load instructions which address-check result may be wrong.

After knowing the address-check result of a load instruction that performs address check, the instruction initiates the verification/invalidation of the chain of speculatively issued dependent instructions. It first verifies/invalidates issued instructions directly dependent (in the serial verification-flow graph) on the load instruction. These verified/invalidated instructions are used, in

next cycle, to verify/invalidate issued instructions that directly depend (in the serial verification-flow graph) on them. Thus, each cycle, issued load instructions that perform address check and verified/invalidated instructions are used to verify/invalidate issued instructions.

We next describe the base structure of the Verification Issue Queue. After that, we describe the use of this base structure when the address check is performed non-speculatively, speculatively and with an Enhanced verification-flow graph.

Verification Issue Queue: elements and input/output signals

The issued instructions, other than load instructions that perform address check, wait in the Verification Issue Queue until knowing if its source operands have became valid or misspeculated. The VIQ has two components (Figure 5.12): a) Verify/Invalidate logic and b) Decision logic.

The Verify/Invalidate logic monitors if the source operands of the instructions are valid or misspeculated. When all source operand of an instruction become valid or one of them is misspeculated, the logic marks the instruction as verified or invalidated respectively.

For each issued instruction, the Decision logic determines if: a) the instruction can be removed, b) the instruction must be re-issued or c) for load instructions that perform address check, the data value has been misspeculated and the speculatively issued dependent instructions must be re-issued. The Decision logic considers: a) the instruction has been issued, b) verify/invalidate signals, c) instruction type (load that performs address check, other instruction), d) address-check result, and e) memorization elements related to the instruction.

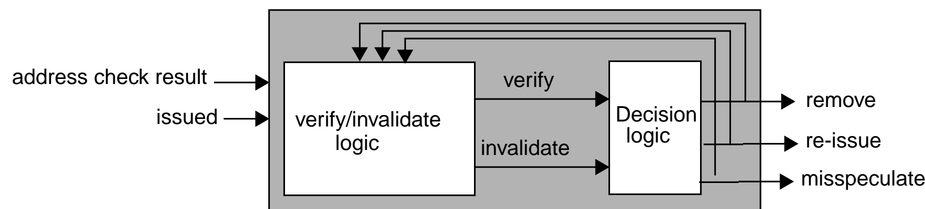


Figure 5.12 Components of the Verification Issue Queue

Decision-logic outputs are used in next cycle to propagate verify/invalidate information to directly dependent (in the verification-flow graph) issued instructions; also, these outputs are sent to the IQ in order to remove/re-issue instructions.

Matrix structure

The core of Verify/Invalidate logic is a matrix structure (Figure 5.13), similar to the matrix used in the IQ device for tracking dependences. This matrix structure monitors if the source operands of the instructions are valid or misspeculated. It has as many rows as the number of instructions in the IQ, and as many columns as the number of physical registers.

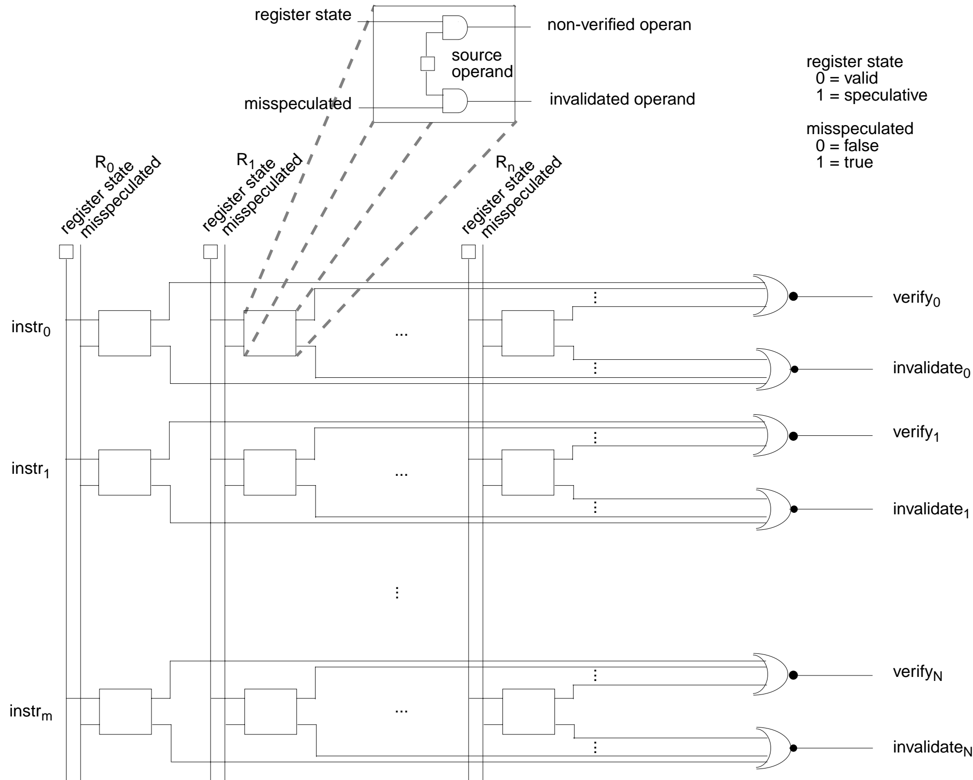


Figure 5.13 Matrix structure of the Verification Issue Queue (VIQ)

The columns are wires that cross all rows and each row includes a bit for each column. Each column marks the register state (speculative, valid) and if the data value has been misspeculated. The state of a physical register is speculative until the Decision logic indicates that the producer instruction, which computes the data value, can be removed. The misspeculated signal is set only for one cycle, when Decision logic indicates that the instructions that have consumed misspeculated data value must be re-issued.

When an instruction is inserted in a matrix entry (row), the bits related to the source operands of the instruction are set. This information is built at the Register Rename stage and inserted in VIQ after this stage.

Each crosspoint of the dependence matrix contains a logical circuit that determines if the source operand is valid or it is misspeculated (Figure 5.13). For each row, the outputs of the logical circuits are used to compute the verify/invalidate signals. Verify signal is activated when all source operands of the instruction are valid. Invalidate signal is activated if any source operand of the instruction is misspeculated. These signals are evaluated every cycle.

Load instructions that perform address check

A load instruction that performs address check establishes if its destination register is valid or misspeculated using two informations related to the same instance of the dynamic load instruction: a) address-check result and b) verify signal of the load instruction (source operand is valid). Then, load instructions that perform address check must be tagged in order to take into account both informations before the Decision Logic takes a decision. For this, we use the *instruction type* bit (load instruction that performs address check, other instruction).

However, for incorrect address-check results, the chain of dependent instructions will be re-issued immediately because the computed address is used for initiating a new memory access after the address-check stage, and this chain of instructions had used misspeculate data values.

Decision logic

Each entry of the matrix structure has a *valid* bit which indicates if the instruction has been issued and it is waiting for the verification/invalidation of its source operands. This bit is set each time the instruction is issued and it is unset (non-valid) when Decision logic decides that the instruction must be re-issued.

For each instruction, the decision logic uses verify/invalidate signals of the matrix structure, valid bits of the entries, address-check results, instruction type and memorization elements to generate remove, re-issue and misspeculate signals.

For a load instruction that performs address check, address-check result and verify signal of the source operand must be paired before deciding if data value is valid. Because both informations may be generated at different time, a memorization element in Decision logic is needed. After pairing both informations, on a correct address-check result, the remove signal is set. In other case, incorrect-address check result, the misspeculate signal is set only for one cycle, and in next cycle remove signal is set[4]. The actions performed in other cases are specific of the issue and address check policies of the load instruction that performs address check.; that is, if the address check is non-speculative/speculative.

For the other instruction type, verify/invalidate signals drive directly remove and re-issue&misspeculate signals respectively (Table 5.3).

matrix structure outputs	Decision logic outputs
verify	remove
invalidate	re-issue & misspeculate

Table 5.3 Decision logic outputs for instructions that non perform address check

Figure 5.14 shows the interface between the Verification Issue Queue elements. Every cycle,

4. As the source operand of the load instruction has been verified, the memory access, initiated after address check, will service the valid data value.

the *removal circuit* and the *register-verify circuit* are notified of the removed/re-issued and the re-issued/misspeculated instructions respectively. The *removal circuit* removes an instruction when then remove signal is activated; also, sets the entry as non-valid when the re-issue signal is activated. The *register-verify circuit* sets the destination-register state to valid on a remove signal, and activates the misspeculate-column signal on a misspeculate signal.

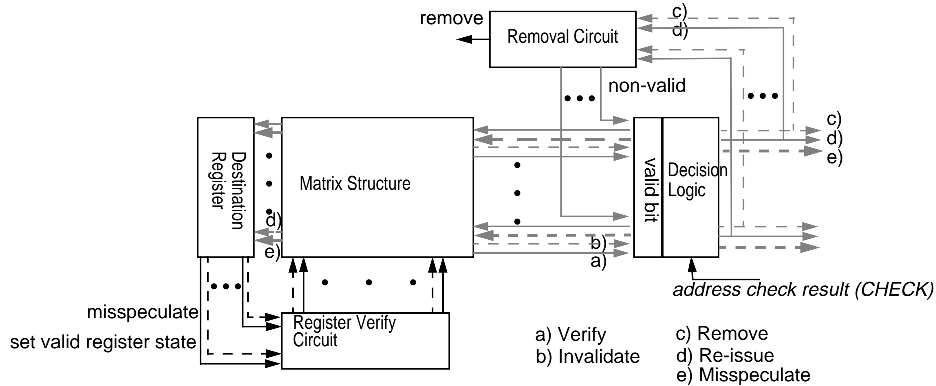


Figure 5.14 Structure of the Verification Issue-Queue

5.4.4 Pipeline timing considerations on the communication between IQ, VIQ and check devices

The IQ and the VIQ devices are communicated in order to coordinate the actions performed by each device. The IQ device communicates to the VIQ which instructions have been issued, and the VIQ communicates to the IQ which instructions must be removed/re-issued, and which register data values have been misspeculated. Moreover, the unit that performs address checks must communicate the address-check result to the VIQ. The pipeline timing of the communication signals must be considered in order to identify critical communication paths.

In order to avoid speculative address checkings, a possible design consists in issuing address-predicted load instructions after validating their source operands. Therefore, these instructions must wait in the IQ until validating their source operands. However, source operands are validated in the VIQ. Then, the VIQ must inform to the IQ that all source operands of the instructions are validated.

To not delay the issuing of an address-predicted load instruction, information on its source-operand validity is needed at the same cycle at which the instruction would be woken-up because the source operand will be available. Therefore, the VIQ must communicate to the IQ the information at this cycle. In Figure 5.15-a is shown an example, where stage usage is represented from the IQ stage. Each instruction is represented by two rows: the upper row shows IQ, Register Read and Execute stages; the lower row shows the VIQ device.

On cycle 1, concurrently to waking-up and selection actions, the VIQ device computes

verify/invalidate signals, and decides the actions to be performed for all instructions allocated in the VIQ; however, these decisions will be effective only for issued instructions. For this, before activating on cycle 2 the column signals in the matrix structure of the VIQ device, the VIQ must known the instructions selected for execution in the previous cycle.

For an address-predicted load instruction, the VIQ device communicates to the IQ if the source operand of the instruction is verified/invalidated (output of the matrix structure). Then, the IQ device uses this information to not issue the instruction until source operand is verified, although the ready bit is set. This can be observed on cycle 3 of Figure 5.15-a. The source operand of the load instruction is available, it is validated, and the load instruction is selected by the select logic.

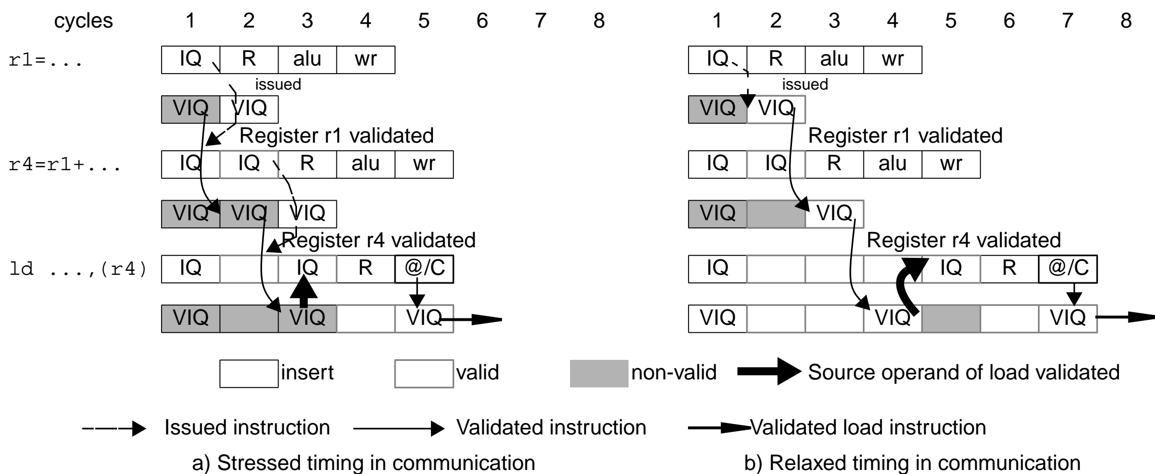


Figure 5.15 Communication between VIQ and IQ devices

We next analyse the cycle-time usage in the IQ and the VIQ devices. The matrix structures of both devices are similar; both have the same number of columns and the same number of rows. However, the VIQ matrix doubles the number of signals in rows and columns with respect to the IQ matrix. Then, the VIQ area is bigger than IQ area, and VIQ wire delays are larger than IQ wire delays. Therefore, VIQ signals (verify, invalidate) are known later than IQ signals (ready). Figure 5.16 shows an of the relative time of the actions performed in both devices.

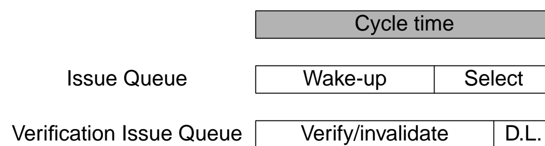


Figure 5.16 Qualitative cycle-time distribution of IQ and VIQ components

To not waste issue slots and to allow issuing younger load instructions while waiting the

source-operand validation, the source-operand information must be know before selection logic begins to work (note that the selection logic prioritizes instructions by instruction age). Therefore, with our previous discussion on time distribution of both devices, this communication affects the cycle time, because the selection logic of IQ must wait until matrix structure of VIQ finishes its computation.

Then, we analyse another design alternative. The VIQ device tracks the source-register state of an instruction the next cycle after issuing it (Figure 5.15-b). Then, the validation of all source registers of an instruction is communicated to the IQ in the cycle that follows the validation of the last source operand of the instruction. In Figure 5.15-b, the source operand of the load instruction is validated on cycle 4, and IQ is informed at the beginning of cycle 5. Moreover, this design relaxes the communication time needed to communicate (from IQ to VIQ) which instructions have been issued on the previous cycle, because this information is used by the Decision logic after the matrix structure finishes its work.

Because the issuing of the address-predicted load instructions is delayed, destination-register validation is also delayed. This delay produces a reduction in the effective size of the IQ, because more entries are allocated to speculatively issued instructions waiting for verification. Moreover, address-misprediction recovery action is initiated later than in the previous design.

To reduce these drawback effects, one may consider the use of instructions older than the producer instruction for validating data values. For instance, we can use all grand-grand-parent instructions of the source operands for validating source operands of an address-predicted load instruction. However, the effectiveness of this technique is limited because the chain of dependent instructions, from a grand-grand-parent instruction to the source operand, can not include address-predicted load instructions.

In next section we develop a design that issues address-predicted load instructions when its source operands are validated and, in other section, we develop a design that uses an enhanced verification-flow graph that validates the source operand of the instructions using instructions older, in the instruction dependent chain, than the parent instruction.

In a later section, we relax the issue policy for address-predicted load instructions. They will be issued with speculative operands and then, address-check result will be speculative until source operands will be verified. With the issue-policy relaxation, the communication restrictions between the VIQ and the IQ do not delay the issuing of address-predicted load instructions.

Address-check result is used by the VIQ to know if the data value of an address-predicted load instruction has been misspeculated. As we use a specific circuit to perform address checks (Section 5.2.2), we assume that address-check result can be used by the Decision Logic of the VIQ at the same cycle in which it is computed. The Decision Logic combines this result with the verify/invalidate signals that computed by the Matrix structure also at this cycle.

While the Decision logic of the VIQ uses local signals of each entry, the Selection logic of the IQ uses signals from all entries. Then, the work time of the Decision logic is shorter than that of the Select logic (Figure 5.17).

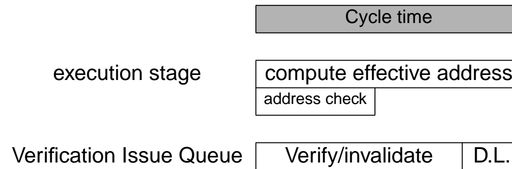


Figure 5.17 Qualitative cycle-time distribution of address checking and VIQ components

5.5 Serial verification through the verification-flow graph

Using serial verification, the verification-flow graph is identical to the data-flow graph, but the latency of all instructions is one cycle.

Firstly, we describe the basic mechanism that supposes that: a) each predicted load instruction waits for execution in the IQ until its operand is validated (address checking is not speculative) and b) each non-predicted load instruction does not perform address checking and may be executed with speculative operands. In later case, if the load instruction is misspeculated, it and their speculatively issued dependent instructions must be invalidated and re-issued. Thus, only address-predicted load instructions perform address check.

Secondly, we describe an extension of the base mechanism where address-predicted load instructions perform checks with speculative source operands and also, non-predicted load instructions perform speculative checks when they are re-issued. Moreover, the design supposes that the number of concurrent instances of a dynamic load instruction, in execution phase, is limited to one instance. Finally we extend the previous design to allow several concurrent instances of a dynamic load instruction.

5.5.1 Non-speculative address check

An address-predicted load instruction must wait in the IQ until validating its source operand. For this, we use the functionality of the VIQ for tracking the source-operand states of the issued instructions with valid bit set. Thus, the VIQ informs to the IQ of the validation of the source operand of the address-predicted load instruction. Then, the instruction is allowed to be selected for issue. When the instruction is issued, the address check is performed and the result is sent to the Decision logic of the VIQ. Now, the Decision logic is aware of both the source-operand validity and the address-check result. Then, the Decision logic decides removing (or re-issuing&misspeculating) the speculatively issued dependent instructions on a correct (or incorrect) address-check result. Figure 5.18 shows a pipeline example of an address-predicted load instruction which is not issued until validating its source operand.

Previous operations require that the VIQ must observe two times the predicted load

instruction. The first time is used for knowing when the source operand becomes valid, and the second time is used for pairing the validity of the source operand and the address-check result. To make this process, the VIQ uses a memorization element that marks if the address-predicted load instruction is observed for the first time or for the second time. This memorization elements is named *check* bit (on, off).

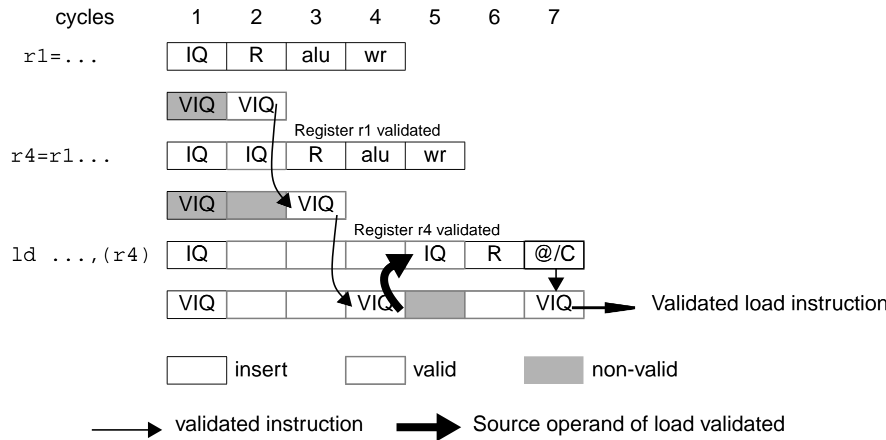


Figure 5.18 Example of a non-speculative address check

To avoid selecting for issue an address-predicted load instruction which source operand is not yet valid, we use the mechanism that makes instructions non-visible to the select logic. That is, when the address-predicted load instruction is inserted in the IQ, its *no-request* bit is set.

To know when the source operand is validated, the valid bit, of the associated entry in VIQ, is set at the same time as the instruction is inserted in IQ. Moreover, the check bit is set to off for marking that the load instruction is being observed by the VIQ for the first time.

While the load instruction is waiting for validating its source operand, the invalidate signals, that may be generated by the matrix structure due to a misspeculated source operand, are filtered in the Decision logic by using the instruction type bit (address-predicted load instruction). When the source operand is valid, the instruction must be made visible in IQ; consequently, the Decision logic activates the re-issue output signal. The *removal circuit* of the IQ uses the re-issue signal for unseting the *no-request* bit of the instruction and, after that, it may be selected for issue. Moreover, in the VIQ, the *removal circuit* unsets the valid bit and the Decision logic sets to on the check bit.

The second time that VIQ observes the address-predicted load instruction, the Decision logic decides, if address-check result is correct, removing the instruction and validating its destination register. In the other case, two cycles are used for performing the recovery action. In the first cycle, the misspeculate signal is activated for re-issuing the directly dependent issued instructions; in the second cycle, the remove signal is activated for removing the

address-predicted load instruction and validating the destination register.

Table 5.4 summarizes the signals activated by the Decision logic considering the observation phase (check bit) and the outputs of the matrix structure.

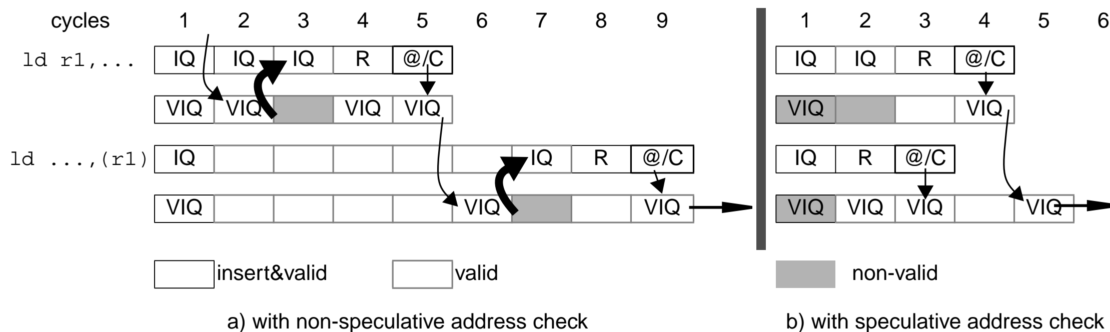
first time (check bit off)	
matrix structure outputs	Decision logic
verify	re-issue
invalidate	none

second time (check bit on)		
matrix structure outputs		Decision logic
verify		remove
invalidate	first cycle	misspeculate
	second cycle	remove

Table 5.4 Decision-logic results using non-speculative address check

5.5.2 Speculative address check

Base design delays the verification of an address-predicted load instruction that depends on other address-predicted load instruction. In Figure 5.19-a is showed an example. Once the address check of the producer load instruction is known to be correct, the destination register state is set to valid. Then, the dependent address-predicted load instruction may be issued and checks its predicted address. After knowing that address-check result is correct, the destination register of the younger load instruction is set to valid. The number of cycles elapsed between both address checks is four cycles.



a) with non-speculative address check b) with speculative address check

Figure 5.19 Validation of dependent predicted load instructions
a) with non-speculative address check, b) with speculative address check

The delay required to verify dependent address-predicted load instructions may be reduced by allowing speculative address checks with speculative operands. In Figure 5.19-b is shown the same example as in Figure 5.19-a, but the younger address-predicted load instruction performs address check with an speculative operand. As in Figure 5.19-a, the address-check result of the younger load instruction is correct. Then, the instruction waits for the validation of its source operand in order to propagate the verification to its issued dependent instructions. Two cycles later, the younger load instruction knows the validity of its source operand, and then the validation of its destination register is propagated. Moreover, if the younger load instruction is address mispredicted, the recovery action can be initiated immediately after receiving the

address-check result. Then, if producer load is correctly predicted, the recovery time is reduced with respect to the base design.

We next describe an extension of the base mechanism where address-predicted load instructions perform address checks with speculative source operands. Moreover, we suppose that the number of concurrent instances of a dynamic load instruction is limited to one instance. That is, the Decision logic will decide to re-issue a load instruction after pairing two informations: a) address-check result, and b) if source operand is valid or misspeculated.

To perform the address check, an instance of a dynamic load instruction uses the predicted address or the address computed in its previous instance; if address-check result is incorrect, a memory access with the computed address is initiated. Moreover, when address-check result is incorrect, the recovery action will be initiated immediately, because it is hoped that the source operand will be validated later. Also, non-address-predicted load instructions perform address check if they are re-issued. First instance of a non-address-predicted load instruction returns correct as address-check result and performs a memory access with computed address.

Address-check result may be received in the VIQ before knowing if the source operand is verified/invalidated or vice versa. Moreover, while waiting address-check result, the source operand may be first invalidated and later verificated. Then, as address check has used a misspeculated operand, the Decision logic must take into account the invalidate signal to perform actions. Thus, as the address-check result and the invalidate signal are only activated during one cycle, they must be stored in memorization elements. These elements are named *check result* bit and *misoperand* bit.

The Decision logic may take several actions to deal with a load instruction. The Decision logic can remove or re-issue the load instruction and, after an incorrect address-check result, can re-issue the chain of issued instructions dependent on the load instruction. Thus, the remove/re-issue signals are activated after pairing both load informations. However, the misspeculate signal is activated when the address-check result is known to be incorrect, independently of the actions performed later, when load information is paired.

To remove a load instruction, by correctness, its source operand must be known to be verified and its address-check result must be received. However, the misspeculated source operand case (re-issue) is an implementation restriction; only one instance of a dynamic load instruction is allowed in execution phase (without have performed the address check).

When address-check result is incorrect, the misspeculate signal is activated. Then, after pairing load information, independently of the actions performed due to the address-check result, the Decision logic considers the operand validity. If source operand is verified, the load instruction is removed; otherwise, if the source operand is invalidated, the load instruction is re-issued, because address check has used a misspeculate data value as source operand. Therefore, in former case the Decision logic activates the remove signal and in the later case

activates the re-issue signal.

Table 5.5 summarizes the outputs produced by the Decision Logic considering its input signals. In case of receiving the verify signal of the source operand earlier or at the same cycle as the check result, and the check-result is incorrect, the Decision Logic performs a two-cycle response: on first cycle generates the misspeculate signal, and on second cycle generates the remove signal.

matrix structure outputs	check result	Decision logic outputs: when address-check result is received	Decision logic outputs: when both load informations are paired
verify	correct	none	remove
	incorrect	misspeculate	
invalidate	correct	none	re-issue
	incorrect	misspeculate	

Table 5.5 Outputs of the Decision logic with speculative address checks

Finally, we discuss the effect of the speculative address checking on updating the address-prediction tables. The last address-check result may be not useful to update the address predictor because the last address check may have used an address different from the predicted address. Therefore, the address computed by the last instance of the load instruction must be compared to the predicted address before updating the address predictor on commit time. For this, the LAQ structure must store both the predicted address and the last computed address.

5.5.2.1 Speculative address check and concurrent execution of several instances of a dynamic load instruction

The previous design delays the re-issue of a load instruction that performs address check when its source operand is known to be misspeculated and its address-check result is not yet known, because this design allows only one instance of a dynamic load instruction in execution phase. The delay depends on the number of cycles (latency) lapsed from the issue cycle (IQ stage) until the VIQ device knows the address-check result, because some misspeculate signals received during these cycles are not attended immediately.

Figure 5.20-a shows an example where a load instruction depends on two mispredicted address-predicted load instructions. Misspeculate signals of the address mispredicted load instruction are delayed between them two cycles. We suppose, but not showed, that consumer load instruction has been executed previously with speculative operand, and in the first re-execution address-check result is correct and in the second re-execution the address-check is incorrect but its source operand has already been verified. Re-executions are showed in consecutive rows, before the row that shows the activity of the VIQ for the instruction. On cycle 7, the invalidation of the source operand of the load instruction is received; however, the decision of re-issuing the load is taken on cycle 8, after receiving the address-check result. Figure 5.20-b shows only the differences with respect Figure 5.20-a when several instances of a dynamic load instruction can be in execution phase concurrently. We can observe that the re-issue of the load

instruction is initiated one cycle early, because the address-check result is not needed to take the decision.

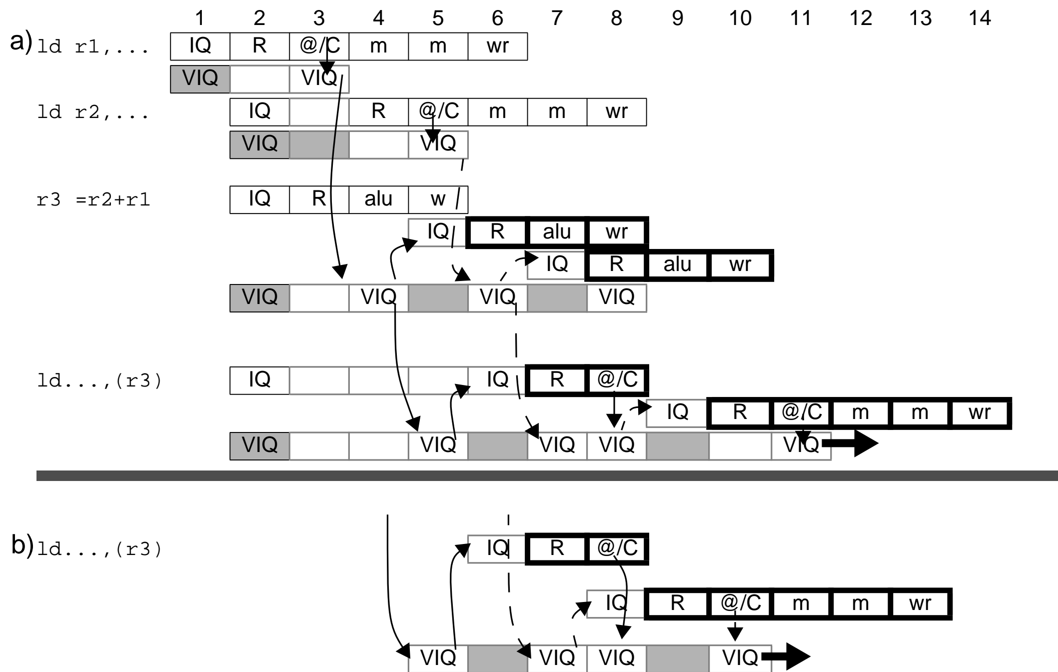


Figure 5.20 Example with one/several concurrent instances of a load instruction: a) one instance, b) several instances

The design of Section 5.5.2 can be extended to support the concurrent execution of several instances of a dynamic load instruction. When the Decision logic knows that the source operand of a load instruction that performs address check is misspeculated, the Decision logic activates the re-issue signal of the load instruction, without waiting for the address-check result of the instance. Later, if address-check result is correct, the Decision logic does not activate any signal. However, if address check-result is incorrect, the Decision logic activates the misspeculate signal in order to re-issue the chain of dependent issued instructions. This it is needed because the new instance, when issued, will perform address check with the computed address of the previous instance.

Address-check results are received in the same order as the re-issue signals of the load instructions are activated. Then, its is sufficient to count the number of pending address-check results that must be received, before pairing the verification signal of source operands with the address-check result of last instance of the dynamic load instruction.

Table 5.6 summarizes the outputs produced by the Decision Logic considering its input signals. We suppose that the number of pending address-checks results related to a load instruction is zero when, after issuing an instance of the load instruction, the next address-check

result received by the VIQ corresponds to the last issued instance. In case of receiving the verify signal of the source operand earlier or at the same cycle as the check result, and the check result is incorrect, the Decision Logic performs a two-cycle response: on first cycle generates the misspeculate signal, and on second cycle generates the remove signal. When the number of pending address-check results is zero, this table is equal to Table 5.5; in the other case, incorrect address-check results produce that the Decision logic activates the misspeculate signal.

			Decision logic outputs		
matrix-structure outputs	address-check result	number of pending address-check results	when address-check result is received	when misspeculate operand is received	when both load informations are paired
verify	correct	= 0	none	n.a.	remove
		> 0			none
	incorrect	= 0	misspeculate		remove
		> 0			none
invalidate	correct	= 0	none	re-issue	n.a
		> 0			
	incorrect	= 0	misspeculate		
		> 0			

Table 5.6 Outputs of the Decision logic with concurrent executions of several instances of a load instruction

5.5.3 Processor Performance

Branch instructions are resolved when their source operands are valid. In this subsection we evaluate the influence of the serial verification on branch-resolution time. To do not bias the evaluations, we use 4-way and 8-way processor configurations with a number of issue-queue entries equal to the number of reorder-buffer entries. For 4-way processors we use 128 entries and for 8-way processors we use 256 entries.

Next example depicts the execution of a sequence of instructions, and shows the branch instruction is resolved earlier using the serial verification than using the verification on commit. The presented results assume speculative-address checks and concurrent execution of several instances of a dynamic load instruction.

Example

Figure 5.21 shows the cycle-by-cycle execution of some predicted load instructions, and how the serial verification through the verification-flow graph propagates the verification to their dependent instructions. The contents of the speculative state of the operands of the instructions are also depicted. For instance, the speculative state (1, 0) of instruction *b* on cycle 3 means that the state of the first operand (r1) is speculative (1), and the state of the second operand (r9) is non speculative (0); we depict these contents only after issuing/re-issuing an instruction and after updating their value. We have assumed that both speculative accesses (instructions *a* and *c*) have finished before inserting the dependent instructions *a* into the issue queue; also, the address prediction of instruction *a* is incorrect (KO), and the address prediction of the instruction *b* is correct (OK). Finally, the initial state of all registers is non speculative.

The instructions dependent on the predictions are executed speculatively. First, instructions *b* and *f*, next instructions *d* and *g*, and finally instruction *e*. Note that the state of some operands are set to speculative (r2, r4 and r6).

After detecting the correct prediction of load instruction *c* on cycle 7, all its directly dependent instructions are notified that register r3 has become non speculative. As both operands of instruction *f* are known to be non speculative, the verification is propagated to its dependent instructions (instruction *g*) on next cycle.

After detecting the incorrect prediction of load instruction *a* on cycle 10, memory is accessed again. As dependent instructions are re-issued, the non speculative state is propagated.

We can observe that the branch instruction (instruction *g*) is resolved when its operand has been verified (cycle 9). Note that the branch resolution has been advanced several cycles with respect to the commit stage (cycle 18).

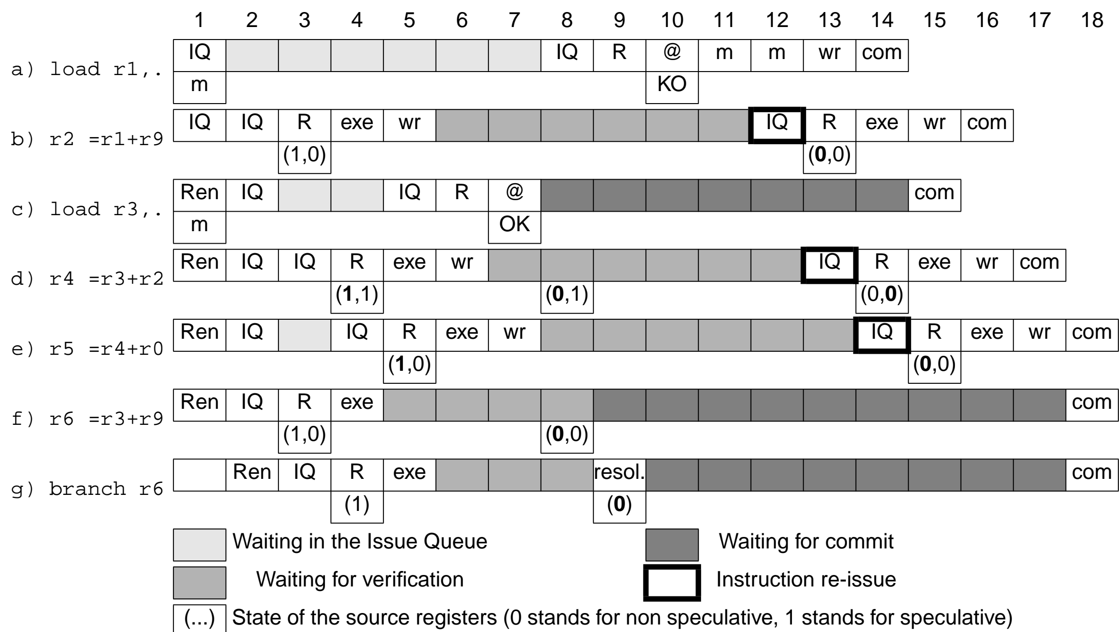


Figure 5.21 Example of the serial verification
(using this verification mechanism, the branch resolution of instruction *g* has been advanced 9 cycles with respect to the implicit verification on commit)

Results

Figure 5.22 shows the performance of the baseline processors and the address-speculative processors with both the implicit verification on commit and the serial verification through verification-flow graph.

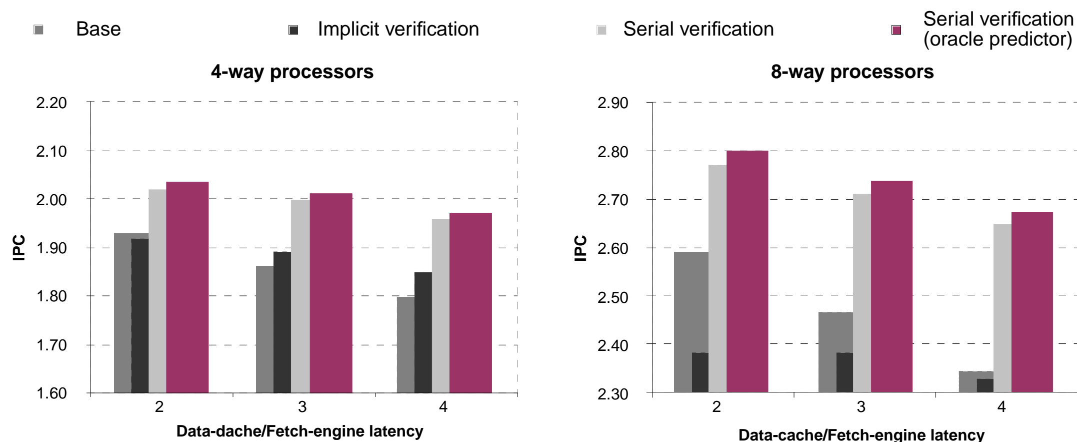


Figure 5.22 IPC versus data-cache latency in baseline and in address-speculative processors with implicit and serial verification mechanism

We can observe that the address-speculative processors with the serial verification represent a performance improvement with respect to the baseline processors. Using the real predictor, this improvement ranges from 5% to 9% (4-way processors) and from 7% to 13% (8-way processors).

Comparing both address-speculative processors (serial verification versus implicit verification) using the real predictor, the verification mechanism has a great impact on processor performance: the use of the serial verification improves the performance around 5% (4-way processors) and from 13% to 17% (8-way processors). The instructions that take advantage of the faster verification process are the mispredicted branch instructions, because they must be resolved after verifying their operands. Consequently, using the serial verification, some mispredicted branch instructions are verified before they reach the head-entry of the re-order buffer (for instance, all the branch instructions independent on address predictions).

The degradation on the performance of the serial mechanism due to address mispredictions can be observed by comparing the results for the real and the oracle predictor. This degradation is around 1% in 4-way processors and around 2% in 8-way processors. Note that in the evaluated processors, address mispredictions have two negative impacts: issuing mispredicted memory accesses and consuming issue slots. However, these issue slots would be unused unless the issued instructions delayed the issuing of younger instructions with correct operands.

Our results show that the verification mechanism is critical for dealing with mispredicted branch instructions.

5.6 Enhanced verification-flow graph

The serial verification propagates verifications serially by traversing level-by-level the verification-flow graph at a uniform one cycle latency. The potentiality of address prediction and

speculative execution is related to propagating the verifications through the speculatively issued instructions faster than executing them.

In integer benchmarks, most instructions have single-cycle latency. The serial verification of a large dependence chain of speculatively issued instructions then exposes a latency similar to that of executing it. Assuming an unbounded number of functional units, both latencies are equal; consequently, the verification of the instructions would be advanced with respect to their non speculative execution as many cycles as the memory latency[5].

Figure 5.23 shows an example of this situation. Instruction *a* is a correctly predicted load instruction and instructions *b*, *c* and *d* are a chain of instructions dependent on this prediction. Analogously, instruction *e* is an incorrectly predicted load instruction and instructions *f*, *g* and *h* are a chain of instructions dependent on this prediction. After the address-prediction checking of instruction *a*, instructions *b*, *c* and *d* get verified on cycles 11, 12 and 13 respectively. After the address-prediction checking of instruction *e*, instructions *f*, *g* and *h* must be re-executed and their non speculative results are obtained on cycles 14, 15 and 16 respectively. Comparing both dependent chains, the verification is advanced with respect to the non-speculative execution as many cycles as the memory latency.

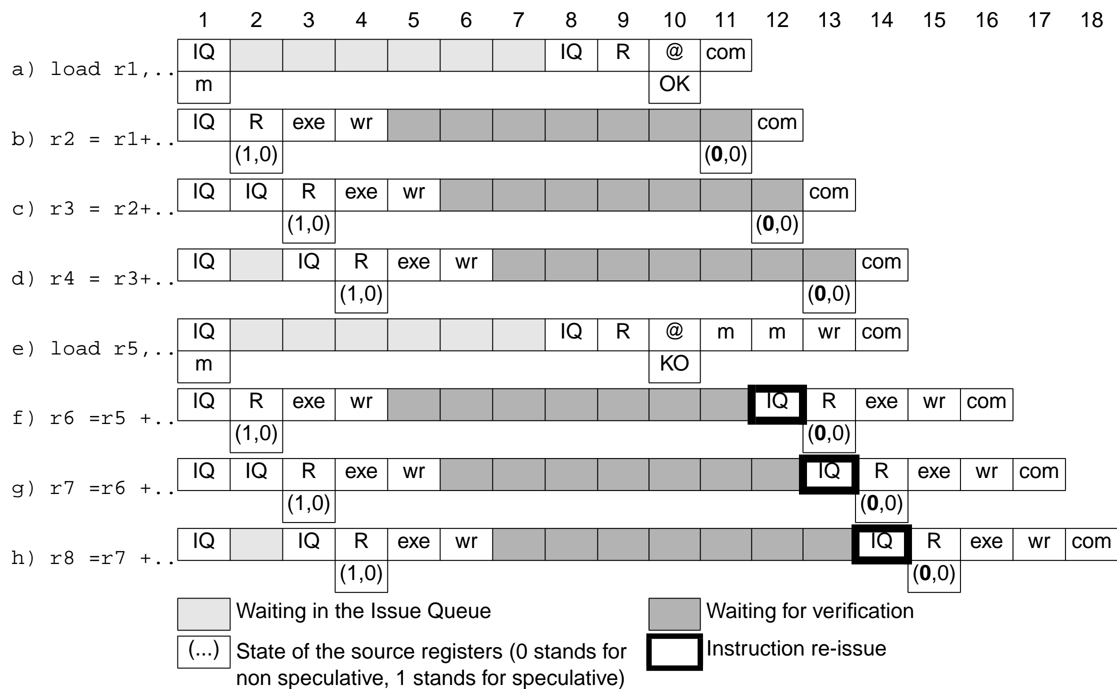


Figure 5.23 Example of the serial verification of large chains of dependent instructions

5. We suppose that address-prediction checkings are notified concurrently to effective-address computation.

Therefore, on correct address predictions, verifying the chain of speculatively issued instructions in only one cycle is useful for resolving early some branch instructions.

In this section we present a design more aggressive than the serial verification flow-graph. We will build a new verification-flow graph with less levels than the serial verification-flow graph. The goal of this design is to reduce the number of cycles needed to verify/invalidate instructions that depend on an address-predicted load instruction.

The Enhanced serial verification-flow graph (EVG) is built as follows. From the data-flow graph, starting at nodes that represent an instruction that performs address check, each output arc of a node is traversed, and the output arcs of the reached nodes are traversed recursively, until reaching nodes that represent an instruction that performs address check. Then, each traversed node (including the nodes that finish the recursions) is connected directly to the starting node. Consequently, only one step will be required to verify/invalidate source operands of instructions directly/indirectly dependent on an instruction that performs address check.

The verify/invalidate mechanism that uses the EVG works like the mechanism described for the serial verification-flow graph. Once the Decision logic of the VIQ takes actions driven by a load instruction that performs address check, the EVG is traversed at an uniform single-cycle rate by level. The difference between both mechanisms is the dependence information stored in the matrix structure of the mechanism; using this mechanism, each row of the matrix identifies the input arcs of a node in the EVG. We next describe how this information is built.

Collapsed Graph Table

To build the rows of the matrix structure we use a table named Collapsed Graph Table (CGT) with as many entries as architectural registers, and each entry has a bit vector with as many bits as physical registers. The CGT is indexed by architectural-register identifier, is updated only with information known in the rename stage, and is read also in this stage to build row information.

A load instruction that performs address check sets the bit that identifies its destination register in the CGT entry of its destination architectural register. Any other instruction reads the CGH entries related to its source architectural registers, applies the OR function to the bit vectors, and stores the resultant bit vector in the CGT entry of it destination architectural register.

Figure 5.24 shows an example of the computation of these bit vectors. We present a chain of instructions, and the sequence of CGT contents after processing each one of these instructions. CGT's contents are presented partially; that is, only the rows and columns used in the example are shown. Moreover, we assume that register renaming has not modified the registers identifiers of the instructions, all load instructions perform address check, and the value of the non-shown registers is established by instruction that store an immediate value. Instruction *a* sets the bit that identifies r1 register; the instruction *b* sets the bit that marks its dependence on register r1; instruction *c* behaves like instruction *a*; instruction *d* sets bits that marks its dependence of registers r1 and r5; instruction *e* behaves like instructions *a* and *c*; finally, instruction *f* sets the

bits that marks its dependence on registers r1 and r6.

CGT contents after processing instructions

a) LD r1, . . . (a)

architectural \ physical	1	2	3	5	6
1	1				
2					
3					
4					
5					
6					

b) r3 = r1 + r9. (a, b)

architectural \ physical	1	2	3	5	6
1	1				
2					
3	1				
4					
5					
6					

c) LD r5, . . . (a, b, c)

architectural \ physical	1	2	3	5	6
1	1				
2					
3	1				
4					
5				1	
6					

d) r2 = r3 + r5 (a, b, c, d)

architectural \ physical	1	2	3	5	6
1	1				
2	1			1	
3	1				
4					
5				1	
6					

e) LD r6, 5(r5) (a, b, c, d, e)

architectural \ physical	1	2	3	5	6
1	1				
2	1			1	
3	1				
4					
5				1	
6					1

f) r4 = r6 + r3 (a, b, c, d, e, f)

architectural \ physical	1	2	3	5	6
1	1				
2	1			1	
3	1				
4	1				1
5				1	
6					1

Figure 5.24 Computation of the Collapsed Graph Table
For each table, each row identifies an architectural register, and each columns identifies a physical register. We assume that Register Rename stage does not modify register identifiers.

Matrix structure

Each instruction reads the CGT entries associated to its source architectural registers, applies the OR-function to the read bit vectors, and stores the result in its assigned entry of the matrix structure of the VIQ device.

Figure 5.25 shows an example of the computation of the matrix structure. We present the same sequence of instructions as in Figure 5.24, and the sequence of matrix contents after inserting in the VIQ each one of these instructions.

matrix-structure contents after processing instructions

a

	1	2	3	5	6
a) LD r1, . . .					
b) r3 = r1 + r9.					
c) LD r5, . . .					
d) r2 = r3 + r5					
e) LD r6, 5(r5)					
f) r4 = r6 + r3					

a, b

	1	2	3	5	6
a) LD r1, . . .					
b) r3 = r1 + r9.	1				
c) LD r5, . . .					
d) r2 = r3 + r5					
e) LD r6, 5(r5)					
f) r4 = r6 + r3					

a, b, c

	1	2	3	5	6
a) LD r1, . . .					
b) r3 = r1 + r9.	1				
c) LD r5, . . .					
d) r2 = r3 + r5					
e) LD r6, 5(r5)					
f) r4 = r6 + r3					

a, b, c, d

	1	2	3	5	6
a) LD r1, . . .					
b) r3 = r1 + r9.	1				
c) LD r5, . . .					
d) r2 = r3 + r5	1			1	
e) LD r6, 5(r5)					
f) r4 = r6 + r3					

a, b, c, d, e

	1	2	3	5	6
a) LD r1, . . .					
b) r3 = r1 + r9.	1				
c) LD r5, . . .					
d) r2 = r3 + r5	1			1	
e) LD r6, 5(r5)				1	
f) r4 = r6 + r3					

a, b, c, d, e, f

	1	2	3	5	6
a) LD r1, . . .					
b) r3 = r1 + r9.	1				
c) LD r5, . . .					
d) r2 = r3 + r5	1			1	
e) LD r6, 5(r5)				1	
f) r4 = r6 + r3	1				1

Figure 5.25 Computation of the Matrix structure of the VIQ
For each table, each row identifies an VIQ entry, and each column identifies a physical register

Thus, after verifying instruction *a*, instruction *b* is verified. After verifying instruction *c*, instructions *d* and *e* are verified.

All the mechanisms described for the serial verification (non-speculative address check, speculative address check) can be also implemented using the enhanced verification-flow graph.

5.6.1 Processor Performance

In this subsection we evaluate the influence of the enhanced verification on branch-resolution time. The processor parameters that have been used are the same than the parameters used for the evaluation of the serial verification.

Next example depicts the execution of a sequence of instructions, and shows the branch instruction is resolved earlier using the enhanced verification that using the serial verification.

Example

Figure 5.26 shows the cycle-by-cycle execution of some predicted load instructions and how the enhanced verification propagates the verification to their dependent instructions. We present the execution of the same instructions as those in Figure 5.21.

When each instruction is inserted in the IQ, the VIQ tracks the destination registers of the non-verified load instructions that the instruction depends on.

After detecting the correct prediction of load instruction *c* on cycle 7, all its dependent (directly and indirectly) instructions are notified that register r3 has become non speculative. We can observe that the branch instruction (instruction *g*) depends directly or indirectly on only this predicted load instruction; then, the branch instruction can be resolved on cycle 8. In this example, the enhanced verification-flow graph allows advancing branch resolution one cycle with respect to the verification-flow graph.

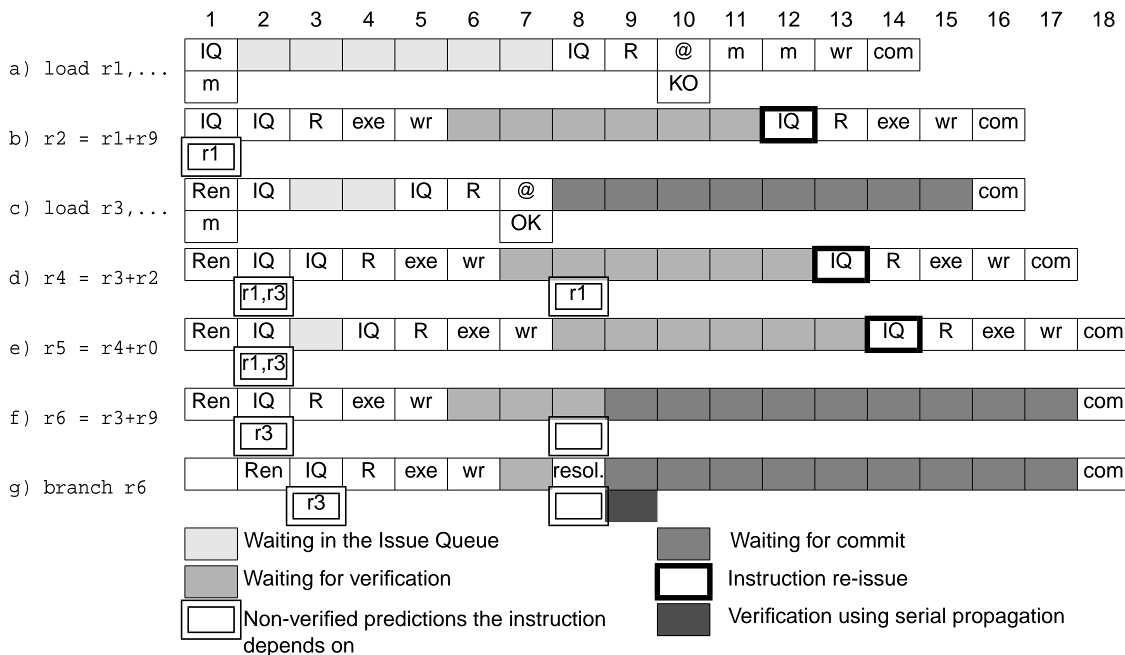


Figure 5.26 Example of the enhanced verification

Results

Figure 5.27 shows the performance of the baseline processors and the address-speculative processors with serial and enhanced verification.

We can observe that the address-speculative processors with enhanced verification represent a performance improvement with regard to the baseline processors. Using the real predictor, this improvement ranges from 6% to 11% (4-way processors) and from 8% to 15% (8-way processors); the degradation due to address mispredictions is about 1% (4-way processors) and 2% (8-way processors).

Comparing both address-speculative processors (serial verification versus enhanced verification), we can appreciate a uniform improvement due to the faster verification process; this improvement is around 1% (in 4-way processors) and 2% (in 8-way processors). The instructions that take advantage of the faster verification process are the mispredicted branch instructions that depend on correctly predicted load instructions.

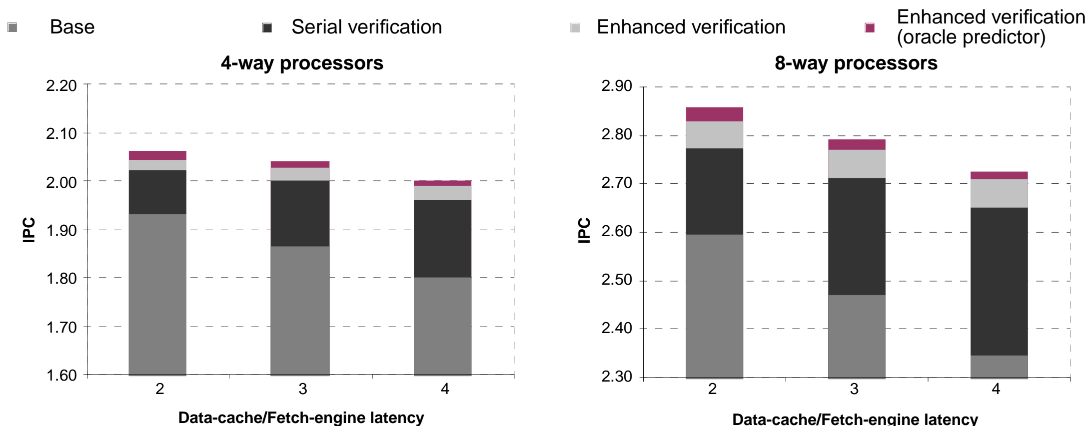


Figure 5.27 IPC versus data-cache latency on baseline processors and address-speculative processors with serial and enhanced verification

5.7 Processors with the issue queue decoupled from the reorder buffer

Existing superscalar processors (such as Alpha 21264 [Kess99], Sparc V [Dief99], Pentium 4 [Carm00]) decouple the issue queue from the reorder buffer, because the issue queue is less scalable than the reorder buffer. Every instruction should then be maintained in each structure for a different time interval: the reorder buffer holds an instruction while it is in-flight and the issue queue holds an instruction while it is waiting for the availability of operands or functional units. In this scenario, an instruction is removed from the issue queue after being issued, and it is extracted from the reorder buffer on commit stage. Consequently, existing superscalar processors have a number of issue-queue entries smaller than the number of reorder-buffer entries.

Re-issue recovery mechanisms based on keeping the speculatively issued instructions in the issue queue require a mechanism for removing the verified instructions from the issue queue. Previous designed mechanism can be used for this purpose.

For baseline processors, the issue-queue size determines the ability of the scheduler for issuing out-of-order the fetched instructions as soon as their source operands are ready. In address-speculative processors, the effective issue-queue size is reduced because the issue queue keeps the speculatively issued instructions until they are verified. Then, the issue-queue becomes a critical resource because it limits the number of instructions that the scheduler analyses to be executed.

In following subsections we first evaluate the influence of the issue-queue size on the performance of baseline and address-speculative processors with the issue queue decoupled from the reorder buffer; for address-speculative processors we use the serial verification and the enhanced verification that implement the mechanism described in Section 5.5.2.1. In the evaluations we focus on the integer issue queue because we simulate integer benchmarks.

5.7.1 Influence of issue-queue size on the performance of baseline processors

First, we evaluate the influence of the issue-queue size on the performance of baseline processors. In these processors, the instructions are removed from the issue queue as soon as they are issued because no re-issue mechanism is needed.

Figure 5.28 shows our results; every graph connects results for the same first-level data-cache latency. For 4-way processors, we have evaluated issue queues with 15, 20 and 25 entries because the integer issue-queue of current processors does not exceed 20 entries (20 entries in Alpha 21264, 18 entries in AMD Athlon and 20 entries in Intel P6). As a reference, we also present results for large issue queues with 64 entries and 128 entries. For 8-way processors, we scale the previous issue-queue sizes by two. We maintain unaltered the reorder-buffer size with respect to the previous evaluations: 128 entries for the 4-way processors and 256 entries for the 8-way processors.

In 4-way processors, we can observe that using a 25-entry issue queue, the performance is close to saturation; the performance is below saturation from 0.1% (2-cycle latency) to 2.3% (4-cycle latency). In 8-way processors, using a 50-entry issue queue, the performance is below saturation from 1.6% to 3.2%. Consequently, the performance of the baseline processors is almost saturated using issue queues with a number of entries significantly smaller than the number of reorder-buffer entries.

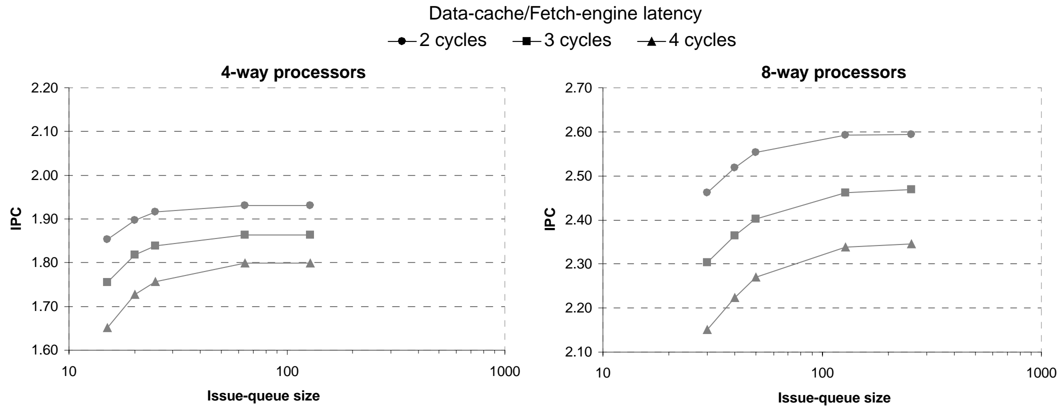


Figure 5.28 IPC versus issue-queue size on baseline processors

5.7.2 Serial verification

In Section 5.5, we described a serial verification mechanism that detects if all the operands of an instruction are known to be non speculative. In this subsection, we apply this mechanism to removing the instructions from the issue queue in addition to performing the verification process. Each instruction will be removed from the issue queue one cycle after the VIQ notifies that the instruction has been verified.

Figure 5.29 shows the impact of the issue-queue size on the performance of an address-speculative processor with non-delayed speculative issue and serial verification mechanism.

Our results show that address-speculative processors outperform baseline processors. For instance, in 4-way processors with a 25-entry issue-queue, the performance improvement ranges from 4% (2-cycle latency) to 9% (4-cycle latency); in 8-way processors, these improvements range from 7% to 14%.

We can observe that the issue-queue size needed to saturate the performance is larger in address-speculative processors than in baseline processors. This is due to the fact that address-speculative processors use some issue-queue entries to maintain the speculatively issued instructions until they become non speculative; then, these entries are not used by the scheduler to look-ahead for independent non-issued instructions. For instance, in 4-way processors, using the 25-entry issue queue, the performance is below saturation from 1.6% (2-cycle latency) to 1.9% (4-cycle latency). In 8-way processors, using the 50-entry issue queue, the performance is below saturation from 2% to 2.3%. We present results for 4-way processors with 32 issue-queue entries, and for 8-way processors with 64-entry issue-queue entries to show this effect.

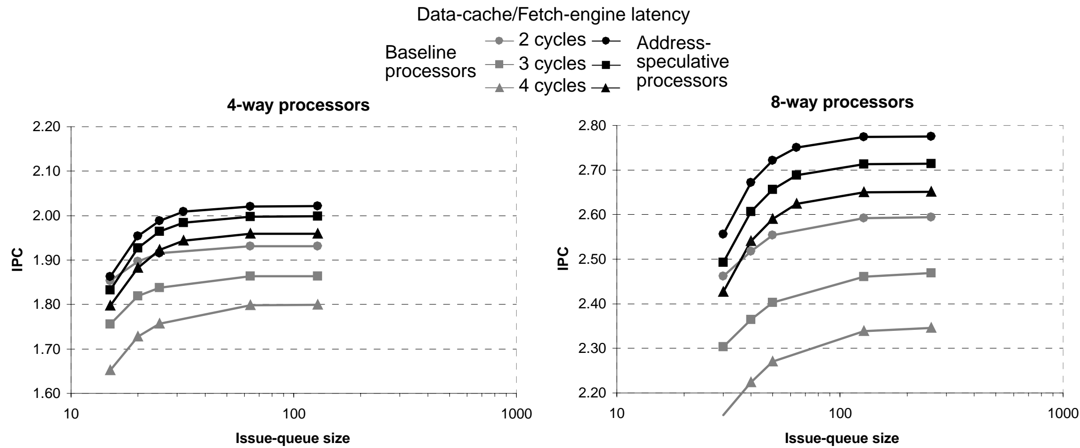


Figure 5.29 IPC versus issue-queue size on baseline processors and address-speculative processors with serial verification

Figure 5.30 shows the impact of the data-cache latency on the performance of the same address-speculative processors. We can observe that address-speculative processors are less sensitive to data-cache latency than baseline processors. For instance, in 4-way processors with 25-entry issue queues, increasing cache latency from two to four cycles produces an 8% performance degradation in baseline processors and 3% in address-speculative processors. Therefore, as the cache-access latency increases, the improvement due to address prediction also increases.

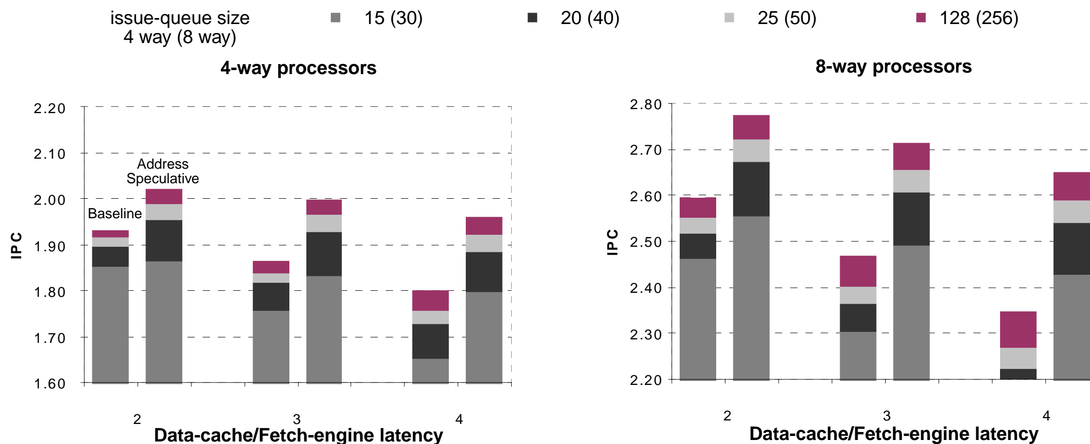


Figure 5.30 IPC versus cache latency on baseline processors and address-speculative processors with serial verification

5.7.3 Enhanced verification

In Section 5.6, we described a enhanced verification mechanism that detects if all the operands of an instruction depend on correctly predicted load instructions. In this subsection, we apply the enhanced verification mechanism to removing the instructions from the issue queue. Each

instruction will be removed from the issue queue one cycle after the VIQ notifies that the instruction has been verified. The use of the enhanced mechanism reduces the pressure on the issue queue with respect to the use of the serial mechanism, since the enhanced mechanism frees some issue-queue entries earlier than the serial mechanism. Freeing issue-queue entries as soon as possible is useful because it may allow advancing the insertion of younger instructions into the issue queue

Figure 5.31 shows the impact of the issue-queue size on the performance of address-speculative processors with non-delayed speculative issue and both the serial and the enhanced verification mechanisms.

Our results show the performance effect of the enhanced mechanism compared with the serial mechanism; it is similar to reducing both the cache latency and the fetch-engine latency by one cycle. For instance, graphs related to the 3-cycle latency, serial verification, 8-way processors and to the 4-cycle latency, enhanced verification, 8-way processors are almost overlapped.

We can observe that address-speculative processors with the enhanced mechanism are closer to saturation than address-speculative processors with the serial mechanism. For instance, in 4-way processors, using the 25-entry issue queue and the enhanced verification mechanism, the performance is below saturation from 1.5% to 1.7% (versus 1.6% to 1.9% using the serial mechanism); in 8-way processors, performance is below saturation from 1.6% to 2% (versus 2% to 2.3% using the serial mechanism).

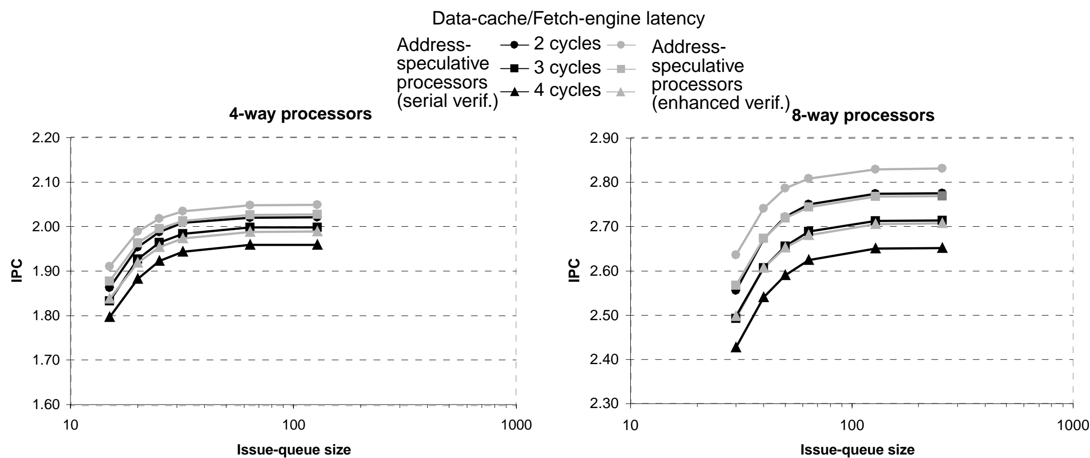


Figure 5.31 IPC versus issue-queue size on address-speculative processors with serial and enhanced verification

5.7.4 Influence of increasing address-predictor latency on the performance of address-speculative processors

To issue a speculative memory access, the address predictor must produce a prediction and the predicted address must be inserted into the Load-Address Queue. Our previous evaluations considered that the predicted effective addresses are inserted into the Load-Address Queue during the first cycle after the fetch engine provides the fetched instruction; on next cycle, the speculative memory access may be issued. These previous evaluations represent an upper bound for the address-prediction look-ahead. However, the latency of the address predictor plus the transit time from the address predictor to the Load-Address Queue may avoid issuing the speculative memory access so early.

In this subsection we evaluate the effect of the early issue of the speculative memory accesses on the performance of address-speculative processors. We define the *prediction-latency increment* as the number of additional cycles (in relation to our previous evaluations) before starting the speculative memory accesses. We evaluate prediction-latency increments from one to three cycles (all the previous evaluations assumed a prediction-latency increment of zero cycles).

Figure 5.32 presents results for baseline and non-delayed address-speculative processors that use the enhanced verification mechanism and the oracle address predictor. The horizontal axis stands for the cache latency and the prediction-latency increment, and the vertical axis stands for the IPC. We present results for 64-entry issue queue, 128-entry reorder buffer 4-way processors and for 128-entry issue queue, 256-entry reorder buffer 8-way processors.

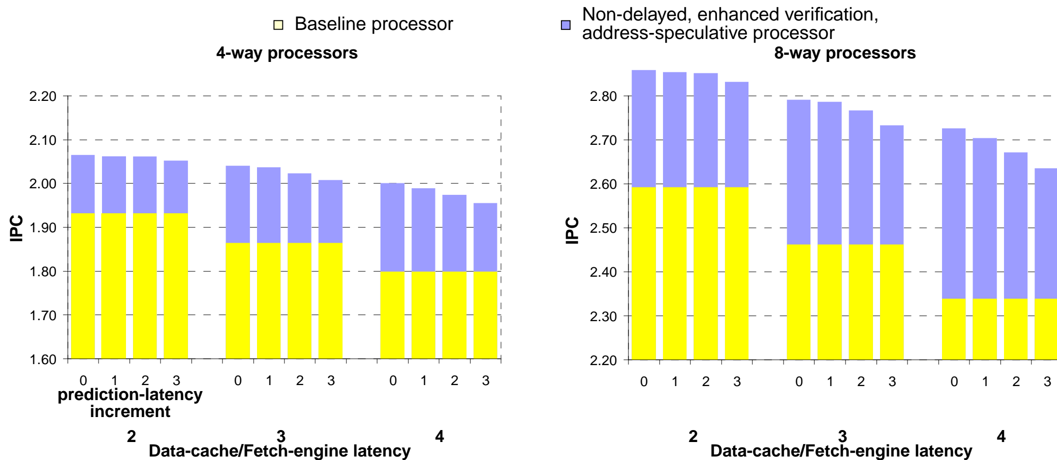


Figure 5.32 IPC versus address-predictor latency and data-cache latency on address-speculative processors with non-delayed speculative issue

We can observe that the performance degradation due to increasing the prediction-latency increment in non-delayed address-speculative processors ranges from 0.6% to 2.2% (4-way processors), and from 1% to 3.3% (8-way processors).

Using the non-delayed policy, an instruction dependent on a predicted load instruction may be issued speculatively as soon as the speculative memory access has finished. Then, delaying the issuing of a speculative memory access is delaying the issuing of its dependent instructions. Consequently, it is mitigating the performance impact of address prediction.

5.8 Delayed speculative issue

Some existing superscalar processors use a speculative technique named latency prediction. It is employed by the scheduler of some high-performance processors to deal with instructions whose latency is unknown when they are issued; for instance, load instructions ([Kess99]). Schedulers that predict the latency of the load instructions are able to schedule optimistically the instructions dependent on the load instructions. Thus, on a correct latency prediction, these schedulers achieve a back-to-back execution of the load instruction and its dependent instructions.

There are two implicit characteristics of latency prediction: a) the instructions dependent on the latency-predicted instruction are issued after issuing the latency-predicted instruction and b) the number of cycles from issuing a latency-predicted instruction until knowing the correctness of the prediction is fixed. The designer of a recovery mechanism for latency mispredictions may make the most of both conditions in order to design a recovery mechanism specific for latency mispredictions. In Chapter 6 we propose a recovery mechanism (named *Recovery Buffer*) specific for latency mispredictions that allows extracting the speculatively issued instructions as soon as they are issued.

The non-delayed speculative issue is not a particular case of latency prediction because the instructions dependent on a predicted load instruction may be issue before issuing the predicted load instruction. Then, its recovery mechanism for address mispredictions must be different from a recovery mechanism specific for latency mispredictions.

We propose the use of a speculative-issue policy for address predictions that can be considered a particular case of latency prediction; this allow us to use any recovery mechanism specific for latency mispredictions. We must guarantee that the instructions dependent on a predicted load instruction will be issued after issuing the effective-address computation of the load instruction; in terms of latency prediction, the latency of the load instruction is predicted to be one cycle. In this work, we refer to this policy as delayed speculative issue and in our evaluations we will assume the use of the recovery mechanism specific for latency mispredictions proposed in Chapter 6 (the *Recovery Buffer*).

Figure 5.33 presents an example of both non delayed (a) and delayed (b) speculative issue. Using delayed speculative issue, the instruction dependent on the predicted load is issued after issuing the effective-address computation of the load. Consequently, on a correct prediction, the predicted load instruction appears to be a one-cycle-latency instruction.

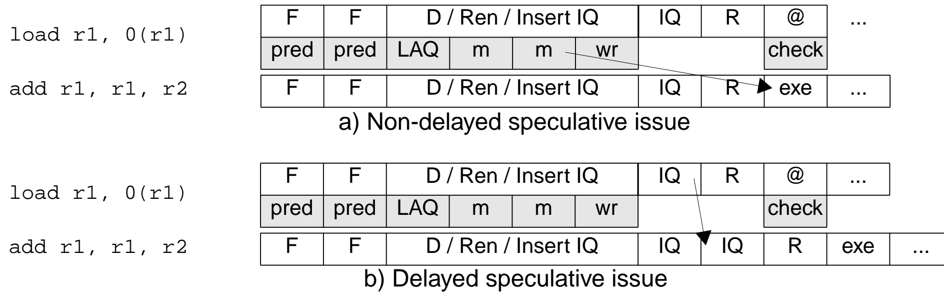


Figure 5.33 Execution of a predicted load instruction and the speculative issue of a dependent instruction: a) non delaying the issuing of the dependent instruction and b) delaying it

The use of the delayed speculative issue may represent a performance degradation in relation to the non-delayed speculative issue, because the instructions dependent on the predicted load instruction are speculatively issued after issuing the effective-address computation of the predicted load instruction. This degradation may be significant if the base register of the predicted load instruction depends on a large dependence chain or on a data-cache miss.

In Chapter 6, we evaluate several recovery mechanisms for latency mispredictions. These recovery mechanisms re-issue mispredicted instructions, but they differ on the structure that records the speculatively issued instructions: the issue queue or a new structure named *Recovery Buffer*. On one hand, using the issue queue, a speculatively issued instruction remains in the issue queue until verifying it. On the other hand, using the recovery buffer, a speculatively issued instruction is removed from the issue queue after issuing it; these instructions are inserted into the recovery buffer and they are removed either when they are verified or when they are re-issued. Both recovery mechanisms present a drawback respect the recovery mechanism used in previous sections: on a latency misprediction, the latency-misprediction recovery mechanisms suffer an one-cycle penalty. This penalty is needed to perform an enhanced invalidation of the instructions dependent on the misprediction. As in the previous evaluations, branch instructions are resolved non speculatively; also, on a latency misprediction, only the instructions dependent on the misprediction are re-issued.

In this and following sections we evaluate address-speculative processors with delayed speculative issue and a recovery mechanism based in the recovery buffer. This recovery mechanism allows us to reduce the pressure on the issue queue because instructions are removed from the issue queue as soon as they are issued.

5.8.1 Delayed speculative issue versus baseline processors

We compare the performance of address-speculative processors with delayed speculative issue versus baseline processors.

Figure 5.34 shows the impact of the issue-queue size on the performance of baseline processors and address-speculative processors with delayed speculative issue; every graph

connects results for the same data-cache latency.

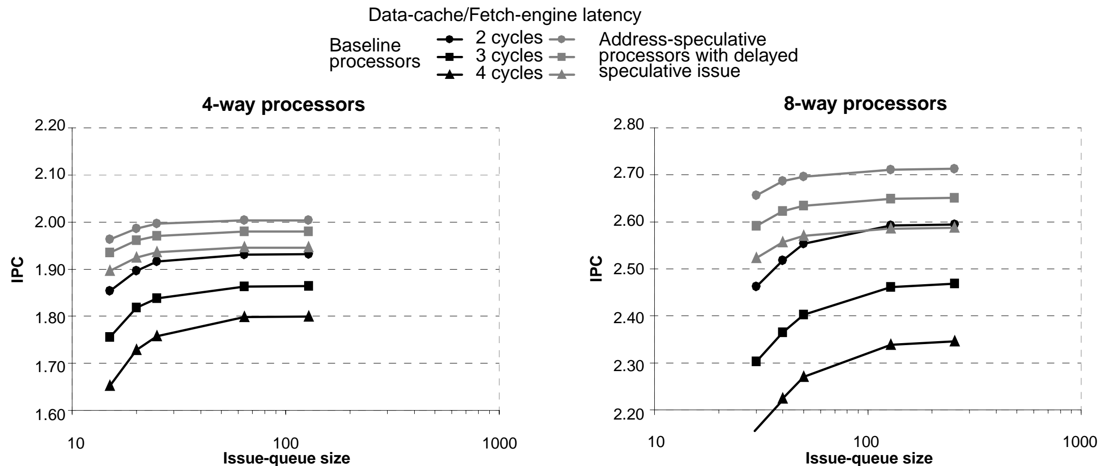


Figure 5.34 IPC versus issue-queue size in baseline processors and address-speculative processors with delayed speculative issue

We can observe that the evaluated address-speculative processors outperform the baseline processors. For instance, in 4-way processors with a 25-entry issue queue, the improvement ranges from 4% (2-cycle latency) to 10% (4-cycle latency); in 8-way processors with a 50-entry issue queue, the improvement ranges from 6% (2-cycle latency) to 13% (4-cycle latency).

Furthermore, the performance of the evaluated address-speculative processors are less sensitive to small issue-queue sizes. For instance, in 4-way processors with 20-entry issue queues, baseline processors are below saturation from 2% (2-cycle latency) to 4% (4-cycle latency). However, address-speculative processors are below saturation around 1%. Consequently, the smaller the issue-queue size, the larger the performance impact of address prediction with delayed speculative issue.

Finally, the evaluated address-speculative processors tolerate cache latency better than baseline processors. For instance, in 4-way processors with 25-entry issue queues, the performance degradation observed when cache latency increases from two to four cycles is 8% (baseline processors) and 3% (address-speculative processors); in 8-way processors with 50-entry issue queues, the degradation is 11% and 5% respectively.

5.9 Non-delayed versus delayed speculative issue

In this section we evaluate the effect of some processor parameters (issue-queue size, cache latency and issue width) on the performance of both non-delayed and delayed address-speculative processors.

Figure 5.35 shows the performance of baseline and both delayed and non-delayed

address-speculative processors. The horizontal axis stands for the first-level-cache latency and each bar is related to a processor model (baseline, serial-verification non-delayed issue, enhanced-verification non-delayed issue and delayed issue), vertical axis stands for the IPC. Each bar also stacks results for several issue-queue sizes.

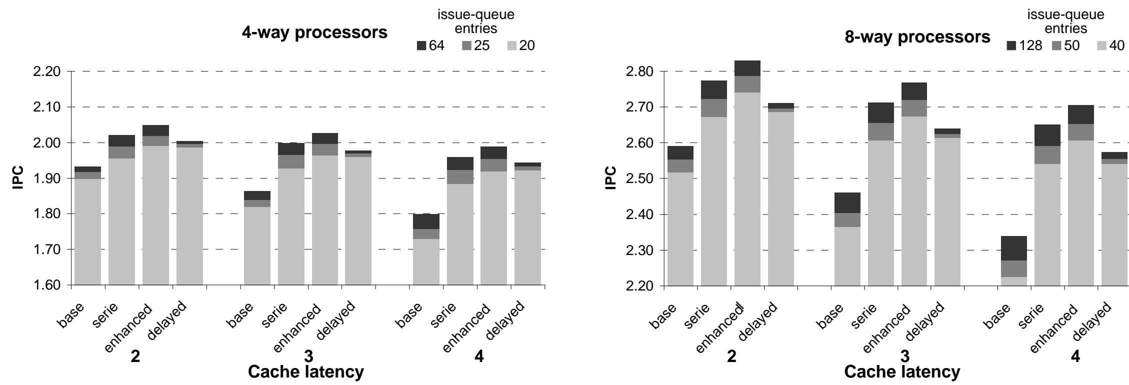


Figure 5.35 Performance of baseline and address-speculative processors

For a cache latency and issue-width, all evaluated address-speculative processors outperform the baseline processors, independently of the issue-queue size. Then, address speculation may produce a performance impact bigger than that of enlarging the issue queue of a baseline processor.

The issue-queue is one of the most critical structures of superscalar processors [PJS97]. Consequently, we are interested in finding out the sensitivity of the evaluated processors to the issue-queue size. Our results show that while address-speculative processors with non-delayed speculative issue are able to exploit large issue-queue sizes, the performance of delayed-issue address-speculative processors almost saturate using relatively small issue queues. For instance, in 4-way processors with 20-entry issue queues, while the performance of delayed-issue address speculative processors is 1% below saturation, the performance of non-delayed-issue address-speculative processors is around 3% below saturation. For 8-way processors with 40-entry issue queues, the results are similar.

This behaviour can be explained because an address-speculative processor with delayed speculative issue which recovery mechanism uses the recovery buffer extract the instructions from the issue queue as soon as they are issued. However, an address-speculative processor with non-delayed speculative issue retains the speculatively issued instructions into the issue queue until they become non speculative.

Comparing all address-speculative processors, our conclusions depend on the issue-queue size. In 4-way processors, for small issue queues (20 entries), the delayed-issue processors achieve performances around 2% larger than that of non-delayed processors with serial verification, and a similar performance to that of non-delayed processors with enhanced

verification. For medium issue queues (25 entries) delayed processors outperform non-delayed processors with serial verification around 0.5%, and present around an 1% performance degradation in relation to the non-delayed processors with enhanced verification. Finally, for large issue queues (64 entries), performance of delayed processors degrades around 0.5% and 2% with respect to non-delayed processors with serial and enhanced verification respectively.

In 8-way processors, for small issue queues (40 entries), delayed processors achieve similar performances to that of non-delayed processors with serial verification, and around 2% degradation in relation to the non-delayed processors with enhanced verification. For medium issue queues (50 entries), performance degradation is around 1% (serial) and 3% (enhanced). For large issue queues (128 entries), degradation reaches 2% and 4% respectively.

The performance difference between both speculative issue policies is due to two factors. First, using the delayed policy, the instructions dependent on a predicted load instruction are issued after issuing the effective-address computation of the predicted load instruction; however, this restriction is not present using the non-delayed policy. Second, the recovery mechanism used by address-speculative processors with non-delayed speculative issue suffers an one-cycle penalty on each address misprediction.

From these evaluations we conclude that the delayed policy combined with the recovery mechanism based on the recovery buffer may be an attractive alternative to the non-delayed policy. However, several aspects may made the delayed alternative still more attractive. For instance, we have considered oracle latency prediction for the load instructions; without this oracle prediction, the non-delayed alternative would suffer an additional pressure over the issue queue because more instructions speculatively issued will be kept into the issue queue[6].

5.9.1 Effect of address mispredictions on performance

In this subsection we present an evaluation of the impact of address mispredictions on the performance of both address-speculative processors. Figure 5.36 depicts the performance of address-speculative processors with both real and oracle address predictors; we present results for 64-entry issue queue, 128-entry reorder buffer 4-way processors and for 128-entry issue queue, 256-entry reorder buffer 8-way processors.

6. Instructions dependent on a load instruction will not be removed from the issue queue until knowing if the load instruction is a hit or a miss in the first-level data cache.

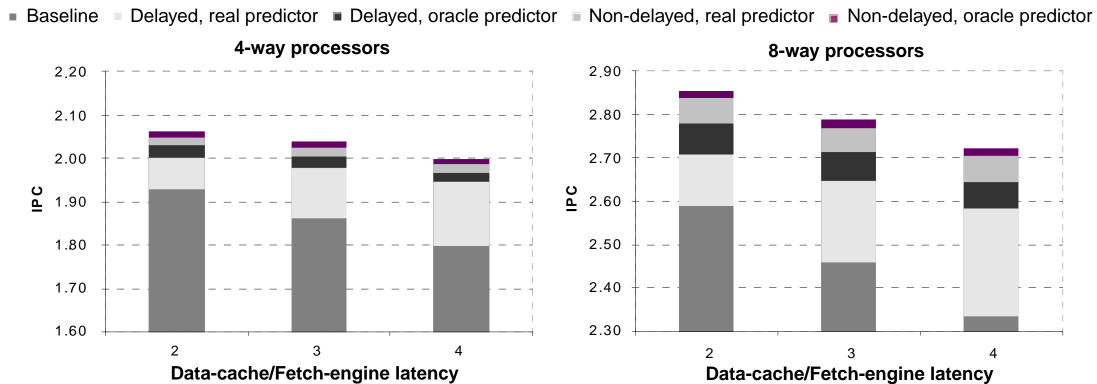


Figure 5.36 IPC versus data-cache latency on address-speculative processors with real and oracle address predictors

Our results show that delayed address-speculative processors are more sensitive to address mispredictions than non-delayed address-speculative processors. For instance, in 4-way processors, the performance degradation due to address mispredictions in delayed processors is around 1.2%; however, in non-delayed processors the degradation is around 0.6%. In 8-way processors, these percentages are 2.4% for the delayed processors and 0.7% for the non-delayed processors.

Delayed address-speculative processors are more sensitive to address mispredictions than non-delayed processors because the recovery mechanism used by the delayed address-speculative processors suffers a one-cycle penalty after each misprediction to perform the enhanced invalidation of the instructions dependent on the misprediction. Non-delayed address-speculative processors does not perform enhanced invalidation; the issued instructions dependent on the misprediction impact on processor performance only if they pollute caches or they avoid issuing misprediction-independent younger instructions.

We can conclude that address mispredictions reduce the potential performance improvement of address prediction. This reduction is specially significant if the recovery mechanism suffers a fixed penalty for each misprediction. In this scope, better confidence mechanisms should be proposed in order to reach a compromise between the captured predictability and the accuracy of a predictor.

5.10 Related works

Several works have presented evaluations of the impact of address prediction and speculative execution on processor performance.

Reinman and Calder [ReCa98] presented an evaluation of speculation techniques that can be applied to load instructions: dependence prediction, address prediction, value prediction and memory renaming. They evaluated these techniques over a future generation micro-architecture

that fetches up to 8 instructions per cycle, issues up to 16 instructions per cycle, couples the issue queue and the reorder buffer into a 512-entry RUU, and has a first-level cache latency of four cycles. They considered two recovery mechanisms: a squash recovery that flushes-out all the instructions after a mispredicted load instruction and re-fetches them from instruction cache, and a conventional selective re-issue mechanism. In their evaluations, they varied the address-predictor model (Last-Address Predictor, Stride Address Predictor, Context Address Predictor and Hybrid Address Predictor), the confidence estimator (conservative, conventional and oracle) and the recovery mechanism (squash recovery coupled with the conservative estimator, and the re-issue recovery coupled with the conventional estimator); however, the authors do not present the implementation of the recovery mechanisms.

There is a subtle difference between their oracle predictor and our oracle predictor: the low-confident address predictions that are correct. Both oracle predictors filter the address mispredictions; however, our oracle predictor also filters the low-confident address predictions (although they were correct predictions). Then, their oracle predictor can capture more predictability than their real predictor, consequently, the difference of performance impact between the real and the oracle predictors are due to both the effect of mispredictions and the effect of the captured-predictability increment. Moreover, the wide issue width of the processor can mitigate the effect of re-issuing the instructions dependent on a misprediction.

Their results show that when the squash recovery mechanism is used (and conservative confidence estimation), the impact of address prediction is small (limited by a 1.04 speed-up). However, using the re-issue recovery mechanism (and conventional confidence estimation), the impact is closer (up to 1.10 speed-up) to that of the oracle predictor (1.13).

Bekerman et al. [BJR+99] evaluated the performance impact of address prediction and speculative execution; they used a new prediction model: the Global-Correlated Context-Address Predictor (Section 2.5). They focused on a prediction-table configuration and analysed the impact of pipelining address prediction (prediction tables are not updated immediately). Their results showed that pipelining address prediction affects the performance impact of address prediction, but this impact is still significant (in SPEC95-INT benchmarks, speed-up decreases from 1.22 to 1.17).

Ahuja et al. [AEK+01] presented an evaluation of the potential impact of address prediction. They proposed a new prediction model, the Dependence-based Address Predictor (Section 2.5), and evaluated its impact assuming no wrong predictions. They showed that their predictor is suitable for pointer-chasing programs.

Some works have proposed recovery mechanisms that can be applied to address-speculative processors with the non-delayed speculative issue.

Akkary and Driscoll [AkDr98] proposed a two-level issue queue. After decoding the instructions, they are sent to the rename stage (and after that, they are inserted into the issue

queue) and they are inserted into a trace buffer. Instructions are removed from the issue queue as soon as they have been issued, but they remain in the trace buffer until they have been committed. On a misprediction, the trace buffer detects which instructions depend on the misprediction, groups them, and sends these instruction blocks to the rename stage.

Sato [Sato99] also proposed a two-level structure: the scheduling window and the instruction buffer. The scheduling window is equivalent to the issue queue; it holds non-issued instructions. The instruction buffer is equivalent to the RUU; it holds all the fetched instructions that have not been committed. On a misprediction, instructions are re-issued from the instruction buffer. There is a key difference between the recovery buffer and this scheme. While the recovery buffer is a simpler structure because it records a valid scheduling of the instructions, Sato's proposal must replicate the scheduling logic in the scheduling window and the instruction buffer. Moreover, due to the large size of the instruction buffer, its select logic and its wake-up logic are pipelined.

5.11 Conclusions

The evaluations presented in this chapter show that address-speculative processors have a potential performance improvement in relation to existing superscalar processors. Address-speculative processors are more tolerant to cache latency than baseline processors; then, address speculation can mitigate the performance degradation produced by increasing cache latency. Moreover, address speculation may produce a performance impact bigger than that of enlarging the issue queue of a baseline processor.

Many parameters influence the effective improvement due to address prediction. To begin with, the performance of address-speculative processors is very sensitive to branch-instruction resolutions. We have decided that branch instructions are resolved non speculatively; that is, all the operands of the branch instruction must be known to be non speculative before resolving the branch instruction. Then, a verification mechanism must be responsible for tracking the speculative state of the operands of each instruction. We have evaluated several verification mechanisms (implicit, serial and enhanced) and our results show that the performance of address-speculative processors is very sensitive to this verification mechanism because a slow verification mechanism increases the branch penalty of the mispredicted branch instructions.

As the issue-queue size has a large influence on processor performance, we have also evaluated the sensitivity of address-speculative processors to the issue-queue size. We have considered two speculative issue policies for the address-speculative processors: non-delayed and delayed speculative issue. Although the non-delayed policy has a larger performance potential than the delayed one, the delayed policy may be a cost-effective alternative to the non-delayed one. While the first policy needs a recovery mechanism with a conventional recovery mechanism that puts additional pressure on the issue queue, the second policy may use a recovery mechanism (the Recovery Buffer) that puts no additional pressure on the issue queue.

Our results show that, depending on the number of issue-queue entries of the processor, address-speculative processors with delayed speculative issue (and the recovery mechanism based on the Recovery Buffer) may be a cost-effective alternative to address-speculative processors with non-delayed speculative issue.

Moreover, we expect that the delayed speculative issue may result more attractive if some additional aspects are taken into account. For instance, in this chapter we have considered oracle latency prediction for the load instructions. To deal with latency mispredictions, a conventional re-issue recovery mechanism (similar to the mechanism used by the non-delayed speculative issue) must keep the instructions dependent on a latency prediction into the issue queue until verifying the prediction; consequently, the pressure over the issue queue is increased. However, the Recovery Buffer can also be applied to latency mispredictions; consequently, these instructions will put no additional pressure on the issue queue. Then, we expect that the degradation due to supporting load-latency prediction will be smaller in address-speculative processors with delayed speculative issue than in address-speculative processors with the non-delayed speculative issue.

From our results, we suggest that the scenario where address prediction is expected to have a larger impact is on processors with a large cache latency (3 or 4 cycles), a wide issue mechanism (8 way). Furthermore, depending on the number of issue-queue entries of the processor, the delayed speculative issue is an attractive alternative to the non-delayed speculative issue.

In the evaluations performed in this chapter some directions of the design space have been unexplored. For instance, in our evaluations address-prediction tables are updated immediately and non speculatively on decode stage. However, an implementation of the address predictor must take into account the latency needed to update the prediction tables and the speculative nature of the effective addresses available on decode stage. Both factors may affect the captured predictability and the accuracy of the address predictor. The performance of the address-speculative processors may then decrease. Moreover, we have shown the degradation produced by address mispredictions. Further work should be performed in order to develop better confidence mechanisms that will reduce this degradation.

5.12 Detailed results

5.12.1 4-way processors

Figure 5.37 shows the results of the evaluated 4-way processors on each benchmark.

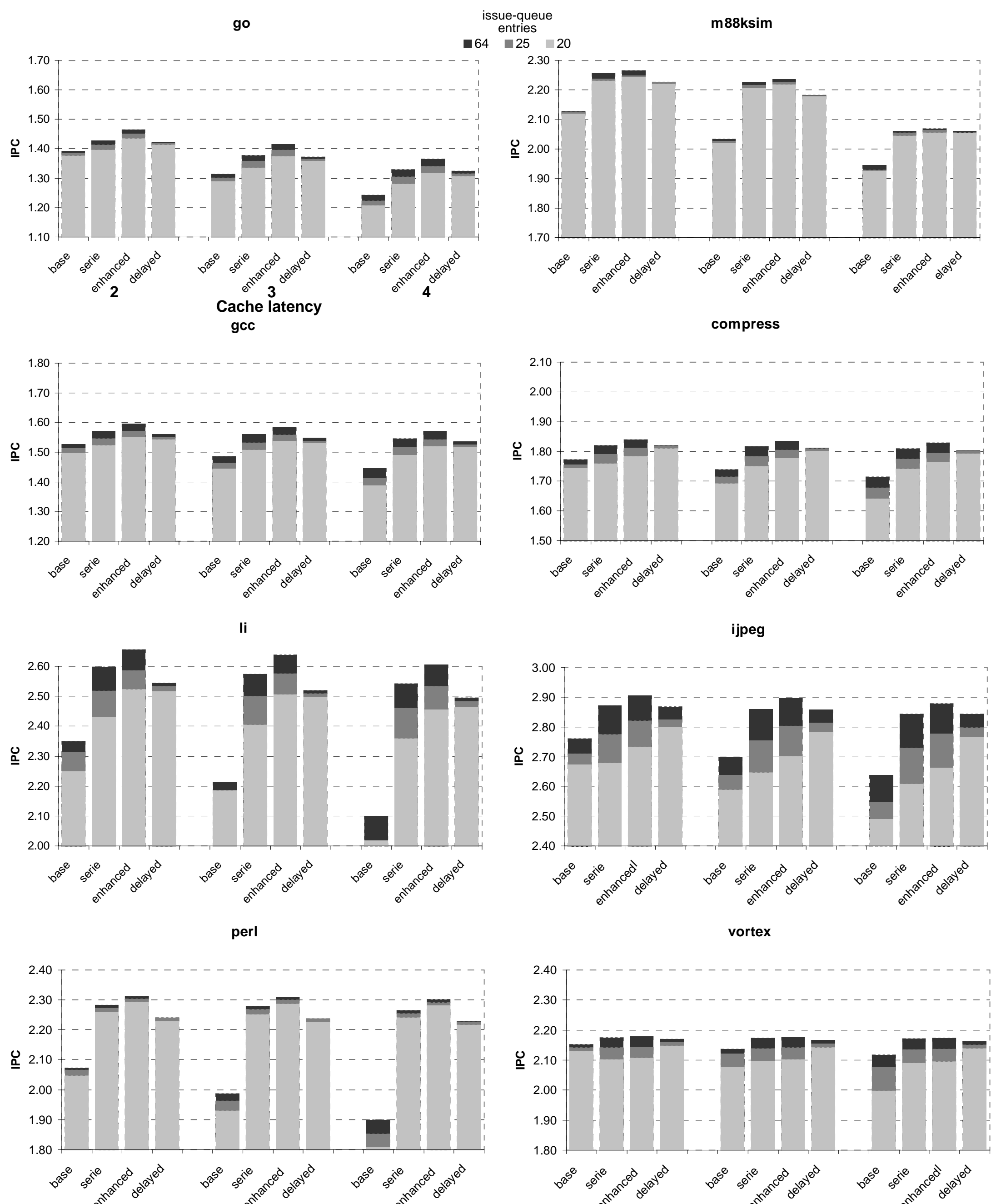


Figure 5.37 Performance of 4-way baseline and address-speculative processors on each SPEC95-INT benchmark

5.12.2 8-way processors

Figure 5.37 shows the results of the evaluated 8-way processors on each benchmark.

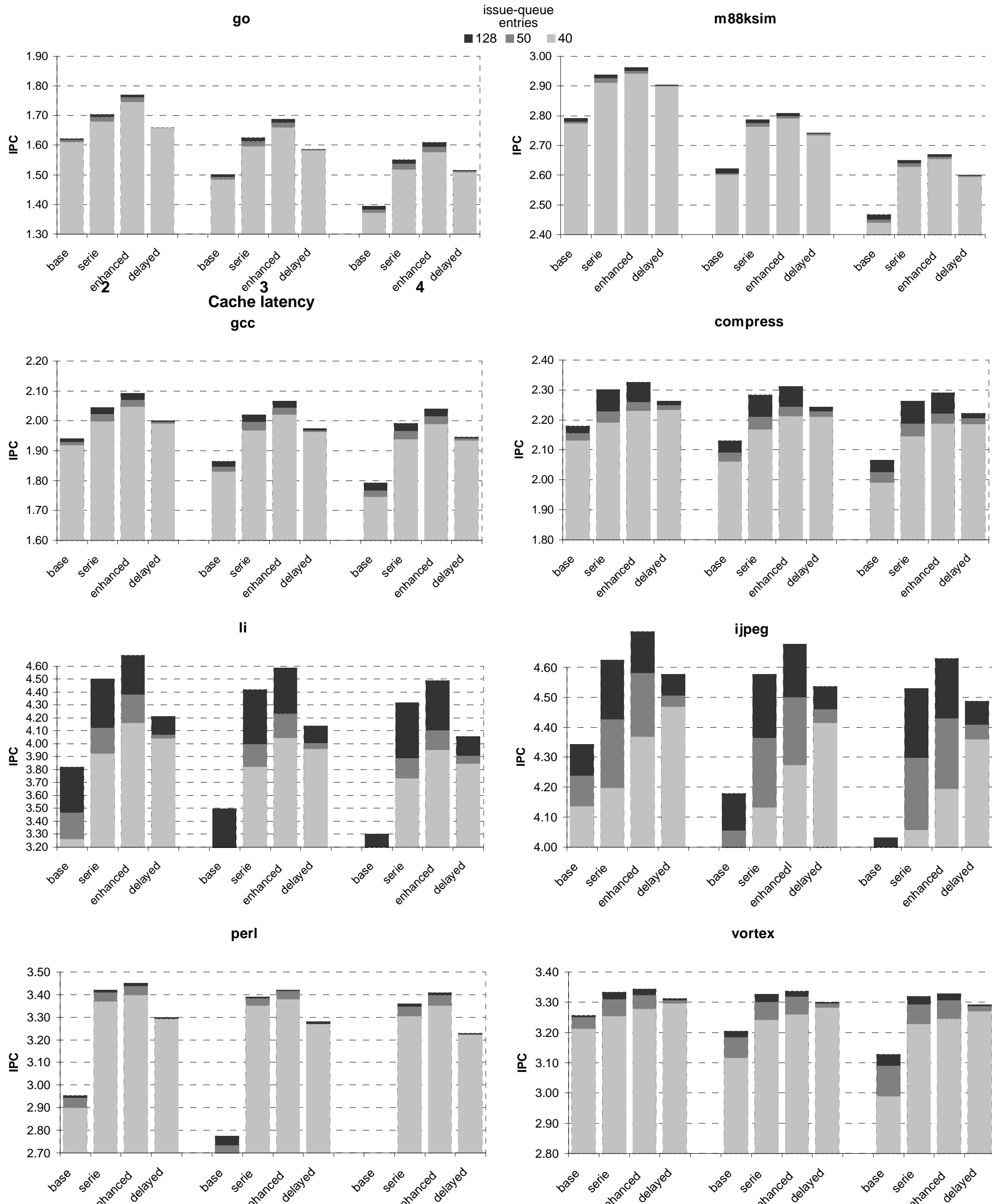


Figure 5.38 Performance of 8-way baseline and address-speculative processors on each SPEC95-INT benchmark

6 RECOVERY MECHANISMS FOR LATENCY MISPREDICTIONS

Latency prediction is a speculative technique used by some superscalar processors to deal with instructions whose latency is variable (for instance, load instructions). Latency prediction allows the scheduler to planify optimistically the instructions dependent on variable-latency instructions. Although prediction techniques have great performance potential, their gain can vanish due to

misprediction handling; for instance, holding speculatively scheduled instructions in the issue queue reduces the queue's ability to look-ahead for independent instructions. This chapter evaluates a recovery mechanism for latency mispredictions that retains the speculatively issued instructions in a structure apart from the issue queue: the recovery buffer. This chapter is organized as follows. Section 6.1 introduces the chapter. Section 6.2 characterizes the recovery process for latency mispredictions. Section 6.3 outlines the recovery process when speculative instructions are retained in the issue queue. In Section 6.4 the recovery process using the recovery buffer is designed. Section 6.5 gives performance results of the recovery-buffer mechanism compared with the conventional recovery mechanism. Finally, Section 6.6 presents the conclusions of this work.

6.1 Introduction

In dynamically-scheduled superscalar processors, instructions wait in the issue queue for the availability of operands and functional units [Toma67][Hunt95][Yeag96][PJS97]. To issue instructions out-of-order to the functional units, the issue queue has two components: a) *wakeup logic* and b) *select logic*. The *wakeup logic* keeps monitoring the dependencies among the instructions in the issue queue and, when the operands of a queued instruction become available, this logic will mark the instruction as ready. The *select logic* selects which instructions will be issued to the functional units on the next cycle.

If only instructions with known latency are considered, a mechanism that counts latencies and wakes-up dependent instructions can be included in the issue logic. However, to deal with instructions with unknown latency, the functional units must send a signal to the *wakeup logic*; then, with high clock rates, wire delays may prevent back-to-back execution of dependent instructions [CaGo00][Matz98]. Therefore, a valuable mechanism that deals with unknown-latency instructions is latency prediction. If the predicted latency is optimistic, instructions dependent on the predicted instruction can be scheduled speculatively. However, a recovery mechanism is needed on mispredictions to nullify and to re-issue the speculatively issued instructions.

A simple alternative for the recovery mechanism is squashing; all the instructions younger than the mispredicted instruction are flushed-out from the processor, and these instructions are later re-fetched from the instruction cache. This process is identical to the branch-misprediction recovery mechanism.

However, to reduce the penalty of the recovery process, finer recovery mechanisms are needed. For instance, they can benefit from the fact that the instructions that must be re-issued have already been fetched. In this case, the mechanism must provide storage to keep the speculatively issued instructions until the prediction is verified.

A conventional solution is to maintain the chain of speculatively issued instructions (and probably other independent instructions) in the issue queue [Kess99][Rote99] until latency

prediction is verified. However, unless the issue-queue size is increased, processor performance can suffer because this solution reduces the ability of the scheduler to look-ahead for independent instructions. On the other hand, increasing the issue-queue size will be limited by future wire delays [AHKB00]; therefore, as the issue queue is in the critical path, this solution is a limited alternative.

Another approach consists in extracting the issued instructions from the issue queue after being issued, and storing them in a recovery buffer, apart from the issue queue, until latency prediction is verified. New instructions can then be inserted in the freed issue-queue entries and the look-ahead ability of the issue queue is maintained. On a misprediction, the re-issue is performed from the recovery buffer and, to reduce the complexity of the re-issue logic of the recovery buffer versus that of the issue queue, the recovery buffer maintains the relative issue cycles between the instructions. Moreover, on mispredictions, the recovery buffer increases the amount of in-flight instructions, because it holds issued instructions dependent on latency-mispredicted instructions.

A scope where this work can be applied is load-use delay. Load instructions have unknown latency because latency depends on the location of the data in the memory hierarchy. Moreover, tag-checking is in the critical path to wake up the dependent instructions. In first-level caches, data-array contents can also be obtained before tag-checking result [MIPS]. Consequently, waiting until tag-checking to wake up the dependent instructions can reduce the performance. For instance, in a 4-way processor executing integer benchmarks, performance degradation is about 6% when load-use delay increases from 3 to 4 cycles. Other scope where this work can be applied is address prediction, as it has been shown in Chapter 5.

This chapter applies latency prediction in the instruction scheduler, and evaluates the performance of the recovery-buffer mechanism versus keeping the speculatively issued instructions in the issue queue. Moreover, two issued-instruction nullification policies are evaluated: a) nullifying all the instructions potentially dependent on the mispredicted instruction, b) nullifying only the chain of instructions dependent on the mispredicted instruction.

The evaluations are focused on load-latency prediction [Carm00][Dief99][HoLa99][Kess99] and, as high first-level-cache hit rates are expected, the prediction is that all load instructions are hits in data cache. As a side effect, cache tag-checking can be moved out of the critical path. Evaluations show that the recovery-buffer mechanism outperforms the conventional recovery mechanism. For integer benchmarks, the recovery-buffer mechanism allows a issue-queue-size reduction of about 20-25% without performance decrease.

6.2 Background

Figure 6.1 shows two cases where latency prediction is profitable (stages between instruction-fetch stage and rename stage are not shown as they are not relevant to this work). In Figure 6.1.a, as tag-checking is performed before waking-up the dependent instruction, it

increases load-use delay if data-array access is a cache hit. Using latency prediction, tag-checking can be decoupled from data availability, and thus load-use delay is reduced by two cycles. Another pipeline design (Figure 6.1.b), includes a stage between the issue queue and the functional units. In this case, to support back-to-back execution of dependent instructions, *wakeup logic* must wake-up the dependent instructions before tag-checking.

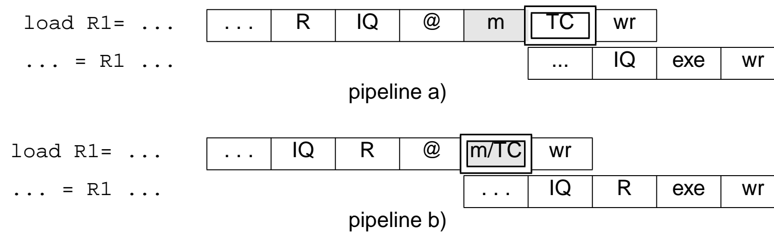


Figure 6.1 Pipeline designs without latency prediction
(stages: read registers (R), issue queue (IQ), compute address (@), execute (exe), data-array access (m), tag-checking (TC), write registers (wr))
a) Registers are read before IQ stage; tag-checking is performed one cycle after data availability; the issue queue stores the values of the registers or a functional-unit identifier.
b) Registers are read after IQ stage; simultaneous tag-checking and data availability.

In both cases, predicting hit latency is useful to execute the chain of dependent instructions without delay if memory access is a cache hit. Without latency prediction, load-use delay is four cycles; with latency prediction, load-use delay is two cycles. However, a recovery mechanism is required on a latency misprediction because the dependent instructions use incorrect data in their computations. From now on, the pipeline design b) on Figure 6.1 is used in the examples on this chapter.

For instance, latency prediction is also useful in: a) way prediction in associative caches, b) bank prediction in multi-banked caches, c) caching physical registers when they are read after the issue stage ([BTME02]), d) first-level cache with ECC correction logic, e) pipelining the scheduling logic ([BSP01]).

Figure 6.2 is used to introduce the terminology of this chapter. Assume that on cycle 1 a load instruction is issued with a data-cache latency of two cycles and a tag-check latency of three cycles. When hit latency is predicted, the speculative instructions potentially dependent on the load instruction, directly or through a dependent chain, are issued on cycles 3, 4 and 5 (shadowed instructions). We name these cycles *speculative window* (SW); that is, the cycle range from waking-up the first potential dependent instruction until tag-checking. We name *verification delay* to the duration of the speculative window (three cycles in the example). We also name *independent window* (IW) to the cycle range between issuing the load and the beginning of the speculative-window; the instructions issued during the independent window are independent of the load. An instruction is inside a window if it is issued during a cycle of the window. A *wave of instructions* represents all the instructions issued during a cycle.

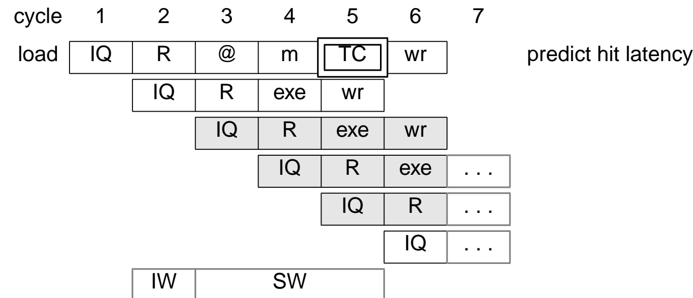


Figure 6.2 Instruction flow after issuing a load instruction (LD) with predicted hit latency (IW is the Independent Window, SW is the Speculative Window)

6.2.1 Recovery on a mispredicted latency

Known scopes where the processor executes instructions speculatively are branch prediction and memory-dependence prediction.

Branch prediction is used to speculatively execute predicted-path instructions. These instructions may be issued before issuing the predicted branch instruction. The prediction is performed by the fetch unit and the verification is performed by the branch instruction when it is executed. On a misprediction, wrong-path instructions are squashed and the fetch unit is redirected to the new path.

Memory-dependence prediction is used to execute a load instruction and its dependent instructions before knowing the addresses accessed by older store instructions. The speculative instructions are issued after issuing the predicted load instruction. The prediction is performed by a load instruction and the verification by an older store instruction. On a misprediction, the instructions that must be re-executed have already been fetched.

Usually, these predictions rely on a general recovery mechanism that flushes-out the entire instruction pipeline [Kess99].

Unlike the previous prediction types, latency prediction shows **all** the following characteristics:

- The verification of the prediction is performed by the predicted instruction.
- The speculative instructions are issued after issuing the predicted instruction.
- When a latency-predicted instruction is issued, the cycles where the dependent instructions can be speculatively issued are known (the speculative window).
- On a misprediction, the instructions that must be re-executed are the same as those that are nullified.

These characteristics allow the design of a simple recovery mechanism that is slightly aggressive from a performance point of view[1]. When a misprediction is detected, all instructions issued inside the speculative window are nullified and their dependent instructions are slept. After that, the nullified instructions are re-issued in proper time: the instructions independent of the latency-predicted instruction are re-issued on next cycles, and the dependent instructions will be re-issued when data is available. In the following sections we analyse two structures for keeping the instructions while predicted latency is verified: the issue queue and the recovery buffer.

The previous approach loses on every misprediction a number of cycles equal to the speculative-window size. Figure 6.3 shows an example where 3 cycles are lost on a misprediction. A better mechanism is also evaluated in this chapter; the mechanism only nullifies the instructions dependent on the mispredicted instructions.

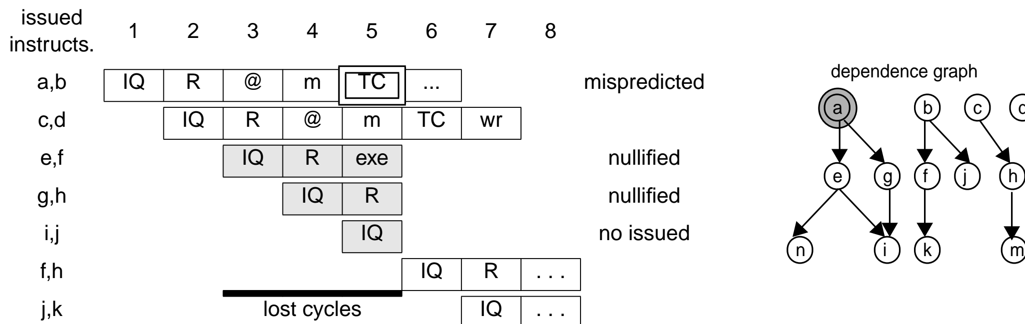


Figure 6.3 Example of an instruction flow with a latency-mispredicted instruction
Load instruction **a** is a latency-mispredicted instruction. On cycle 5, the misprediction is detected, instructions **i** and **j** are not issued and do not wakeup their dependent instructions; also instructions **e**, **f**, **g**, **h** are nullified and their dependent instructions (**i**, **k**, **n** and **m**) are slept in the issue queue. On cycle 6, nullified instructions **f** and **h** are re-issued and their dependent instructions are woken-up. On cycle 7, instruction **j** is re-issued and instruction **k** is issued. Instructions dependent on load instruction **a** are re-issued (not shown) once the memory hierarchy provides data.

Our evaluations assume that no instruction is issued on cycles where latency mispredictions are detected; that is, on the last cycle of the speculative window of a mispredicted instruction, the instructions selected to be issued are not issued (cycle 5 in Figure 6.3).

In summary, this chapter presents evaluations of two approaches:

---

1. Alpha 21264 processor handles this situation with a mini-restart mechanism. All integer instructions issued during the speculative window are “pulled back” into the issue queue to be re-issued later [Kess99].

- A conservative approach, named *non-selective*, that assumes that all issued instructions are inside a potential speculative window. On mispredictions, it nullifies all the instructions inside the speculative window.
- An aggressive approach, named *selective*, that considers only the instructions dependent on latency-predicted instructions. On mispredictions, it nullifies only the instructions of the speculative window dependent on mispredicted instructions.

These approaches represent two extreme cases, although several intermediate approaches could be designed.

6.2.2 Base Pipeline and Issue Queue

Base Pipeline (Figure 6.4). In this chapter we use the baseline processor described in Section 5.2.1. After fetching the instructions, they are decoded and renamed. A renamed instruction resides in the issue queue until its source operands have been computed and it has been selected for execution. After it has been executed, it is marked in the Reorder Buffer (ROB) as completed. After that, it is committed when all previous instructions in program order have been marked as completed and have been committed. When an instruction is committed, the previous physical register mapped to the destination logical register of the instruction is freed. The ROB records all in-flight instructions.

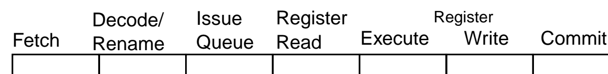


Figure 6.4 Base processor pipeline

Base Issue Queue. The structure of the issue queue has been described in Section 5.4.2.

6.3 Keeping issued instructions in the issue queue

In regular operation, without latency prediction, instructions are removed from the issue queue as soon as they are issued. However, with latency prediction, some instructions must be re-issued when a misprediction is detected.

To perform a fast recovery, a possibility is to keep each issued instruction in the issue queue until the instruction is known to be unnecessary for a recovery action [Kess99]. During these cycles, the issued instruction should be nonvisible to the *select logic*. A *no-request bit* is then added to each dependence-matrix row; the bit is set when its instruction is issued.

When an issued instruction is known not to be needed in a latency-mispredicted recovery action, it can be removed from the issue queue. Otherwise, on a misprediction, it must be nullified. It is made visible again to the *select logic*, and its destination register is set as not available to delay the issue of its dependent instructions until it has been re-issued.

Two control circuits perform these operations: the *removal circuit* and the *register-scoreboard circuit*. Figure 6.5 shows the interface between them and other issue-queue elements.

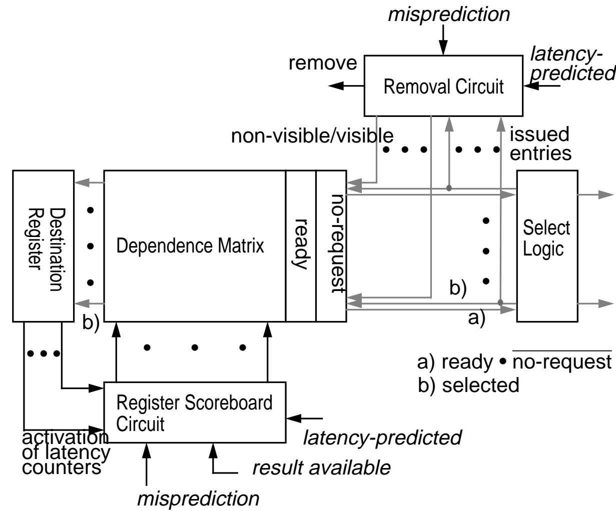


Figure 6.5 Issue-Queue structure.

The *removal circuit* is used to remove issued instructions from the issue queue (if it is safe to remove them), as well as to make them visible again (if they must be re-issued).

The *register-scoreboard circuit* is used for activating latency counters (for each issued instruction) or for unsetting columns (for each nullified instruction). As *ready bits* are re-evaluated every cycle, nullified instructions will re-evaluate their ready bits and all instructions dependent on the nullified instructions will be slept.

On every cycle, the *removal circuit* is aware of the issued instructions and the *register-scoreboard circuit* is notified of their destination registers. Both circuits are also notified of which issued instructions are latency predicted, and of the results of the prediction verifications. Moreover, the register-scoreboard circuit keeps track of the mispredicted-data availability.

This chapter presents the evaluation of two extreme approaches that differ on two aspects:

- When an instruction can be removed from the issue queue.
- Which instructions are nullified on a misprediction.

6.3.1 Non-selective nullification

The simplest recovery mechanism conservatively assumes that all the instructions are issued inside a potential speculative window and, on a misprediction, they are dependent on a latency-predicted instruction. All issued instructions are then retained in the issue queue during a

number of cycles equal to the verification delay minus one. After that, if no latency-misprediction is detected, the instructions can be removed from the issue queue. Otherwise, on a misprediction, all the instructions issued on the speculative window of the mispredicted instruction must be nullified and re-issued (for instance, instructions **e**, **f**, **g** and **h** in Figure 6.3). Consequently, the columns related to these instructions must be reset and the issue-queue entries must be made visible again.

To perform both actions, the *register scoreboard circuit* and the *removal circuit* respectively track the destination registers and the issue-queue entries of the instructions issued each cycle. The information related to a cycle can be discarded as soon as the instruction wave is outside any speculative window.

On a misprediction, both circuits aggregate the information related to the speculative window of the mispredicted instruction. After that, the *register scoreboard circuit* clears the columns related to the destination registers of the instructions to be nullified, and the *removal circuit* unsets the no-request bits of the issue-queue entries of these instructions. Moreover, the *register scoreboard circuit* also clears the destination register of the mispredicted instruction.

Among nullified instructions, there may be instructions independent of the mispredicted instruction. These instructions will immediately compete to be selected for issue because their source operands are still available (for instance, instructions **f** and **h** in Figure 6.3).

A possible implementation of the tracking mechanism uses bit vectors; every cycle, a bit vector is allocated in every circuit. Every bit vector of the *register scoreboard circuit* has as many bits as physical registers; setting its i-th bit indicates that the instruction that produces the i-th register has been issued on the related cycle. Every bit vector of the *removal circuit* has as many bits as issue-queue entries; setting its j-th bit indicates that the instruction allocated in the j-th issue-queue entry has been issued on the related cycle. The amount of bit vectors of each circuit is equal to the verification delay minus one.

On a misprediction, both circuits aggregate the information by OR-ing the bits vectors related to the cycles of the speculative window. The resultant bit vectors are used to clear the columns and the no-request bits on a single cycle.

6.3.2 Selective nullification

The previous mechanism is simple but conservative because it assumes that all the issued instructions are inside a potential speculative window and, on a misprediction, independent instructions inside this speculative window are also nullified. A more selective mechanism keeps in the issue queue only instructions dependent on a latency-predicted instruction not yet verified, and nullifies just these instructions on a misprediction. For instance, in Figure 6.3, this mechanism does not nullify instructions **f** and **h**, and retains in the issue queue only the instructions **e** and **g**.

We suppose that the cycle following the issue of an instruction is used to compute the dependence of the issued instructions on a latency-predicted instruction not yet verified. Then, independent instructions are removed from the issue queue one cycle after issuing them, and dependent instructions are kept in the issue queue while the prediction is not yet verified.

Register-scoreboard circuit tracks dependencies and notifies them (not shown in Figure 6.5) to the *removal circuit* to track the issue-queue entries of the dependent instructions. To do so, each circuit uses bit vectors with the same size as in the previous subsection, but managed differently. When a latency-predicted instruction is issued, a bit vector is allocated in each control circuit; bit vectors of the *register-scoreboard circuit* are initialised by setting the destination register of the instruction. These bit vectors are updated in successive cycles with the destination registers and the issue-queue entry numbers of the dependent issued instructions.

Register-scoreboard circuit tracks dependencies using as inputs the identifiers of the source operands of the issued instructions. If any source operand is marked in the bit vector of a latency-predicted instruction, the control circuit sets the destination register of the issued instruction in this bit vector. Then, a bit vector shows the registers dependent on the related latency-predicted instruction.

On mispredictions, each circuit uses the bit vector related to the mispredicted instruction. Bit vectors are freed as in the non-selective mechanism. Thus, the amount of bit vectors of each circuit is equal to the issue-width of latency predicted instructions times the verification delay minus one.

6.4 Keeping issued instructions in the recovery buffer

Keeping speculatively-issued instructions in the issue-queue reduces the queue's ability to look-ahead for independent instructions. This section develops a recovery mechanism that keeps issued instructions in a structure apart from the issue queue while they can be nullified: the recovery buffer.

Figure 6.6 shows the placement of the recovery buffer in the pipeline. In every cycle, instructions can be issued from the issue queue, from the recovery buffer or from both structures to the execution pipelines; in the latter case, each pipeline is fed prioritarily from the recovery buffer.

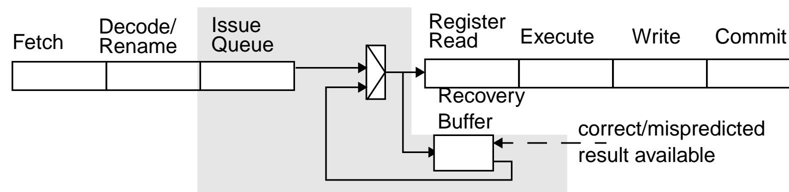


Figure 6.6 Placement of the recovery buffer in the processor pipeline

After issuing the instructions, they are removed from their source structures and are stored in the recovery buffer, and they remain there while they can be nullified.

Each recovery-buffer entry stores all the instructions issued on the same cycle, i.e., an instruction wave. If no instruction is issued on a cycle, the related recovery-buffer entry is kept empty. Thus, the recovery-buffer entries are time-ordered in issue order; that is, the relative issue cycles among instruction waves are maintained. For instance, on cycle 5 of Figure 6.3, the recovery buffer holds the following instruction waves: (**e**, **f**) and (**g**, **h**).

When a prediction is verified and it turns out to be correct, the recovery-buffer entries related to the speculative window of the latency-predicted instruction are freed. However, on a misprediction, the instructions dependent on the mispredicted instruction are retained in the recovery buffer until they can be re-issued. For instance, in example of Figure 6.3, instruction waves (**e**) and (**g**) would be retained.

For each latency-mispredicted instruction, the recovery buffer identifies the range of recovery-buffer entries related to the instruction. Then, when the result of a mispredicted instruction is available, the re-issue logic of the recovery buffer scans the entry range (one entry per cycle) related to the instruction to re-issue its dependent instructions. For instance, in the example of Figure 6.3, the re-issue logic scans the entries that hold the instruction waves (**e**) and (**g**).

As in the previous models, on cycles where a misprediction is detected (that is, the last cycle of the speculative window of the mispredicted instruction), the instructions selected to be issued are not issued and remain in their source structure. Furthermore, in the issue-queue structure, instructions dependent on the nullified instructions are slept until nullified instructions are re-issued (Section 6.3).

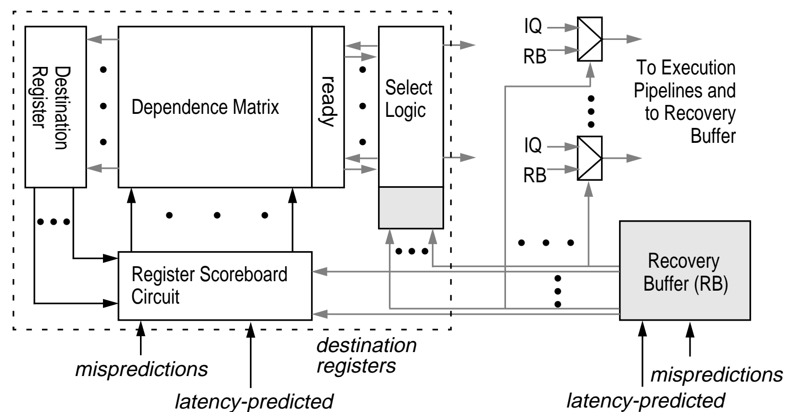


Figure 6.7 Interface between the issue queue and the recovery buffer

Figure 6.7 shows the interface between the issue queue and the recovery buffer. The *removal circuit* is not shown because issued instructions are always removed from the issue queue without waiting for the prediction verification; also, neither are the *no-request* bits needed.

Execution pipelines can be fed from both the issue queue and the recovery buffer. The multiplexers are controlled by signals generated by the recovery buffer. These signals are also used by the *select logic* to avoid selecting some instructions due to the higher priority of the instructions re-issued from the recovery buffer.

To wake-up the issue-queued instructions dependent on the re-issued instructions, the recovery buffer notifies the destination registers of the re-issued instructions at every cycle.

The re-issue logic of the recovery buffer has a lower complexity than the issue logic of the issue queue, because the former makes use of the scheduling performed when the instructions were previously issued. This logic is described in the next section.

6.4.1 Recovery-Buffer organization

The recovery buffer has three instruction storage components (Figure 6.8): *pending-verification buffer*, *first-level buffer* and *second-level buffer*. The pending-verification buffer stores latency-predicted instructions not yet verified. The first-level buffer stores issued instructions potentially dependent on the instructions stored in the pending-verification buffer. The second-level buffer stores issued instructions dependent on latency-mispredicted instructions.

The issued instruction waves are stored in the first-level buffer and they are removed from it when they are outside all the potential speculative windows. The number of cycles that an instruction wave remains in this buffer is then fixed and equal to the verification delay minus one.

When an instruction wave leaves the first-level buffer, each one of the instructions is either moved to the second-level buffer or discarded. Moreover, the latency-predicted instructions of the wave are either moved to the second-level buffer or to the pending-verification buffer. These decisions are taken by considering if the instructions are included in the speculative window of a mispredicted instruction, and if they are dependent on a mispredicted instruction.

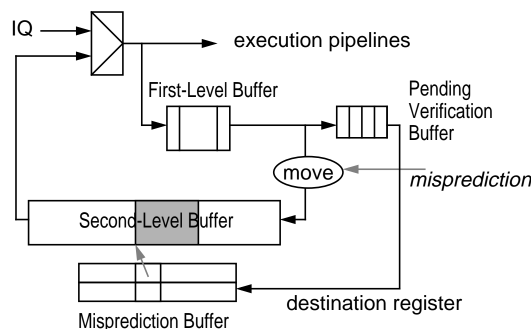


Figure 6.8 Recovery-Buffer organization

The number of entries of the pending-verification buffer is equal to the duration of the independent window. So when a latency-predicted instruction leaves this buffer, its prediction is verified.

On a misprediction, the recovery buffer allocates an entry in a structure named *misprediction buffer*. This entry stores the destination register of the mispredicted instruction and a pointer to the first entry of the second level buffer related to the mispredicted instruction.

After that, during a number of cycles equal to the verification delay minus one, the instruction waves that leave the first-level buffer are analysed looking for instructions dependent on the mispredicted instruction. The dependent instructions are moved to an empty entry of the second-level buffer, and the independent instructions are discarded. Concurrently, execution pipelines are fed with ready instructions

For instance, in Figure 6.3, instruction **a** is moved to pending-verification buffer on cycle 4. If **a** is a latency-mispredicted instruction, instructions **e** and **g** are moved to second-level buffer on cycles 6 and 7.

Re-issue logic of the recovery buffer. The idea is to make use of the scheduling performed when the instructions were previously issued. The re-issue logic is based on the fact that the recovery-buffer entries are time ordered and only one entry is analysed on a cycle. So, the re-issue logic does not need to account explicitly for instruction latencies. It is enough to account for the status (availability) of the physical registers.

The status of the physical registers can be maintained locally because the recovery buffer analyses all issued instructions. When an instruction is issued, its destination register is marked as available in the recovery buffer. In addition, the recovery buffer is notified of the misprediction; so the status of the destination registers of the nullified instruction can be updated locally as not-available.

Figure 6.9 shows the re-issue logic of the recovery buffer. We distinguish three components: two dependence matrices (similar to the matrix described in Section 6.2.2) and a register scoreboard circuit without latency counters. A dependence matrix is used by instruction waves leaving the first-level buffer, and the other one is used by instruction waves re-issued from second-level buffer. The number of rows of both dependence matrices is equal to the processor issue width. Register-scoreboard circuit controls columns of both dependence matrices. A column is set or unset in both matrices at the same time.

Some input signals of the register-scoreboard circuit are generated by the first-level buffer to unset columns. Other inputs are generated by the second-level buffer to set columns. The remaining inputs are used to set/unset columns related to the destination register of mispredicted instructions.

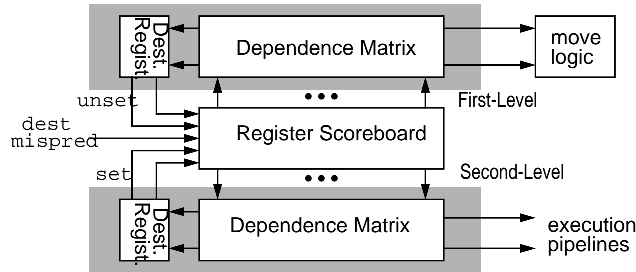


Figure 6.9 Re-issue logic of the recovery buffer

When an instruction wave leaves the first-level buffer, it is scanned using the associated dependence matrix. This matrix has two functions: updating the status of the destination registers of the wave and, after detecting a misprediction, discriminating dependent from independent instructions.

The dependence matrix associated to the second-level buffer is used when the result of a mispredicted instruction is available. This matrix has two functions: checking the availability of the source registers of an instruction wave and updating the status of the destination registers of the re-issued instructions.

When a latency-predicted instruction leaves the pending-verification buffer, its prediction has been verified. On a misprediction, the register-scoreboard circuit unsets the column associated to its destination register.

First-level buffer. When an instruction wave leaves the first-level buffer, the columns related to the destination registers of the wave are set, and the wave is analysed in the dependence matrix of the first-level buffer. We differentiate two cases. In the first case, the instruction wave is not included in the speculative window of a mispredicted instruction. In this case, the dependence matrix indicates that all source registers of the instructions are available.

In the second case, the instruction wave is included in the speculative window of a mispredicted instruction. As the column associated to the destination register of the mispredicted instruction has been previously unset, the dependence matrix discriminates between instructions dependent and independent of the mispredicted instruction. Columns associated to the destination registers of the dependent instructions are unset. By unsetting the columns, the chain of instructions dependent on the latency instruction is detected on consecutive cycles.

Moreover, the output of the dependence matrix is used to indicate to the move logic which instructions of an instruction wave must be stored in the second-level buffer.

Second-level buffer. The dependence matrix associated to the second-level buffer is used to re-issue instructions when the result of a mispredicted instruction is available. From the misprediction buffer a pointer to a second-level buffer entry and the identifier of the destination

register of the mispredicted instruction are obtained. This register identifier is used to set the associated column of the matrices. After that, from the pointed entry, a fixed range of second-level buffer entries is scanned to re-issue the chain of dependent instructions. At each cycle a single second-level buffer entry is scanned.

Note that on a cycle as many mispredictions may be detected as the number of latency-predicted instructions simultaneously issued. A new misprediction can also be detected while another is being processed; that is, while moving dependent instructions form the first-level buffer to the second-level buffer. Then, in the scanned range of second-level buffer entries, instructions dependent on several mispredictions may be stored. Moreover, data can return from memory hierarchy in an order different from misprediction-detection order.

The role of the second-level-buffer dependence matrix is to identify ready instructions in the scanned range of entries. In order to do this, in each cycle an instruction wave is analysed. Ready instructions are re-issued and the columns associated to their destination registers are set.

6.4.2 Recovery buffer with selective nullification

This mechanism nullifies only instructions dependent on a latency-mispredicted instruction. First, we describe the management of the storage components of the recovery buffer and their sizes. Second, we use an example to show the detail of the mechanism. Finally, we present an optimization in the re-issue logic to avoid performance degradations.

Storage components of the Recovery Buffer. All storage components are handled using the FIFO policy. The instructions remain in the first-level buffer and in the pending-verification buffer for a fixed number of cycles. Instruction waves remain in the first-level buffer for a number of cycles equal to the verification delay minus one. Latency-predicted instructions remain in the pending-verification buffer a number of cycles equal to the duration of the independent window. The second level buffer and misprediction buffer are handled as a circular queue, and after re-issuing the instruction, its storage is freed using a FIFO policy.

Each misprediction requires a number of second-level-buffer entries equal to the verification delay minus one.

In the worst case, when the speculative windows of the mispredictions do not overlap, the maximum number of second-level buffer entries needed is equal to the number of pending mispredictions supported by the processor times the verification delay minus one. Although second-level-buffer size can be large, the accesses to this buffer can be pipelined without performance degradation.

Example. Figure 6.10 shows an example where the speculative windows of the mispredicted instructions (**a** and **c**) overlap by two cycles; instructions issued on cycles 8 and 9 can be dependent on both latency-predicted instructions.

On cycle 9, instruction **a** leaves the pending-verification buffer and a misprediction is detected. In this moment, the instructions waves stored in the first-level buffer are (**d**), (**e**) and (**f**). A misprediction-buffer entry is allocated to store the destination register of the mispredicted instruction and the current tail pointer to the second-level buffer.

During a number of cycles equal to the verification delay minus one, a second-level-buffer entry is allocated every cycle; an instruction wave is also analysed every cycle. Every entry will store the instructions of the analysed wave that are dependent on the mispredicted instruction.

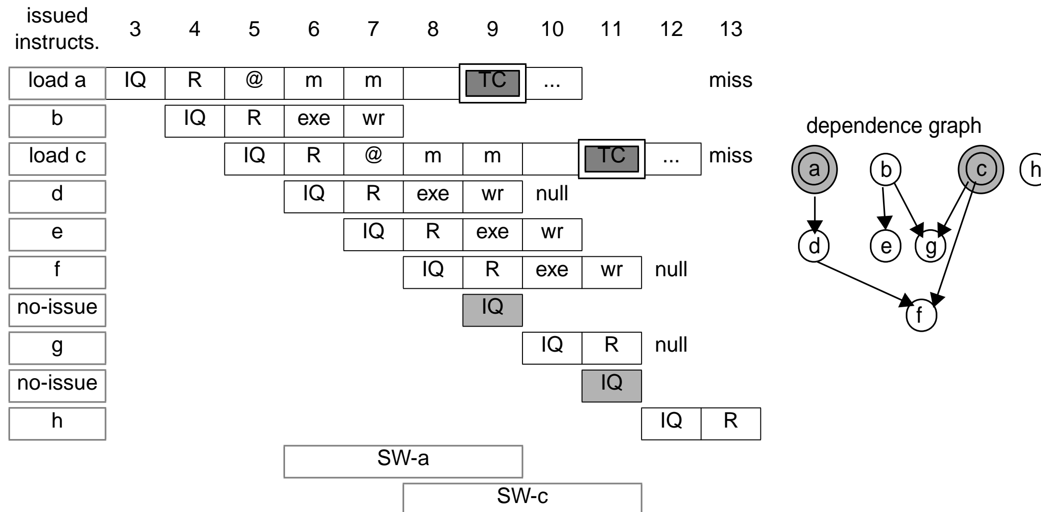


Figure 6.10 Recovery buffer with selective nullification
(load instructions **a** and **c** are latency-mispredicted instructions)

On cycle 10, instruction **d** is inserted in the second-level buffer. On cycle 11, the misprediction of instruction **c** is detected and a misprediction-buffer entry is allocated. Furthermore, because instruction **e** is independent of the misprediction, a second-level-buffer entry is left empty. On cycle 12, instruction **f** is moved to the second-level buffer; note that this instruction is in the speculative windows of both mispredicted instructions. On cycles 13 and 15, second-level-buffer entries are kept empty. On cycle 14, instruction **g** is moved.

When the result of the mispredicted instruction **a** is available (cycle *n*), instruction **d** will be reissued. On cycle *n*+1 no instructions will be re-issued because the related second-level buffer entry is empty. On cycle *n*+2, instruction **f** is not re-issued because it must wait until the availability of the result of the mispredicted instruction **c**.

6.4.2.1 Re-issue optimization

In some cases, when the result of a mispredicted instruction is available, the first entries of the scanned range of second-level-buffer entries are empty. One such case is when the speculative windows of several latency-mispredicted instructions overlap (note that instruction-issue is

stalled when a misprediction is detected). Another case is when no instruction dependent on the latency-mispredicted instruction has been issued on the first cycles of its speculative window.

The cycles used to scan the first empty entries are not needed for a proper scheduling of the next not empty entries; consequently, these cycles constitute a delay. This situation can be critical if the latency of the instructions to be re-issued is long (for instance, floating-point operations).

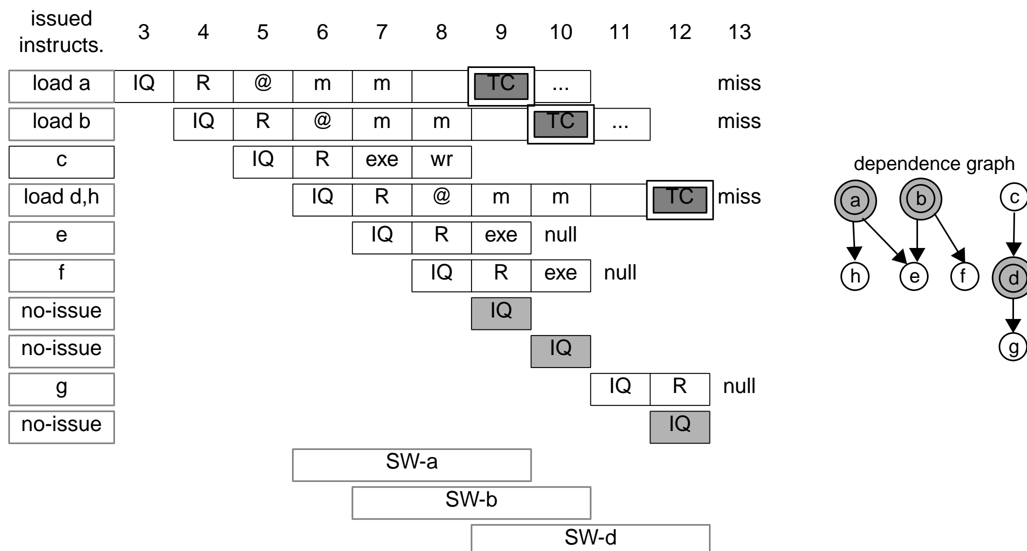


Figure 6.11 Instruction-flow example using the recovery buffer with selective nullification and the re-issue optimization
(loads **a**, **b** and **d** are latency-mispredicted instructions; on the first cycles of the speculative window of instruction **d**, no instructions are issued)

Figure 6.11 shows an example where, on the overlapped cycle between two speculative windows (cycle 9), no instructions are issued due to the mispredicted instruction **a**. Moreover, on cycle 10, no instructions are issued due to the mispredicted instruction **b**. These cycles are the first cycles of the speculative window of instruction **d**.

In the second-level buffer, the relevant entry ranges stores the following instruction waves: {(**h**), (**e**), (**f**), (**-**), (**-**) and (**g**)}. When data of mispredicted instruction **d** is available, the entry range {(**-**), (**-**), (**g**)} is scanned to re-issue the dependent instructions. Consequently, no instructions will be re-issued from second-level buffer on the first and the second cycle of the scanning process.

In order not to delay the re-issue of instructions, the cache controller can notify to the second-level buffer data arrival several cycles in advance. Then, re-issue-logic can skip the first empty entries of the entry range.

6.4.3 Recovery buffer with non-selective nullification

This mechanism nullifies all the instructions issued inside the speculative window of a latency-predicted instruction when a misprediction is detected. The nullified instructions independent of the misprediction are re-issued without delay from the first-level buffer. Concurrently, the dependent instructions are moved to the second-level buffer.

An example is shown Figure 6.12. The latency prediction of instruction **a** is verified on cycle 5. A misprediction is detected and the instruction waves (**e**, **f**) and (**g**, **h**) are nullified. On cycle 6, the instruction wave (**e**, **f**) is analysed in the dependence matrix associated to the first-level buffer; after that, instruction **e** is recorded in the second-level buffer and instruction **f** is re-issued from the first-level buffer. Analogously, on cycle 7, the dependent instruction **g** is moved to the next entry of the second-level buffer and instruction **h** is re-issued. Concurrently, some instructions are issued from the issue queue (**j** and **k** on cycles 6 and 7).

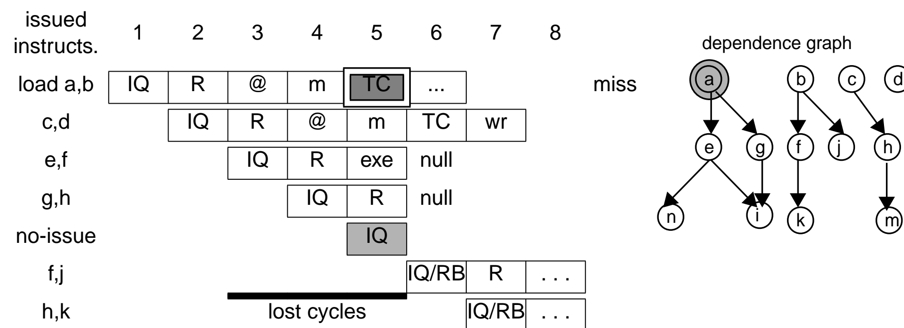


Figure 6.12 Instruction-flow example of the recovery buffer with non-selective nullification (load **a** is a latency-mispredicted instruction. IQ/RB stands for cycles where instructions are issued from the issue queue and re-issued from the recovery buffer).

6.4.4 Effect of wrong-path instructions on recovery-buffer structures

When a branch misprediction is detected, wrong-path instructions are squashed and their physical destination registers are freed. Age identifiers are used to detect the instructions to be squashed, and shadow map tables are used to re-establish the mapping from architectural to physical registers.

Recovery-buffer structures also require attention when a branch misprediction is detected. The local status of the physical registers must be repaired and the wrong-path instructions stored in these structures must be squashed. These actions are not performed immediately; they are performed concurrently with the regular operation of the recovery buffer.

In the recovery buffer, an age identifier is stored with each instruction[2]. In addition, a new recovery-buffer structure (the squashed-range buffer) holds the range of age identifiers of the

2. This identifier can be the processor age identifier, or the recovery buffer can build a local age identifier from the processor age identifier.

squashed instructions. An entry of this structure is freed when the age identifier of a committed instruction is younger than the youngest limit of the range.

Regular operations of the recovery buffer that analyse instruction waves are: a) when an instruction wave leaves the first-level buffer and b) when an instruction is re-issued from the second-level buffer. In both cases, the local status of the physical registers is updated. To squash wrong-path instructions, the age identifiers of the analysed instructions are checked with entries of the squashed-range buffer. If the age identifier is included in a squashed range, the instruction is squashed and neither recorded in the second-level buffer nor re-issued.

When a mispredicted instruction must be squashed, previous regular operations squash neither it nor its dependent instructions (all of them are wrong-path instructions). This case is managed by the freeing algorithm of the second-level buffer. As this buffer and the misprediction buffer are managed as circular queues, the age identifier of the latency-mispredicted instructions will achieve the head entry of the misprediction buffer. The age identifier of the head entry is then compared with the squashed-range-buffer entries. If the age identifier is included in a squashed range, the related instruction is squashed.

The local status of the physical registers freed by squashed instructions is repaired in time by the algorithm described in Section 6.4.1. This algorithm updates the local status of a physical register when its producer instruction leaves the first-level buffer. The freed physical registers are assigned to a producer instruction in the new path, and younger instructions use it as a source register. When the producer instruction leaves the first-level buffer, before its consumer instructions, the local status of the register is repaired. Before this time, no instruction needs to check the local status of this register.

6.5 Evaluation

6.5.1 Evaluation environment

We have evaluated our proposals over all the SPEC95 benchmarks (Section A.2). Our evaluations used a cycle-by-cycle simulator (Section A.1.2) that models the processors described in Figure 5.1, and assume hit-latency prediction for all load instructions. Table A.3 and Table A.4 show the simulation intervals selected for each integer and floating-point benchmark respectively. Table 6.1 shows the miss rate of the 64K first-level data cache for each benchmarks in its selected simulation interval.

The evaluations varied the following processor parameters: the verification delay and the issue-queue size.

- The verification delay was defined in Section 6.2 as the duration of the speculative window; we consider 2-cycle, 3-cycle and 4-cycle verification delays.

- In current processors, the integer issue queue typically does not exceed 20 entries. For instance, 20 entries in Alpha 21264, 18 entries in AMD Athlon and 20 entries in Intel P6. Moreover, the number of in-flight instructions (reorder-buffer size) is typically 2 to 4 times bigger. In this chapter, the issue-queue sizes of the baseline processor model are 20-entry integer issue queue, and 15-entry floating-point issue queue. We present results also for 15-entry and 25-entry integer issue queues and for 10-entry and 20-entry floating-point issue queues.

SPEC INT		SPEC FP	
Benchmark	%miss	Benchmark	%miss
go	1.9	tomcatv	18.6
m88ksim	0.7	swim	6.1
gcc	2.2	hydro2d	15.0
compress	12.5	mgrid	4.0
li	3.3	applu	7.1
ijpeg	0.6	turb3d	4.1
perl	0.5	fpppp	0.2
vortex	3.0	wave5	7.5

Table 6.1 Miss rates for the SPEC95 benchmarks in a direct-mapped 64K first-level data cache.

6.5.2 Results

We present the results of the integer benchmarks separately from the results of the floating-point benchmarks because the behaviour of the recovery-buffer mechanisms depends on the computational latency of the instructions.

In figures, the following acronyms are used for identifying the evaluated mechanisms: a) IQNS: keeping in the Issue Queue with No Selective nullification, b) IQS: keeping in the Issue Queue with Selective nullification, c) RBNS: Recovery Buffer with No Selective nullification, and d) RBS: Recovery Buffer with Selective nullification.

6.5.2.1 Integer benchmarks

In all the evaluations performed in this section we used a 10-entry floating-point issue queue and the following integer issue-queue sizes: 15, 20 and 25 entries. First, the sensitivity of the evaluated mechanisms to the verification delay is shown. In Figure 6.13, each graph is related to an issue-queue size, the horizontal axis stands for the verification delay and the vertical axis for the harmonic mean of the IPC's of the integer benchmarks.

The sensitivity to the verification delay of RBNS and RBS remains small when the verification delay or the issue-queue size increases. Performance differences between RBNS and RBS are small (less than 1.3%). As the instructions are removed from the issue queue after issuing them, the differences arise because RBNS re-issues nullified instructions independent of the mispredicted instructions.

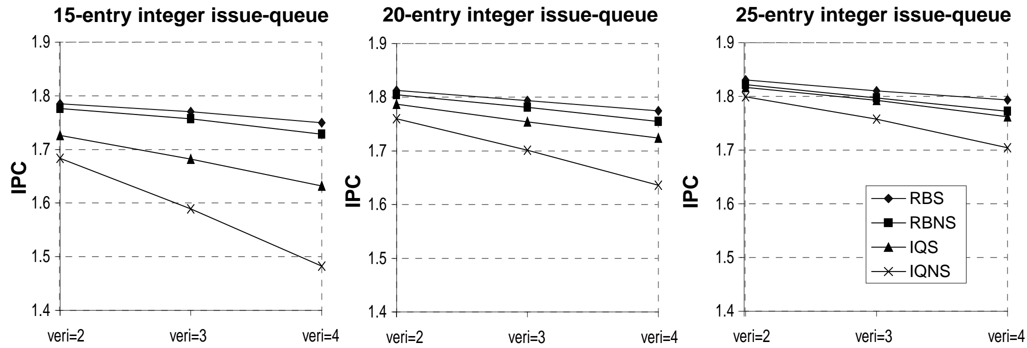


Figure 6.13 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks
(each graph is related to an integer issue-queue size; the vertical axis stands for the IPC, and the horizontal axis stands for the verification delay).

However, the sensitivity to the verification delay of IQNS and IQS is very significant for the 15-entry issue queue, and decreases as the issue-queue size increases. This result shows that keeping all instructions in the issue queue a number of cycles equal to the verification delay (IQNS) can reduce performance significantly.

IQS retains in the issue queue only instructions dependent on the latency-predicted instructions but its performance is smaller than the performance of RBNS, although both are close in a 25-entry issue-queue. The recovery buffer enables the issue-queue entries to be freed of some of the instructions dependent on the mispredicted instructions. These freed entries can be assigned to new instructions, increasing the look-ahead ability of the instruction scheduler. Moreover, this ability overcomes the lost issue slots for re-issuing independent instructions.

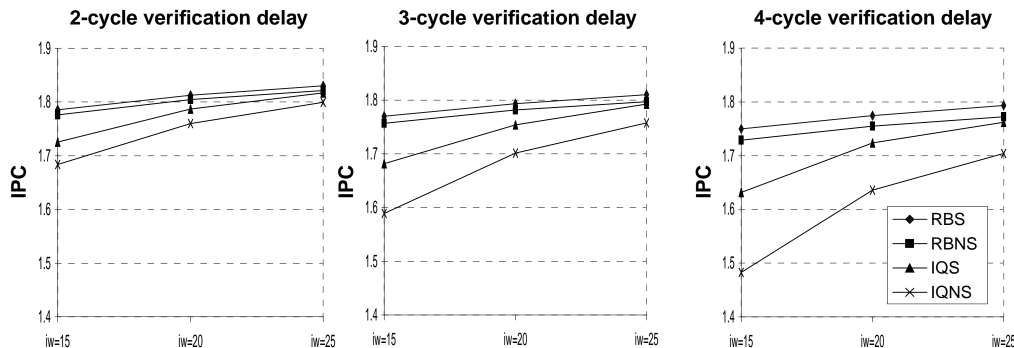


Figure 6.14 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks
(each graph is related to a verification delay; the vertical axis stands for the IPC, and the horizontal axis stands for the integer issue-queue size).

The ability of the recovery buffer to free issue-queue entries allows the use of a smaller issue queue. Figure 6.14 shows almost the same information as Figure 6.13, but grouping data by verification delay and putting the issue-queue size in the horizontal axis. In all cases, RBNS and RBS can obtain the same performance as IQNS and IQS, but with a reduction of the issue-queue size of between 20% to 25%[3].

The implementation of the selective nullification in the issue queue may be critical with intensive one-cycle operations. Comparing non-selective mechanisms, the longer the verification delay, the larger the issue-queue size can be reduced. RBNS is therefore an attractive solution.

6.5.2.2 Floating-point benchmarks

In all the evaluations performed in this section we used a 20-entry integer issue queue and the following floating-point issue-queue sizes: 10, 15 and 20 entries.

In the evaluated mechanisms, floating-point benchmarks show (Figure 6.15 and Figure 6.16), a different behaviour than integer benchmarks. Non-selective mechanisms are sensitive to the verification delay while selective mechanisms are not. This makes IQS performance better than RBNS performance when issue-queue size increases.

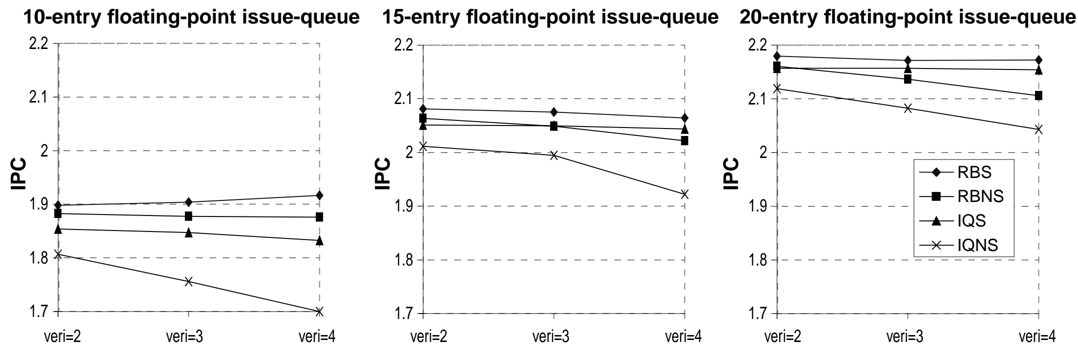


Figure 6.15 Influence of both the recovery mechanism and the verification delay on the performance of processors executing floating-point benchmarks (each graph is related to a floating-point issue-queue size; the vertical axis stands for the IPC, and the horizontal axis stands for the verification delay).

The best (RBS) and the worst (IQNS) mechanisms are the same for both classes of benchmarks. Furthermore, RBS is almost insensitive to the verification delay while IQNS is very sensitive to it.

3. This reduction is significant because the delay of the issue logic of the issue queue depends quadratically on the product of the instruction-issue width and the instruction-window size [PJS97].

The behaviour of the floating-point benchmarks is related to the computational latency of the floating-point instructions; four-cycle latency for FPadder and FPmultiplier are used, and the evaluated verification delays range from two to four cycles. These latencies forbid the existence of a chain of dependent instructions larger than one instruction in the speculative window of a FPload instruction. Therefore, only the first data-flow level dependent on a FPload is included in its speculative window. Other data-flow levels are held in issue-queue entries and the scheduler cannot look-ahead because the issue queue is full. Thus, the capacity of the recovery buffer to store dependent instructions is only slightly used.

The value retrieved by a load instruction is not used by many instructions either; that is, its fan-out is small. Then, in a non-selective mechanism, a large number of the independent instructions[4] are often re-issued. As the latency of the re-issued instructions is long in floating-point benchmarks, the execution of the chain of dependent instructions is significantly delayed, and potential ILP is lost.

The implementation of selective mechanisms for integer-benchmarks can be critical because the latency of most ALU operations is one cycle. However, as the computational latency of the FP operations is longer, the implementation of selective mechanisms for them is less critical.

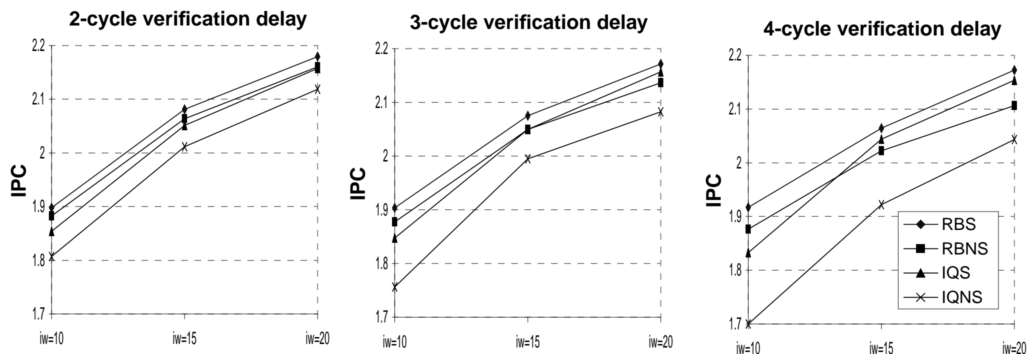


Figure 6.16 Influence of both the recovery mechanism and the issue-queue size on the performance of processors executing floating-point benchmarks
(each graph is related to a verification delay; the vertical axis stands for the IPC, and the horizontal axis stands for the floating-point issue-queue size).

6.6 Conclusions

This chapter addresses recovery mechanisms for dealing with latency-predicted instructions. It compares the performance of a conventional mechanism that keeps issued instructions in the issue queue with a mechanism that stores these instructions in a recovery buffer apart from the

---

4. Using a 15-entry floating-point issue queue and a 4-cycle verification delay, 85% of the nullified instructions on floating-point benchmarks are independent of the mispredicted instructions. In integer benchmarks, using a 20-entry integer issue-queue and a 4-cycle verification delay, this percentage drops to 53%.

issue queue. It also compares selective with non-selective instruction nullifications on mispredictions.

We designed a recovery-buffer mechanism and we evaluated it in the context of load-latency prediction. Our results show that, under the same nullification conditions, the recovery-buffer mechanism outperforms the mechanism that retains the instructions in the issue queue. Moreover, the recovery-buffer mechanism is less sensitive to the verification delay of the predictions. For integer benchmarks, it allows a reduction in the issue-queue size of between 20-25% without performance decrease. It enables the issue-queue logic to free entries and to insert new instructions in the issue queue, and therefore increases the capacity of the scheduler to look-ahead for independent instructions.

We also showed that for issue queues feeding functional units with intensive one-cycle latency operations, the simple recovery-buffer mechanism with non-selective nullification is an attractive solution. On the other hand, for issue queues feeding functional units where the latency of most instructions is long, the use of selective nullification is preferable. Note that, in this case, selective nullification is not critical due to the long latency of the operations.

On each latency misprediction, the recovery buffer suffers a one-cycle penalty where no instruction is neither issued nor re-issued; this cycle coincides with the last cycle of the speculative window of the mispredicted load instruction. This penalty cycle is used to avoid issuing instructions that may be dependent on the misprediction and to update the issue queue in order to sleep all the instructions dependent on the nullified instructions. As a future work, the recovery buffer with selective nullification may be enhanced in order to reduce the performance degradation produced by this penalty cycle. The idea is re-issuing instructions from the recovery buffer while the issue queue is being updated due to a latency misprediction.

We may take advantage of several facts:

- Most second-level caches provide tag-checking result before data-array contents. Moreover, tag-checking latency of second-level caches is slightly larger than tag-checking latency of first-level caches.

- We hope that the chain of instructions dependent on the mispredicted load instruction is short. That is, the number of instructions independent on the misprediction increases as the speculative window is traversed.

We then propose delaying latency-misprediction handling until knowing tag-checking result on second-level cache. That is, we will extend the speculative window until tag-checking in second-level cache.

- When a miss on the first-level cache is detected, no recovery action is started. Consequently, instructions dependent on the mispredicted load instruction may still be issued. Issuing them allows freeing some issue-queue entries and inserting new instructions in the issue queue; however, these dependent instructions must be re-issued later. Instructions independent on the memory access may also be issued. We hope that most issued instructions will be independent on the memory access due to the short dependence-chain lengths.

- On a second-level cache hit, the second-level cache will provide the data-array contents after a fixed number of cycles. When data is available, two actions are performed concurrently: the issue queue is updated, and the first instruction wave dependent on the mispredicted load is re-issued from the recovery buffer. That is, the issue queue is being updated while an instruction wave is being re-issued from the recovery buffer. On next cycles, ready instructions are issued from the issue queue and the remaining instruction waves are re-issued from the recovery buffer.

- On a second-level-cache miss, the issue queue is updated and all the issued instructions dependent on the load are kept in the recovery buffer until the memory system provides the accessed data. In this case, no instruction is neither issued nor re-issued on the penalty cycle.

Consequently, the one-cycle penalty where no instructions is neither issued nor re-issued arises only for memory references that miss in both cache levels. We expect this reduction on the number of cycles where no instruction is issued or re-issued will offer an attractive performance.

Further work is needed to evaluate the use of the recovery-buffer mechanism in other kinds of prediction scenarios, such as limited cases of value prediction; for instance, performing value prediction on load instructions and issuing their dependent instructions after issuing the predicted load instruction. We are also interested in studying how the recovery-buffer mechanism can be integrated into mechanisms that perform a dynamic data-flow pre-scheduling [MiSe01] and use an issue queue for accounting latency mispredictions or for waiting the result of unknown-latency instructions [CaGo00].

6.7 Detailed results

Figure 6.13 and Figure 6.15 presented average results for integer and floating-point benchmarks respectively. Following subsections present individual results for each benchmark.

6.7.1 Integer benchmarks

Figure 6.17 and Figure 6.18 show the results obtained in each SPEC95-INT benchmark.

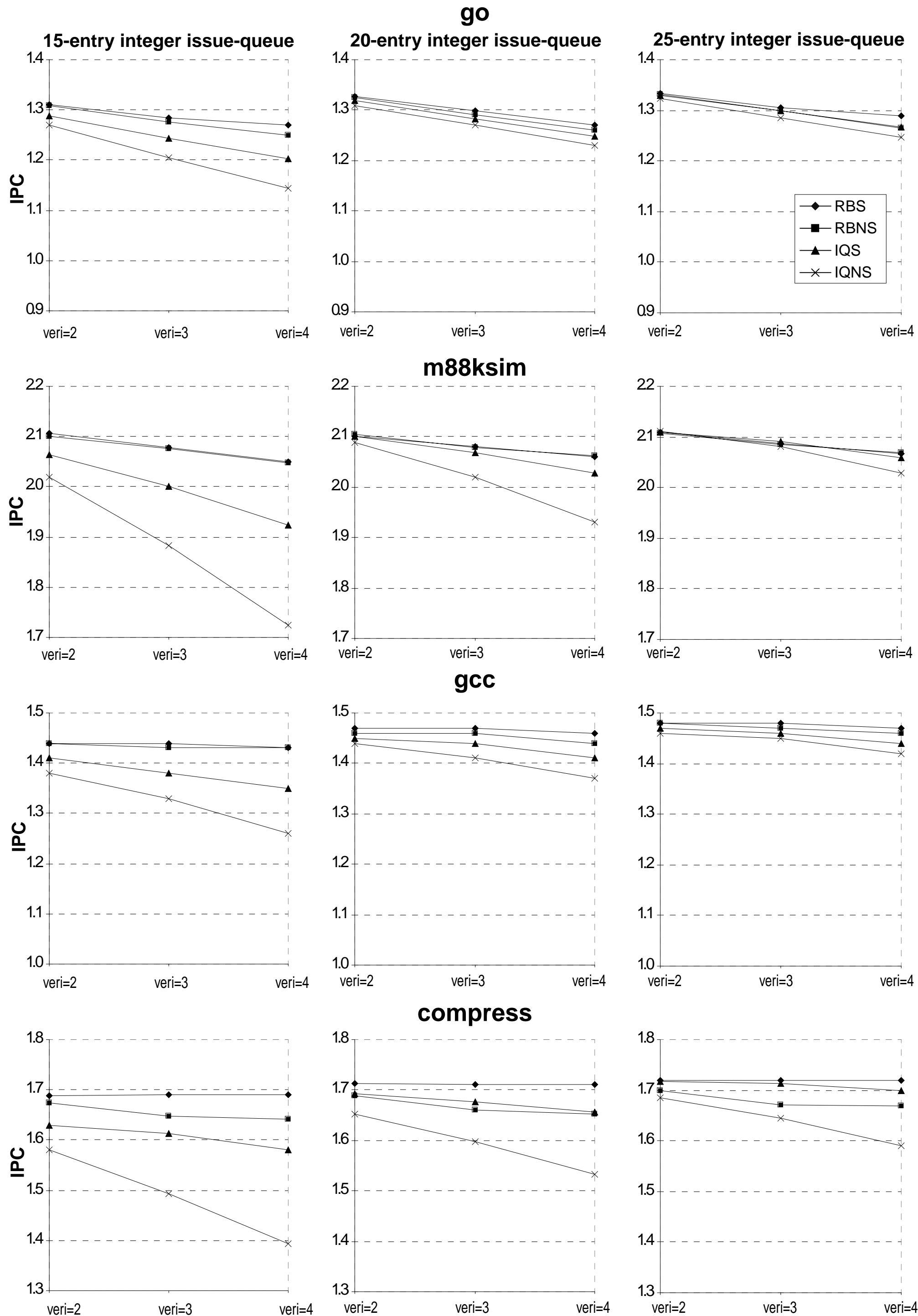


Figure 6.17 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks *go*, *m88ksim*, *gcc* and *compress*

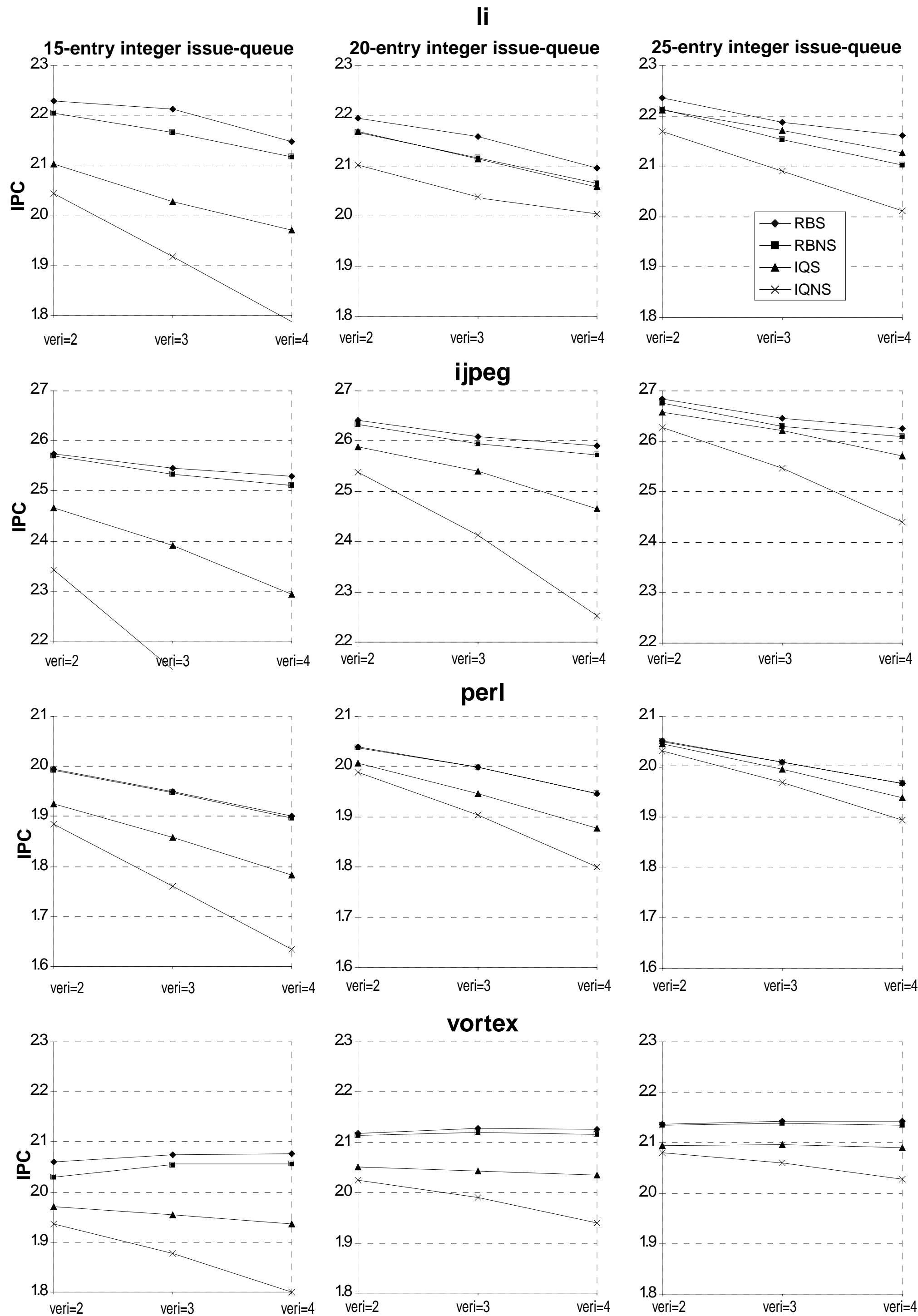


Figure 6.18 Influence of both the recovery mechanism and the verification delay on the performance of processors executing integer benchmarks *li*, *ijpeg*, *perl* and *vortex*

6.7.2 Floating-point benchmarks

Figure 6.19 and Figure 6.20 show the results obtained in each SPEC95-FP benchmark.

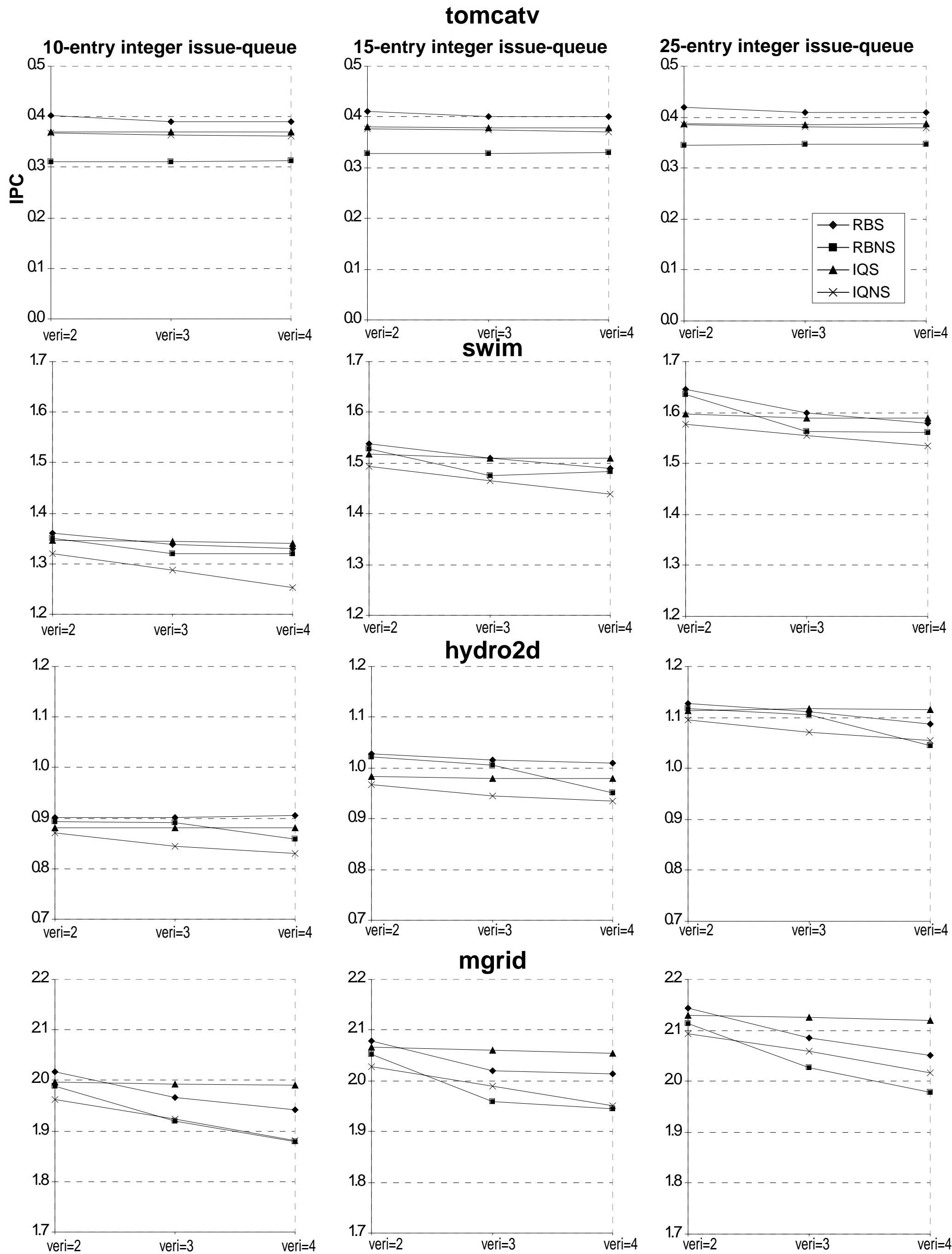


Figure 6.19 Influence of both the recovery mechanism and the verification delay on the performance of processors executing floating-point benchmarks *tomcatv*, *swim*, *hydro2d* and *mgrid*

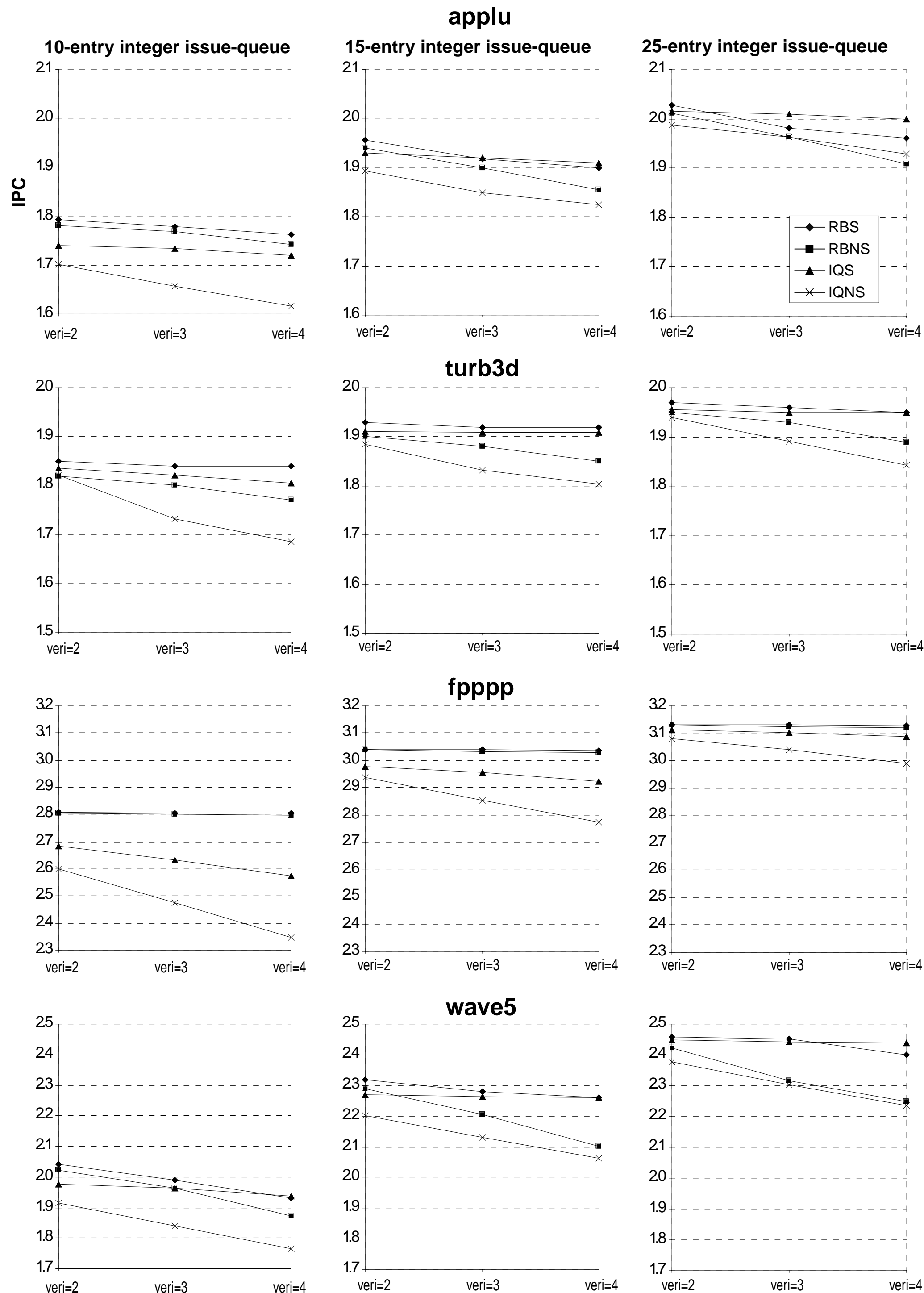


Figure 6.20 Influence of both the recovery mechanism and the verification delay on the performance of processors executing floating-point benchmarks *applu*, *turb3d*, *fpppp* and *wave5*

7 CONCLUSIONS

This chapter lists the conclusions of this work and describes some directions of future research

7.1 Summary

This thesis is concerned with exploiting the predictability of the effective address computed by the load instructions. Our goal was evaluating the effectiveness of address prediction to improve the performance of superscalar processors.

First, we analysed the phenomenon of address predictability. We studied the sources of address predictability, focusing on the high-level language structures that are compiled to predictable load instructions; we found usual high-level structures that produce predictable load instructions. Also, we evaluated the performance of conventional address predictors (Last-Address Predictor, Stride Address Predictor and Context Address Predictor) on both integer and floating-point benchmarks.

After that, we proposed several techniques that try to use more efficiently the area-cost devoted to an address predictor: we refer to these techniques as filtering and narrowing techniques. The application of these techniques is two folded. First, given an address predictor, they allow obtaining a new, similar-performance address predictor that needs a smaller area cost. Second, given an area cost, they allow obtaining an address-predictor configuration that best fits the available area cost. Our results allows area-cost reductions around 50% without performance effects on the Last-Address Predictor; moreover, these techniques can be applied to other address predictors (Stride-Address Predictor, Context-Address Predictor).

Next, we have applied address prediction to superscalar processors in order to execute speculatively instructions dependent on the predicted load instructions; we refer to these processors as address-speculative processors.

To begin with, our evaluations focused on the interaction between address prediction and branch-prediction resolution. This interaction appears because we assumed that the operands of a branch instruction must be known to be non-speculative before resolving the branch instruction. We describe several verification mechanisms that notify when a branch instruction can be resolved. From our results we conclude that address prediction is a promising alternative to improve the performance of superscalar processors; however, the verification process must be performed without delaying branch-instruction resolutions because these resolutions are critical on processor performance.

We then analysed address-speculative processors where the issue queue is decoupled from the reorder buffer. In this scope, the permanence of the instructions in the issue queue should be as short as possible; moreover, the selected recovery mechanism may influence on this permanence. We have considered two recovery mechanisms: a conventional recovery mechanism that keeps the speculatively issued instructions in the issue queue until they are known to be non speculative, and a recovery mechanism that records them in a separated structure; however, the latter mechanism restricts the speculative issue of the instructions until issuing the predicted load instruction. Our results show that restricting the speculative issue may be an attractive alternative.

We also evaluated the effect of some processor-design trends on the performance impact of address-speculative processors: increasing the cache latency and widening the issue width. Our results show that the performance impact of address prediction gets larger as data-cache latency increases and as the issue width widens.

This document finishes with the proposal and evaluation of recovery mechanisms for a speculative technique named latency prediction; these mechanisms differ on the structure used to record the speculatively issued instructions: the issue queue or a new structure named recovery buffer. Also, we have evaluated two nullification policies (non selective and selective) for each of these mechanisms. We applied latency prediction to predict the latency of the load instructions in order to schedule optimistically its dependent instructions. From our results, the

nullification policy to be used depends on the dominating instruction latency: while the non-selective policy is enough in scopes where the dominating latency is one cycle (integer issue queue), the selective policy must be used in scopes where the dominating latency is several cycles and the instruction-level parallelism is large (floating-point issue queue).

7.2 Future directions

In this section we detail some possible directions of further research:

To achieve a deeper understanding of the phenomenon of the address predictability, the analysis presented in Chapter 2 must be extended. First, we should consider new address-predictor models; for instance, address predictors that correlate the addresses computed by several load instructions and the results computed by arithmetic instructions. Also, we should analyse the load instructions that remain unpredictable. Finally, other ISA's must be studied because some predictability is inherent to the ISA and to the compiler technology.

Our study has analysed some design parameters of address-speculative processors. However, other parameters have not been evaluated such as the degradation imposed by a) performing address checks out of the processor core, b) wire delays in the communication between the address-check unit, the verification issue queue and the issue queue, c) non-immediate prediction-table updates, and d) banking prediction tables. We hope that these new evaluations will lead us to determine which are the most critical design parameters of address-speculative processors, and to design a cost-effective micro-architecture for an address-speculative processor.

Finally, in this work we have evaluated our proposed recovery mechanism (the Recovery Buffer) in two scopes: latency prediction and address prediction. We should evaluate the performance of the Recovery Buffer in processors that perform both kinds of predictions simultaneously, because we expect that the Recovery Buffer will reduce the pressure on the issue queue with respect to other recovery mechanisms.

A EVALUATION ENVIRONMENT

This appendix details the evaluation environment built for the experiences reported in this thesis. Section A.1 describes the tools used to evaluate the proposals of this thesis. Section A.2 lists and characterizes the benchmark suite run in the evaluations.

A.1 Evaluation tools

All the proposals presented in this thesis have been analysed using evaluation tools. These tools can be divided into two categories: instrumented executables and processor simulators. This section describes both types and their purpose in this thesis.

A.1.1 Instrumented executables

An instrumented executable not only behaves like the uninstrumented executable, but also executes some added code inserted at some specific points (for instance, before every load instruction). This added code performs the analysis programmed by the user (for instance, the evaluation of the data-cache hit rate). Instrumented executables perform analysis with a small execution-time overhead. However, they cannot model accurately a cycle-by-cycle evolution of the processor components.

ATOM instrumentation tool

The instrumented executables used in this work have been generated using the ATOM instrumentation tool [EuSr94]. To instrument a binary executable, ATOM requires two input files: the instrumentation file and the analysis file.

The instrumentation file tells ATOM:

- Which instructions of the executable must be instrumented (for instance, every load instruction, the first instruction of every basic block,...)
- The analysis routine that must be called at every instrumentation point
- The parameters of the analysis routines (for instance, the PC of the instrumented instruction, the target of a conditional branch, the effective address of a load instruction, register values,...).

The analysis file provides the implementation (in C language) of the analysis routines referenced in the instrumentation file.

Instrumented executables in this thesis

These kinds of simulators have been used in this thesis to collect basic statistics related to the address predictors; for instance, the amount of predictions and the misprediction rate.

A.1.2 Processor simulators

A processor simulator models the internal timing of a processor at the cycle level. Working at this level, the simulator can model the relationship between pipeline stages and processor components, and can model events that will not affect the final architectural state; that is, events related to non-committed instructions. However, the cost of this detailed simulation is a large execution time.

Simple Scalar tool set

The simulators used in this work derive from the *sim-outorder* simulator of the Simple Scalar tool set version 3.0 [BAB96]. The *sim-outorder* simulator models an out-of-order, superscalar processor and can be fully parametrized; for instance, fetch width, branch prediction, decode width, reorder-buffer size, issue width, functional units, latencies, memory system,... Moreover,

the source code of this simulator is generally available, so researchers can adapt the simulator to their own.

Processor simulators in this thesis

Although *sim-outorder* simulator models the micro-architectural behaviour of an out-of-order superscalar processor, *sim-outorder* does not cover some aspects relevant to this thesis. For instance:

- *sim-outorder* combines the Issue Queue and the Reorder Buffer in the same structure: the Register Update Unit (RUU) [SoVa87]. This leads to the simulation of processor configurations with unrealistic instruction-window capacities. Moreover, some processors have separate instruction windows for integer and for floating-point instructions.

- *sim-outorder* simulates explicitly only one decode stage; more decode stages are simulated implicitly by increasing the branch misprediction latency. However, all the actions related to the decode stage are performed on the same simulated cycle.

- *sim-outorder* assumes that source registers are read before issuing the instructions using the reservation-station model. Moreover, it assumes that registers are read in a single cycle.

- There is no support for address-prediction mechanisms.

- *sim-outorder* deals with only one misprediction type: branch mispredictions. Consequently, recovery mechanisms for other types of mispredictions were not provided.

- *sim-outorder* assumes perfect cache-hit/miss prediction for load instructions, because the latency of a load instruction is unrealistically assumed to be known in its issue cycle.

Consequently, to evaluate the proposals presented in this thesis, the *sim-outorder* simulator has been heavily modified to model them.

Processor-simulation validation

The validation of a cycle-by-cycle simulator is a difficult task due to the complex relationships between processor components. Several strategies have been used to validate the simulators used in this work:

- The benchmark results obtained during the simulation of the benchmarks have been compared to the results obtained in a native execution of the benchmarks.

- Consistence checkings are performed through all the simulated pipeline stages.

- Several simulation actions (for instance issue-queue occupancy,...) were performed redundantly in order to detect incoherencies.

- Source-level debugging was used to perform step-by-step executions of the modified simulator-code fragments.
- A revision control system (RCS) was used to maintain the different versions of the simulators.

A.2 Benchmarks

A.2.1 Benchmark description

All the evaluations presented in this thesis are based on the SPEC95 benchmark suite [SPEC95]. This suite is composed of eight integer programs and ten floating-point programs. Table A.1 lists these benchmarks and describes their function.

Benchmark		Description
Integer Benchmarks (SPEC95-INT)	go	*go* game player
	m88ksim	Motorola 88000 simulator
	gcc	C compiler
	compress	File compression and decompression
	li	Lisp interpreter
	ijpeg	Graphic compression
	perl	Perl interpreter
	vortex	Object-oriented data base
Floating-point Benchmarks (SPEC95-FP)	tomcatv	Mesh-generation program
	swim	Solver of shallow water equations
	su2cor	Quantum-physics application
	hydro2d	Solver of hydrodynamical equations
	mgrid	Multigrid solver
	applu	Solver of partial differential equations
	turb3d	Turbulence simulator
	appsi	Atmospheric model
	fpppp	Quantum-chemistry application
	wave5	Solver of Maxell's equations

Table A.1 SPEC95 benchmark description

These benchmarks have been compiled on an AXP-Alpha 21264 using the highest optimization level of the native compiler of the machine. Table A.2 summarizes the execution characteristics of these benchmarks using their reference input data set (the input data set for benchmarks *gcc* and *perl* is composed of only one of the reference input files). The table shows

the number of user-level committed instructions, the percentage of load instructions (loads to zeroed registers have been ignored) and the percentage of store instructions.

Benchmark	Input Data Set	Instructions ($\times 10^9$)	%Loads	%Stores
go	reference	29.7	28.80	9.97
m88ksim	reference	78.7	24.64	10.38
gcc	cp-decl.i	0.4	26.84	11.55
compress	reference	66.9	18.72	12.44
li	reference	43.1	25.47	15.82
ijpeg	reference	39.7	18.08	7.33
perl	primes.pl	51.5	22.50	7.19
vortex	reference	89.3	25.60	16.38
tomcatv	reference	29.8	28.19	7.82
swim	reference	37.4	25.93	8.24
su2cor	reference	40.7	17.99	8.80
hydro2d	reference	48.9	23.88	6.96
mgrid	reference	81.1	43.19	1.83
applu	reference	50.1	24.09	9.14
turb3d	reference	88.8	24.07	16.61
apsi	reference	33.0	19.23	10.51
fpppp	reference	136.4	32.07	11.82
wave5	reference	35.3	22.35	12.68

Table A.2 SPEC95 benchmark characterization

Our evaluation tools based on instrumented executables run all these benchmarks until completion. However, due to simulation-time constrains, our cycle-by-cycle simulations focus on an execution interval specific for each benchmark. Next section describes how these intervals have been selected.

A.2.2 Simulation intervals for cycle-by-cycle simulations

A cycle-by-cycle simulation of the complete execution of the SPEC-95 benchmarks with their reference input data set is not feasible. Even when 500.000 instructions per second are simulated, a complete simulation of *compress* takes 37 hours. Most computer-architecture researchers must deal with this problem and apply several solutions.

Lee et al. [LWY00] decided to use of much smaller input data sets in order to simulate the whole execution of the benchmarks. This may cause an imbalance between the initialization phase and the steady phase of the benchmark, and could lead to wrong conclusions.

Other studies extrapolate the program behaviour from the behaviour in a selected execution interval that uses the reference input data set; that is, sample a full-length run with the reference input. However, there are several approaches that follow this idea:

- Sato [Sato00] simulates only the first 100 million of committed instructions of the benchmarks. These results may not be extrapolated because the behaviour of the initialization phase of a benchmark can be different from the behaviour of the steady phase.

- Berkeman et al. [BJR+99] use a trace of 30 million of consecutive instructions (including kernel activity). They affirm that these traces are representative of the entire execution, although the authors do not explain how the traces have been obtained.

- Other works skip a certain amount of the initial instructions of the execution; after skipping them, they simulate a limited amount of instructions. Most of these papers select an arbitrary interval that is the same for all benchmarks ([GoGo97a] skips the first 100-million instructions and simulates the following 50-million instructions). Consequently, their results are not clearly extrapolable.

- Reinman and Calder [ReCa98] skip an initial number of instructions that depend on the simulated benchmark, but they do not justify these amounts. They then simulate 100-million instructions. Chrysos and Emer [ChEm98] also skip the initialization code, but they do not specify how it is measures. They then simulate until the retirement of 100-million instructions.

- Skadron et al. [SAMC98] select the simulated interval by evaluating the branch-misprediction rate for every 50-million of committed instructions and choosing an interval with a representative behaviour.

This work follows the same approach as [SAMC98]. In order to select a significant simulation interval, the evolution of a performance metric during program execution has been evaluated. This metric has then been plotted every 500 millions of committed instructions. Finally, a representative simulation interval has been selected.

Integer benchmarks

The metric chosen is related to the topic covered by this thesis: the predictability captured by a last-address predictor (Section 2.4.1). Figure A.1 shows the evolution of captured predictability (vertical axis) during execution time (horizontal axis). Horizontal scales differ because every graph is related to the complete execution of a benchmark, and every benchmark executes a different number of instructions (Table A.2). Due to the execution length of benchmark *gcc*, it has been sampled every 50-million instructions.

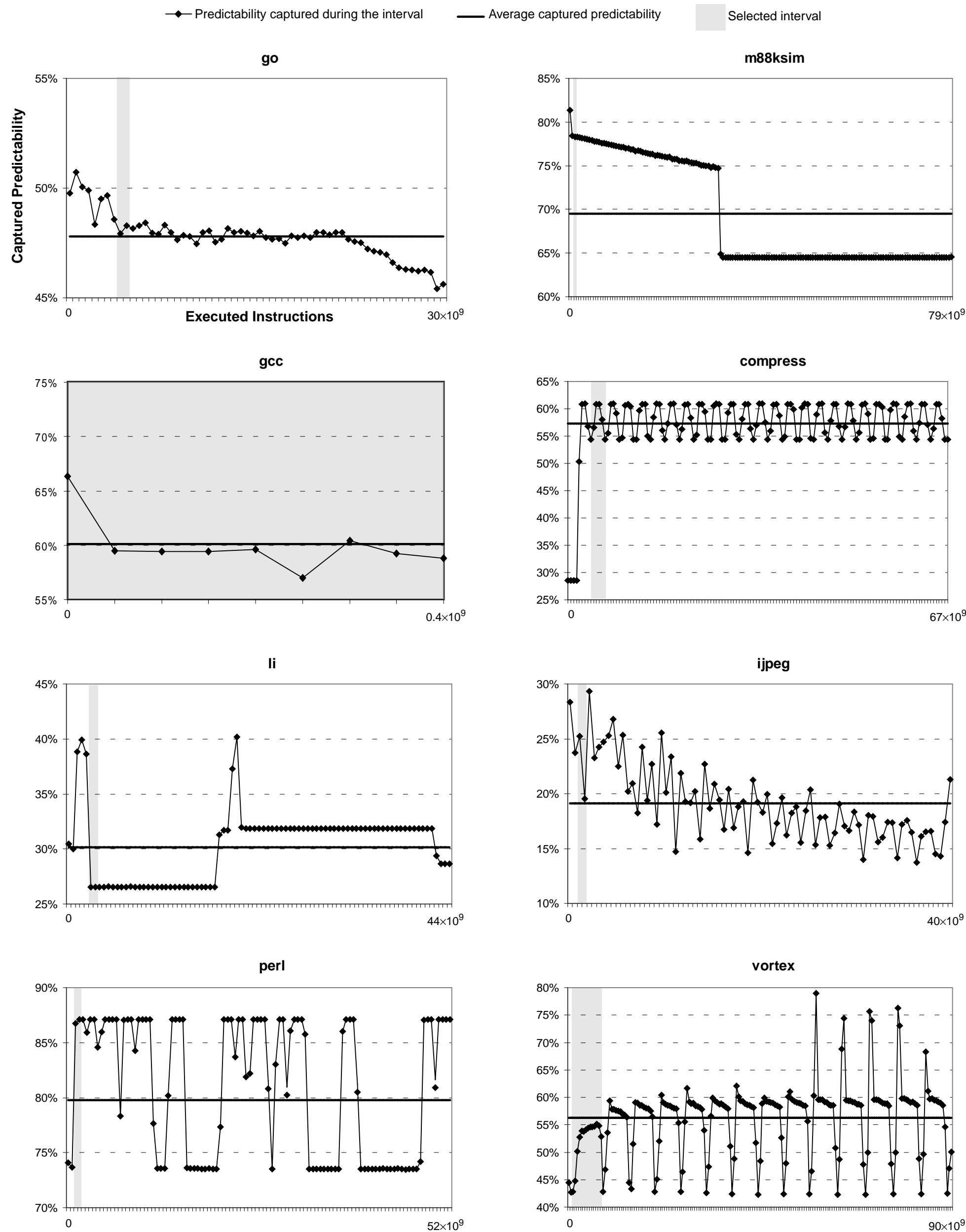


Figure A.1 Evolution of the predictability captured by a last-address predictor during the execution of the SPEC95-INT benchmarks

The regular behaviour of some benchmarks can be explained from their input data sets.

- *m88ksim* performs two micro-architecture simulations where the first lasts half the time of the second.

- *compress* performs a main loop with 25 iterations. Every period of the graph is related to an iteration.

- *vortex* performs a main loop with 14 iterations. Every period of the graph is related to an iteration.

From these graphs, we have selected the simulation intervals related to the steady phases of the benchmarks (shaded zones of the graphs). Table A.3 presents the number of initial instructions skipped (initialization phase), and the number of instructions executed after the initialization phase. These intervals had been used in all the cycle-by-cycle simulations of this work.

Benchmark	Instructions skipped ($\times 10^6$)	Instructions simulated ($\times 10^6$)
go	4.000	1.000
m88ksim	1.000	1.000
gcc	0	all
compress	4.500	2.500
li	2.500	1.000
ijpeg	1.000	1.000
perl	1.000	1.000
vortex	1.000	7.000

Table A.3 Selected simulation intervals for SPEC95-INT benchmarks

Floating point benchmarks

For floating-point benchmarks last-address predictors perform poorly. Consequently we use a different address predictor to select the simulation interval: the stride predictor (Section 2.3.2).

Figure A.2 shows the evolution of captured predictability (vertical axis) during execution time (horizontal axis). Horizontal scales differ because every graph is related to the complete execution of a benchmark, and every benchmark executes a different number of instructions (Table A.2).

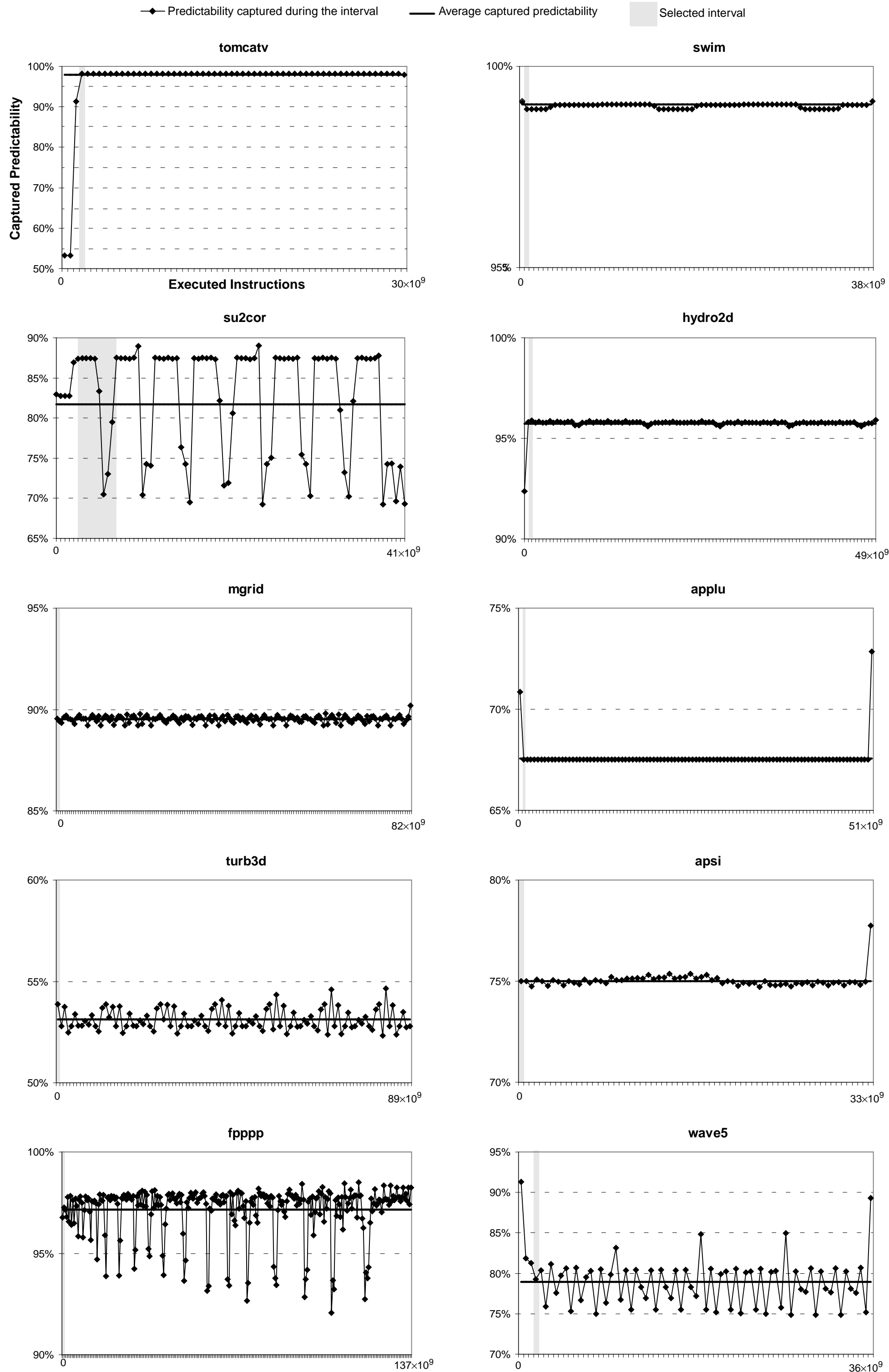


Figure A.2 Evolution of the predictability captured by a stride predictor during the execution of the SPEC95-FP benchmarks

We may observe that floating-point benchmarks exhibit more regular behaviour than integer benchmarks.

From these graphs, we have selected the simulation interval related to the steady phases of each benchmark (shaded zones of the graphs). Table A.4 presents the number of initial instructions skipped (initialization phase), and the number of instructions executed after the initialization phase. These intervals have been used in all the cycle-by-cycle simulations of this work.

Benchmark	Instructions skipped ($\times 10^6$)	Instructions simulated ($\times 10^6$)
tomcatv	1.500	500
swim	500	500
su2cor	2.000	5.000
hydro2d	500	500
mgrid	0	500
applu	500	500
turb3d	0	500
apsi	0	500
fpppp	0	500
wave5	1.500	500

Table A.4 Selected simulation intervals for SPEC95-FP benchmarks

In order to determine the representative execution intervals of the benchmarks, a more in-depth analysis should be carried out. However, such an analysis is beyond the scope of this thesis.

REFERENCES

[AEK+01] P.S. Ahuja, J. Emer, A. Klauser and S.S. Mukherjee. (2001). Performance Potential of Effective Address Prediction of Load Instructions. In *Workshop on Memory Performance Issues of the 28th International Symposium on Computer Architecture*.

[AHKB00] V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger. (2000). Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, pp. 248-259.

[AkDr98] H. Akkary and M. Discroll. (1998). A Dynamic Multithreading Processor. In *Proceedings of the 31th International Symposium on Microarchitecture*, pp. 226-236.

[Alph99] Alpha 21264 MicroProcessor Data Sheet. (1999). Compaq Computer Coorporation (EC-R4CFA-TE).

[APS95] T.M. Austin, D.N. Pnevmatikos and G.S. Sohi. (1995). Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 369-380.

[AuSo95] T.M. Austin and G.S. Sohi. (1995). Zero-Cycle Load: Microarchitecture Support for Reducing Load Latency. In *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 82-92.

[Bela66] L.A. Belady. (1966). A Study of Replacement Algorithms for Virtual Storage Computers. In *IBM Systems Journal*, Vol. 5 (2), pp. 78-101.

[BMP+98] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai and J.P. Shen. (1998). Load Execution Latency Reduction. In *Proceedings of the 12th International Conference on Supercomputing*, pp. 29-36.

[BAB96] D. Burger, T.M. Austin and S. Bennet. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison. July 1996.

[Bhan96] D. P. Bhandarkar. (1996). Alpha, implementations and architecture. Digital Press

[BJR+99] M. Bekerman, S. Jourdan, R.Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz and U. Weiser. (1999). Correlated Load-Address Predictors. In *Proceedings of the International Symposium on Computer Architecture*, pp. 54-63.

[BrMa99] D. Brooks and M. Martoniosi. (1999). Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 13-22.

[BSP01] M. Brown, J. Stark and Y. Patt. (2001). Select-Free Instruction Scheduling Logic. In *Proceedings of the 34th International Symposium on Microarchitecture*.

[BTME02] E. Borch, E. Tune, S. Manne and J. Emmer. (2002). Loose Loops Sink Chips. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

[CaGo00] R. Canal and A. González. (2000). A Low-Complexity Issue Logic. In *Proceedings of the 14th International Conference on Supercomputing*, pp. 327-335.

[Carm00] D. Carmean. (2000). Inside the Pentium 4 Processor Microarchitecture. Intel Developer Forum.

[ChBa95] T.F. Chen, J.L. Baer. (1995). Effective Hardware-Based Data Prefetching for High-Performance Processors. In IEEE Transactions on Computers 44 (5), pp. 609-623.

[ChEm98] G. Z. Chrysos and J. S. Emer. (1998) Memory Dependence Prediction using Store Sets. In *Proceedings of the International Symposium on Computer Architecture*, pp. 142-153.

[ChPu97] M.J. Charney and T.R. Puzac. (1997). Prefetching and memory system behaviour of the SPEC95 benchmark suite. In IBM Journal of Reseach and Development 41(3), pp. ????

[CoLl92] J. Cortadella and J.M. Llabería. (1992). Evaluation of A+B=K Conditions without Carry Propagation. In *IEEE Transactions on Computers*, 41(11), pp. 1484-1488.

[CRT99] B. Calder, G. Reinman and D.M. Tullsen. (1999). Selective Value Prediction. In *Proceedings of the 26tn International Symposium on Computer Architecture*, pp. 64-74.

[Dief99] K. Diefendorff. (1999). Hal Makes Sparcs Fly. In Microprocessor Report 13(15), pp 5-13.

[DrHo98] K. Driesen and U. Holzle. (1998). The cascaded predictor: Economical and Adaptative Branch Target Prediction. In *Proceedings of the 31th International Symposium on Microarchitecture*, pp. 249-258.

[EdMu98] A.N. Eden and T. Mudge. (1998). The YAGS Branch Predictor Scheme. In *Proceedings of the 31th International Symposium on Microarchitecture*, pp. ?????

[EiVa93] R.J. Eickemeyer and S. Vassiliadis. (1993). A load-instructions unit for pipelined processors. In *IBM Journal of Research and Development* 37 (4), pp. 547-564.

[EuSr94] A. Eustace and A. Srivastava. ATOM: A Flexible interface for Building High Performance Program Analysis Tools. WRL Technical Note TN-44, Digital Research Laboratory, Palo Alto. July 1994.

[FaFi98] J.A. Farrell and T.C. Fischer. (1998). Issue Logic for a 600 MHz Out-of-Order Execution Microprocessor. *IEEE Journal of Solid-State Circuits*, Vol 33(5), pp 707-712.

[Fagi95] B. Fagin. (1995). Partial Resolution in Branch Target Buffers. In *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 193-198.

[FaPa91] M. Farrens and A. Park. (1991). Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 128-137.

[Gabb96] F. Gabbay. (1996). Speculative Execution based on Value Prediction. Electrical Engineering Department (Israel Institute of Technology). EE-TR-1080

[GaMe97] F. Gabbay and A. Mendelson. (1997). Can Program Profiling Support Value Prediction? In *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 270-280.

[GoGo97a] J. González and A. González. (1997). Memory Address Prediction for Data Speculation. In *Proceedings of the EuroPar Conference*, pp 1084-1091.

[GoGo97b] J. González and A. González. (1997). Speculative Execution via Address Prediction and Data Prefetching. In *Proceedings of the 11th International Conference on Supercomputing*, pp. 196-203.

[GVD01] B. Goeman, H. Vandierendonck y K. De Bosschere. (2001). Diferential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In *The Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 207-216.

[Half96] T.R. Halfhill. (1996) AMD K6 Takes on Intel P6. Byte, Vol. 1, pp. 67-72.

[HoLa99] T. Horel and G. Lauterbach. (1999). UltraSparc III: Designing Third-Generation 64-Bit Performance. *IEEE MICRO*, Vol. 19, pp. 73-85.

[Hunt95] D. Hunt. (1995). Advanced Performance Features of the 64-bit PA-8000. In *Proceedings of the COMPCON*, pp. 123-128.

[JaMu98] B. Jacob and T. Mudge. (1998). Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, Vol 18(4), pp. 60-75.

[JoGr97] D. Joseph and D. Grunkwald. (1997). Prefetching using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 252-263.

[Kess99] R.E. Kessler. (1999). The Alpha 21264 Microprocesor. *IEEE MICRO*, Vol 19 (2), pp. 24-36.

[LiSh96] M.H. Lipasti and J.P. Shen. (1996). Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 226-237

[LWS96] M.H. Lipasti, C.B. Wilkerson and J.P. Shen. (1996). Value Locality and Load Value Prediction. In *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147.

[LWY00] S. Lee, Y. Wang and P. Yew. (2000). Decoupled Value Prediction on Trace Processors. In *Proceedings of the Symposium on High-Performance Computer Architecture*, pp. 231-240.

[Matz98] D. Matzke. (1998). Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, Vol. 30(9), pp. 37-39.

[McFa93] S. McFarling. (1993). *Combining Branch Predictors*. Western Research Laboratory (Digital). WRL-TN-36

[MIPS] MIPS. MIPS R4000 Microprocessors User's manual

[MiSe01] P. Michaud and A. Seznec. (2001). Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 27-36.

[MLC97] E. Musoll, T. Lang and J. Cortadella. (1997). Exploiting the locality of memory references to reduce the address bus energy. In *Proceedings of the International Symposium on Low Power Design and Electronics*, pp. 202-207.

[MLO98] E. Morancho, J.M. Llabería and À. Olivé. (1998). Split Last Address Predictor. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pp. 230-239.

[MLO99] E. Morancho, J.M. Llabería and À. Olivé. (1999). Looking at History to Filter Allocations in Predictions Tables. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pp. 314-319.

[MLO00] E. Morancho, J.M. Llabería and À. Olivé. (2000). Two-Level Address Storage and Address Prediction. In *Proceedings of the 6th International Euro-Par Conference (EuroPar'2000)*, pp. 960-964.

[MLO01] E. Morancho, J.M. Llabería and À. Olivé. (2001). Recovery Mechanism for Latency Misprediction. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, pp. 118-128.

[Mosh98] A.I. Moshovos. (1998). *Memory Dependence Prediction*. PhD Thesis, University of Wisconsin - Madison.

[PJS97] S. Palacharla, N.P. Jouppi and J.E. Smith. (1997). Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture*, pp 206-218.

[PMT99] L. Piñuel, R. Moreno and F. Tirado. (1999). Cost-effective and Accurate Hybrid Value Predictor. In *Proceedings of the X Jornadas de Paralelismo*, pp. 125-130.

[ReCa98] G. Reinman and B. Calder. (1998). Predictive Techniques for Aggfressive Load Speculation. In *Proceedings of the 31st International Symposiun on Microarchitecture*, pp. 127-137.

[RFK+98] B. Rychlik, J.W. Faistl, B.P. Krug, A.Y. Kurland, J.J. Sung, M. Velev and J.P. Shen. (1998). *Efficient and Accurate Value Prediction Using Dynamic Classification*. Carneige Mellon University, CMμART-1998-01.

[RFKS98] B. Rychlik, J.W. Faistl, B.P. Krug and J.P. Shen. (1998). Efficacy and performance impact of value prediction. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 148-154.

[Rote99] E. Rotenberg. (1999). *Trace Processors: Exploiting Hierarchy and Speculation*. PhD Thesis, University of Wisconsin-Madison.

[SaAr00] T. Sato and I. Arita. Table Size Reduction dor Data Value Predictors by Exploiting Narrow Width Values. In *Proceedings of the International Conference on Supercomputing*, pp.196-205.

[SAMC98] K. Skadron, P.S. Ahuja, M. Martonosi and D.W. Clark. A quantitative evaluation of branch prediction's impact on instruction-window size and cache size. Technical Report TR-578-98, Princenton Department of Computer Science, April 1998.

[SaSm97] Y. Sazeides and J.E. Smith. (1997). The Predictability of Data Values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248-258.

[SaSm98] Y. Sazeides and J.E. Smith. (1998). Modeling Program Predictability. In *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 73-84.

[Sato98] T. Sato. (1998). Data Dependence Speculation using Data Address Prediction and its Enhancement with Instruction Reuse. In *Proceedings of the 24th Euromicro Conference*, pp. 285-292.

[Sato99] T. Sato. (1999). A Simulation Study of Pipelining and Decoupling a Dynamic Instruction Scheduling Mechanism. In *Proceedings of the EUROMICRO Conference, Workshop on Digital System Design: Architectures, Methods and Tools*, pp. 178-185.

[Sato00] T. Sato. (2000). A Simulation Study of Combining Load Value and Address Predictions In *International Journal of High Speed Computing*, vol 10 (3), pp. 301-325.

[Saze99] Y. Sazeides. (1999). *An Analysis of Value Predictability and its application to a Superscalar Processor*. PhD Thesis, University of Wisconsin-Madison.

[SBP00] J.Stark, M.D. Brown and Y.N. Patt. (2000). On Pipelining Dynamic Instruction Scheduling Logic. In *Proceedings of the International Symposium on Microarchitecture*, pp. 57-66.

[Sezn94] A. Seznec. (1994). Decoupled sectored caches: reconciliating low tag volume and low miss rate. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 384-393.

[Sezn96] A. Seznec. (1996). Don't use the page number, but a pointer to it. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 104-113.

[Sima00] D. Sima. (2000). The Design Space of Register Renaming Techniques. In *IEEE Micro*, Vol 20(5), pp. 70-83.

[Smit81] J.E. Smith. (1981). A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148.

[Smit82] A.J. Smith. (1982). Cache Memories. ACM Computing Surveys, vol 14 (3), pp. 473-530.

[SoSo98] A. Sodani adn G.S. Sohi. (1998). Understanding the Differences between Value Prediction and Instruction Reuse. In *Proceedings of the 31th International Symposim on Microarchitecture*.

[SoVa87] G.S. Sohi and S. Vajapeyam. (1987). Instruction Issue Logic for Hihg-Performance, Interruptable Pipelined Processors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 27-34.

[SPEC95] The Standard Performance Evaluation Corporation. *http://www.specbench.org/osg/cpu95/news/cpu95descr.html*, September 1995.

[SrWa94] A. Srivastava and D.W. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. WRL Research Report 94-1, Digital Research Laboratory, Palo Alto. February 1994.

[ThFr01] R. Thomas and M. Franklin. (2001). Using Dataflow Based Context for Accurate Value Prediction. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, pp. 107-117.

[Toma67] R.M. Tomasulo. (1967). An Efficient Algorithm for Exploting Multiple Arithmetic Units. I*BM Journal of Research and Development*, vol 11, pp 25-33.

[TyAu97] G.S. Tyson and T.M. Austin. (1997). Improving the Accuracy and Performance of Memory Comunication Through Renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp 218-227.

[WaFr97] K. Wang and M. Franklin. (1997). Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 281-290.

[WSY97] H. Wang, T. Sung and Q. Yang. (1997). Minimizing Area Cost of On-Chip Cache Memories by Caching Address Tags. *IEEE Transactions on Computers*, 46 (11): 1187-1201.

[Yeag96] K.C. Yeager. (1996). The MIPS R10000 Superscalar Microprocessor. *IEEE MICRO*, Vol. 16(2), pp 28-41.

[YePa92] T.Y. Yeh and Y.N. Patt. (1992). Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 124-134.

[ZZY97] C. Zhang, X. Zhang and Y. Yan. (1997). Two Fast and High-Associativity Cache Schemes. *IEEE Micro*, Vol 17(5), pp. 40-49.