“Lucian Blaga” University of Sibiu
“Hermann Oberth” Engineering Faculty
Computer Science Department

Advanced Prediction Methods Integrated Into Speculative Computer Architectures

PhD Thesis

Author:
Árpád GELLÉRT, MSc

PhD Supervisor:
Professor Lucian N. VINȚAN, PhD

PhD Co-supervisor:
Professor Theo UNGERER, PhD

SIBIU, 2008

Universitatea „Lucian Blaga” din Sibiu
Facultatea de inginerie „Hermann Oberth”
Catedra de Calculatoare şi Automatizări

Metode avansate de predicție integrate în arhitecturi cu procesări speculative

Teză de doctorat

Autor:
Asist. univ. ing. Árpád GELLÉRT, MSc

Conducători ştiințifici:
Prof. univ. dr. ing. Lucian N. VINȚAN
Prof. univ. dr. Theo UNGERER (cotutelă)

SIBIU, 2008

Mulțumiri

În primul rând doresc să mulțumesc conducătorului meu de doctorat prof. univ. dr. ing. Lucian VINȚAN, pentru încrederea acordată, pentru deosebita responsabilitate a coordonării sale ştiinţifice, pentru discuțiile profesionale extrem de stimulative pe care le-am avut şi pentru întreg sprijinul acordat cu multă generozitate. De asemenea, mulțumesc conducătorului de doctorat prin cotutelă, prof. dr. Theo UNGERER de la Universitatea din Augsburg, Germania, pentru discuțiile utile şi pentru sprijinul oferit. Țin să mulțumesc colegului meu dr. Adrian FLOREA pentru ajutorul acordat şi pentru sfaturile sale deosebit de utile. Mulțumesc şi d-lui Dr. Colin EGAN de la Universitatea din Hertfordshire pentru colaborarea sa precum şi pentru observațiile şi sugestiile făcute. De asemenea, doresc să mulțumesc colegilor de la Catedra de Calculatoare şi Automatizări din cadrul Universității „Lucian Blaga" pentru sprijinul acordat şi pentru climatul favorabil asigurat. Nu în ultimul rând, doresc să mulțumesc familiei pentru sprijinul necontenit şi pentru răbdarea de care a dat dovadă în această perioadă.

Cercetările prezentate în această lucrare au fost parțial susținute prin granturile CNCSIS TD-248/2007-2008 şi A-39/2007-2008 respectiv prin proiectul HPC-EUROPA (RII3-CT-2003-506079) din cadrul programului FP6 „Structuring the European Research Area", cu suportul Comunității Europene (Research Infrastructure Action).

"Nu ştim cum să prezicem cu certitudine
ce se va întâmpla într-o situaţie dată,
singurul lucru care poate fi estimat fiind
probabilitatea diferitelor evenimente"

Richard Feynman

Rezumat

Paralelismul la nivelul instrucţiunilor este limitat de dependenţele existente între instrucţiuni, iar pentru eliminarea lor microprocesoarele moderne, unele din ele prezentate în Capitolul 2, folosesc tehnici speculative. Principalul obiectiv al acestei teze îl reprezintă creşterea performanţelor unor microarhitecturi superscalare şi SMT (Simultaneous Multithreading) prin tehnici anticipatorii dinamice precum predicţia branch-urilor, predicţia valorilor şi reutilizarea instrucţiunilor. Această lucrare aduce contribuţii originale în identificarea branch-urilor dificile şi îmbunătăţirea predictibilităţii lor, în caracterizarea comportamentului acestora din punct de vedere al gradului de aleatorism, respectiv în dezvoltarea unor tehnici de reutilizare şi predicţie selective a valorilor instrucţiunilor în cadrul arhitecturilor superscalare şi al celor cu fire multiple de execuţie.

Instrucţiunile de ramificaţie, generate de construcţii de limbaj de tipul *if*, *switch*, *for*, *while* etc., reprezintă un obstacol major în exploatarea paralelismului la nivelul instrucţiunilor (ILP). Rezultate statistice bazate pe simulări laborioase pe benchmark-uri reprezentative arată că o instrucţiune de ramificaţie apare la fiecare 5 – 8 instrucţiuni executate, ceea ce înseamnă că rata de aducere a instrucţiunilor este limitată la cel mult 8, aducerea simultană a mai multor instrucţiuni fiind inutilă. Pentru creşterea gradului de paralelism la nivelul instrucţiunilor procesoarele moderne folosesc predictoare markoviene, neuronale, bayesiene, bazate pe arbori de decizie sau pe algoritmi de tipul *support vector machine* etc., simplificate pentru a putea fi implementate în *hardware*. Prin predicţia dinamică a branch-urilor pot fi procesate mai multe *basic block*-uri în paralel. Pentru îmbunătăţirea performanţei instrucţiunile de ramificaţie trebuie identificate şi atât direcţia cât şi adresa de salt trebuie predicţionate corect. Factorul de superscalaritate al microprocesoarelor devine din ce în ce mai mare, permiţând rate de procesare mai agresive pentru îmbunătăţirea performanţelor. Procesoarele cu factor mare de superscalaritate pot fi afectate din punct de vedere al performanţelor în cazul predicţiilor greşite când contextul CPU trebuie refăcut şi instrucţiunile trebuie reexecutate pe căile corecte. De aceea, performanţa globală depinde foarte mult de acurateţea predictorului de salturi. Având în vedere faptul că numărul de instrucţiuni executate per ciclu creşte neliniar cu acurateţea predicţiei, este foarte importantă îmbunătăţirea acurateţii predictoarelor actuale. Calitatea unui model de predicţie este dependentă de calitatea informaţiei disponibile. Este foarte importantă alegerea caracteristicilor pe baza cărora se generează predicţia. Marea majoritate a predictoarelor de salturi se bazează pe mai multe informaţii de intrare (adresa instrucţiunii de salt, istoria locală, istoria globală, informaţii de cale etc.) fără să ţină cont de cauzele reale (ex. instrucţiuni de salt nepolarizate) care produc o acurateţe scăzută şi implicit performanţe mai slabe.

În Capitolul 3 am demonstrat că o instrucţiune de ramificaţie într-un anumit context dinamic al informaţiei de predicţie este greu de prezis dacă este nepolarizată în acel context, oscilând între *taken* (saltul se face) respectiv *not taken* (saltul nu se face) într-un mod nedeterminist, entropic (comportament dezordonat). Cu alte cuvinte, o instrucţiune de ramificaţie dinamică este nepredictibilă cu o anumită informaţie de predicţie, dacă este nepolarizată în contextul dinamic considerat şi comportamentul în acel context nu poate fi modelat prin procese stohastice Markov. Am identificat aceste branch-uri dificile şi am încercat îmbunătăţirea predictibilităţii lor prin extinderea informaţiei de predicţie. Pe baza unor simulări laborioase am

arătat că procentajul branch-urilor nepolarizate, pe istoria locală şi globală, este semnificativ: în medie între 6% şi 24% pe benchmark-urile SPEC 2000, în funcţie de contextul de predicţie folosit şi de lungimea acestuia (16-28 biţi). De asemenea, cercetările noastre au arătat că adăugarea informaţiei de cale (formată din PC-urile ultimelor *k* branch-uri) la cele clasice de istorie locală/globală determină o polarizare mai ridicată doar în cazul folosirii unor contexte de istorie locală/globală scurte (sub 16 biţi). Istoriile locale şi globale suficient de lungi aproximează foarte bine informaţia de cale.

În Capitolul 4 am arătat că pentru anumite instrucţiuni de ramificaţie, informaţiile de predicţie limitate (istorie locală, istorie globală şi calea spre branch-ul supus predicţiei) folosite de predictoarele actuale nu sunt întotdeauna suficient de relevante şi, din această cauză, ele nu pot fi predicţionate cu acurateţe. Acurateţea cea mai ridicată pe branch-urile nepolarizate, de doar 77,30%, am obţinut-o cu *piecewise linear branch predictor* [Jim05]. De aceea, este deosebit de importantă găsirea unor informaţii relevante care determină comportamentul instrucţiunilor de ramificaţie, pentru a fi utilizate de predictoare mai eficiente. Am dezvoltat diferite predictoare de valori Markoviene care folosesc istoria comprimată a precedentelor condiţii de salt, ale cărei elemente pot fi -1, 0 sau 1, în funcţie de semnul diferenţei dintre operanzi. Nici aceste predictoare puternice, capabile să exploateze corelaţia dintre comportamentul branch-ului şi istoria condiţiilor, n-au reuşit să îmbunătăţească predictibilitatea acestor branch-uri dificile. De asemenea, am îmbunătăţit predictoare convenţionale (GAg, PAg) şi neuronale, prin utilizarea condiţiei de salt precedente – Previous Branch Condition (PBC) – sub forma unei diferenţe pe 32 de biţi dintre operanzi. Chiar şi predictorul *piecewise linear branch predictor* îmbunătăţit, cel mai performant pe branch-urile nepolarizate, obţine o acurateţe modestă de 78,3% pe branch-urile nepolarizate, în timp ce acurateţea globală a predicţiei este de 95,45%. Aşadar, branch-urile nepolarizate sunt caracterizate de acurateţi de predicţie scăzute, indiferent de informaţia de predicţie folosită, reprezentând astfel o limitare fundamentală în domeniul predicţiei branch-urilor. Astfel, creşterea acurateţii de predicţie a acestor instrucţiuni de ramificaţie nepolarizate constituie o problemă deschisă, deoarece fiecare procent de astfel de instrucţiuni reduce decisiv acurateţea predicţiei şi implicit performanţa de procesare.

Pornind de la această provocare tehnică, în Capitolul 5 am realizat un studiu comparativ privind gradul de aleatorism al secvenţelor de simboluri (*taken* şi *not taken*) generate de branch-uri polarizate respectiv nepolarizate. Pe baza cercetării bibliografice efectuate, am dezvoltat patru metrici pentru caracterizarea comportamentului unui branch din punct de vedere al gradului de aleatorism: complexitatea Kolmogorov a secvenţei de program care generează branch-ul, rata de compresie, entropia discretă respectiv acurateţea de predicţie cu HMM (Hidden Markov Models) a secvenţei generate de branch. Rezultatele simulărilor efectuate pe şase benchmark-uri de numere întregi din suita SPEC 2000 arată că toate cele patru metrici de caracterizare intrinsecă a unei secvenţe binare din punct de vedere al gradului de aleatorism asociat, converg în aceeaşi direcţie. Ele sunt foarte utile arhitectului de microprocesoare întrucât arată dacă un anumit branch dinamic este sau nu este „aleator” sau „nepredictibil”. În cazul în care metricile utilizate arată în mod clar că branch-ul nu este unul intrinsec aleator, arhitectul are şanse reale de îmbunătăţire a predictorului aferent. În cazul aleatorismului ridicat, răspunsul este unul pesimist întrucât secvenţa este una intrinsec, şi deci iremediabil, aleatoare. De precizat că aleatorismul comportării acestor branch-uri este o consecinţă a complexităţii uriaşe a programelor care le generează, după cum arătăm în lucrare.

Instrucţiunile cu latenţă ridicată reprezintă o altă sursă de limitare a paralelismului la nivelul instrucţiunilor. În Capitolul 6 am arătat că 28,68% din instrucţiunile de ramificaţie (5,61% fiind chiar nepolarizate) sunt dependente de instrucţiuni cu latenţă ridicată (Load-uri critice, înmulţiri, împărţiri). Aceste dependenţe reprezintă o sursă importantă de penalităţi datorate predicţiilor greşite, afectând serios performanţa globală a procesorului. De aceea, impactul negativ al branch-urilor, în special al celor nepolarizate, asupra performanţei globale poate fi atenuat anticipând rezultatele instrucţiunilor cu latenţă ridicată. Am dezvoltat un mecanism de anticipare selectivă a valorilor instrucţiunilor cu latenţă de execuţie ridicată, care include o schemă de reutilizare pentru instrucţiunile Mul şi Div, respectiv un predictor de valori pentru instrucţiunile Load critice. Rezultatele simulărilor efectuate, au arătat creşteri de performanță (IPC) de 3,5% pe benchmark-urile de numere întregi respectiv 23,6% pe cele flotante şi o scădere importantă a consumului relativ de energie (a EDP-ului) de 6,2% respectiv 34,5%.

Tot în Capitolul 6 am arătat că există o corelație temporală între numele registrelor şi valorile memorate în acestea. De aceea am extins predicția dinamică a valorilor prin introducerea conceptului de predicție a valorilor centrată pe contextul CPU (registre) şi nu pe instrucțiuni. Practic se prezice valoarea registrului destinație curent bazat pe analiza valorilor anterioare ale acestuia. Localitățile valorilor obținute pe anumite registre ale arhitecturii MIPS au fost remarcabile conducând la concluzia că predicția valorilor poate fi aplicată cu succes cel puțin centrat pe aceste registre favorabile, prin ataşarea câte unui predictor la nivelul acestora. Astfel se reduce semnificativ numărul predictoarelor şi scade corespunzător complexitatea şi consumul de putere statică/dinamică. Rezultatele evaluărilor au arătat că predictorul hibrid cu prioritizare dinamică, format dintr-un predictor adaptiv pe două niveluri şi unul incremental, exploatează cel mai eficient această corelație, depăşind chiar şi hibridul mult mai complex format dintr-un predictor PPM (Prediction by Partial Matching) şi unul incremental.

După ce am arătat utilitatea anticipării selective a instrucțiunilor cu latență ridicată într-o arhitectură superscalară, în Capitolul 7 am analizat eficiența acestor metode şi într-o arhitectură SMT, focalizându-ne pe aceleaşi instrucțiuni: Mul şi Div respectiv Load-uri critice. Rezultatele au arătat îmbunătățiri IPC pe toate configurațiile SMT evaluate. Cu cât numărul de fire este mai mare, cu atât creşterea de performanță devine însă tot mai puțin semnificativă, datorită exploatării tot mai eficiente a unităților de execuție partajate de către procesorul SMT. Plastic spus, cu motorul SMT mergând în plin, sporul de performanță aferent tehnicilor anticipative implementate adițional devine mai mic. Cele mai bune performanțe medii, de 2,29 IPC pe benchmark-urile de numere întregi respectiv de 2,88 IPC pe cele flotante, s-au obținut cu şase fire de execuție.

În Capitolul 8 sunt trecute succint în revistă contribuțiile ştiințifice ale acestei lucrări şi sunt evidențiate câteva dintre direcțiile viitoare de cercetare.

“Lucian Blaga” University of Sibiu
“Hermann Oberth” Engineering Faculty
Computer Science Department

Advanced Prediction Methods Integrated Into Speculative Computer Architectures

PhD Thesis

Author:
Árpád GELLÉRT, MSc

PhD Supervisor:
Professor Lucian N. VINȚAN, PhD

PhD Co-supervisor:
Professor Theo UNGERER, PhD

SIBIU, 2008

Acknowledgments

First of all I express my sincere consideration and deep gratitude to my PhD supervisor Professor Lucian VINȚAN for his responsible and valuable scientific coordination, for providing stimulating discussions focused on my PhD work and for his generous support. My full recognition to my PhD co-supervisor Professor Theo UNGERER from the University of Augsburg (Germany) for the useful discussions and for all his various support. I express my gratitude to Dr. Adrian FLOREA for his continuous help and his very useful advices. Also my gratitude to Dr. Colin EGAN from the University of Hertfordshire (UK) for his research collaboration. I am also grateful to my colleagues from the Computer Science Department of "Lucian Blaga" University, for their support and for a good working environment assured during my PhD programme. Finally, I would like to thank my family for their patience and for all they have done for me.

This work was supported in part by the Romanian Agency for Academic Research (CNCSIS) through the research grants TD-248/2007-2008 and A-39/2007-2008. It was also partially carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community – Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme.

Contents

1. Introduction — *5*

2. Speculative Computer Architectures — *7*

2.1. Speculative Architectures with Reorder Buffer — **8**

2.1.1. Speculative Dynamic Scheduling with Reorder Buffer — 8

2.1.2. The Architecture of Sim-Outorder — 10

2.1.3. Checkpoint Processing Architectures — 14

2.2. Advanced Dynamic Branch Prediction — **18**

2.3. Dynamic Value Prediction — **24**

3. Finding Difficult-to-Predict Branches — *31*

3.1. Related Work — **31**

3.2. Methodology of Identifying Unbiased Branches — **34**

3.3. An Analytical Model for Determining Relative IPC Speedup — **37**

3.4. Experimental Results — **38**

3.4.1. Pattern-Based Correlation — 39

3.4.2. Path-Based Correlation — 43

3.4.3. Evaluating Relative IPC Speedup Through an Analytical Model — 46

3.5. Summary — **47**

4. Predicting Unbiased Branches — *48*

4.1. Related Work — **48**

4.1.1. Branch Prediction Based on Data Value Information — 48

4.1.2. State-of-the-Art Branch Predictors — 52

4.2. Branch Prediction Using State-of-the-Art Predictors — **55**

4.2.1. The Perceptron-Based Branch Predictor — 55

4.2.2. The Idealized Piecewise Linear Branch Predictor — 56

4.2.3. The Frankenpredictor — 56

4.2.4. The O-GEHL Predictor — 57

4.3. Value-History-Based Branch Prediction with Markov Models — **57**

4.3.1. Local Branch Difference Predictor — 59

4.3.2. Combined Global-Local Branch Difference Predictor — 60

4.3.3. Branch Difference Prediction by Combining Multiple Partial Matches — 61

4.4. Using Previous Branch Condition as Prediction Information — **61**

4.4.1. The GAg Predictor Using Global PBC Value — 62

4.4.2. The PAg Predictor Using Local PBC Value — 62

4.4.3. The Piecewise Linear Branch Predictor Using PBC Value — 63

4.4.3.1 The Piecewise Linear Branch Predictor Using Global PBC Value — 63

4.4.3.2 The Piecewise Linear Branch Predictor Using Local PBC Value — 64

4.5. Experimental Results — **64**

4.5.1. Evaluating State-of-the-Art Branch Predictors — 65

4.5.1.1. Evaluating the Perceptron-Based Branch Predictor — 65

4.5.1.2. Evaluating the Idealized Piecewise Linear Branch Predictor — 66

4.5.1.3. Evaluating the Frankenpredictor — 67

4.5.1.4. Evaluating the O-GEHL Predictor — 68

4.5.2. Evaluating Markovian Value-History-Based Branch Predictors — 69

4.5.2.1. Evaluating Local Branch Difference Predictors — 69

4.5.2.2. Evaluating Combined Global and Local Branch Difference Predictors — 72

4.5.2.3. Branch Difference Prediction by Combining Multiple Partial Matches — 73

4.5.3. Evaluating PBC-Based Branch Predictors — 75

4.5.3.1 Evaluating the Global-PBC-Based GAg Predictor — 75

4.5.3.2 Evaluating the Local-PBC-Based PAg Predictor — 76

4.5.3.3 Evaluating the Global-PBC-Based Piecewise Linear Branch Predictor — 76

4.5.3.4 Evaluating the Local-PBC-Based Piecewise Linear Branch Predictor — 77

4.5.3.5 Prediction Accuracy Improvements with PBC — 78

4.6. Summary — **78**

5. Better Understanding Unbiased Branches Using Random Degrees 79
5.1. Related Work 79
5.1.1. What is a Random Sequence? 79
5.1.2. Prediction with Hidden Markov Models 80
5.2. Random Degree Metrics for Characterizing Unbiased Branches Behavior 81
5.2.1. Random Degree Metric Based on Hidden Markov Models 81
5.2.1.1. First Order HMMs 82
5.2.1.2. A Possible Generalization: Superior Order HMMs 86
5.2.2. Random Degree Metric Based on Discrete Entropy 90
5.2.3. Random Degree Metric Based on Compression Rate 91
5.2.4. Random Degree Metric Based on Kolmogorov Complexity 91
5.3. Evaluation Results 92
5.3.1. Random Degree Evaluation with HMMs 92
5.3.2. Random Degree Evaluation Based on Discrete Entropy 93
5.3.3. Random Degree Evaluation Based on Compression Rate 95
5.3.4. Random Degree Evaluation Based on Kolmogorov Complexity 95
5.4. Summary 97

6. Exploiting Selective Instruction Reuse and Value Prediction in a Superscalar Architecture 98
6.1. Related Work 98
6.2. Anticipating Long-Latency Instructions Results 101
6.2.1. Selective Dynamic Instruction Reuse 102
6.2.2. Selective Load Value Prediction 103
6.2.3. Simulation Methodology 104
6.2.4. Experimental Results 105
6.3. Contributions to Dynamic Value Prediction: CPU Context Prediction 108
6.3.1. Register Value Predictors 108
6.3.1.1. Last Value Predictors 109
6.3.1.2. Stride Predictors 110
6.3.1.3. Context-Based Predictors 110
6.3.1.4. Hybrid Predictors 111
6.3.2. Simulation Methodology 113
6.3.3. Experimental Results 113
6.4. Summary 118

7. Enhancing the Simultaneous Multithreading Paradigm Through Selective Instruction Reuse and Value Prediction 119
7.1. Related Work 119
7.1.1. Multithreading Architectures 119
7.1.2. Simultaneous Multithreading 121
7.2. Selective Instruction Reuse and Value Prediction in SMT Architectures 123
7.3. Simulation Methodology 125
7.4. Experimental Results 126
7.5. Summary 128

8. Conclusions and Further Work 129

References 132

Glossary 142

*"The best way to predict the future
is to invent it"*

Alan Kay

1. Introduction

The number of instructions that can be processed simultaneously in multiple instruction issue (MII) microprocessors is limited by dependencies existing between instructions. To eliminate these dependencies modern architectures, some of them presented in Chapter 2 as prerequisites for this work, rely heavily on speculation. The main goal of this thesis is to increase instruction-level parallelism (ILP) and therefore the overall performance of superscalar and multithreaded microarchitectures through advanced dynamic anticipatory techniques like branch prediction, value prediction and instruction reuse. This work brings original contributions in identifying difficult-to-predict branches and improving their predictability, in characterizing the randomness of their behavior, and in developing some selectively applied value prediction and instruction reuse methods.

Branch instructions, appearing in high level program constructs like *if*, *switch*, *for*, *while*, etc., are a major bottleneck in the exploitation of ILP, since (in general-purpose code) conditional branches occur approximately every 5 – 8 instructions [Hen03]. Therefore, almost all present-day multiple instruction issue microprocessors are using advanced branch prediction techniques in order to increase ILP. Several prediction methods have been developed based on some well-known learning algorithms (Markovian, neural, Bayesian, decision trees, support vector machine, etc.) simplified for efficient hardware implementation. Through dynamic branch prediction microprocessors are speculatively processing multiple basic blocks in parallel and therefore their ability to increase ILP is stronger. In order to improve performance, branches must be detected within the dynamic instruction stream, and both the direction taken by each branch and the branch target address must be correctly predicted. Furthermore, predictions must be completed in time to fetch instructions from the branch target address without interrupting the flow of new instructions to the processor pipeline [Vin07]. In the case of misprediction, the CPU context must be recovered and the correct paths have to be reissued. As instruction issue width and the pipeline depth of MII processors are getting higher (allowing more aggressive clock rates in order to improve the overall performance), accurate dynamic branch prediction becomes more essential [Spr02]. Very high prediction accuracy is required because an increasing number of instructions are lost before a branch misprediction can be corrected. As an example, the performance of the Pentium 4 equivalent processor degrades by 0.45% per additional misprediction cycle, and therefore the overall performance is very sensitive to branch prediction. Taking into account that the average number of instructions executed per cycle (IPC) grows non-linearly with the prediction accuracy [Yeh92], it is very important to further increase the accuracy achieved by present-day branch predictors. From a technological point of view, modern high-end processors use quite large tables for branch direction and target prediction [Sez02], and they are accessed every cycle resulting in significant energy consumption, sometimes more than 10% of the total chip power [Cha03]. Therefore, power consumption is another important constraint of all present-day branch predictors.

The quality of a prediction model is highly dependent on the quality of the available data. Especially the choice of the *features* to base the prediction on is important. The vast majority of branch prediction approaches rely on usage of a greater number of input features without taking into account the real causes (indirect jumps and calls and, especially, unbiased branches) that produce a lower accuracy and implicit lower performance. In Chapter 3 we identified difficult-to-predict branches as being unbiased branches that have a "random" dynamic behavior, and tried to improve their predictability through context length extension. In Chapter 4 we showed that present-day branch predictors cannot accurately predict these branches due to their limited prediction information (branch address, local/global branch history, path). Therefore we improved several state-of-the-art branch predictors with additional prediction information, namely the previous branch condition or even a compressed branch condition history, in order to improve their prediction accuracy. We also showed in Chapter 5 that sequences generated by unbiased branches are characterized by high random degrees.

Long-latency instructions, especially critical Loads due to their memory wall problem (the increasing gap between processor and memory speeds), represent another source of ILP limitation. A solution to reduce the number of cache misses consists in prefetching speculatively data from memory to cache. Multithreading can also reduce the effects of the memory wall by hiding memory latency through issuing into the pipelines instructions from different idle threads. Value Prediction (VP) is another technique that increases performance by eliminating true data dependency constraints. VP architectures allow data dependent instructions to issue and execute speculatively using the predicted value. The speculative executions are validated when the correct values are known. If the value was correctly predicted the critical path is reduced, otherwise the instructions executed with wrong entries must be executed again. On the other hand, dynamic instruction reuse is a non-speculative microarchitectural technique that exploits the repetition of dynamic instructions with the same input values. The main benefit of reusing long-latency instructions consists in unlocking dependent instructions.

In Chapter 6 we developed a superscalar architecture that selectively anticipates the values produced by long-latency instructions. We focused on Multiply, Division and Loads with miss in the L1 data cache. Thus, we implemented a Dynamic Instruction Reuse scheme for the Mul/Div instructions and a simple Last Value Predictor for the critical Load instructions. We also extended dynamic VP by introducing the concept of register-centric prediction instead of instruction-centric prediction. The register value prediction technique consists in predicting registers' next values based on the previously seen values. It executes the subsequent data dependent instructions using the predicted values. In Chapter 7 we evaluated a simultaneous multithreaded architecture enhanced with selective instruction reuse and value prediction to anticipate the results of long-latency instructions.

Finally, Chapter 8 concludes the thesis pointing out the original contributions and suggests some further work directions.

"When speculation has done its worst,
two and two still make four"

Samuel Johnson

2. Speculative Computer Architectures

All processors since about 1985 use pipelining in order to improve performance by overlapping the execution of instructions. A pipeline acts like an assembly line with instructions processed in phases. With simple pipelining, only one instruction at a time is introduced into the pipeline, but multiple instructions may be in different phases of execution concurrently. In the case of superscalar processors, more than one instruction at a time can be introduced into multiple pipelines to be executed simultaneously. This potential execution overlap among independent instructions is called instruction-level parallelism (ILP). There are some features of both programs and processors that limit the amount of parallelism such as structural hazards, data hazards and control stalls. In particular, to exploit instruction-level parallelism it must be determined which instructions can be executed in parallel. If two instructions are parallel and no structural hazards exist, they can be executed simultaneously in a pipeline without causing any stalls, assuming that the pipeline has sufficient resources. If two instructions are dependent they are not parallel and must be executed in order. There are three different types of dependences: data dependences, name dependences and control dependences.

An instruction is data dependent if it uses the result produced by another instruction. Data dependences can be overcome through hardware techniques (dynamic instruction reuse, value prediction) and software techniques (by reorganizing the code). When two dependent instructions are close enough to change the order of access to the operand involved in the dependence, a data hazard occurs. Considering two successive instructions *i* and *j*, a RAW (read after write) data hazard occurs when instruction *j* tries to read a source before *i* writes it, so *j* incorrectly gets the old value. A WAW (write after write) data hazard occurs when instruction *j* tries to write an operand before it is written by *i*. A WAR (write after read) data hazard occurs when instruction *j* tries to write a destination before it is read by *i*.

Name dependences occur when two instructions use the same register or memory location. Instructions involved in name dependence can be executed simultaneously or reordered if the register or memory location used by the instructions is changed so the instructions do not conflict. This renaming can be more easily done for register operands (register renaming), either statically by a compiler or dynamically by the hardware.

Control dependences are generated by branch instructions. An instruction that is control dependent on a branch cannot be executed until the branch direction is known. Control stalls can be eliminated or reduced by a variety of hardware techniques (branch prediction) and software techniques (static scheduling).

A major limitation of the simple pipelining techniques is that they all use in-order instruction issue and execution. Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed. Out-of-order execution introduces the possibility of data hazards. Hennessy and Patterson in [Hen03] explore an important technique, called dynamic scheduling, in which the hardware rearranges the instruction execution in order to reduce the stalls. In a dynamically scheduled pipeline, all instructions are dispatched in order, however, they can be stalled or bypass each other in the *issue* stage and thus execute out of order.

2.1. Speculative Architectures with Reorder Buffer

Branch prediction is a mechanism that reduces control stalls in order to improve performance in a multiple instruction issue processor. Control dependences are overcome by speculating on branch outcomes and executing dependent instructions as if the predictions were correct. Obviously it became necessary the integration of branch prediction into dynamically scheduled processors. Predicting the outcomes of conditional branches, more instructions can be fetched in parallel (a part of them are fetched speculatively from the predicted path), increasing in this way the execution window [Smi95]. The fetched instructions are analyzed for true data dependences, issued to the functional units and executed out-of-order, in parallel, based on the availability of the operands. Value prediction is another technique that speculatively forwards predicted instruction results to the dependent instructions. With speculative execution, the architectural storage cannot be updated immediately when instructions complete execution. The results must be held in a temporary status until the architectural state can be updated in sequential program order.

2.1.1. Speculative Dynamic Scheduling with Reorder Buffer

The present-day out-of-order issue superscalar microprocessor model is implemented as a speculative microarchitecture that actually fetches, issues and executes instructions based on branch prediction using Tomasulo's algorithm or closely related algorithms and a structure called *Reorder Buffer* (ROB). Figure 2.1 shows the hardware structure of the processor including the ROB.

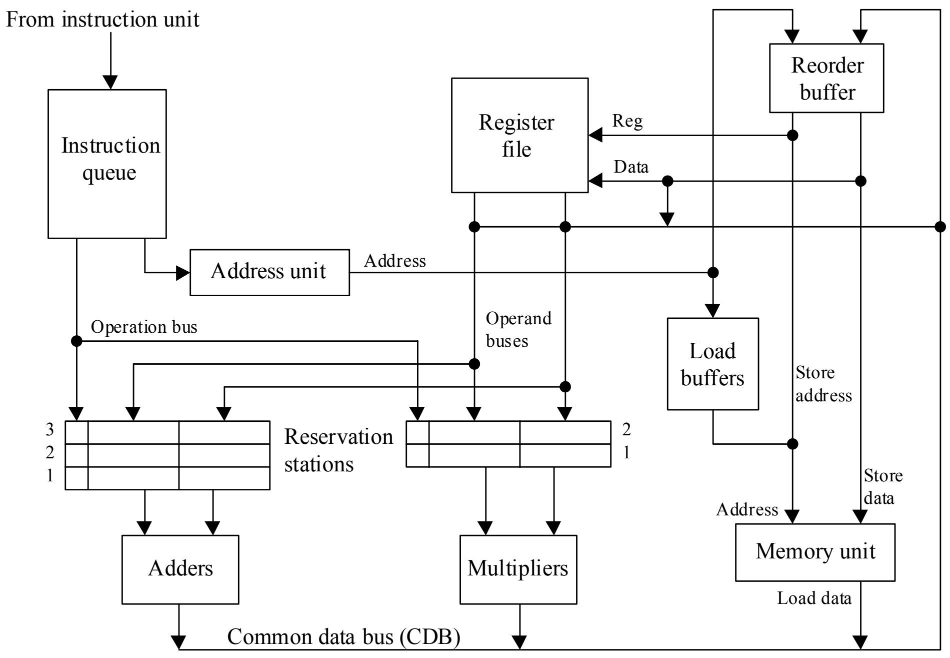


Figure 2.1. Tomasulo's architecture extended to support speculation

The hardware that implements Tomasulo's algorithm [Tom67] can be extended to support speculation, only if the bypassing of results, which is needed to execute an instruction speculatively, is separated from the completion of an instruction (that consists in updating the memory or register file). Doing this separation, an instruction bypasses its results to other instructions, without performing any CPU updates that cannot be canceled. When the instruction is no longer speculative (after its *writeback* stage), it updates the register file or memory; this phase is called instruction *commit*. Separating the bypassing of results from instruction completion makes possible avoiding imprecise exceptions in out-of-order execution, preserving in this way exception behavior. An exception is imprecise if the processor state when the exception raised is not exactly as in the case of sequential execution.

Adding this *commit* phase to the instruction execution sequence, an additional set of hardware buffers is required, which hold the results of instructions that have finished execution but have not yet committed. The reorder buffer provides the register renaming function and it is also used to pass the results of speculatively executed instructions. The reservation stations keep operations and operands only between the time they issue end the time they begin execution.

Each ROB entry contains four fields: *Type*, *Dest*, *Value* and the *Ready* field. The *Type* field indicates whether the instruction is a branch, a Store, or a register operation (ALU operation or Load). The *Dest* field supplies the register number for Loads and ALU operations or the memory address for Stores, where the instruction result must be written. The *Value* field is used to hold the value of the result until the instruction commits. The *Ready* field indicates if the instruction has completed execution and, thus, the value is ready. The ROB completely replaces the Store buffers. The ROB is usually implemented as a circular FIFO queue having associative search facilities.

Each reservation station has the following eight fields:

- *Op* – the operation performed on the source operands (*opcode*);
- Q_j, Q_k – the ROB entries that will provide the source operands, a value of zero indicating that the source operand is already available in V_j, V_k, or that it is unnecessary;
- V_j, V_k – the values of the source operands; for Loads and Stores the V_j field is used to hold the offset;
- *A* – holds the memory address for Loads or Stores: initially holds the immediate field, after the address calculation holds the effective address;
- *Dest* – supply the corresponding ROB entry number representing the destination for the result produced by the execution unit.
- *Busy* – indicates if a reservation station is available or occupied.

The register file has a field Q_i indicating the number of the ROB entry that contains the operation whose result should be stored into the register. The six steps involved in instruction execution are the following [Hen03]:

1. *Fetch* – fetches the next instruction into the instruction queue.
2. *Dispatch* – gets the next instruction from the instruction queue. If all reservation stations are full or the ROB is full, then instruction dispatch is stalled until both structures have available entries. If there is an empty reservation station and the tail of the ROB is free, the instruction is sent to the reservation station. The *Busy* bit of the allocated reservation station is set and the *Ready* field of the ROB entry is reset. The source registers are searched associatively in the *Dest* field of the ROB, considering the last entry in the case of multiple hits, since the ROB entries are allocated in order. If an operand value is available in the ROB (*Ready*=1), it is written from the *Value* field into the reservation station field V_j / V_k. If the operand value is not available (*Ready*=0), the number of ROB entry that will provide the operand is written into the reservation station field Q_j / Q_k. In the case of miss in the ROB the operand value is written from the register set into the reservation station field V_j / V_k. The number of ROB entry allocated for the value of the result is sent into the *Dest* field of the reservation station. The destination register number is written into the *Dest* field of the ROB entry.
3. *Issue* – if an operand is not yet available, the common data bus (CDB) is monitored until it is computed and when the operand is available on the CDB it is placed into the corresponding

reservation stations. In order to avoid structural hazards, modern processors have multiple CDBs and a multiported ROB. When all the operands are available, the instruction is issued to the appropriate functional unit. By delaying instruction execution until the operands are available, RAW dependences are detected.

4. *Execute* – the corresponding functional unit executes the operation. In the case of Loads and Stores the effective memory address is computed in this stage. In the case of a taken branch, usually is calculated the branch's target address.
5. *Writeback* – when the result is available, it is written to the CDB (together with the ROB entry number indicated by the *Dest* field of the reservation station) and from there into the *Value* field of the corresponding ROB entry, whose *Ready* field is set to 1. The *Busy* field of the corresponding reservation station is reset. The result is also written into field V_j / V_k of the reservation stations that are waiting for it. In the case of a Store instruction if the value to be stored is available, it is written into the *Value* field of the ROB entry allocated for that Store. If the value to be stored is not available, the CDB is monitored, and when it is received, the *Value* field of the ROB entry is updated.
6. *Commit* – the normal *commit* case occurs when an instruction reaches the head of ROB having its result available (*Ready*=1) and if no exception occurs. In this case, the result is written from the *Val* field of the ROB entry into the destination register or memory location indicated by the *Dest* field of the ROB entry and, after that, the instruction is squashed from the ROB. Thus, the in order *commit* is guaranteed by the in order *dispatch*, whereas the *issue*, *execute* and *writeback* stages can be processed out of order. When an incorrectly predicted branch reaches the head of the ROB, the ROB is flushed and the execution is restarted with the correct successor of the branch. More precisely, there are implemented two branch recovery strategies: refetch and selective reissue [She03]. The obvious disadvantage of the refetch based recovery is a severe misprediction penalty. The goal of selective reissue is to reduce this penalty. With this approach only dependent instructions are reissued in the case of misprediction. This requires a mechanism for propagating misprediction information through the data flow graph to all dependent instructions.

As it can be observed, in the case of speculative architectures is very important *when* is performed the updating. Using the ROB, speculative executions are possible because the register file or memory is updated with the result of an instruction only when that instruction is no longer speculative.

2.1.2. The Architecture of Sim-Outorder

After more than two decades, simulators have become an integral part of computer architecture research and design process [Yi06]. Their most important advantages, comparing with real processors, are low implementation cost and development time, flexibility and extensibility, allowing the architects to quickly evaluate the performance of a wide range of architectures and to quantify the efficacy of every enhancement. In this work we relied on some commonly used simulators like *Simplesim* [Bur97] and the *M-SIM* [Sha05] which extends the *Simplesim* toolset with support for concurrent execution of multiple threads and power consumption evaluation. Both of them are written in C language and the sources are free in order to be improved and enlarged by researchers.

The *sim-outorder* simulator from the *Simplesim-3.0* toolset [Bur97] is presented in Figure 2.2. It simulates a superscalar architecture that uses a register update unit (RUU) in order to support out-of-order and speculative execution. The RUU is a combination of reservation stations and ROB, and is organized as a circular queue. Each RUU entry contains the following fields:

- *IR* – stores the instruction bits.
- *op* – holds the opcode after the instruction is decoded in the *dispatch* stage.
- *PC* – the instruction address.
- *next_PC* – the next instruction address.
- *pred_PC* – the next predicted instruction address.
- *ea_comp* – non-zero if the operation is an address computation (the first operation in the case of Load and Store instruction preceding the memory access).
- *in_LSQ* – non-zero if the Load/Store operation is in the LSQ.
- *recover_inst* – indicates when an instruction is the start of misspeculation.
- *dir_update* – pointer to the branch predictor state entry.
- *spec_mode* – indicates if the instruction was fetched speculatively.
- *addr* – holds the effective address for Load/Store instructions.
- *tag* – RUU slot tag, used to identify an operation in the RUU.
- *queued* – indicates that the operands are ready and the operation was queued to the *ready_queue*.
- *issued* – indicates that the operation was issued for execution.
- *completed* – indicates that the operation has completed the execution.
- *onames* – output logical register names.
- *odep_list* – dependency list containing a pointer to all dependent RUU entries. These lists are used to limit the number of associative searches in the RUU when operations complete the execution and need to wake up dependent operations.
- *idep_ready* – indicates if the input operands are ready.

For Loads and Stores a Load/Store Queue (LSQ) is also used. The LSQ has the same structure as the RUU. Load and Store instructions are split in two operations: the effective address computation that is inserted into the RUU and the Load/Store operation that is inserted into the LSQ and is activated by the RUU when the address computation is finished. A rename-table structure called Create Vector (CV) holds for each register the last mapped RUU or LSQ entry that will write the result into that register. The CV is divided into a speculative table (maintains the last speculative state of the register file) and a non-speculative table (maintains the last non-speculative state of the register file). The CV is used to handle instruction dependencies: to construct the dependency lists (*odep_list*) and to squash efficiently the RUU and LSQ structures if an exception occurs. An instruction fetch queue (IFQ) is used to hold the instructions fetched from memory. Each IFQ entry has the following fields: *IR* (holds instruction bits), *regs_PC* (instruction address), *pred_PC* (next predicted instruction address) and *dir_update* (pointer to the branch predictor state entry). A ready queue (RQ) is used to hold operations whose operands are ready and an event queue (EQ) holds operations during their execution. Each RQ and EQ location contains only a pointer to the RUU or LSQ entry associated to the operation.

The *sim-outorder* simulator uses a pipeline with five important stages implemented in software: *fetch*, *dispatch*, *issue*, *write back* and *commit*. The classical *execution* stage is distributed into the *dispatch* and *issue* stages as we will detail further. In the software implementation of this superscalar architecture the pipeline stages are executed sequentially and are not overlapped leading in this way to synchronization problems. More exactly, because one cycle of execution in the simulator corresponds to the sequential iteration of all pipeline stages once, the effects of a certain stage are "instantaneously" seen by the next pipeline stages too early, in the current cycle, while they must be seen only in the next cycle. Therefore, in order to eliminate these synchronization problems, the pipeline stages are traversed in reverse order, and thus, the effects of a certain one-cycle operation are visible correctly only in the next cycle (iteration).

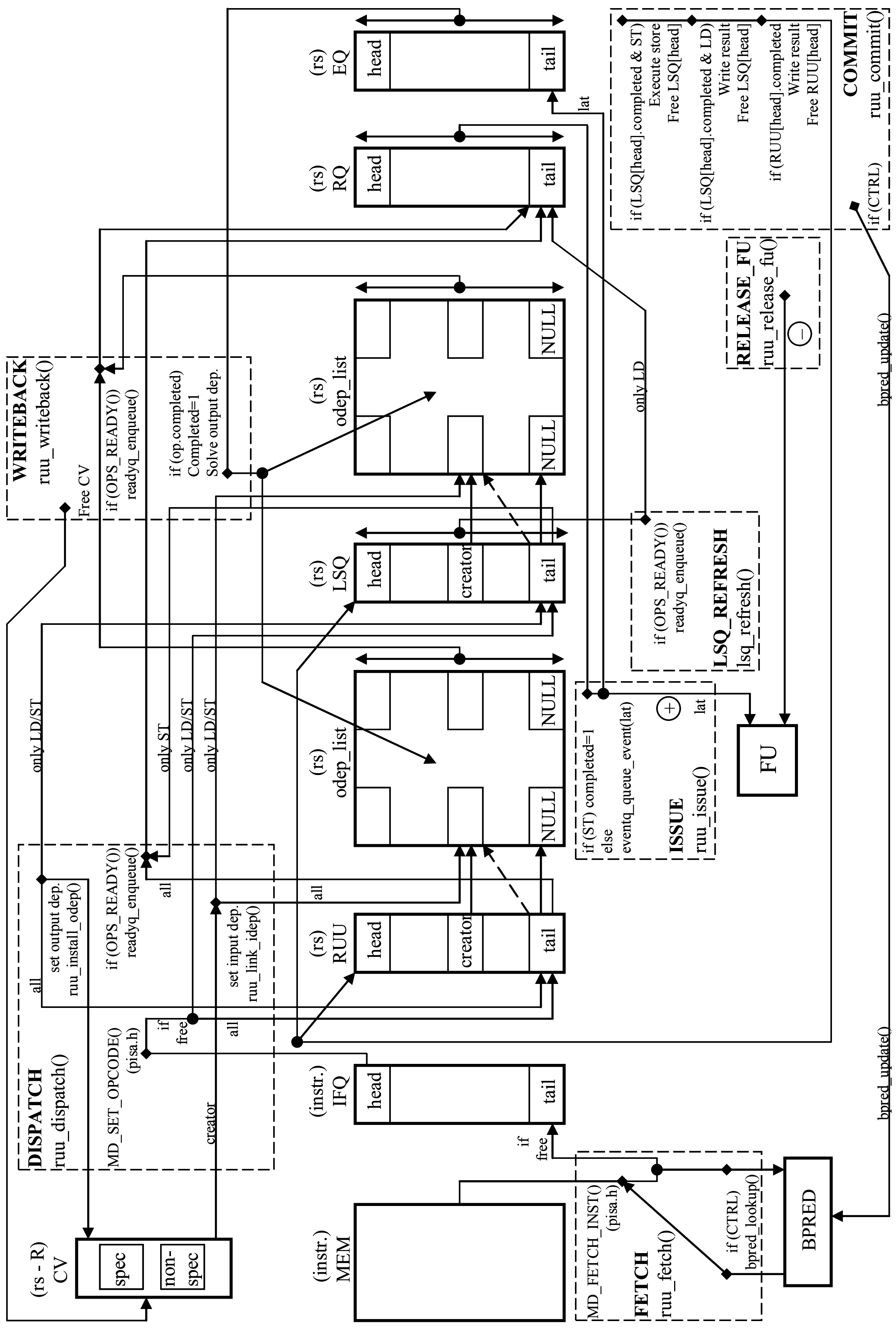


Figure 2.2. The architecture of *Sim-Outorder*

All events (marked with ♦) appear in Figure 2.2 horizontally in chronological order. The seven execution steps of *sim-outorder* are the following:

1. *Fetch* (*ruu_fetch*) – as many instructions are fetched up (*MD_FETCH_INST*) as one branch prediction and one instruction-cache line support, without overflowing the instruction fetch queue (IFQ). The instructions are inserted into the tail of the IFQ (*fetch_data*). If the simulator is started with a branch predictor, the instructions are pre-decoded in order to identify branches (*MD_SET_OPCODE*). When a branch instruction occurs the next instruction is fetched from the address *pred_PC* predicted using a certain *pred* branch predictor (*bpred_lookup*).
2. *Dispatch* (*ruu_dispatch*) – gets the next instruction from the head of the IFQ, decodes the instruction (*MD_SET_OPCODE*), and inserts it into the tail of the RUU if it is free. For Loads and Stores the effective address computation is inserted into the tail of the RUU, and the Load/Store operation is inserted into the tail of the LSQ. If the RUU/LSQ is full, then instruction *dispatch* is stalled until the structure has available entries. The dispatched instructions are removed from the IFQ. A pointer to the allocated RUU/LSQ entry (*rs*) is introduced into the dependency list (*odep_list*) corresponding to the RUU/LSQ entries – identified based on the CV – that will produce the input operands (*ruu_link_idep*). The output register numbers are written into the *onames* field and a pointer to the allocated RUU/LSQ entry (*rs*) is set to all the output registers in the CV structure (*ruu_install_odep*). If all the input operands are available, a pointer to the allocated RUU/LSQ entry (*rs*) is inserted into the tail of the RQ (*readyq_enqueue*). Actually the simulator "instantaneously" executes the operation in this stage, but correctly simulates its latency through the write-back event in the next stages. In the case of a Store instruction a pointer to the allocated LSQ entry is also inserted into the tail of the RQ (Load operations are queued into the RQ only in the *LSQ-refresh* stage).
3. *Issue* (*ruu_issue*) – tries to issue all instructions from the RQ (*ready_queue*) to free functional units (FU) whose *busy* count is set to the latency value corresponding to the issued operation. A writeback-event is scheduled for each issued operation to the cycle obtained adding its execution latency to the current cycle: a pointer to the corresponding RUU/LSQ entry (*rs*) is inserted together with the scheduled writeback-cycle (*wb_cycle*) into the EQ (*eventq_queue_event*). The EQ (*event_queue*) is sorted from earliest to latest event. The issued operations are evacuated from the RQ. The *issue* stage ends with the execution of the operations at the functional units (the previously presented Tomasulo's architecture has an additional *execute* stage for this operation). Thus, the execution is simulated by scheduling the writeback-event to the cycle obtained by adding the corresponding execution latency to the current cycle. Store operations are executed only in the *commit* stage.
4. *LSQ-refresh* (*lsq_refresh*) – a pointer to each Load operation (*rs*) from the LSQ whose operands are ready is inserted into the RQ (*readyq_enqueue*). Store operations are inserted during the *dispatch* stage.
5. *Writeback* (*ruu_writeback*) – in the case of a misprediction the RUU/LSQ entries corresponding to speculatively fetched instructions are squashed and the CV is reverted to the last non-speculative state. In the normal *writeback* case, for each event from the EQ whose scheduled writeback-cycle is less than or equal to the current execution cycle (the event has already occurred), the result is written from the functional unit (FU) to the RUU/LSQ, and the event is removed from the EQ. If the RUU/LSQ entry of the completed operation is still mapped in the CV to the output registers, the corresponding CV entries are invalidated (assigning NULL), because the construction of the operation's dependency list (*odep_list*) finished. Dependent operations that occur in the future will get the result from the RUU/LSQ or from the register file. Each RUU/LSQ entry that has a pointer in the dependency list (*odep_list*) of the completed operation is updated with the result, and if all its

operands are ready, it is queued into the RQ – its pointer (*rs*) is inserted into the tail of the RQ (*readyq_enqueue*).
6. *FU-release* (*ruu_release_fu*) – the *busy* count of each FU is decremented by 1. An FU is free for another operation when its *busy* count is 0.
7. *Commit* (*ruu_commit*) – the normal commit case occurs when an instruction reaches the head of the RUU/LSQ and its result is available (*completed*=TRUE). The results are written from the head of the RUU/LSQ into the register file. If a Store instruction occurs in the head of the LSQ, the Store data is written to the data cache. At the end of the *commit* stage the head of the RUU/LSQ is freed and, in the case of branch instructions, the used branch predictor *pred* is updated (*bpred_update*).

The *fetch*, *dispatch* and *commit* stages are effectuated in program order avoiding thus imprecise exceptions, while the other stages might be executed out of order. In fact, instruction execution is done “instantaneously” in *ruu_dispatch*. Thus, instructions flow down the pipeline only for timing evaluations. Therefore, there is no need to actually store the result value into the RUU/LSQ structure at the end of the *writeback* stage and there is no need to update the register file in the *commit* stage because that’s already been done in the *dispatch* stage.

2.1.3. Checkpoint Processing Architectures

Another technique that allows speculative execution consists in saving or checkpointing the state of the processor at certain points in a history buffer or a checkpoint, respectively [Smi95]. The architectural state of the processor is updated as instructions execute and when a precise state is needed, it is recovered from the history buffer. In this case, the *commit* phase consists only in the evacuation from the buffer of the unneeded processor states. The reorder buffer technique is more popular than the checkpoint/history buffer method, because, besides providing a precise state, it implements the register renaming function too.

The continuously increasing gap between processor and memory speeds – commonly known as the memory wall – produces a serious performance limitation of high-frequency microprocessors by main memory access latencies. One approach to the memory wall problem was the cache memory that exploits program locality to reduce the number of long-latency accesses to the main memory. Another approach is the out-of-order execution mechanism that can hide long operation latencies in superscalar processors by executing independent instructions while dependent instructions are waiting for their operands. For an L1 cache miss, these independent instructions can often hide the L2 access latency, but the approach is much less effective in the case of L2 cache misses. If the miss latency cannot be hidden, the ROB is blocked until the Load instruction completes. Cache misses often occur in bursts and, thus, when the ROB is unblocked it is blocked again by another L2 miss. A solution to reduce the number of L2 misses consists in prefetching speculatively data from memory to cache. Multithreading can also reduce the effects of the memory wall by hiding memory latency through issuing into the pipelines instructions from different idle threads.

A different approach to tolerate very long memory latencies consists in supporting a substantially increased number of in-flight instructions [Cri05]. Processors that are able to maintain thousands of in-flight instructions can hide the latency of memory operations by overlapping memory accesses with the execution of independent instructions. Unfortunately, supporting a high number of in-flight instructions typically involves scaling up critical processor resources (reorder buffer, instruction queue, physical register file, Load/Store queue) that is impossible in current processor designs because of area, power and cycle time limitations. Cristal et al. [Cri04b, Cri05] recently proposed Kilo-instruction Processors (KIP) that are able to support a high number of in-flight instructions through an intelligent management of the available resources instead of resource enlargement. They showed in [Cri05] that most instructions either have a short or a long flight-time, and, thus, they hold resources for a short or a very long time, respectively. The long flight-time instructions are usually blocked in the ROB because of long-latency memory operations. Kilo-instruction Processors exploit the bimodal flight time distribution by giving critical resources to the short-flight-time instructions and early releasing resources used by long-flight-time instructions, reallocating them later.

Current superscalar processors rely on in-order instruction commit to avoid imprecise exceptions, imposing thus important constraints to the use of the critical processor resources. Cistal et al. showed in [Cri05] that all these resources are highly underutilized and they changed the management of these resources in order to improve performance. They proposed kilo-instruction processors that combine multicheckpointing with techniques for efficient management of the IQs and the physical register file.

In a conventional superscalar processor every decoded instruction requires an entry in the reorder buffer (ROB) until the instruction commits. The ROB keeps a copy of all in-flight instructions, and thus, the processor can restore the correct architectural state at any instruction if an exception occurs. Kilo-instruction processors reduce the size requirement of ROB through checkpointing. A checkpoint is the state of the processor taken at a specific instruction of the program being executed. The state of the processor can be restored to that point if an exception occurs. The state of the processor can be checkpointed for a subset of instructions, and, if there is an exception, the processor can roll the state back to the closest checkpoint prior to the instruction causing the exception. Using a relatively small set of checkpoints for long flight-time instructions considerably reduces ROB requirements, but, obviously, the cost is a longer recovery time when a long-flight-time instruction suffers an exception.

Cristal et al. [Cri02] proposed selective checkpoints taken only when a Load that misses in the L2 cache reaches the head of the ROB. After taking such a checkpoint, the processor can early release the ROB resources, and also the physical registers and LSQ slots used by instructions in the ROB. Instructions independent of the Load with miss in L2 can then use these resources. In the same way, Mutlu et al. [Mut03] create a checkpoint of the architectural state when a Load that misses in the L2 cache reaches the head of the ROB. But their architecture starts executing instructions in a special speculative mode called *runahead* that invalidates the results of the Load and all dependent instructions. Some of the independent instructions executed in runahead mode might miss in the instruction-, data-, or unified caches. The memory system overlaps their miss latencies with the latency of the runahead-causing cache miss. When the runahead-causing Load completes, the processor exits runahead mode by flushing the instructions from its pipeline. It restores the checkpoint and resumes normal instruction fetch and execution starting with the runahead-causing Load. Thus, when the processor returns to normal mode, it can make faster progress without stalling because it has already prefetched into the caches during runahead mode some of the data and instructions needed during normal mode.

In [Cri04b, Cri04a] Cristal et al. replaced the ROB with a structure called pseudo-ROB. Decoded instructions are inserted into the pseudo-ROB in order. The processor removes instructions from the head of the pseudo-ROB at a fixed rate even if they are incomplete. The processor state is recoverable for any instruction from the pseudo-ROB and, therefore, checkpoints are taken only when incomplete instructions reach the head of the pseudo-ROB. This checkpointing mechanism is beneficial to alleviate the impact of branch mispredictions. The authors show that over 90% of mispredictions are caused by branches that are still in the pseudo-ROB and therefore do not need to roll back to the previous checkpoint for recovering the correct processor state, minimizing the misprediction penalty.

A multicheckpointing mechanism was proposed in [Akk03a, Akk03b] by Akkary et al. in order to implement large instruction window processors without requiring large structures. Rather than using a reorder buffer, they use a substantially smaller checkpoint buffer for branch misprediction recovery. Since checkpoints cannot be created at every branch, a branch misprediction causes the re-execution of all instructions between the last checkpointed instruction and the mispredicted branch. This re-execution overhead can be minimized if checkpoints are created on branches with high misprediction probability. Therefore, Akkary et al. use a confidence estimation scheme – a table of 4-bit saturating counters indexed by XORing the branch address with the global branch history – to select low-confidence branches. Correct branch prediction increments the corresponding counter while misprediction resets the counter to zero. Thus, they take a checkpoint when a low-confidence branch reaches the *decode* stage. In addition, the re-execution overhead is also minimized by checkpointing every 256th instruction. To prevent the same branch to be mispredicted again, in the case of re-execution from a checkpoint the branch outcome from the previous aborted execution is used instead of a prediction. Furthermore, once a branch misprediction is resolved and re-execution begins from a prior checkpoint, a new checkpoint is taken on the mispredicted branch, allowing the retirement of instructions between the two checkpoints. The checkpoints are stored in a FIFO buffer. Each entry in the checkpoint buffer has a counter that is used to determine when the corresponding checkpoint can be freed. A counter is incremented when an instruction, associated with the corresponding checkpoint, is allocated and decremented when the instruction completes execution, the overflow being prevented by creating a new checkpoint. A checkpoint is allocated only if a free entry is available in the checkpoint buffer. If a low-confidence branch is fetched and a free entry in the checkpoint buffer is not available, the processor continues fetch, dispatch and execution without creating a checkpoint on that branch. A checkpoint is reclaimed and all its associated instructions are retired when the value of the corresponding counter is 0 – the last instruction belonging to that checkpoint completes – and the next checkpoint is allocated. Their mechanism enables fast branch misprediction recovery and they show that a small number of checkpoints is sufficient for a large instruction window. Their simulation results show that 8 checkpoints are sufficient to support a 2048-entry instruction window, and thus, usually, each checkpoint corresponds to a group of hundreds of instructions.

Every decoded instruction requires an entry in the instruction queue (IQ) until it is issued for execution. In [Cri04b] the authors show that instructions are divided into two groups: instructions blocked for short time in the IQ that are waiting for a functional unit or for results of short-latency operations, but most instructions are blocked for long time, when they are waiting for long-latency instructions to complete, such as Loads with miss in L2 cache. Maintaining these instructions blocked for a long time in the IQ, increases the probability of stalling the processor due to a full IQ. This problem can be overcome by using multilevel IQs [Cri04b] that advantage of different waiting times of instructions in IQs. When long latency instructions – Loads without hit in the first or second level caches and all instructions that depends on – reach the head of the pseudo-ROB, they are removed from the IQ to a slower, but larger and less complex structure, called Slow Lane Instruction Queue (SLIQ). Later, when the long-latency operations are resolved, the dependent instructions are moved back from the SLIQ to the IQ.

Every renamed instruction that generates a result requires a physical register, which is assigned in the *rename* stage and it is released when the next instruction that use the same logical register commits. An analyze regarding the physical register file [Cri04b] shows that registers blocked for a long time and dead registers constitute the largest fraction of allocated registers. A large portion of registers are blocked for long time because the corresponding instructions are waiting for the execution of long-latency operations.

Cristal et al. [Cri04b] integrated into kilo-instruction processors an aggressive register management mechanism, called ephemeral registers, that allows dead registers to be released early and registers blocked for long time to be allocated late. Instead of assigning physical registers to architectural registers, their late register allocation mechanism assigns virtual tags, the physical registers being only assigned when the instructions are issued for execution. In order to implement the early register release technique, each virtual tag has an associated counter. The counter is incremented each time the source register of an instruction is renamed to that virtual tag and is decremented each time the reader instruction is issued for execution. The virtual tag and its associated physical register can be released when the corresponding counter reaches zero and the register has been already written.

Akkary et al. in [Akk03a] proposed another register release scheme implemented by associating a counter to each physical register. A counter is incremented each time the source operand of an instruction is mapped to the corresponding physical register and is decremented each time an instruction actually reads that physical register. A physical register can be released when its counter is 0 and the logical register corresponding to that physical register is renamed again. Since a checkpoint provides the ability to restore the correct architectural state, physical registers must be released only after the corresponding checkpoint is released. Using checkpoints as readers guarantees that physical registers are not released until all checkpoints to which they belong are released. Therefore, when a checkpoint is created, the counters of all the physical registers belonging to the checkpoint are incremented. Similarly, when a checkpoint is released, the counters of all the physical registers that belong to the checkpoint are decremented. Thus, the proposed register file mechanism performs comparable to a larger conventional register file by significantly reducing the average lifetime of the physical registers.

Memory instructions also require an entry in the Load/Store queue (LSQ) until commit. The LSQ assures the program order commitment of all Load and Store instructions, and, therefore, complex memory disambiguation logic is necessary that compares the effective address of each memory operation with the addresses of all previous in-flight memory operations. In the memory disambiguation mechanism used by Cristal et al. [Cri04b], if a Load is issued and an older Store must write the same memory location, the Store result is forwarded to the Load. If the Store result is still not ready, the Load is rejected and reissued again.

Akkary et al. [Akk03b] proposed a hierarchical Store queue organization. When a new Store appears, it is inserted into a fast and small first-level Store queue that holds the last executed Stores. If this first-level Store queue is full, the space for the new Store is assured by removing the oldest Store instruction into a larger and slower second-level Store queue that holds it until commit. A membership test buffer is used to predict if a certain Store instruction is buffered in the second-level Store queue. When a Load instruction is issued, the first-level Store queue and the membership test buffer are accessed. If the Load misses both the first-level Store queue and the membership test buffer, the data is forwarded to the Load from memory. If the Load hits the first-level Store queue, the data is forwarded to the Load. If the Load misses the first-level Store queue, but hits the membership test buffer, the second-level Store queue is accessed. If there is a hit, the data is forwarded to the Load, but in the case of miss the data is forwarded from memory with penalization. This hierarchical Store queue organization with only a few hundreds of entries provides performance close to a usual Store queue with thousands of entries, which is remarkable.

The performance of kilo-instruction processors on integer programs is sometimes limited by hard-to-predict branches and pointer chasing. In general, Loads become very critical when they drive a hard-to-predict branch. Hopefully, the performance on these integer programs can be improved, with selective checkpointing applied on long latency Loads and hard-to-predict branches – identified in [Gel06a, Vin06] based on the polarization index of branch instructions in different contexts, such as local history, global history and path information. Selective checkpointing can be applied on unbiased branch contexts in the same manner as Akkary et al. [Akk03a] created checkpoints on low-confidence branches, and as Chappell et al. [Cha02b] used microthreads only for branch instances likely to be mispredicted.

Pericàs et al. [Per06] introduced the execution locality concept, a property that describes instructions as a function of the number of cycles they wait in the queues until they issue. Thus, instructions depending on cache misses have low execution locality, while the remaining instructions, including those that depend only on cache hits, have high execution locality. The small amount of low execution locality code causes stalls that significantly reduce performance. Based on the observation that low execution locality code is very decoupled from high execution locality code, Pericàs et al. proposed a decoupled microarchitecture (Figure 2.3) that executes low latency instructions on a Cache Processor and high latency instructions on a Memory Processor.

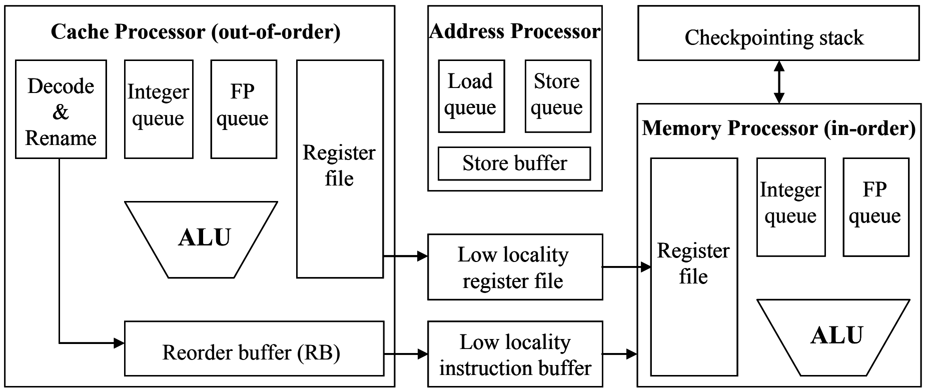


Figure 2.3. The Decoupled Kilo-Instruction Processor

Thus, one pipeline, the out-of-order Cache Processor, exploits instruction-level parallelism, while a second pipeline, the in-order Memory Processor, exploits memory-level parallelism. The instructions are fetched by the Cache Processor where they are waiting to be issued to the functional units. If an instruction turns out – based on a timer – to have long issue latency, it is moved from the Cache Processor into a Low Locality Instruction Buffer (LLIB), where it is waiting until all long-latency Load operations it depends on have finished. When the operands of a long latency instruction are available, they are inserted into the Low Locality Register File (LLRF). Long-latency Loads are executed in the address processor by the LSQ. After a long latency Load completes, the value is kept in the address processor. When the dependent instructions arrive to the head of the LLIB and the Load value is available, they are moved to the Memory Processor to be executed. For the recovery after mispredictions in the Cache Processor an ROB structure is used. In the Memory Processor the recovery is assured through selective checkpointing. Taking a checkpoint involves copying the ready values from the architectural register file into a free entry of the Checkpointing Stack. The state of the Memory Processor is restored from the Checkpointing Stack if an exception occurs.

An important limitation of the above presented decoupled microarchitecture consists in the serialization (in-order execution) of all memory-dependent instructions within the Memory Processor, resulting in about 10% performance loss. Therefore, Pericàs et al. in [Per07] have further developed their decoupled microarchitecture presented in [Per06] by allowing it to scale to multiple cores and multiple threads. This architecture, with variable window size, uses multiple cores called Memory Engines that can be shared among threads. Thus, their flexible multi-core architecture consists of a set of Cache Processors, each one with a static partition of Memory Engines, and a pool of Memory Engines that can be dynamically assigned to different threads. Consequently, the proposed microarchitecture has good potential to adapt to application mixes, because threads without Memory Engine requirements can yield their resources to threads that require more Memory Engines. Moreover, when there are fewer threads than Cache Processors, the active threads can access the dynamic pool of Memory Engines without competition. The evaluation results obtained on the floating-point SPEC 2000 benchmarks by using 16 Memory Engines show a considerable IPC speedup of 12% compared to the baseline decoupled microarchitecture.

2.2. Advanced Dynamic Branch Prediction

Accurate branch prediction is increasingly important in today's high performance superscalar processor designs. A variety of basic branch prediction techniques are presented in [Hen03] such as the *branch prediction buffer* (BPB) or the *branch target buffer* (BTB). The goal of all these mechanisms is to allow the processor to speculatively execute control dependent instructions, thus avoiding stalls and extending the instruction-level parallelism across multiple basic blocks.

The BPB and BTB predictors use only the recent behavior of a single branch to predict the future behavior of that branch. Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors* [Hen03], and they were introduced independently by Yeh and Patt [Yeh92] and by Pan et al. [Pan92]. Two-level predictors use two levels of branch history information to make predictions. The first level is a branch history register (BHR) that records the outcomes of the last *k* branches encountered. The second level is a pattern history table (PHT) with entries having the same fields as a BTB entry. The PHT is indexed using a concatenation of the lower portion from branch's PC with the BHR representing the context of the branch [Pan92]. After a branch is resolved, its outcome is shifted left into the BHR, and the corresponding PHT entry is also updated.

In [Yeh92] there are presented three important alternative two-level branch predictors and other variations are introduced in [Yeh93]. The simplest two-level branch predictor called GAg (Figure 2.4), uses a global BHR and a global PHT. Since the outcomes of different branches update the same history register and the same pattern history table, the information of both branch history and pattern history is influenced by results of different branches. Initially all these predictors did not have the tag checking mechanism, using only the global/local histories as pointers to the prediction table; however, according to [Vin00a], in Figure 2.4 we added this mechanism that reduces branch interferences.

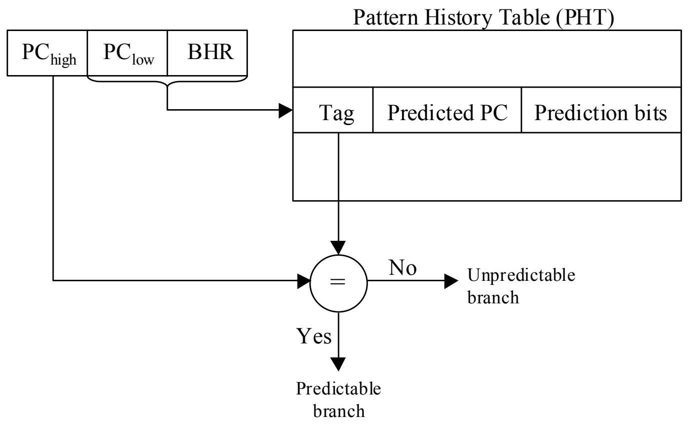


Figure 2.4. Two-level branch predictor (GAg)

In order to reduce the branch interference, in [Yeh92] the authors introduced a two-level branch prediction mechanism called PAg that uses a per-address branch history table (PBHT) and a global pattern history table. The PBHT consists of multiple local branch history registers, each of them being associated to distinct static branch instructions. A local BHR records the last *k* outcomes of the same static branch. Since all branches update the same PHT, the pattern history interference still exists. In order to completely remove the interference in both levels, in [Yeh92] the authors proposed another two-level branch predictor called PAp that uses a per-address branch history table and a per-address pattern history table. Thus, the PAp scheme keeps separate history and pattern information for each distinct static branch.

All two-level branch predictors presented in [Yeh92] use only global or only local branch history information. The global branch history is used to exploit correlation between the outcome of a branch and the outcomes of other branches. In contrast, the local history exploits correlation between the outcomes of a single branch. However, there exist branches that are not predictable based only on global or local history. Therefore, in [Vin00a, Cha02a] the authors introduced the two-level branch predictors, which employ both the global and local branch history information simultaneously.

McFarling [McFar93] proposed a new technique that combines the advantages of two different types of branch predictors. His technique uses 2-bit up-down counters to keep track of which predictor is currently more accurate for each branch. The hybrid predictor uses the more accurate component predictor to generate the prediction. He also describes a method of increasing the usefulness of branch history by hashing it together with the branch address, instead of concatenating it with the branch address as Pan and Rahmeh have done in [Pan92]. McFarling demonstrated that the eXclusive OR (XOR) of the branch address with the global history has more information than either component alone. Combining a predictor that use local branch history with a predictor that use global branch history hashed together with branch address, he obtained a prediction accuracy of 98.1% on the SPEC89 benchmarks [SPEC].

Hybrid predictors provide high prediction accuracy, but they must bring data from several component prediction tables to compute a final prediction. Unfortunately, this complexity adds more gate delay to the process of making a prediction. Because the branch predictor is on the critical path for fetching instructions, it must deliver a prediction in a single cycle. Jiménez proposed in [Jim03b] an alternative predictor design that completely hides prediction latency so that accuracy and hardware budget are the only factors that affect the efficiency of the predictor. The key idea is to organize the predictor so that a small set of candidate entries from the prediction table is prefetched several cycles before the prediction is needed. Because more possible target instructions are fetched and executed, when it becomes known which entry from the prediction table must generate the prediction, the final prediction can be selected in a single cycle.

Falcón et al. in [Fal04] introduced the prophet/critic hybrid branch predictor, which has two component predictors that play the role of either prophet or critic. The prophet is a conventional predictor that uses branch history to predict the direction of the current branch. The critic uses both the history and the future of the branch to give a critique of the prediction provided by the prophet for the current branch. Thus, the critique is used to generate the final prediction for the branch. In a conventional hybrid predictor, both components are accessed in parallel. In the prophet/critic hybrid predictor, although both prophet and critic predict the same branch, the predictions are not initiated at the same time. The prophet generates the prediction for the current branch in an early pipeline stage, and goes on along the predicted path generating new predictions, thus providing branch future. This allows the output of the prophet (branch future) to be used as input to the critic, which provides its critique some cycles later. This critique either agrees or disagrees with the prophet prediction, and determines the final prediction for the branch. When the prophet mispredicts a branch, the critic uses its future bits to train its prediction structures. When the branch is encountered again, the critic uses the future bits as context to identify if the prophet is likely to be wrong and should be overridden, thus increasing the prediction accuracy.

Dynamic branch prediction with neural techniques was first proposed by Vintan [Vin99a], exploring the use of learning vector quantization (LVQ) method. In [Vin00b] Vintan analyzed the suitability for branch prediction of the LVQ and a Multi-Layer Perceptron (MLP) with a single intermediate layer using the *backpropagation* learning algorithm. The author compared the performance of a two-level adaptive branch predictor with the LVQ and MLP neural branch predictors. Both the classical and neural schemes predicted based on the same information (PC, LHR, GHR). While the LVQ predictor achieved results comparable to an equivalent conventional predictor, both the statically pre-trained MLP and dynamically trained MLP outperformed the two-level adaptive branch predictor. Taking into account that all branches were predicted using only one global neural predictor (LVQ or MLP) instead of a local neural predictor per branch, the obtained results encouraged other researches in neural branch prediction domain.

As a consequence Jiménez and Lin [Jim03a] proposed a two-level scheme that uses fast per branch single-layer perceptrons instead of the commonly used two-bit saturating counters. The branch address is hashed to select the perceptron, which is used to generate a prediction based on global branch history. The perceptron, one of the simplest neural networks, is a natural choice for branch prediction because it can be efficiently implemented in hardware. Other neural methods, such as backpropagation, radial basis networks, Elman networks and Learning Vector Quantization, were studied in [Ste01, Kim03, Ega03] but these methods are less attractive because of excessive implementation costs. The single-layer perceptron consists of one artificial neuron providing weighted connections between several input units and one output unit. A perceptron learns a target boolean function $t(x_1,...,x_n)$ of n inputs. In the case of branch prediction, the x_i are the bits of the branch history register, and the target function predicts whether a particular branch will be taken. Intuitively, a perceptron keeps track of positive and negative integer correlations between branch history and the branch being predicted, each weight w_i representing the correlation between the output of an already executed branch x_i and the output of the branch being predicted (w_0 is the bias weight). The output y of a perceptron is computed as

$$y = w_0 + \sum_{i=1}^{n} x_i \cdot w_i \tag{2.1}$$

The inputs are bipolar, thus each x_i is either –1, meaning not taken or 1, meaning taken. A negative output is interpreted as predict not taken while a positive output is interpreted as predict taken. Figure 2.5 presents the structure of a perceptron.

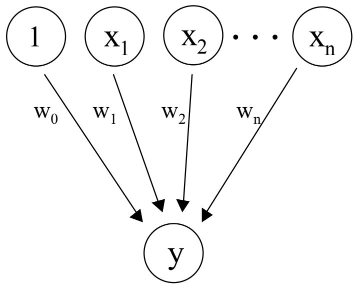


Figure 2.5. The structure of a simple perceptron

Once the perceptron output y has been computed and the branch executed, the following formula is used to train the perceptron:

$$\begin{aligned}&\textit{if } sign(y) \neq t \textit{ then}\\ &\qquad w_i = w_i + t \cdot x_i, \quad i = 0,...,n\end{aligned}$$

where t is the target behavior of the branch: -1 if the branch was not taken, or 1 if it was taken. The weight w_i is incremented when the branch outcome agrees with x_i, and it is decremented in the case of disagreement. Thus, for a mostly agreed connection (positive correlation), the weight becomes large, and when there is mostly disagreement (negative correlation), the weight becomes negative with large magnitude. In both cases, the weight has a large influence on the prediction. When there is a weak correlation, the weight remains close to 0 contributing little to the output. Jiménez trained the predictor using a special case of Rosenblatt's perceptron learning rule [Mit97]:

$$E(W) = \frac{1}{2} \cdot (t - o)^2 \tag{2.2}$$

For each input unit i, $i = 1,...,n$, calculate the error

$$\Delta w_i = -\alpha \cdot \frac{\partial E(W)}{\partial w_i} = -\alpha \cdot (t-o) \cdot \frac{\partial (t-o)}{\partial w_i} = -\alpha \cdot (t-o) \cdot \left(-\frac{\partial o}{\partial w_i} \right) = \alpha \cdot (t-o) \cdot x_i$$

- if $t = o$ the weights are not updated;
- if $t = 1,\ o = -1,\ \Delta w_i = \alpha \cdot (t + t) \cdot x_i = t \cdot x_i$ for $\alpha = 0.5$;
- if $t = -1,\ o = 1,\ \Delta w_i = \alpha \cdot (t + t) \cdot x_i = t \cdot x_i$ for $\alpha = 0.5$.

Their predictor achieved increased accuracy by using long branch histories without requiring exponential resources. Thus, for a 4KB hardware budget they improved misprediction rates for the SPEC 2000 benchmarks [SPEC] by 10.1%. In [Jim02] they developed a perceptron-based predictor that use both local and global branch history in the prediction process, increasing the accuracy by 14% over the McFarling-style hybrid predictor [McFar93]. In [Jim01b] they compared the perceptron predictor with a Multi-Layer Perceptron (MLP) using the backpropagation learning algorithm. At each history length, the perceptron predictor was more accurate than the MLP. Although backpropagation should be able to asymptotically exceed the accuracy of the perceptron, the longer training time causes it to be slightly less accurate overall.

Hunt et al. [Hun03] compared different perceptron-based predictors: global predictors trained on global branch history, local predictors with each of them trained on history of a single branch, and combined predictors that use both local and global branch histories. They obtained the best prediction accuracy of 96.41% with the combined predictor. They also demonstrated that the perceptrons trained on bipolar data outperform the equivalent systems trained on binary data.

In [Jim03c] Jiménez improved the previously presented neural architecture obtaining prediction accuracy far superior to conventional predictors but with a latency comparable to predictors from industrial designs. He used a neural predictor that selects the vector of weights used to generate prediction, according to the path leading up to a branch – based on all branch addresses belonging to that path –, rather than according to the branch address alone as the original perceptron does. This selection mechanism improves significantly the prediction accuracy, because, due to the path information used in the prediction process, the predictor is able to exploit the correlation between the output of the branch being predicted and the path leading up to that branch. On the other hand, the prediction latency of path-based neural predictors is lower (almost completely hidden), because the computation of the output can begin far in advance of the effective prediction, each step being processed as soon as a new element of the path is executed.

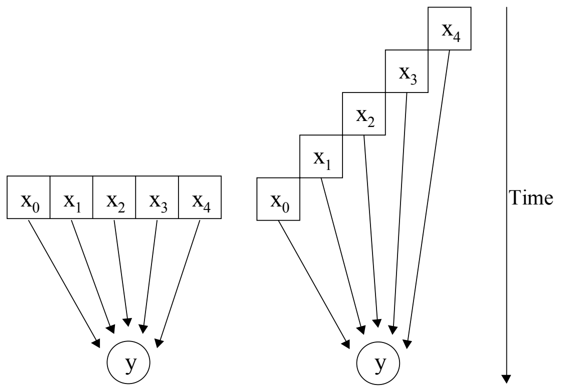


Figure 2.6. The output computation process with simple perceptron versus path-based perceptron

The most critical-timing operation is the sum of the bias weight and the current partial sum. Figure 2.6 depicts the difference between the original perceptron and the path-based neural predictor. The path-based neural predictor improves the instructions-per-cycle (IPC) rate of an aggressively clocked microarchitecture by 16% over the original perceptron predictor [Jim01a].

Seznec [Sez04] proposed an improved perceptron-based predictor called redundant history skewed perceptron (RHSP). The author demonstrated that the accuracy that can be achieved by a perceptron-based predictor is significantly better than the one achieved by the original perceptron predictor. This accuracy increase is allowed by the combination of three techniques: use of a redundant history, pseudo-tagging, and skewing. Their experiments showed that the use of a redundant history introducing up to four bits to represent a branch significantly improves the potential accuracy of the predictor. Pseudo-tagging is introduced in order to decrease aliasing impact on the perceptron table by using a few bits of the address as part of the input vectors. Thus, when several branches share a perceptron, it will predict them correctly if it is able to linearly separate their addresses. The skewing technique that also contributes to higher prediction accuracy consists in splitting the table of perceptrons in distinct physical tables. These distinct tables are indexed with different hashing functions. Seznec also reduced the complexity of the computation and introduced the ahead-pipelined RHSP that initiates prediction computation ahead. The simulations showed lower prediction latency and the same prediction accuracy.

Fern et al. [Fer04] proposed a dynamic decision tree (DDT) for hardware prediction. The main idea of dynamic feature selection using DDT is to provide branch prediction selecting and storing information about only the most relevant features from the larger feature set. They use at each decision tree node a correlation feature selector mechanism to select the most predictive feature from a large set of candidate features. These candidate features are the bits of the local and global branch history. The correlation feature selector associates a signed counter to each feature in the set. Thus, a large counter magnitude for a feature indicates its strongly positive or negative correlation with the branch outcome. The correlation feature selector is updated incrementing the counters for the features that agree with the target outcome, and decrementing the remaining counters. In the prediction process, the most correlated feature is selected from the set of candidate features, including the constant feature. The purpose of the XOR operations is to select either the value or its negation based on whether the correlation counter indicates a positive or negative correlation.

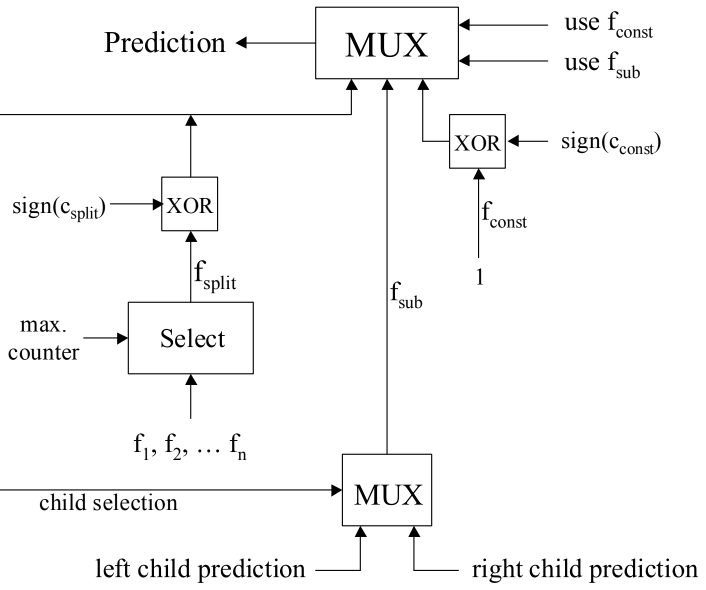


Figure 2.7. Prediction process of an internal DDT node

The DDT maintains a summary information for each node indicating whether the prediction should be made using a branch history feature, the constant feature, or the prediction of the selected child. A child is chosen based on the sign of the correlation between the selected feature and the target outcome. Leaf nodes behave identically to internal nodes except they do not have child predictors.

The DDT-based predictor is implemented using a prediction table. Each location of the table stores the information of the nodes for a particular tree. In prediction mode the prediction table is accessed with the lower-order bits of PC, and the summary information for each node is read out. The summary information at every node is used in conjunction with the feature vector to select the decision to be made at that node: either to predict based on a single feature or to pass the prediction to the appropriate child. Once the parallel decision operations have occurred, the prediction process identifies a single path of activated nodes from the root to one leaf. The only operation that uses time proportional to the depth of the tree is the flow of the prediction up the tree on the selected path from the leaf to the root. When a target outcome is resolved the correlation feature selectors are updated. The simulations on the SPEC'95 benchmarks [SPEC] indicated that the DDT-based branch predictor performs comparable to conventional two-level predictors with similar storage requirements. In domains with many features the DDT has an advantage in terms of time over perceptrons, because its prediction time depends only on tree depth and not on the number of features.

Other state-of-the-art branch prediction schemes, more related with our work, are presented in Chapter 4.

2.3. Dynamic Value Prediction

Value Prediction (VP) is a relatively new technique that is built on the concept of value locality and increases performance by eliminating true data dependencies. The main aim is to early predict instruction results during their *fetch* or *decode* stages and to speculatively issue and execute data dependent instructions using the predicted values. If the prediction is incorrect, recovery mechanisms must be employed to squash speculative results and reexecute all instructions that have already used the mispredicted value. An important challenge of the VP technique is to compress the program's dynamic critical path and therefore to solve the so-called Issue Bottleneck. If the predicted instructions do not belong to the critical path, the technique is obviously not efficient in reducing the limitation of the critical path. Therefore, the VP technique tries to avoid a fundamental limitation in the present-day computing paradigm, the Read After Write (RAW) data hazards, thus the intrinsic sequential program execution.

Lipasti et al. [Lip96a] first introduced Value Locality as the third facet of the locality concept (temporal and spatial). They defined the value locality as "the likelihood of the recurrence of a previously-seen value within a storage location inside a computer system". Measurements using SPEC'95 benchmarks show that value locality on Load instructions is about 50% using a history of one (producing the same value like the previous one) and 80%, using a history of 16 previous instances. Based on the dynamic correlation between Load instruction addresses and the values being loaded, Lipasti et al. proposed a new data-speculative micro-architectural technique entitled *Load Value Prediction* that can effectively exploit value locality. Load value prediction is useful only if it can be done accurately since incorrect predictions can lead to increased structural hazards and longer execution latency. Classifying the static Loads separately based on their dynamic behavior (unpredictable, predictable and constants), the full advantage of each case can be extracted. The cost of mispredictions can be avoided by detecting the unpredictable Loads and also the cost of memory access through identifying highly predictable Loads. The proposed Load Value Prediction Unit consists of a direct mapped Load Value Prediction Table (LVPT) for generating last value predictions, a direct mapped Load Classification Table (LCT) that maintains 2-bit saturating counters in order to classify Loads as unpredictable, predictable or constants and a Constant Verification Unit (CVU) used for constant Loads. The CVU assures coherence between the LVPT value and the real value from the main memory. If a LVPT entry is classified as being constant, its LVPT index and memory address are stored in the associative CVU table. Any entry with data address matching a subsequent Store instruction is invalidated in the CVU table, and the corresponding LCT counter transits into the predictable state.

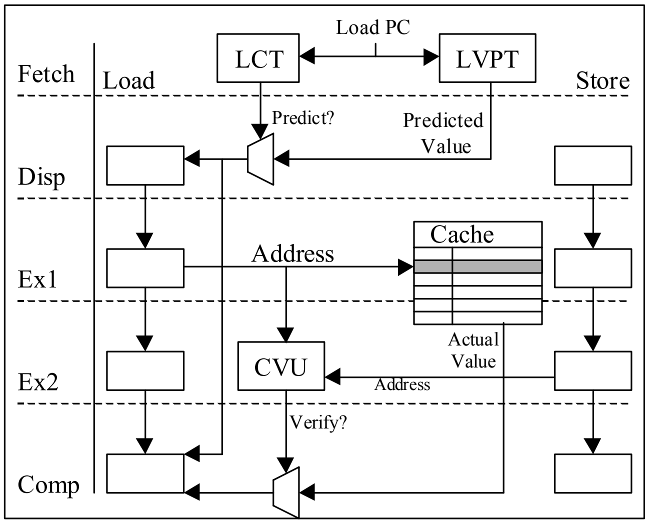


Figure 2.8. LVP Mechanism

The LVPT and LCT structures are indexed by the lower part of the PC and simultaneously accessed in the *fetch* stage of the Load in order to generate a value prediction and to determine whether or not a prediction should be made, respectively. If the corresponding confidence counter is in the unpredictable state, no prediction is generated. For Loads classified as predictable, the LVPT is used to predict the value that was previously loaded by that instruction from memory (last value prediction) and to forward it to the dependent instructions. When the Load completes, the predicted and actual values are compared, the LVPT and LCT are updated, and the correct path is reexecuted in the case of misprediction (recovery). For constant Loads that find a match in the CVU, accessing the conventional memory system is completely avoided.

Lepak and Lipasti [Lep00a] introduced the Store locality concept and Store prediction methods, with good results especially for multiprocessor systems. Similarly with the approach of Load instructions, the Store value locality was measured using PC (instruction-centric) or data address (memory-centric). In both cases the value locality degree is between 30% and 70%. The authors introduced the "silent Store" concept, meaning that a Store writes the same value like its previous instance (34% – 68% of dynamic Store instructions are silent Stores). Removing these Store instructions at some points in the program's execution (either statically at compile time, or dynamically at run time), some potential benefit can be gained in execution time and/or code size. They describe how is enhanced the performance of uniprocessor programs by squashing silent stores: the pressure on cache write ports and on Store queues is reduced and the data bus traffic outside the processor chip is also decreased. The free silent Store squashing concept is based on idle read port stealing to perform Store verifies and aggressive Load/Store Queue to exploit temporal and spatial locality for Store squashing [Lep00b].

In [Saz97] Sazeides and Smith developed an empirical classification of value sequences produced by instructions. There are two kinds of value predictability existing in programs: value repetition and value computability. In order to capture these certain types of value predictability, the authors have been proposed two distinct main categories of predictors: computational and contextual. Two important characteristics were also defined for understanding prediction behavior. One is the Learning Time (LT), which is the number of values that have to be observed before the first correct prediction. The second is the Learning Degree (LD), which is the percentage of correct predictions following the first correct prediction.

Computational predictors are predicting the next value based on some previous values in an algorithmic manner, therefore according to a deterministic recurrence formula. The simplest computational predictors are the *last value predictors* (LVP) that perform a trivial computational operation: the identity function. The next value of a static instruction is predicted as being the most recent value produced by that instruction. LVP were used for the first time in [Lip96a] to predict Load values and in [Lip96b] the value prediction process was extended to other instruction types.

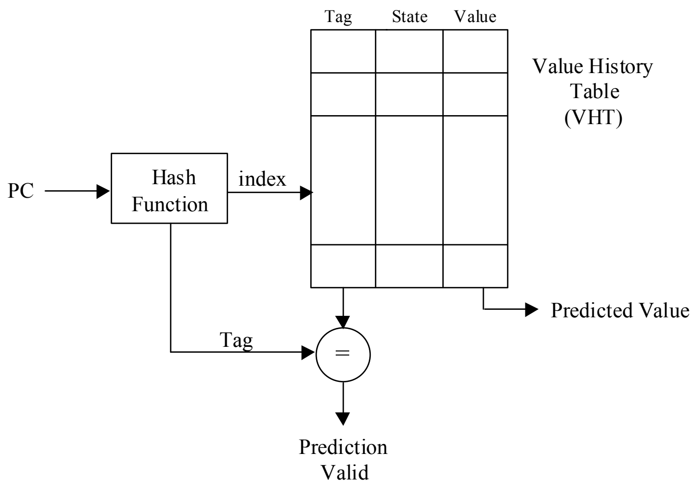


Figure 2.9. Last Value Predictor

The value history table is indexed by the instruction address. Each entry in the prediction table has three fields: *Tag*, *State* and *Value*. The *Tag* field stores the identity (the lower part of the PC) of the instruction that is currently mapped to that entry, and the *Value* field stores the last result for that instruction. The *State* field represents a saturating confidence counter (automaton), which is incremented when the prediction is correct and is decremented otherwise. The verification of the values generated by the VHT is necessary. The state of the confidence counter will be changed according to the comparison between the predicted and actual values.

In [Lip96b] Lipasti and Shen introduced another computational predictor, the stride predictor, and Sazeides and Smith in [Saz97] generalized the idea. A stride sequence is a value sequence in which the next value can be computed by the immediate previous value and a stride. Stride predictors in their simplest form predict the next value by adding the most recent value to the difference of the two most recent values produced by an instruction.

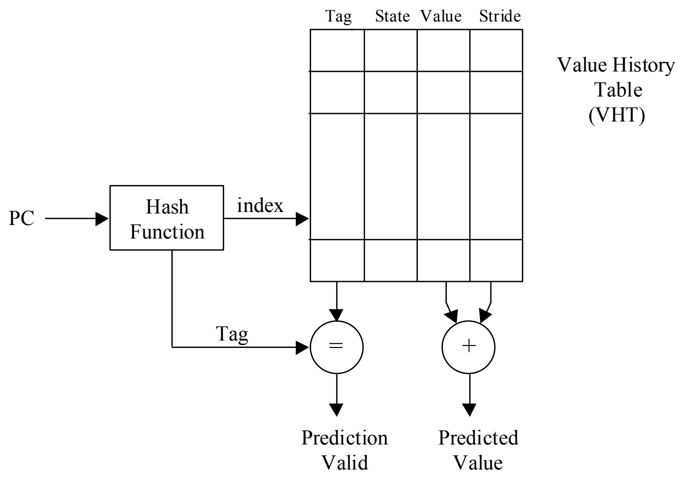


Figure 2.10. Stride Predictor

The experimental results indicated that the performance of computational prediction varies between instruction types indicating that its performance can be further improved if the prediction function matches the functionality of the predicted instruction.

Figure 2.11 presents a generic scheme for a context-based predictor. During the instruction *fetch* stage the context from VHT (Value History Table) is addressed using the PC. This context will address the VPT (Value Prediction Table). A location from VPT contains two fields: *Val* and *Confid*. The *Val* field stores the last instruction value(s), and the *Confid* field stores the confidence degrees attached to each value from the *Val* field.

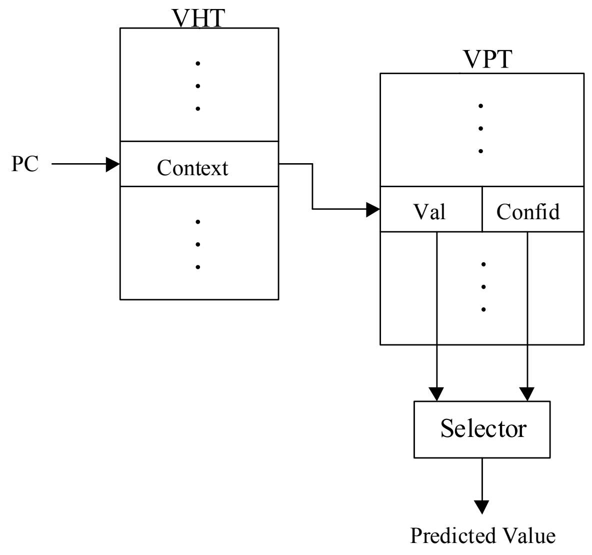


Figure 2.11. A generic context-based predictor scheme

The value with the highest confidence is predicted only if this confidence is greater than a certain threshold. Practically the scheme represents a simplified feasible implementation of the generic PPM predictor (because it counts the frequencies for each value following a certain context). Obviously, there might be some interferences in the PHT. An interesting solution in this sense is given in [Des02] where the authors proposed to use a second hashing function, independent of the first one. They use independent hashing as confidence mechanism for value prediction. Figure 2.12 shows this hashing mechanism. The interferences are strongly reduced with great benefits on prediction accuracy.

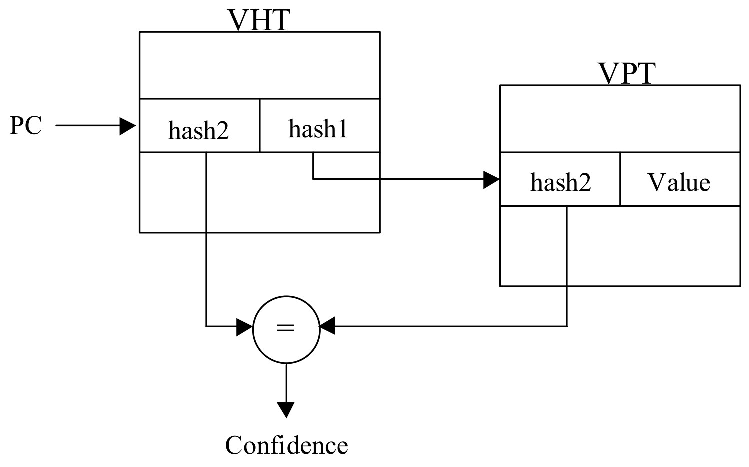


Figure 2.12. Independent hashing

In [Wan97] Wang and Franklin introduced a two-level value predictor for data value prediction (see Figure 2.13). The VHT has four fields: *Tag*, *LRU*, *Data Values*, and *Value*

History Pattern. The *Data Values* field stores up to four most recent unique values. There is a statistical explanation for using only the last four values: 15% – 45% of instructions produce only one value in their last 16 dynamic instances and 28% – 67% produce maximum four distinct values. The four values are associated with the binary encoding {00, 01, 10, 11}. So long as the different instances of a static instruction keep producing one of these four values, the next value can be predicted by selecting one of the four outcomes. When a fifth unique value is produced, it replaces from the *Data Value* field the least recently seen value, based on the *LRU* field that keeps a counter for each stored value. The *Value History Pattern* (*VHP*) field stores a *2p* bit pattern representing the last *p* outcomes of an instruction. Because there are four possible outcomes for an instruction {00, 01, 10, 11}, two bits are required to store each outcome. The *VHP* field is used to index a second prediction level, the Pattern History Table (PHT). For each possible *2p* bit pattern, four independent up/down counter values {C_0, C_1, C_2, C_3} are stored in the PHT, representing a condensed history of the previous outcomes of the pattern. When a prediction is to be made, the maximum counter is determined from the selected PHT entry, and the outcome corresponding to that counter is predicted. If $MAX(C_0, C_1, C_2, C_3) = C_K$ then the outcome of the MAX circuit is the binary code of *K* on two bits. A prediction is furnished only if the maximum counter value is greater than a specific threshold value. The two-level predictor is updated as follows. The *VHP* field of the selected VHT entry is shifted left by two bits and the new outcome is entered. The counter from the selected PHT entry corresponding to the correct outcome is incremented by 3, and all the other counters are decremented by 1.

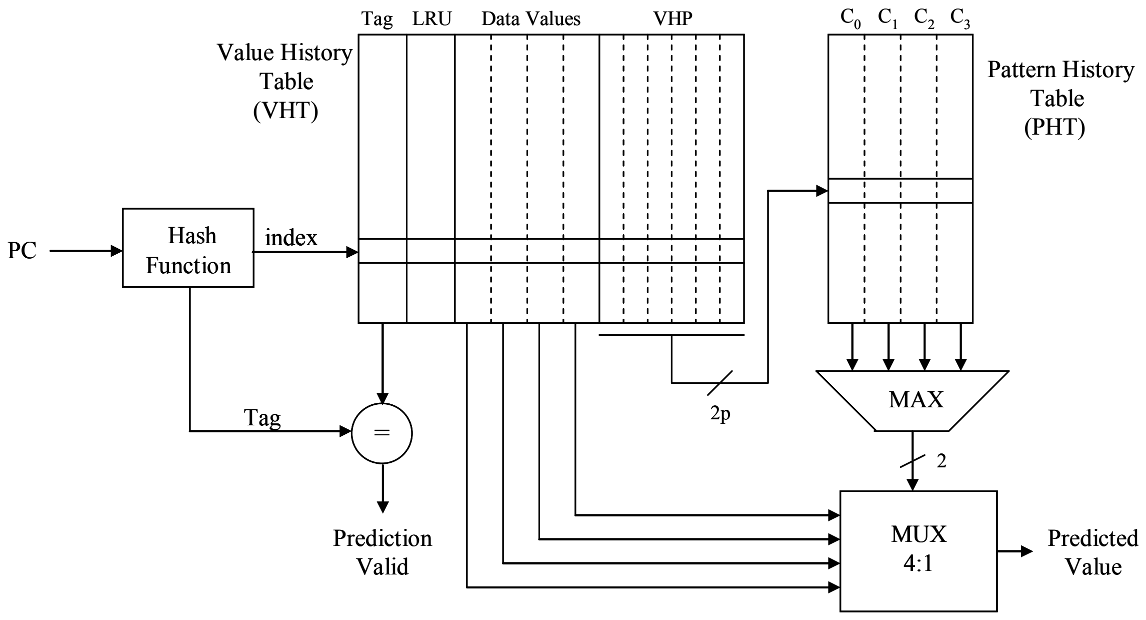


Figure 2.13. Two-level adaptive value predictor

The results of laborious simulations on SPEC benchmarks [SPEC] pointed out that a single predictor cannot capture all the various types of predictability patterns that occur in programs. This suggests that a hybrid scheme might be useful for enabling high prediction accuracy at lower cost [Wan97]. Although the hybrid value predictors can provide more correct predictions than single predictors, they consume more hardware resources. More important, they can waste the limited available hardware resources, since every instruction being predicted occupies a unique entry in each of the component predictors. In [Wan97] Wang and Franklin proposed a hybrid of two-level and stride predictors, with a fixed prioritization of its component predictors (Figure 2.14).

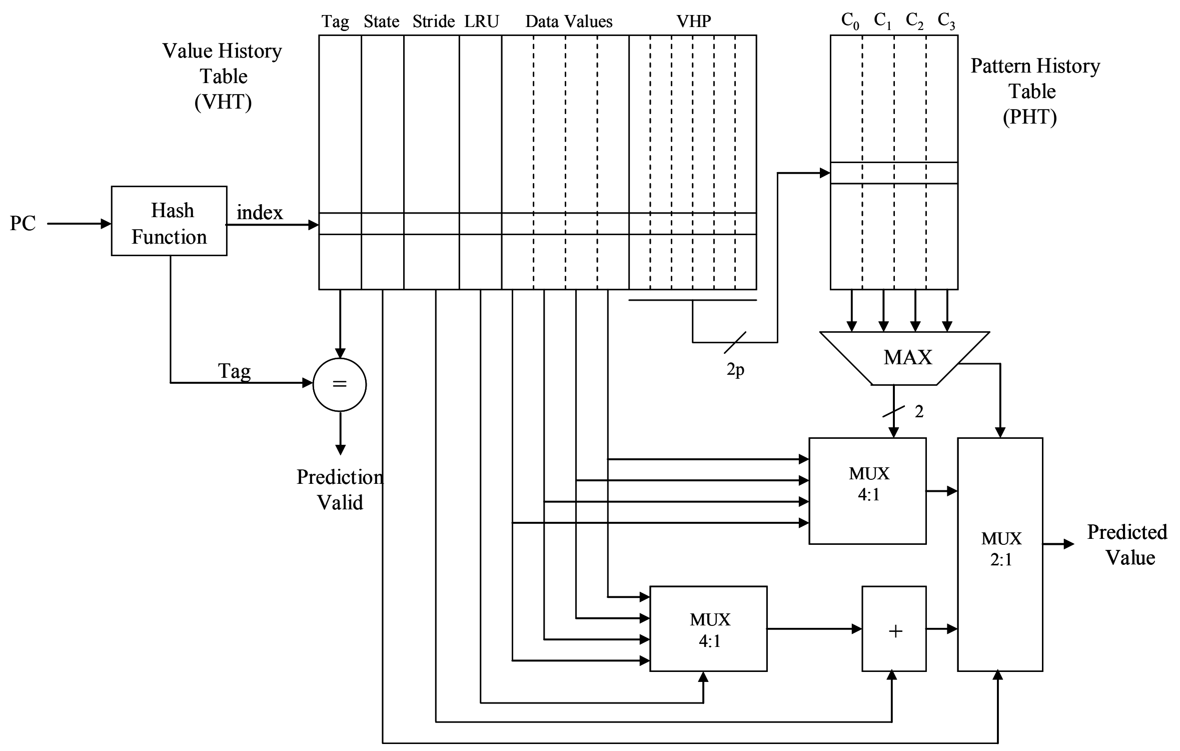


Figure 2.14. Hybrid (two-level, stride) predictor

In the proposed hybrid predictor the two-level predictor has always priority, thus the stride predictor is used only when the two-level predictor does not make a prediction. In our opinion, this fixed prioritization is not optimal but it is quite simple to be implemented; a dynamic prioritization based on some confidences should be better, but in this case a dynamic (adaptive) metapredictor should be necessary in order to select the best predictor at a certain moment. Rychlik et al. in [Ryc98] combined a last-, a stride-, and a two-level value predictor to an overall hybrid value predictor. In order to efficiently use the hardware resources, they provided a dynamic classification scheme that distributes instructions into proper component predictors during run-time, but unfortunately, without prediction accuracy improvements. Wang et al. [Wan99] modified the dynamic classification scheme by reclassifying instructions after they cannot be predicted well by their previously assigned component predictor. Their modification improved this kind of hybrid value predictor.

Calder et al. [Cal99] proposed some selective techniques in order to reduce the pressure on the prediction tables, by filtering the instructions that accessed these resources. The ideal case is to select those dynamic instructions that belong to the critical path. For simplicity, the authors proposed a technique that gives priority for prediction to those instructions that belong to the current longest data dependence chain from the instruction window. Their results show that concentrating only on Loads, is a reasonable filtering approach since Load latencies are responsible for most of the critical paths in integer programs. It is also important to concentrate on Store instructions that can provide significant gains, even if those instructions are hard to predict. For prediction tables of 1024 entries they report an average performance growth of about 11%, comparing with a classical superscalar structure.

Gabbay and Mendelsohn [Gab98] developed a register-file predictor that is the closest predecessor to our register value prediction technique (presented in Chapter 6). They predict the destination value of a given instruction according to the last previously seen value and the stride of its destination register. They have also proposed a dedicated analytical model for determining the speedup involved by a value prediction architecture. Unfortunately the authors did not pursue further this particular idea by systematically developing new register-centric predictors and evaluating them through simulations.

Zhou et al. [Zho03] have studied a new type of value locality, named computational locality in the global value history. They demonstrated that value locality also exists in the global value history, which is the value sequence produced by all dynamic instructions according to their execution order. As a consequence, a novel predictor scheme, the so-called *gDiff* predictor, is proposed to exploit one special and common case of this computational locality, stride-based global locality. Thus, the *gDiff* predicts based on the formula $X_n = X_{n-k} + D$, where D is the stride value. Experiments show that very strong stride-based locality exists within global value histories. Predicting all value-producing instructions, the *gDiff* can achieve a prediction accuracy of 73%.

Other state-of-the-art value predictors, including our original register value prediction techniques, are presented in Chapter 6.

"Prediction is very difficult,
especially about the future"

Niels Bohr

3. Finding Difficult-to-Predict Branches

Since the performances of modern speculative architectures highly depend on branch prediction accuracy, we will further focus on some branch prediction limitations, namely, on hard-to-predict branches. Our first goal is to identify difficult branches in the SPEC 2000 benchmarks [SPEC]. We consider that a branch in a certain context is difficult-to-predict if it is unbiased (the branch behavior is not sufficiently polarized for that certain context) and the taken and not taken outcomes are non-deterministically shuffled. The second goal is to improve prediction accuracy for branches with low polarization rate, introducing new feature sets that will increase their polarization rate and, therefore, their predictability.

3.1. Related Work

Representative hardware and compiler-based branch prediction methods have been developed in recent years in order to increase instruction-level parallelism. Branch prediction is an important component of modern microarchitectures, despite of their deeper pipelines that increased misprediction latency. Therefore, improvements in terms of branch prediction accuracy are essential in order to avoid the penalties of mispredictions. In this section we presented only the works that are most closely related to our proposed novel approach.

Chang et al., introduced in [Cha94] a mechanism that classifies branches into groups of highly biased (mostly-one-direction branches) and unbiased branches, in an attempt to reduce the impact of aliasing. By profiling, branches were classified according to their dynamic taken rate and assigned to the most appropriate dynamic predictor. With their branch classification model the authors showed that using a short history for the biased branches and a long history for the unbiased branches improves the performance of the global history Two-Level Adaptive Branch predictors. In contrast to our work, the authors are classifying branches irrespective of their attached context (local and global histories, etc.) involving thus an inefficient approach. Due to this rough classification the corresponding predictors are not optimally chosen, simply because it is impossible to find an optimal predictor for some classes.

Mahlke et al., proposed in [Mah94] a compiler technique that uses predicated execution support to eliminate branches from an instruction stream. Predicated execution refers to the conditional execution of an instruction based on the value of a boolean source operand – the predicate of the instruction. This architectural support allows the compiler to convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions. Predicated instructions are fetched regardless of their predicate value. Thus, instructions whose predicate value is true are executed normally, whereas instructions whose predicate is false are nullified. Predicated execution offers the opportunity to improve branch handling in superscalar processors. Eliminating frequently mispredicted branches may lead to a substantial reduction in branch prediction misses, and as a result, the performance penalties associated with the eliminated branches are removed. The authors use compiler support for predicated execution based on a structure called hyperblock. The goal of hyperblock formation is to group basic blocks eliminating unbiased branches and leaving highly biased branches. They selected the unbiased branches based on taken frequency distributions.

Their experimental results show that leaving only highly biased branches with predicated execution support, the prediction accuracy is higher.

Nair has first introduced dynamic branch prediction based on path correlation [Nair95]. The basic observation behind both pattern-based and path-based correlation is that some branches can be more accurately predicted if the path leading to these branches is known. Path-based correlation attempts to overcome the performance limitations of pattern-based correlation arising from pattern aliasing situations, where knowledge of the path leading to a branch results in higher predictability than knowledge of the pattern of branch outcomes along the path. Nair proposed a hardware scheme which records the path leading to a conditional branch in order to predict the outcome of the branch instruction more accurately. He adapted a pattern-based correlation scheme, replacing the pattern history register with a *g*-bit path history register which encodes the target addresses of the immediately preceding *p* conditional branches. Ideally, all bits of the target address should be used to ensure that each sequence of *p* addresses has a unique representation in the register. Since such schemes are too expansive to be implemented in hardware, Nair used a simplified scheme which uses a subset of *q* bits from each of the target addresses. Limiting the number of bits from the branch address could result path aliasing – the inability of the predictor to distinguish two distinct paths leading to a branch. Unfortunately, this path correlation scheme does not show any significant improvement over pattern-based correlation [Nair95]. Nair's explanation for this is that for a fixed amount of hardware in the prediction tables, path-based correlation uses a smaller history than pattern-based correlation because the same number of bits represents fewer basic blocks in the path history register than branch outcomes in the pattern history register. Despite this, path based correlation is better than pattern-based correlation on some benchmarks – especially when history information is periodically destroyed due to context switches –, indicating that with a better hashing scheme the pattern correlation schemes could be outperformed.

A quite similar approach is proposed by Vintan and Egan in [Vin99b] – their paper represents the genesis of the original work presented in this chapter. The authors illustrated, based on examples, how a longer history could influence the behavior of a branch (changing it from unbiased to biased). They also showed that path information could also reduce branch entropy. The main contribution of this paper is related to the prediction accuracy gain obtained by extending the correlation information available in the instruction *fetch* stage. Based on trace-driven simulation the authors proved for relatively short global branch history patterns, that a path-based predictor overcomes a pattern-based predictor at the same hardware budget. The main difference, comparing with Nair's approach, is that here the authors are using both the path- and history information in order to do better predictions. They show that a scheme based on this principle performs better than a classical GAp scheme, at the same level of complexity. Particularly useful information has been gleaned regarding the interaction between path length and the number of replacements required in the PHT.

Desmet et al. [Des04] proposed a different approach for branch classification. They evaluated the predictive power of different branch prediction features based on the *Gini-index* metric, which is used as selection measure in the construction of decision trees. Actually, the *Gini-index* is a metric of informational energy and in this case is used to identify the branches with high entropy. In contrast to our work Desmet used as input features both dynamic information (global and local branch history) and static information (branch type, target direction, ending type of taken-successor-basic-block). V. Desmet compared in her PhD thesis [Des06] different branch prediction information, including local/global branch history and path information, from the entropy point of view. An important difference between our approach and Desmet's is that we measured per dynamic branch-context polarization and presented the average percentage of branch contexts having polarization less than 0.95, whereas Desmet measured per branch entropy and presented the average entropy.

Yokota et al. present in [Yok08] the information entropy concept from the branch prediction point of view. They proposed two entropy measures: Branch History Entropy (BHe) representing the entropy of global branch history and Branch Instruction Entropy (BIe) for local branch history. They also defined the entropy of prediction function, called Table Entry Entropy (TEe), as the entropy of the input sequence to a prediction function, and Table Reference Entropy (TRe) representing the number of active table entries determined based on the number of references. The authors measured these entropies in every 1,000,000 branches time-window from the SPEC 2000 benchmarks. They show that the BHe, BIe and TEe entropies are correlated with prediction limits and can derive expected prediction performance. Thus, BHe and BIe show prediction limits by global and local history, while TEe shows theoretical limits on the predictor organization.

In [Hei99a] the authors identified some program constructs and data structures that create "hard to predict" branches. In order to accurately predict difficult branches the authors find additional correlation information beyond local and global branch history. In their approach the prediction table is addressed by a combination between structural information, value information and history of values that are tested in the condition of respective branch. Unlike our work, Heil et al. did not use the path history information in order to do better predictions. Using the proposed prediction method based on data values significantly improves prediction accuracy for some certain difficult branches but the overall improvements are quite modest. However there are some unsolved problems: they tested only particular cases of difficult branches, and also, they did not approach branch conditions with two input values. Their final conclusion suggests that researchers must focus on the strong correlation between instructions producing a value and the branch condition that would be triggered by that certain value.

Chappell et al. [Cha02b] investigated difficult-to-predict branches in a Simultaneous Subordinate Micro-Threading (SSMT) architecture. The authors defined a difficult path as having a terminating branch which is poorly predicted when it executes from that path. A path represents a particular sequence of control-flow changes. It is shown that between 70% and 93.5% of branch mispredictions are covered by these difficult paths, involving thus a significant challenge in the branch prediction paradigm. The proposed solution in dealing with these difficult predictable branches consists in dynamically constructing micro-threads that can speculatively and accurately pre-compute branch outcomes, only along frequently mispredicted paths. Obviously, micro-thread predictions must arrive in time to be useful. Ideally, every micro-thread would complete before the fetch of the corresponding difficult branch. By observing the data-flow within the set of instructions guaranteed to execute each time the path is encountered, it can be extracted a subset of instructions that will pre-compute the branch. The proposed micro-architecture contains structures to dynamically identify difficult paths (Path Cache), construct micro-threads (Micro-Thread Builder) and communicate predictions to the main thread. The proposed technique involves a realistic average speedup of up to 10%, but the average potential speedup through perfect prediction of these difficult branches is about 100%, suggesting the fertility of the idea. Unfortunately the authors did not investigate why these paths, and their associated final branches, are difficult to predict. In other words, a very important question is: why these "difficult paths" frequently lead to mispredictions? We could hope that we already gave the answer in our paper [Vin06], because these "difficult branches" might be, at least partially, exactly the unbiased branches in the sense defined by us during the paragraph 3.2. They could be more predictable even in a single threaded environment, by sufficiently growing history pattern length or extending prediction information, as we show in this chapter. Thus, our hypothesis is that the SSMT environment represents a sufficient solution in order to partially solve these difficult branches, as the authors have shown, but not a necessary one.

Gao et al. have focused in [Gao08] on hard-to-predict branches that depend on long-latency cache-missing Loads. These dependences involve high-penalty mispredictions becoming serious performance obstacles and causing significant performance degradation in executing instructions from wrong paths. The authors describe the correlation existing between these Load-dependent hard-to-predict branches and the addresses of the producer Loads. This correlation is based on the observation that major data structures from some memory-intensive applications (especially those with heavy pointer chasing) tend to remain stable. If a branch is dependent on such stable data, the Load address instead of the Load value is sufficient to determine the branch outcome. Therefore, the branch can be solved once the corresponding Load address is known, much earlier than the Load value. The authors exploit the address-branch correlation through a dedicated scheme consisting in the hardware that dynamically captures Load/Branch pairs and in an Address-Branch Correlation Based Predictor (ABC). In the ABC predictor, stable address-branch correlation information is maintained within a prediction table. When a producer address is known, this prediction table is accessed to see whether the address has stable correlation with a consumer branch. In the case of hit, the branch outcome is predicted and the prediction is used as either a prioritized one when the branch has not been fetched yet or an overriding one when the branch has already been fetched based on the prediction of the primary branch predictor. The experimental results performed on a set of memory-intensive SPEC 2000 benchmarks show that augmenting a 16KB TAGE branch predictor with a 9KB ABC predictor reduces the execution time by 6.3% and the energy consumption by 5.2%.

Another class of hard-to-predict branches are indirect jumps, which are used to implement common programming language constructs such as virtual function calls, switch-case statements, polymorphism and interface calls. Unfortunately, the prediction accuracy of indirect branches is still very low because many indirect branches have multiple targets that are difficult to predict even with specialized hardware. In [Flo05a, Flo04] Florea extracted some typical features and corpus of procedural and object-oriented applications or execution characteristics of desktop applications that generate indirect jumps and calls. Starting from the necessity of implementing new performing indirect branch prediction schemes, but taking into account their hardware feasibility desiderate, the author showed that a modified Target Cache structure, based on confidence mechanism and indexed with extended global correlation information, represents a more simpler and feasible solution that could replace the more complex PPM (prediction by partial matching) predictor. He also determined based on laborious simulations what is the optimum search pattern when different contexts are used. Using profile information, Florea and Vintan developed in [Flo05b] a hybrid predictor with arity-based selection that improves indirect branch prediction accuracy reaching in average 93.77%, which is comparable with a more complex multi-stage cascaded predictor.

Kim et al. proposed in [Kim07] a new technique for handling indirect branches, called Virtual Program Counter (VPC) prediction, which seems to be the first low-cost dynamic mechanism that uses the existing conditional branch prediction hardware to predict the targets of indirect branches, without requiring any program transformation or compiler support. The key idea of the proposed Virtual Program Counter (VPC) technique is to treat an individual indirect branch as a sequence of multiple virtual conditional branches in order to predict them in hardware more accurately (using history information). VPC prediction dynamically de-virtualizes an indirect branch. Unlike compiler-based de-virtualization, VPC prediction can be applied to any indirect branch regardless of the number and locations of its targets. A main advantage is that in this way any existing conditional branch predictor can be used instead of special predictors dedicated to indirect branches (indirect jumps, indirect calls), maintaining thus low costs and complexity. Therefore, further improving conditional branch prediction will involve automatically improving VPC technique. The evaluations showed that VPC prediction improves average performance by 26.7% compared to a commonly used branch target buffer. Unfortunately, VPC prediction is a multi-step iterative algorithm, therefore taking many (multiple) cycles. This essential timing problem is not quite clearly solved in the paper.

3.2. Methodology of Identifying Unbiased Branches

Based on our previous work already published in [Gel06a, Vin06, Oan06, Gel07c] we are presenting in this paragraph the methodology of finding difficult-to-predict branches, as they are defined in our approach. As we have already pointed out in Chapter 2, for each processed dynamic branch, the prediction is achieved based on some binary context information (local or global branch history, the path leading up to the branch, etc.). We have statistically observed that some dynamic branches occurring in certain contexts have a highly unbiased behavior. We consider that a branch in a context is difficult-to-predict if it is unbiased – meaning that the branch behavior (taken / not taken) is not sufficiently polarized for that certain context (local branch history, global history, etc.) – and the taken and not taken outcomes are shuffled. Therefore, we evaluate the impact of unbiased branches on different commonly used features.

We called feature the binary context on *p* bits of prediction information such as local history, global history or path. Each static branch finally has associated *k* dynamic contexts in which it can appear ($k \le 2^p$). A context instance is a dynamic branch executed in the respective context. We introduce the polarization index (P) of a certain branch context as follows:

$$P(S_i) = \max(f_0, f_1) = \begin{cases} f_0, & f_0 \ge 0.5 \\ f_1, & f_0 < 0.5 \end{cases} \tag{3.1}$$

where:

- $S = \{S_1, S_2, ..., S_k\}$ = set of distinct contexts that appear during all branch instances;
- k = number of distinct contexts, $k \le 2^p$, where p is the length of the binary context;
- $f_0 = \dfrac{T}{T + NT}$, $f_1 = \dfrac{NT}{T + NT}$, NT = number of *not taken* branch instances corresponding to context S_i, T = number of *taken* branch instances corresponding to context S_i, $(\forall) i = 1, 2, ..., k$, and obviously $f_0 + f_1 = 1$;
- if $P(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the context S_i is completely biased (100%), and thus, the branch is highly predictable;
- if $P(S_i) = 0.5, (\forall) i = 1, 2, ..., k$, then the context S_i is totally unbiased, and thus, the branch might be not predictable if the *taken* and *not taken* outcomes are shuffled.

If the *taken* and *not taken* outcomes are grouped separately, even in the case of a low polarization index, the branch is predictable. The unbiased branches are not predictable only if the *taken* and *not taken* outcomes are chaotically shuffled, because in this case, the predictors cannot learn their behavior. We introduce the distribution index (shuffle degree) for a certain branch context, defined as follows:

$$D(S_i) = \begin{cases} 0, & n_t = 0 \\ \dfrac{n_t}{2 \cdot \min(NT, T)}, & n_t > 0 \end{cases} \tag{3.2}$$

where:

- n_t = the number of branch outcome transitions, from *taken* to *not taken* and vice-versa ($0 \to 1$ or $1 \to 0$), in a certain context S_i;
- $2 \cdot \min(NT, T)$ = maximum number of possible transitions;
- k = number of distinct contexts, $k \le 2^p$, where p is the length of the binary context;
- if $D(S_i) \to 1, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is "contradictory" (unfavorable cases), and thus its learning is impossible;
- if $D(S_i) \to 0, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is constant (favorable cases), and it can be learned.

As it can be observed in Figure 3.1, we want to systematically analyze different feature sets used by different present-day branch predictors in order to find and, hopefully, to reduce the list of unbiased branch contexts (contexts with low polarization *P*).

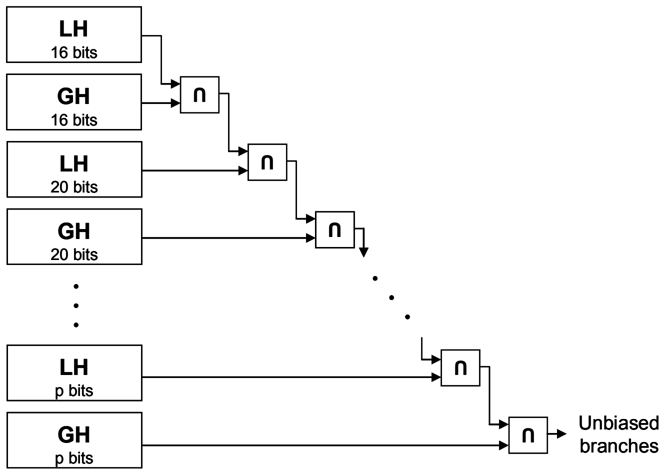


Figure 3.1. Reducing the number of unbiased branches through feature set extension

We approached an iterative methodology: we evaluate and reduce the number of unbiased branches by passing them through successive cascades of different prediction contexts (feature sets). Gradually this list is shortened by increasing the lengths of feature sets (from 16 to 28 bits) and reapplying the algorithm. Thus, the final list of unbiased branches contains only the branches that were unbiased throughout all their contexts, being therefore identified as difficult-to predict. For the final list of unbiased branches we will try to find new relevant feature sets in order to further improve their polarization index and, therefore, the prediction accuracy.

This approach is more efficient than one which repeats each time the algorithm on all branches. Beside producing some unpleasant aspects related to simulation time (days / benchmark) and memory (gigabytes of memory needed), the second method would prove even not very accurate. This is because some of the branches that are not solved by a long context can be solved by a shorter one. Through our iterative approach we avoided the occurrence of false problems extending the context.

In [Oan06] we have studied the polarization of branches but using a little different simulation methodology. We evaluated local history concatenated with global history. The simulation methodology is presented in Figure 3.2.

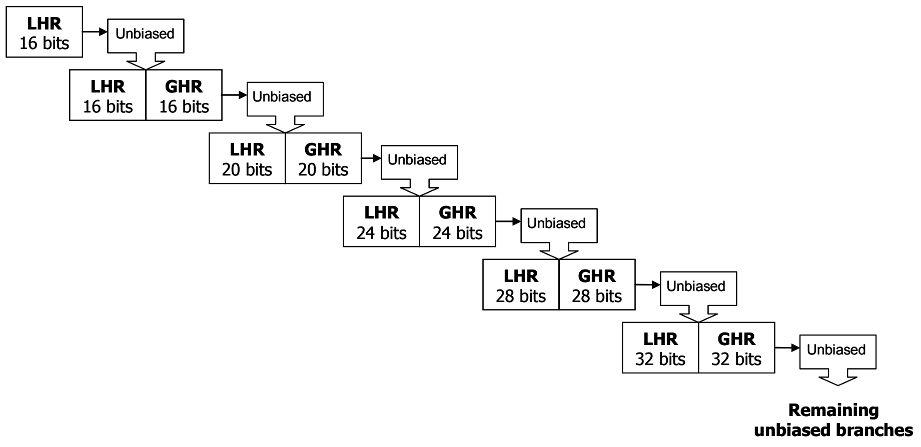


Figure 3.2. Identifying unbiased branches by using the local history concatenated with the global history

Figure 3.3 presents a suggestive example on how unbiased branch contexts can be solved through their extension. We considered that a branch context is unbiased if its polarization index (see relation (3.1)) is less than 0.95. The branch contexts with polarization greater than 0.95 are quite predictable and will obtain relatively high prediction accuracies (around 95%). More details are presented in [Flo07a, Flo07b] on a real example from the Stanford *Perm* benchmark.

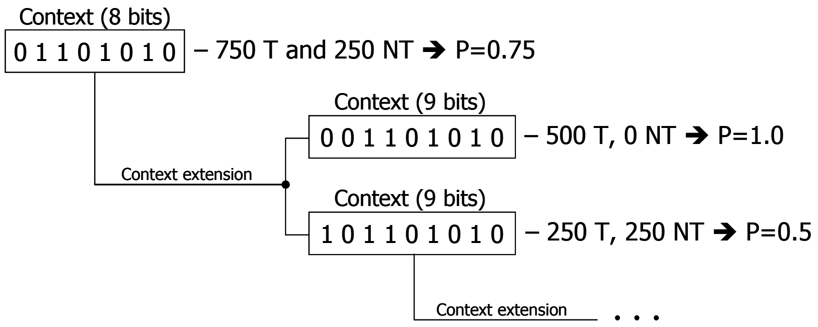


Figure 3.3. The goal of context extension

In our experiments we concentrated only on benchmarks with a percentage of unbiased branch context instances (obtained with relation (3.3)), greater than a certain threshold (T=1%) considering that the potential prediction accuracy improvement is not significant in the case of benchmarks with percentage of unbiased context instances less than 1%. If the percentage of unbiased branch contexts is 1%, even if they would be solved, the prediction accuracy would increase with maximum 1%. This maximum can be reached when the predictor solves all discovered difficult-to-predict branches.

$$T = \frac{NUB_i}{NB_i} = 0.01 \tag{3.3}$$

where NUB_i is the total number of unbiased branch context instances on benchmark i, and NB_i is the number of dynamic branches on benchmark i (therefore, the total number of branch context instances).

3.3. An Analytical Model for Determining Relative IPC Speedup

High branch prediction accuracy is vital especially in the case of multiple instruction issue processors. Further, we assume the analytical models proposed in [Cha94, Vin07], a superscalar processor that ignores stalls like cache misses and bus conflicts, focusing only on the penalty introduced by branch misprediction. Considering as Branch Penalty (*BP*) the average number of cycles wasted for each dynamic instruction due to a branch misprediction, the following relation can be written:

$$BP = C \cdot (1 - Ap) \cdot b \cdot IR \ \ [wasted\ clock / instruction] \tag{3.4}$$

where we denoted:

C = number of penalty cycles wasted due to a branch misprediction;
Ap = prediction accuracy;
b = the ratio of branches (the number of branches reported to the total number of instructions);
IR = the average number of instructions that are executed per cycle (the superscalar factor of architecture; >1).

Further, we computed how many cycles the execution of each instruction take for a real superscalar processor that includes a branch predictor:

$$CPI_{real} = CPI_{ideal} + BP \ \ [clock\ cycle / instruction] \qquad (3.5)$$

where:

CPI_{ideal} = the average number of cycles per instruction considering perfect branch prediction $(Ap = 100\% \Rightarrow BP = 0)$. It is obvious that $CPI_{ideal} < 1$.

CPI_{real} = the average number of cycles per instruction considering real branch prediction $(Ap < 100\% \Rightarrow BP > 0 \Rightarrow CPI_{real} > CPI_{ideal})$.

Therefore, the real processing rate (the average number of instructions executed per cycle) results immediately from the following formula:

$$IR_{real} = \frac{1}{CPI_{real}} = \frac{1}{CPI_{ideal} + BP} \ \ [instruction / clock\ cycle] \qquad (3.6)$$

Relation (3.6) proves the non-linear correlation between processing rate (*IR*) and prediction accuracy (*Ap*). With these metrics, we adapted the model to our results. Further, we use the following notations:

x = the ratio of biased context instances;
$1 - x$ = the ratio of unbiased context instances.

Since Ap_{global} represents a weighted mean among prediction accuracies applied both to biased and unbiased branches, it can be determined the biased prediction accuracy Ap_{biased}.

$$Ap_{global} = x \cdot Ap_{biased} + (1-x) \cdot Ap_{unbiased} \qquad (3.7)$$

Therefore, further we determined how much is influenced the branch penalty (*BP*) by the growth of the context length and what is the speedup in these conditions. For this, we softly modified Chang's model (3.8) [Cha94] by substituting *Ap* with our Ap_{global}, according to relation (3.7). Thus, the penalty introduced by mispredicting biased branches is the term $(1 - Ap_{biased}) \cdot x$, and it is $(1 - x)$ by mispredicting unbiased branches $(Ap_{unbiased} = 0)$.

$$BP = C \cdot (1 - Ap) \cdot b \cdot IR \qquad (3.8)$$

$$BP = C \cdot b \cdot IR \cdot (1 - x \cdot Ap_{biased}) \qquad (3.9)$$

A lower percentage of unbiased branches $(1 - x)$ obtained by extending the context length, leads to a reduction of branch penalty (*BP*) according to (3.9), and implicitly to a greater *IR* according to (3.6). It can be written:

Context (Features Set) *Length* $\nearrow$ => *x* $\nearrow$ => *BP* $\searrow$ => *IR* $\nearrow$ => $\exists$ *Relative Speedup*>0.

$$Relative\ Speedup = \frac{IR(L) - IR(16)}{IR(16)} \geq 0 \qquad (3.10)$$

We computed the relative *IR* speedup according to relation (3.10), where *L* is the feature's length, $L \in \{20, 24, \text{and } 28\}$.

3.4. Experimental Results

All simulation results are reported on 1 billion dynamic instructions skipping the first 300 million instructions from the SPEC 2000 benchmarks [SPEC] and on all instructions from the

INTEL benchmarks [CBP04]. We note with LH(p) a local history of *p* bits, GH(p) a global history of *p* bits, LH(p)-GH(p) their concatenation, and PATH(p) a path consisting in *p* PCs.

3.4.1. Pattern-Based Correlation

We started our study evaluating the branch contexts from SPEC 2000 benchmarks [SPEC] on local branch history of 16 bits. In Table 3.1, for each benchmark we presented the percentages of branch contexts with polarization indexes belonging to five different intervals. The column *Dynamic Branches* contains the number of all dynamic conditional branches for each benchmark, whereas the *Static Branches* column contains the number of static branches. For each benchmark we generated using relation (3.1) a list of unbiased branch contexts, having polarization less than 0.95. We considered that the branch contexts with polarization greater than 0.95 are predictable and will obtain relatively high prediction accuracies (around 0.95), therefore, in these cases we considered that the potential improvement of the prediction accuracy is quite low.

SPEC 2000	Dynamic Branches	Static Branches	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
			[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
bzip	42591123	211	15.86	16.50	8.58	6.94	52.12	11252986	26.42%
gzip	71504537	136	15.08	15.63	11.03	9.50	48.76	27692102	38.73%
mcf	118321124	370	10.06	10.50	8.17	8.52	62.74	6812313	5.76%
parser	85382841	1777	6.67	5.90	3.68	4.56	79.19	17589658	20.60%
twolf	70616018	239	14.49	12.72	6.92	5.34	60.54	31763071	44.98%
gcc	90868660	17248	3.06	2.68	1.72	2.30	90.24	9809360	10.80%
Average	79880717	3330	10.87	10.65	6.68	6.19	65.59	17486582	24.55%

Table 3.1. Polarization rates of branch contexts on local history of 16 bits from the SPEC 2000 benchmarks

The column *Unbiased Context Instances* contains – for each benchmark – the number of unbiased context instances and their percentage reported to all context instances (dynamic branches). As it can be observed in Table 3.1, the relatively high percentages of unbiased branches (at average 24.55%) show high improvement potential from the predictability point of view.

We continue our work analyzing a global branch history of 16 bits only on the local branch contexts that we have already found unbiased for local branch history (see Table 3.1 – last column). In other words, we used a dynamic branch in our evaluations only if its 16 bit local context is one of the unbiased local contexts. In Table 3.2, for each benchmark we presented again the percentages of branch contexts with polarization indexes belonging to five different intervals.

SPEC 2000	Simulated Dynamic Branches		Simulated Static Branches	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
bzip	11252986	26.42%	83	19.34	16.62	14.36	13.80	35.88	9969701	23.40%
gzip	27692102	38.73%	62	8.98	10.09	9.01	10.88	61.04	20659305	28.89%
mcf	6812313	5.76%	25	14.57	11.94	9.25	8.13	56.10	3887052	3.28%
parser	17589658	20.60%	707	6.87	6.98	5.71	6.18	74.26	11064817	12.95%
twolf	31763071	44.98%	132	8.46	7.43	6.39	9.89	67.83	22893014	32.41%
gcc	9809360	10.80%	4923	4.02	4.13	3.14	3.56	85.15	3563776	3.92%
Average	17486582	24.55%	988	10.37	9.53	7.97	8.74	63.37	12006278	17.48%

Table 3.2. Polarization rates of branch contexts on global history of 16 bits evaluating only the unbiased local branch contexts of 16 bits from the SPEC 2000 benchmarks

The *Simulated Dynamic Branches* column contains the number of evaluated dynamic branches and their percentages reported to all dynamic branches. The *Simulated Static Branches* column represents the number of static branches evaluated within each benchmark. We generated for each benchmark using relation (3.1) a list of unbiased branch contexts on local and global history of 16 bits, having polarization less than 0.95. The last column contains the number of unbiased branch context instances and their percentages reported to all dynamic branches. Analyzing comparatively Tables 3.1 and 3.2, we observe that the global branch history reduced the average percentage of unbiased branch context instances from 24.55% to 17.48%. The high percentages of unbiased branch context instances in the case of *bzip, gzip* and *twolf* benchmarks represent a potential prediction accuracy improvement.

We have also analyzed the XOR between the global branch history of 16 bits and the lower part of the branch address (PC bits 18 – 3). We used again only the branch contexts we found unbiased for the previous feature sets (local and global branch history of 16 bits). In other words, we used a dynamic branch in our evaluations only if its 16-bit local context is one of the unbiased local contexts (Table 3.1), and its 16-bit global context is one of the unbiased global contexts (Table 3.2). As our simulations show [Gel06a, Gel07c], this feature does not reduce the percentage of unbiased branches (17.47%) more than the global branch history did (17.48%).

For the determined unbiased branch contexts we are analyzing now if the *taken* and *not taken* outcomes are grouped separately. This is necessary, because if the branch outcomes are not shuffled they are predictable using corresponding two-level adaptive predictors, but if these outputs are shuffled the branches are not predictable. We used relation (3.2) in order to determine the distribution indexes for each unpredictable branch context per benchmark. We evaluated only the unbiased dynamic branches obtained using all their contexts of 16 bits. Table 3.3 shows for each benchmark the percentages of branch contexts with distribution indexes belonging to five different intervals in the case of local branch history, and in the same way, Table 3.4 presents the distribution indexes in the case of global history.

Tables 3.3, 3.4 show that in the case of unbiased branch contexts, the *taken* and *not taken* outcomes are not grouped separately, more, they are highly shuffled.

SPEC 2000	Simulated Dynamic Branches		Simulated Static Branches	Distribution Rate (D) [%]				
				[0, 0.2)	[0.2, 0.4)	[0.4, 0.6)	[0.6, 0.8)	[0.8, 1.0]
bzip	9969752	23.40%	75	6.78	6.45	44.00	16.80	25.98
gzip	20659339	28.89%	51	5.10	5.38	38.70	20.98	29.85
mcf	3887069	3.28%	19	9.21	11.02	46.30	13.32	20.15
parser	11064250	12.95%	483	20.23	9.50	42.44	9.63	18.19
twolf	22893094	32.41%	110	14.63	5.81	43.42	16.71	19.43
gcc	3564489	3.91%	2553	39.07	9.11	33.32	6.00	12.50
Average	12006332	17.47%	548	15.83	7.87	41.36	13.90	21.01

Table 3.3. Distribution rates on local history of 16 bits evaluating only the branches that were unbiased on all their 16 bit contexts (on local and global history) in the SPEC 2000 benchmarks

SPEC 2000	Simulated Dynamic Branches		Simulated Static Branches	Distribution Rate (D) [%]				
				[0, 0.2)	[0.2, 0.4)	[0.4, 0.6)	[0.6, 0.8)	[0.8, 1.0]
bzip	9969752	23.40%	75	0.25	2.94	32.24	37.43	27.13
gzip	20659339	28.89%	51	0.26	2.18	26.45	35.19	35.91
mcf	3887069	3.28%	19	0.27	4.30	37.75	34.38	23.31
parser	11064250	12.95%	483	6.92	14.62	36.63	19.33	22.50
twolf	22893094	32.41%	110	0.84	5.12	26.84	28.44	38.75
gcc	3564489	3.91%	2553	8.10	18.03	38.66	16.06	19.15
Average	12006332	17.47%	548	2.77	7.86	33.09	28.47	27.79

Table 3.4. Distribution rates on global history of 16 bits evaluating only the branches that have all their 16 bit contexts unbiased in the SPEC 2000 benchmarks

The percentage of unbiased branch contexts having highly shuffled outcomes (with distribution index greater than 0.4) is 76.3% in the case of local history of 16 bits (see Table 3.3), and 89.37% in the case of global history of 16 bits (see Table 3.4). A distribution index of 1.0 means the highest possible alternation frequency (with taken or not taken periods of 1). A distribution index of 0.5 means again a high alternation, since, supposing a constant frequency, the taken or not taken periods are only 2, lower than the predictors' learning times. In the same manner, periods of 3 introduce a distribution of about 0.25, and periods of 5 generate a distribution index of 0.15, therefore we considered that if the distribution index is lower than 0.2 the taken and not taken outcomes are not highly shuffled, and the branch's behavior could be learned.

We continued our evaluations extending the lengths of feature sets from 16 bits to 20, 24 and 28 bits, our hypothesis being that the longer feature sets will increase the polarization index and, therefore, the prediction accuracy. Table 3.5 shows the percentages of unbiased branch contexts after each context length extension.

Benchmark	**16 bits**		**20 bits**		**24 bits**		**28 bits**	
	LH(16)	**GH(16)**	**LH(20)**	**GH(20)**	**LH(24)**	**GH(24)**	**LH(28)**	**GH(28)**
bzip	26.42	23.40	15.24	14.62	9.98	8.92	6.40	5.35
gzip	38.73	28.89	24.82	24.07	19.23	18.85	14.95	14.55
mcf	5.76	3.28	2.66	2.58	2.22	2.17	1.83	1.81
parser	20.60	12.95	9.18	8.39	5.95	5.46	3.86	3.56
twolf	44.98	32.41	24.83	22.99	17.42	7.28	5.95	5.67
gcc	10.80	3.92	2.26	1.94	1.35	1.20	~~0.85~~	-
Average	24.55	17.48	13.17	12.43	9.36	7.31	6.60	6.19

Table 3.5. The percentages of unbiased context instances (P<0.95) from the SPEC 2000 benchmarks after each context length extension

As it can be observed, in the case of the *gcc* benchmark, extending the feature set length to 28 bits, the percentage of the unbiased context instances is less than the threshold T of 1% (see relation (3.3)), and thus we eliminated it from our next evaluations. Therefore we consider that the conditional branches from the *gcc* benchmark are not difficult to predict using feature lengths of 28 bits. As a consequence, the results obtained with the *gcc* benchmark are not included in the average results from the last two columns. Despite of the feature set extension, the number of unbiased dynamic branches remains still high (6.19%), and thus, it is obvious that using longer feature sets is not sufficient.

The global history solves at average 2.56% of the unbiased dynamic branches not solved with local history (according to Figure 3.4). The hashing between global history and branch address (XOR) behaves just like the global history, and it does not improve further the polarization rate [Gel06a, Gel07c].

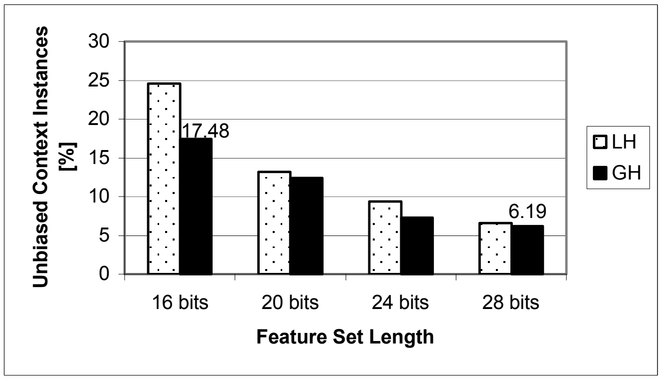


Figure 3.4. Reduction of average percentages of unbiased context instances (P<0.95) in the SPEC 2000 benchmarks by extending the lengths of feature sets

In Figure 3.4 it can be also observed that increasing the branch history, the percentage of unbiased dynamic branches decreases, suggesting a correlation between branches situated at a large distance in the dynamic instruction stream. The results also show that the "ultimative predictibility limit" of history context-based prediction is approximatively 94%, considering biased branches as perfectly predictable and unbiased branches as completely unpredictable. A conclusion based on our simulation methodology is that 94% of dynamic branches can be solved with prediction information of up to 28 bits (some of them are solved with 16 bits, others with 20, 24 or 28 bits).

In [Oan06] we have studied the polarization of branches by evaluating local history concatenated with global history, according to the methodology presented in Figure 3.2. The evaluation results presented in Table 3.6 and Figure 3.5 show that these longer contexts, due to their better precision, have higher polarization index.

Benchmark	LH(16)-GH(0)	LH(16)-GH(16)	LH(20)-GH(20)	LH(24)-GH(24)	LH(28)-GH(28)	LH(32)-GH(32)
bzip	26.42	12.83	7.53	4.70	3.08	2.10
gzip	38.73	24.58	17.84	12.67	9.12	6.16
mcf	5.76	3.09	2.44	2.09	1.78	1.49
parser	20.61	7.42	4.7	3.01	1.98	1.40
twolf	44.98	23.94	12.79	8.28	5.70	3.90
gcc	10.85	2.50	1.41	~~0.88~~	~~0.58~~	~~0.39~~
Average	24.56	12.39	7.80	6.15	4.33	3.01

Table 3.6. The percentages of unbiased context instances in the SPEC 2000 benchmarks, after each context length extension, obtained by using the local history concatenated with the global history

Comparing our results, it is obvious that a certain feature set LH(p)-GH(p) from Table 3.6 is approximatively equivalent in terms of polarization rate with feature set GH(p+4) from Table 3.5. In other words, the same percentage of unbiased context instances is obtained for both LH(p)-GH(p) and GH(p+4) feature sets, but the number of bits in the correlation information is different: (p+p) bits of local and global history and (p+4) bits of global history, respectively. Most of the present-day predictors cannot use very long contexts and also cannot use dynamic reconfigurable history lengths to get the full advantages of the iterative approach.

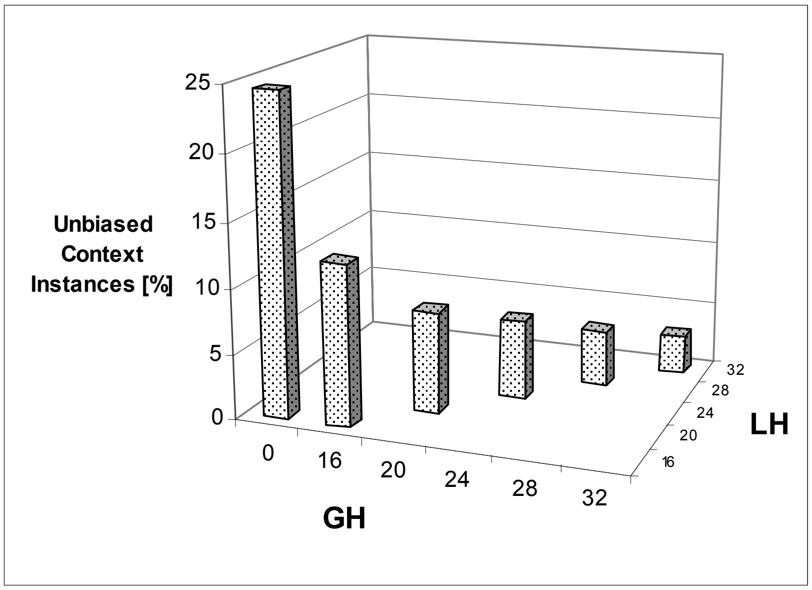


Figure 3.5. The percentages of unbiased context instances in the SPEC 2000 benchmarks, after each context length extension, obtained by using the local history concatenated with the global history

In [Flo07b] we have also detected – on all branches (non-iterative simulation) – the unbiased context instances within the SPEC JVM98 (Java) benchmarks, by extending the global branch history contexts from 8 to 32 bits. The percentage of unbiased branches decreased from 8.87% to 5.80% in these object-oriented Java programs (see Figure 3.6). The reason of the lower percentage of unbiased branches in the SPEC JVM98 benchmarks could be the lower occurance of conditional branches in object-oriented applications compared to procedural applications.

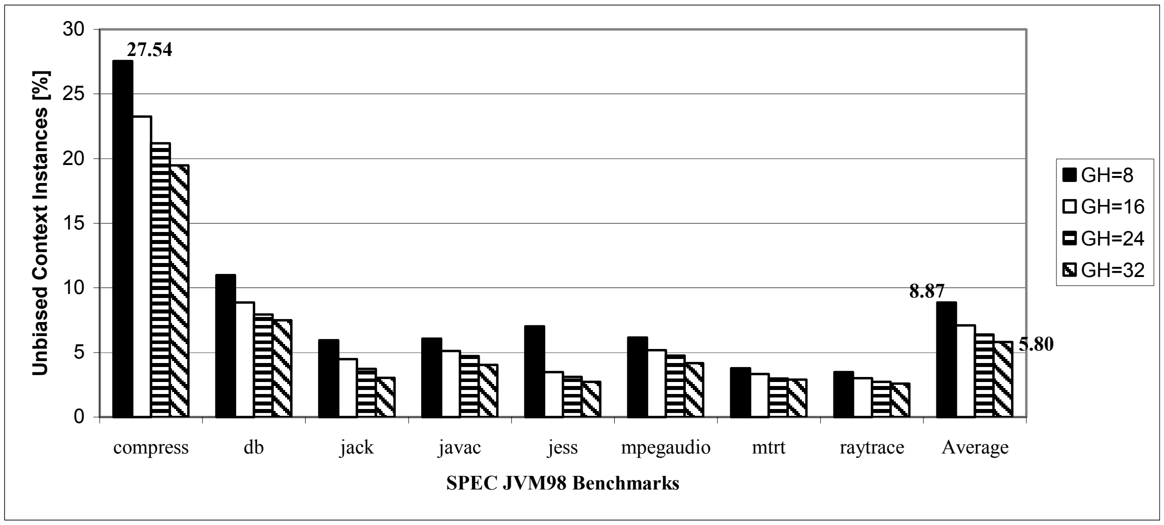


Figure 3.6. The percentages of unbiased context instances in the SPEC JVM98 benchmarks, after each context length extension of global branch history

Taking into account that increasing the prediction accuracy with 1%, the IPC (instructions-per-cycle) is improved with more than 1% (it grows non-linearly) [Yeh92], there are great chances to obtain considerably better overall performances even if not all of the 6.19% difficult predictable branches, from the SPEC 2000 benchmarks, will be solved. Therefore, we consider that this 6.19% represents a significant percentage of unbiased branch context instances, and in the same time a good improvement potential in terms of prediction accuracy and IPC. Focusing on these unbiased branches – in order to design some efficient path-based predictors for them [Nair95, Vin99b] – the overall prediction accuracy should increase with some percents, that would be quite remarkable. The simulation results also lead to the conclusion that as higher is the feature set length used in the prediction process, as higher is the branch polarization index and hopefully the prediction accuracy (Figure 3.4). A certain large context (e.g. 100 bits) – due to its better precision – has lower occurrence probability than a smaller one, and higher dispersion capabilities (the dispersion grows exponentially). Thus, very large contexts can significantly improve the branch polarization and the prediction accuracy, too. However, they are not always feasable for hardware implementation. The question is: what feature set length is really feasable for hardware implementation, and more important, in this case, which is the solution regarding the unbiased branches? In our opinion, as we'll further show, a feasable solution in this case could be given by path-based predictors.

3.4.2. Path-Based Correlation

The path information could be a solution for relatively short history contexts (low correlations). Our hypothesis is that short contexts used together with path information should replace significantly longer contexts, providing the same prediction accuracy. A common criticism for most of the present two-level adaptive branch prediction schemes consists in the fact that they used insufficient global correlation information [Vin99b]. There are situations when a certain static branch, in the same global history context pattern, has different behaviors (taken / not taken), and therefore the branch in that context is unbiased. If each bit belonging to the global history will be associated during the prediction process with its corresponding PC, the context of the current branch becomes more precise, and therefore its prediction accuracy could be better. Our next goal is to extend the correlation information with the path, according to the above idea [Vin99b]. Extending the correlation information in this way, suggests that at different occurrences of a certain static branch with the same global history context, the path contexts can be different.

We started our evaluations regarding the path, studying the gain obtained by introducing paths of different lengths. The analyzed feature consists of a global branch history of 16 bits and the last *p* PCs. We applied this feature only to dynamic branches that we already found unbiased (P<0.95) for local and global history of 16 bits.

SPEC 2000	GH(16)	PATH(1)	PATH(16)	PATH(20)	LH(20)
bzip	23.40	23.35%	22.16%	20.38%	15.24%
gzip	28.89	28.88%	28.17%	27.51%	24.82%
mcf	3.28	3.28%	3.28%	3.20%	2.66%
parser	12.95	12.89%	12.01%	10.95%	9.18%
twolf	32.41	32.41%	31.46%	27.10%	24.83%
gcc	3.92	3.91%	3.56%	3.02%	2.26%
Average [%]	17.48	17.45%	16.77%	15.36%	13.17%
Gain [%]		0.02%	0.70%	2.11%	4.30%

Table 3.7. The gain introduced by the path of different lengths (1, 16, 20 PCs) versus the gain introduced by extended local history (20 bits), in the SPEC 2000 benchmarks

Column GH(16) from Table 3.7, presents for each benchmark the percentage of unbiased contexts using a 16-bit global history. Columns PATH(1), PATH(16) and PATH(20) present the percentages of unbiased context instances obtained using a global history of 16 bits and a path of 1, 16 and 20 PCs, respectively. The last column presents the percentages of unbiased context instances extending the local history to 20 bits (without path). For each feature is presented the average gain opposite to the first column. It can be observed that a path of 1 introduces a not significant gain of 0.2%. Even a path of 20 introduces a gain of only 2.11% related to the more significant gain of 4.30% introduced by an extended local branch history of 20 bits. The results show (Table 3.7) that the path is useful only in the case of short contexts. Thus, a branch history of 16 bits compresses well the path information. In other words, a branch history of 16 bits spreads well the different paths that lead to a certain dynamic branch.

We continue our work evaluating – on all branches (non-iterative simulation) – the number of unbiased context instances (P<0.95) using as prediction information paths of different lengths (*p* PCs) together with global histories of the same lengths (*p* bits). The results are presented in Figure 3.7 where they are compared with the results obtained using only global history.

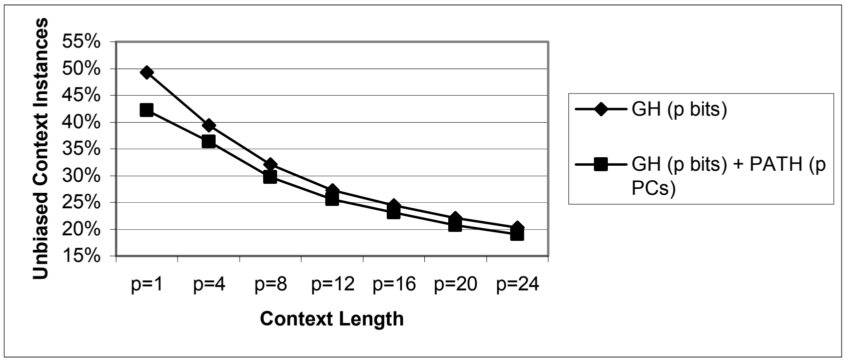


Figure 3.7. The gain introduced by the path for different context lengths – SPEC 2000 benchmarks

Figure 3.7 shows again that the path is relevant for better polarization rate and prediction accuracy only in the case of short contexts and there is only marginal gain with longer history lengths (*p* bits), meaning that a global branch history of more than 12 bits approximates very well the longer path information (*p* PCs).

Desmet shows in her PhD thesis [Des06] that complete path (all branches) is more efficient than simple path (only conditional branches) from the entropy point of view. This is in contradiction with our results presented in Table 3.8, where we compared these types of path from the unbiased branch percentage point of view. This contradiction can be justified by observing the following differences between our measurements:

- Desmet measured per branch entropy and presented the average entropy, while we measured per branch-context polarization and presented the average percentage of branch contexts having polarization less than 0.95;
- Desmet's path consists in the PCs corresponding to the target instructions (as Nair did), while our path consists in the PCs of branches;
- Desmet uses short histories (p=1, 2, 5 PCs), whereas our evaluations were generated on a considerable larger interval (p=1, 4, 8, …, 24 PCs).

As we explain below, paradoxically, the simple path is more rich in information than complete path (for the same number of PCs), justifying our results presented in Table 3.8. Let's consider the following sequence of instructions:

... bne1 ... bne2 ... jr ... bne3 ... bne4 ... bne5=?

If we use a path history of 4 PCs (p=4), then:

- simple path = *bne1*, *bne2*, *bne3*, *bne4*;
- complete path = *bne2*, *jr*, *bne3*, *bne4*.

The unconditional branch *jr* brings less information, because it is always *taken*, and therefore, between *bne2* and *bne3* through *jr* only one path is possible, while through conditional branches two paths are possible. Thus, the path consisting exclusively in conditional branches is better than complete path (see Table 3.8).

Context	**p=1**	**p=4**	**p=8**	**p=12**	**p=16**	**p=20**	**p=24**
GH (p bits)	49.28	39.38	32.08	27.23	24.46	22.08	20.23
GH (p bits) + FullTargetPath (p PCs)	46.74	37.23	30.72	26.50	23.89	21.58	19.88
GH (p bits) + FullPath (p PCs)	43.21	37.03	30.49	26.41	23.86	21.56	19.86
GH (p bits) + CondTargetPath (p PCs)	45.13	36.41	29.76	25.56	23.18	20.77	19.09
GH (p bits) + CondPath (p PCs)	42.19	36.39	29.71	25.51	23.13	20.74	19.01

Table 3.8. Percentages of unbiased branches on the SPEC 2000 benchmarks

We also compared the path consisting in PCs of branches with the path consisting in PCs of target instructions. The path of branch PCs is slightly better, however the difference is quite unsignificant (see Table 3.8).

Further, we present some results obtained applying the same methodology on Branch Prediction World Championship benchmarks – proposed by Intel [CBP04, Loh05a]. We continue to evaluate – on all branches using the non-iterative simulation – paths of different lengths (*p* PCs) used together with global histories of the same lengths (*p* bits). The results are presented in Figure 3.8 where they are compared with the results obtained using only global history. As it can be observed, the results produced (unbiased context instances ratio) by the Intel benchmarks have the same profile like those obtained on the SPEC 2000 benchmarks (Figure

3.7). Actually, rich contexts (long patterns) reduce almost to zero the advantage introduced by using the path information.

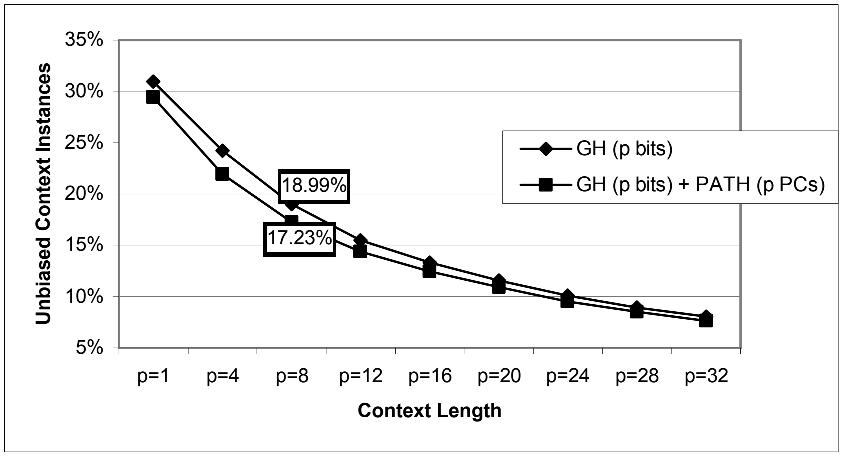


Figure 3.8. The gain introduced by the path for different context lengths – Intel benchmarks

The main difference observed, analyzing Figures 3.7 and 3.8, consists in the different values of these ratios (much higher on SPEC benchmarks) – due to their different characteristics and functions [Loh05a]. However, it must be mentioned that while the SPEC benchmarks were simulated on 1 billion dynamic instructions the Intel benchmarks were entirely simulated, but the total number of dynamic instructions is significantly lower (under 30 million).

3.4.3. Evaluating Relative IPC Speedup Through an Analytical Model

In our simulations presented in [Gel06a] we obtained using the *gshare* predictor [McFar93] the global prediction accuracy Ap_{global} = 93.60% and the accuracy of unbiased branch prediction $Ap_{unbiased}$ = 72.2%. Thus, according to formula (3.7), $0.936 = 0.8253 \cdot Ap_{biased} + 0.1747 \cdot 0.722$, resulting that Ap_{biased} = 0.9813.

Obviously, predicting the unbiased branches with a more powerful branch predictor having, to say, 95% prediction accuracy, determines a gain proportional with ratio of unbiased context instances: $Accuracy\,gain = (0.95 - 0.722) \cdot (1 - x)$. More than that, this accuracy gain involves a processing rate speedup according to (3.4) and (3.6). This gain justifies the importance and the necessity of finding and solving difficult-to-predict branches. Figure 3.9 illustrates the relative *IR* speedup obtained, according to (3.10), by extending the context.

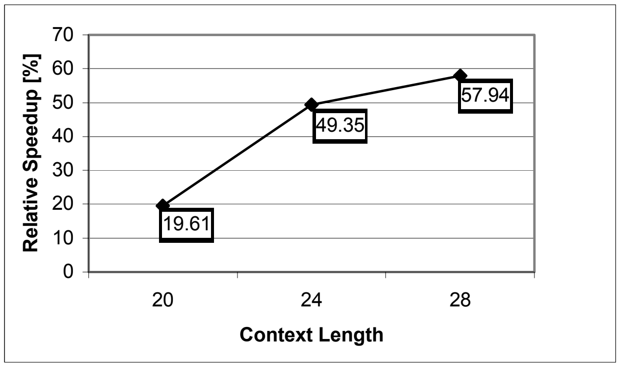


Figure 3.9. The relative *IR* speedup for different increased context lengths reported to the *IR* obtained on 16 bits

The baseline processor model has an IR_{ideal} of 4 [*instruction / clock cycle*] and incorporates a branch predictor with 98.13% prediction accuracy for biased branches. The considered number of penalty cycles wasted due to a branch misprediction in our model is 7. The ratio of simulated branches (the number of simulated branches reported to the total number of simulated instructions) is *b*=8% (see Table 3.1). Figure 3.9 illustrates not only the necessity of a greater number of prediction features to improve the processor performance, but also the necessity of new performing branch predictors that can consider a larger amount of information in making predictions (but whose size does not scale exponentially with the length of the input feature set).

3.5. Summary

We considered that a branch context is unbiased if its polarization index is less than 0.95. In order to reduce the number of unbiased branches, we first increased the lengths of the branch contexts (local/global histories, etc.). We identified and decreased the number of unbiased branches in the SPEC 2000 benchmark suite [SPEC] by passing unbiased branches through successive cascades of different prediction contexts – local history (LH) and global history (GH) – by increasing history information (from 16 to 28 bits). Using a global history context of 16 bits, about 17% of branches are unbiased and unpredictable. This number decreases to about 6% if the context has 28 bits. We consider that this value of 6% is still too high and further investigations are required. The evaluation results also show that the "ultimate predictability limit" of history context-based prediction is about 94%, considering unbiased branches as completely unpredictable. A conclusion based on our simulation results is that about 94% of dynamic branches can be solved with prediction information of up to 28 bits. We have also analyzed the path information and we concluded that a global branch history of more than 12 bits compresses well the path information, and therefore, the gain introduced by the path is not significant.

Summarizing the statistics reported on the SPEC 2000 benchmarks, 546 static branches generate 77,683,129 dynamic instances at average (142,120 instances / static branch). Focusing now on those detected unbiased (with LH=28 bits and GH=28 bits), 113 static branches generate 4,376,664 dynamic instances at average (38,731 instances / static branch). Therefore the unbiased branches are generated by few static branches having many dynamic instances. As a consequence, taking into account the enormous number of dynamic unbiased branches per a static branch, an adequate predictor has plenty of time to learn its behavior. The real problem is to find the right prediction information that changes such unbiased branches into biased ones.

The next chapter investigates the predictability of this remaining 6% of identified unbiased branches and proposes some new, more correlated prediction information in order to increase their prediction accuracy.

"The only relevant test of the validity of a hypothesis is comparison of prediction with experience"

Milton Friedman

4. Predicting Unbiased Branches

In Chapter 3 we showed that the percentages of difficult branches are quite significant (at average between 6% and 24%, depending on the different used prediction contexts and their lengths). This chapter presents some important present-day branch predictors and some condition-history-based branch predictors proposed by us in [Gel07a, Gel07b, Gel07c], all of them being used to evaluate, in terms of prediction accuracy, the unbiased branches identified in Chapter 3.

4.1. Related Work

4.1.1. Branch Prediction Based on Data Value Information

In this section we analyze different proposed techniques that are exploiting the correlation between data values and branch outcomes. An important disadvantage of the approaches that are using the branch register values directly is that these values are rarely available, and therefore, they must be predicted. In general, value prediction is applied wisely due to the relatively high misprediction cost and low prediction accuracy.

In [Che03] the authors proposed a new approach, called ARVI (Available Register Value Information), in order to predict branches based on partial register values along the data dependence chain leading up to the branch. The authors show that for some branches the correlation between such register value information and the branch's outcome can be stronger than either history or path information. Thus, the main idea behind the ARVI predictor is the following: if the essential values in the data dependence chain, that determine the branch's condition, should be identified, and those values have occurred in the past, then the branch's outcome should be known. If the values involved in the branch condition are the same as in a prior occurrence then the outcome of the branch will be the same, too. Thus, if the branch's register values are available then a lookup table can provide the last branch's outcome occurred with the same values. Unfortunately, the branch's register values are rarely available at the time of prediction. However, if values are available for registers along the dependence chain that leads up to the branch, then the predictor can use these values to index into a table and reuse the last behavior of the branch occurred in the same context. Therefore, instead of relying only on branch history or path, the ARVI predictor includes data dependent registers as part of the prediction information. The ARVI predictor uses a Data Dependence Table (DDT) to extract the registers corresponding to instructions along the data dependence chain leading up to the branch. The branch's PC and the identifiers of the data dependent registers are hashed together and used to index the prediction table. The values of the data dependent registers are hashed together and used as a tag to distinguish the occurrences of the same path having different values in the registers. Thus, the ARVI predictor uses both path and value-based information to classify branch instances. A two-level predictor using ARVI at the second level achieves a 12.6% overall IPC improvement over the state-of-the-art two level predictors, for the SPEC'95 integer benchmarks. In our opinion, if dynamic branches that are unbiased in their branch history or path contexts [Vin06] are biased in their value history context, the benefit could be remarkable. An analysis in this sense will be effectuated in this chapter.

Z. Smith in his work [Smi98] showed on the SPEC'95 benchmarks that the majority of mispredicted branches come from few static branches. Therefore, he identified "bad" branches based on the distribution of mispredictions – a function of the number of mispredictions per branch using the *gshare* predictor with 12 history bits. An analysis of branches having a relatively high number of mispredictions shows that they could be really less predictable but without importance due to their relatively low number of dynamic instances, and, on the other hand, some of them could be predictable because the number of mispredictions is, however, far less then the number of branch's dynamic instances. Consequently, there is no strong correlation between branch's predictability or global prediction accuracy and the distribution of mispredictions. In order to increase the predictability of mostly mispredicted branches, Smith evaluated the possibility to predict branch outcomes based on a value history. The idea is to use a context-based predictor whose prediction table is indexed by a register value instead of the XOR between the PC and global history as in *gshare*. Only the first (non-immediate) branch operand is used as prediction context, because, as the author shows, the majority of branches have the second operand equal with zero. However, using both branch operands as prediction information could be better. Using a history of only 2 values together with the value of the outer loop counter (an iteration counter associated to the enclosing loop's branch), Smith obtained a branch prediction accuracy of 93.4%.

In [Hei99b] the authors observed that many important branches that are hard to predict based on branch history and path become easily predictable if data-value information is used. First, they analyzed a technique called *speculative branch execution* that uses a conventional data-value predictor to predict the input values of the branch instruction and, after that, executes the branch instruction using the predicted values. The main disadvantage of this method consists in the relatively high prediction latency, because the operand-value prediction is followed by the pre-calculation of the branch's condition. Therefore, they proposed a Branch Difference Predictor (BDP) which simply holds a history of branch source register differences and uses it in the prediction process. Consequently, the value history information is used directly for branch prediction, reducing thus the latency. Since branch outcomes are determined by subtracting the two inputs, the branch source differences correlate very well with the branch outcomes. The branch difference history is maintained per static branch in a Value History Table (VHT) and it is retrieved using the branch's PC. By using branch differences, the number of patterns is very high, since a certain static branch instruction may produce many values. Thus, predicting all branches through this method leads either to excessive storage space requirements or to significant table interference. Therefore, in their prediction mechanism, only the difficult branches are predicted based on the branch source differences using the Rare Event Predictor (REP), whereas the majority of branches are predicted using a conventional predictor (e.g. *gshare*). They considered that a branch is difficult if it is mispredicted by the conventional predictor. When a branch instruction occurs, the VHT and the REP are accessed in parallel with the PC and global branch history. If the value difference history matches a REP tag, then the REP provides the prediction. If the REP does not contain that certain pattern, the conventional branch predictor generates the prediction. Their results show that the majority of prediction accuracy improvement is gained by using a single branch difference, while adding a second or third difference results in little additional improvement. The BDP reduces the misprediction rate by up to 33% compared to *gshare* and up to 15% compared to *Bi-Mode* predictors, in the SPEC'95 integer benchmarks. A first important difference between Heil's approach and ours is that we are focusing on unbiased branches identified in Chapter 3 (branches with low polarisation degree that tend to shuffle between taken and not taken) instead of Heil's difficult branches (those mispredicted by a conventional predictor). However, the main difference is that we correlate branch outcome with the sign of the condition's difference whereas Heil et al. correlate it with the value of the condition's difference. As we'll further show, using signs instead values involves better prediction accuracies and less storage necessities. Furthermore, we use a sign-history of up to 256 condition differences in contrast to the value-history of up to 3 condition differences exploited in [Hei99b]. Another important difference between the two approaches is the architectural one, since we predict branches using some modified state-of-the-art Markov and neural predictors.

Thomas et al. [Tho03] introduced new branch prediction information that consists in *affector branches*. They identify for each dynamic branch from a long global history, a set of branches called *affectors*, which control the computation that directly affect the source operands of the current dynamic branch. Since affectors have a direct effect on the outcome of a future branch, they have a high correlation with that branch. The affector information is represented as a bitmap having all bits corresponding to the affector branches set to 1 and those of non-affectors set to 0. The affector information is maintained based on runtime dataflow information for each architectural register as entries in an Affector Register File (ARF). When the processor encounters a conditional branch, all entries in the ARF are shifted left by one bit and the least significant bit is made 0. When a register-writing instruction occurs, the ARF entries corresponding to the source registers are ORed together and written into the ARF entry of the destination register with the least significant bit set to 1. Thus, the affector information for the destination register is generated as a union of the affector histories corresponding to the source registers, whereas the least significant bit, set to 1, marks the last branch from the global history as an affector. The affector branch information for a branch instruction is inherited from the affector information corresponding to its source registers. Therefore, when a prediction is to be made for a certain branch, the affector information of its source registers are ORed together in order to determine its affector branches. The authors also proposed different prediction schemes that use the affector branch information.

Constantinides et al. [Con04] presented a method of detecting instruction-isomorphism and its application to dynamic branch prediction. A dynamic instruction is considered isomorphic if its component graph is identical with the component graph of an earlier executed dynamic instruction. The component graph of a dynamic instruction can include information about the instruction, its dynamic data dependence graph and its input data. Two cases of instruction isomorphism can be distinguished: isomorphic-equality and pseudo-isomorphism. In the case of isomorphic equality the instructions are isomorphic and they have the same outputs, whereas in the pseudo-isomorphism case, the instructions are isomorphic but their outputs are not equal. The isomorphism detection process is preceded by component-graph transformations that may convert non-isomorphism to isomorphic-equality by removing information from the component graph that does not affect the outcome of the instruction. The isomorphism detection mechanism contains four units: the Register-Signature File (RSF), the Component Graph Encoding/Transformation mechanism (CGET), the Memory Signature File (MSF) and the Isomorphism Detection Table (IDT). The RSF is accessed with the source register names to read the signatures – encoded component graphs. The CGET takes the instruction's source signatures and creates a new signature, which represents the instruction's encoded/transformed component-graph. If the instruction writes to a register, the new signature is written into the RSF entry corresponding to the destination register. To determine if an instruction is isomorphic with a previously executed instruction, its signature – produced by CGET – is used to access the IDT. The IDT also returns the branch direction in the case of branch prediction. Isomorphism detection must wait for decoded instruction information and, thus, the isomorphic branch predictor has relatively high latency. Therefore, Constantinides et al. proposed a hybrid branch prediction mechanism composed by a fast conventional predictor and a slower isomorphic-based predictor. The isomorphic prediction – available few cycles after the conventional prediction – is used to validate and possibly override the prediction provided by the conventional predictor.

In [Gon99] and [Gon01] González et al. introduced a *branch prediction through value prediction* unit (BPVP) that pre-computes branch outcomes by predicting their input values. Since, the accuracy of value predictors is lower than that of the conventional branch predictors, speculative branch pre-computation must be applied selectively. Therefore, they proposed a hybrid branch prediction mechanism involving a correlating branch predictor (e.g. *gshare*) and a

BPVP that uses a conventional value predictor. The value predictor is used together with an Input Information Table (IIT) and an additional logic to detect the instructions that generate the branch's inputs. Each architectural register has an entry in the IIT that stores the PC of the latest instruction having the corresponding register as destination and the value computed speculatively by the latest *compare* instruction having the corresponding register as destination. The *compare* instructions are speculatively pre-executed according to their predicted inputs and the speculative results are stored in the IIT. The mechanism has different behaviors depending on the branch that is predicted. In the case of branches with inputs produced by arithmetic or Load instructions, the IIT is accessed with the source register names to read the PCs of the latest instructions that had as destination the branch's source registers (detection of the instructions that produces the branch inputs). The PCs are used to access the value predictor that predicts the inputs of the branch. The branch's outcome is speculatively pre-computed based on the predicted inputs. In the case of branches with inputs produced by *compare* instructions, the IIT is accessed with the source register names to read the comparison's speculative result. The outcome of the branch is speculatively pre-computed based on this speculative comparison result. The BPVP-*gshare* predictor achieves a speedup of 8% over the 2bit-gshare predictor. The instruction-centric value prediction within the BPVP should be replaced with register-centric value prediction [Vin05a] (presented in Chapter 6), reducing the complexity, hardware costs and power consumption. Thus, branches should be pre-computed speculatively based on their input values predicted with our most effective register-centric value predictor (a hybrid of two-level and stride).

In [Rot99] call targets are correlated with the instructions that produce them rather than with the call's global history or the previous branch targets. The proposed approach pre-computes virtual function call's (v-call) targets. V-call targets are hard to predict even through path-based schemes that exploit the correlation between multiple v-calls of the same object reference. Object oriented programming increases the importance of v-calls. The proposed technique dynamically identifies the sequence of instructions that computes a v-call target. Based on this instruction sequence is possible to pre-calculate the target before the actual v-call is encountered. This pre-calculation can be used to supply a prediction. The approach reduces v-call target mispredictions with 24% over a path-based two level predictor.

In [Vin03] the authors proposed to pre-compute branches by determining a branch outcome as soon as its operands are available. The instruction that produced the last branch source operand would also trigger the branch condition estimation. As soon as this operation completed, the branch outcome could be immediately resolved. Similarly to branch history prediction, branch information is cached into a "prediction table" (PT). Each PT entry has the following fields: *TAG* (the lower part of the PC), *PC1* and *PC2* (the PCs of the instructions that produced the branch operand values), *OPC* (the opcode of the branch), *nOP1* and *nOP2* (the register names of the branch operands), *PRED* (for the branch outcome) and a *LRU* field (Least Recently Used). The register file has two additional fields for each register: *LP* (the PC of the last producer) and *RC* (a reference counter which is incremented by each instruction that modifies a register linked by a branch instruction stored in the PT and is decremented when the corresponding branch instruction is evicted from the PT). The PC of any non-branch instruction that modifies at least one register is recorded into the supplementary *LP* (Last Producer) field of its destination register. The first issue of a particular branch in the program is predicted with a default value (not taken). After the branch's execution, a PT entry is allocated and updated. Every time after a non-branch instruction – having the corresponding *RC* field greater than 0 – is executed, the *PC1* and *PC2* fields from the PT are searched upon its PC. When a hit occurs, the branch stored in that PT entry is executed and the outcome is stored into the *PRED* bit. When the branch is issued, its outcome is found in the PT, as it was previously computed, and thus its behavior is perfectly known before execution. Even though this concept would provide (almost) perfect prediction accuracy, there was a heavy timing penalty in the case when a branch instruction was dynamically executed immediately after the last source operand has been computed, in fact this is a common case. Based on the pre-computing branch concept [Vin03]

Aamer et al. presented in [Aam03] a study regarding the number of instructions occurred between the execution of the instruction that produced the last operand of a branch and the execution of that branch. Their simulations show that the average distance between the last source producer and branch is less than the ideal theoretical distance. If the operand producer instruction is too close to the corresponding branch then the branch would have to postpone processing for a few cycles, until the operand producer instruction is finished. For these branches a BTB can be used, improving thus the performance. Thus, the branch outcomes can be obtained far enough in advance so that some performance improvement can be still achieved.

Aragón et al. presented in [Ara01] a new approach to improve branch predictors: *selective branch prediction reversal*. The main idea is that many branch mispredictions can be avoided if they are selectively reversed. Therefore, they proposed a Branch Prediction Reversal Unit (BPRU) that reverses predictions of branches likely to be mispredicted, based on the path leading to the branch (including the PC of the input producers) and the predicted values of the branch inputs. The BPRU uses the previously presented BPVP-*gshare* hybrid branch predictor [Gon99] and a Reversal Table (RT). Each entry of the RT stores a reversal counter implemented as an up/down saturating counter, and a tag. When a branch is predicted, the RT is accessed by hashing together the PCs of its input producers, the predicted input values and the path leading to the branch. The most significant bit of the counter indicates if the predicted branch outcome must be reversed. When the correct branch outcome is available, the corresponding RT entry is updated by incrementing the reversal counter if the preliminary branch outcome was correct and decrementing the counter otherwise. The experimental results show average speedups of 6% over the original BPVP-*gshare* and of 14% over the 2bit-gshare predictor.

4.1.2. State-of-the-Art Branch Predictors

Dynamic branch prediction with neural methods, was first introduced by Vintan [Vin99a], and further developed by many researchers, especially by Daniel Jiménez [Jim01a]. Despite the neural branch predictor's ability to achieve very high prediction rates and to exploit deep correlations at linear costs, the associated complexity due to latency, large quantity of adder circuits, area and power are still obstacles to the industrial adoption of this technique. Anyway, the neural methods seem to be successful for future microprocessors taking into account that they were already implemented in Intel's IA-64 simulators. Jiménez and Lin [Jim01a] proposed a two-level scheme that uses fast single-layer perceptrons instead of the commonly used two-bit saturating counters. The branch address is hashed to select the perceptron, which is then used to furnish a prediction based on global branch history. The perceptron's prediction- and learning algorithm was presented in Section 2.2.

A branch may be linearly inseparable as a whole, but it may be piecewise linearly separable with respect to the distinct associated program paths. More precisely, the path-based neural predictor combines path history with pattern history, resulting superior learning skills to those of a neural predictor that relies only on pattern history. To generate a path-based neural prediction [Jim03c], the correlations of each component of the path are aggregated. This aggregation is a linear function of the correlations for that path. Since many paths are leading to a branch, there are many different linear functions for that branch, and they form a piecewise-linear surface separating paths that lead to predicted taken branches from paths that lead to predicted not taken branches. The piecewise linear branch prediction [Jim05], is a generalization of perceptron branch prediction [Jim01a], which uses a single linear function for a given branch, and path-based neural branch prediction [Jim03c], which uses a single global piecewise-linear function to predict all branches. The piecewise linear branch predictors use a piecewise-linear function for a given branch, exploiting in this way different paths that lead to the same branch in order to predict – otherwise linearly inseparable – branches. The piecewise linear branch predictors exploit better the correlation between branch outcomes and paths, yielding an IPC improvement of 4% over the path-based neural predictor [Jim05]. In the weight selection mechanism of the idealized piecewise linear branch predictor, the weight W_{bpg} corresponds to branch b ($1 \le b \le B$), its global history bit g ($1 \le g \le G$) and the p^{th} PC ($1 \le p \le P$) from its path. The Idealized Piecewise Linear Branch Predictor uses dynamically adjusted history lengths [Jim05]. The predictor counts the number of static branches whose bias magnitude, noted $|W_0|$, exceeds 2. If this number exceeds 300, then the predictor switches to lower global and local history lengths, otherwise, it uses higher global and local history lengths. This heuristic is applied after 300,000 branches have passed.

Related to Jiménez's research, we gave an original interpretation of his dynamically adjusting history length mechanism [Jim05], through our previously introduced "unbiased branches" concept [Gel06a, Vin06, Oan06]. Thus, his heuristics work as follows: if more than 300 "relatively biased" branches are encountered (branches having $|W_0|>2$), then it switches to lower global/local history length. Otherwise (meaning that there were encountered many "perfectly unbiased" branches, having $|W_0|\le2$) it switches to higher global/local history length. From our point of view, this is justified by the fact that increasing history length reduces the number of unbiased branches as we have already shown.

A conventional path-based neural predictor achieves high prediction accuracy, but its very deeply pipelined implementation makes it both a complex and power-intensive component, since for a history length of p it uses – to store the weights – p separately indexed SRAM arrays organized in a p-stage predictor pipeline. Each pipeline stage requires a separate row-decoder for the corresponding SRAM array, inter-stage latches, control logic and checkpointing support, all of this adding power and complexity to the predictor. Loh and Jiménez proposed in [Loh05c] two techniques to address this problem. The first decouples the branch outcome history length from the path history length using shorter path history and a traditional long branch outcome history. In the original path-based neural predictor, the path history was always equal to the branch history length. The shorter path history allows the reduction of the pipeline length, resulting in decreased power consumption and implementation complexity. The second technique uses the bias-weights to filter out highly-biased branches (mostly always taken or mostly always not taken branches), and avoids consuming update power for these easy-to-predict branches. For these branches the prediction is determined only by the bias weight, and if it turns out to be correct, the predictor skips the update phase which saves the associated power. The proposed techniques improve the prediction accuracy with 1%, and more important, reduce power and complexity by decreasing the number of SRAM arrays, and reducing predictor update activity by 4-5%. Decreasing the pipeline depth to only 4-6 stages reduces the implementation complexity of the path-based neural predictor.

Tarjan and Skadron introduced in [Tar05] the hashed perceptron predictor, which merges the concepts behind the *gshare* [McFar93] and path-based perceptron predictors [Jim03c]. The previous perceptron predictors assign one weight per local-, global- or path branch history bit. This means that the amount of storage and the number of adders increases linearly with the number of history bits used to make a prediction. One of the key insights of Tarjan's work is that one-to-one ratio between weights and number of history bits is not necessary. By assigning a weight not to a single branch but a sequence of branches (hashed indexing), a perceptron can work on multiple partial patterns making up the overall history. The hashed indexing consists in XORing a segment of the global branch history with a branch address from the path history. Decoupling the number of weights from the number of history bits used to generate a prediction allows the reduction of adders and tables almost arbitrarily. Using hashed indexing, linearly inseparable branches which are mapped to the same weight can be accurately predicted, because each table acts like a small *gshare* predictor [McFar93]. The hashed perceptron predictor improves accuracy by up to 27.2% over a path-based neural predictor.

Loh and Jiménez introduced in [Loh05b] a new branch predictor that takes the advantage of deep-history branch correlations. To maintain simplicity, they limited the predictor to use conventional tables of saturating counters. Thus, the proposed predictor achieves neural-class prediction rates and IPC performance using only simple PHT (pattern history table) structures. The disadvantage of PHTs is that their resource requirements increase exponentially with branch history length (a history length of p requires 2^p entries in a conventional PHT), in contrast to neural predictors, whose size requirements increase only linearly with the history length. To deal with very long history lengths, they proposed a Divide-and-Conquer approach where the long global branch history register is partitioned into smaller segments, each of them providing a short branch history input to a small PHT. A final table-based predictor combines all these per-segment predictions to generate the overall decision. Their predictor achieves higher performance (IPC) than the original global history perceptron predictor, outperforms the path-based neural predictors, and even achieves an IPC rate equal to the piecewise-linear neural branch predictor. Using only simple tables of saturating counters, avoids the need for large number of adders, and in this way, the predictor is feasible to be implemented in hardware.

Seznec recently developed perhaps the most powerful idealistic/realistic branch predictors. His idealistic hybrid GTL predictor is composed by three distinct branch predictors (TAGE, GEHL and Loop) exploiting very deep history correlations and a metapredictor derived from the skewed predictor [Sez07a]. The GEHL predictor is the main component being the most accurate. In this chapter we also used it in predicting our unbiased branches. In [Sez07b] is presented a realistic branch predictor called L-TAGE consisting of a 13-component TAGE predictor and a Loop predictor. Both these predictors won the 2nd Championship Branch Prediction [CBP06] organized by Intel Co., obtaining, at average, 3.314 mispredictions/KI. Even if these predictors are the best known branch predictors they are using the same limited prediction information (branch address, global/local histories and path) that is insufficient for reducing unbiased branches entropy and for accurately predicting them.

In [Gao06] the authors initially implemented a PPM-based branch predictor using as context the global branch history. They associated a signed saturating prediction counter ranging between [-4, 4] to each PC-history pair. The counter was incremented if the branch outcome was taken and decremented otherwise. When both the branch address and history pattern were matched, the corresponding counter provided the prediction. In the case of multiple matches for a branch with different history lengths, the prediction counter corresponding to the longest history was used. However, as they show, the longest history match may not be the best choice, and, therefore, they proposed another scheme called PPM with the confident longest match that uses the prediction counter as a confidence measure. This scheme generates a prediction only when the counter is a non-zero value. The authors observed that in the case of multiple matches with different history lengths, the counters may not agree with each other and different branches may favor different history lengths. Thus, the most important scheme introduced by Gao and Zhou in this paper, predicts branch outcomes by combining multiple partial matches through an adder tree. The *Prediction by combining Multiple Partial Matches* (PMPM) algorithm selects up to L confident longest matches and sums the corresponding counters to furnish a prediction. A bimodal predictor is used to predict branches that are completely biased (either always taken or always not taken) and the PMPM predictor is used to furnish a prediction when a branch is not completely biased. The realistic PMPM predictor has seven global prediction tables indexed by the branch address, global history and path, and also has a local prediction table indexed by the branch address and local history. When the PMPM is accessed for prediction, up to four counters from the global history tables are summed with the counter from the local prediction table, if there is a hit. If the sum is zero, the bimodal predictor is used. Otherwise the sign of the sum provides the prediction. The prediction counter from the bimodal prediction table is always updated. The prediction counter from the local prediction table is always updated in the case of hit, while the counters of the global prediction tables that have been included in the summation are updated only when the overall prediction is wrong or the absolute value of the sum is less than a certain threshold. Their results show that combining multiple partial matches provides higher prediction accuracy than a single partial match, decreasing the average misprediction rate to 3.41%. A first important difference between the approach presented in [Gao06] and our *branch difference prediction by combining multiple partial matches* developed in this chapter is that we are focusing on the unbiased branches identified in Chapter 3 (branches with low polarisation degree that tend to shuffle between taken and not taken) instead of "not fully biased" branches. The authors defined a "fully biased" branch as a branch in a certain dynamic context having set its attached bias counter to a maximum value (the counter is incremented each time that branch has a biased behavior and decremented otherwise). Probably it would be better to say "highly biased" branch instead of "fully biased", meaning that it was highly biased (maximum counter) during the "last" processing period (maximum counter at the current prediction moment). However, the main difference is that they used global branch history, whereas we used global and local branch difference history. Another important difference consists in how the multiple Markov predictions are combined: we used majority vote (more efficient for our approach) instead of the adder tree used by Gao and Zhou.

In [Sri06] the authors proposed a hybrid branch prediction scheme that employs two PPM predictors, one predicts based on local branch history and the other predicts based on global branch history. For both the local and global PPM predictors, if the local and global history were not matched, then shorter patterns are searched, and so on, until a match is found. When a pattern match occurs, the outcome of the branch that succeeded the pattern during its last occurrence is returned as prediction. The two independent predictions are combined through a perceptron. The output of the perceptron is computed as $Y=W_0 + W_1P_L + W_2P_G$, where the inputs P_L and P_G correspond to the predictions generated by the local and global PPM predictor (-1 if not taken and +1 if taken), respectively. The final prediction is *taken* if the output Y is positive and *not taken* if Y is negative. The table of weights is indexed by the lower 20 bits of the branch's PC. The perceptron is updated by incrementing the weights whose inputs match the branch outcome and decrementing those with mismatch. The *Neuro-PPM* branch predictor achieves an average misprediction rate of 3%.

4.2. Branch Prediction Using State-of-the-Art Predictors

4.2.1. The Perceptron-Based Branch Predictor

In [Jim02] Jiménez and Lin developed a perceptron-based predictor that uses both local and global branch history information in the prediction process. Figure 4.1 presents the architecture of their perceptron-based branch predictor.

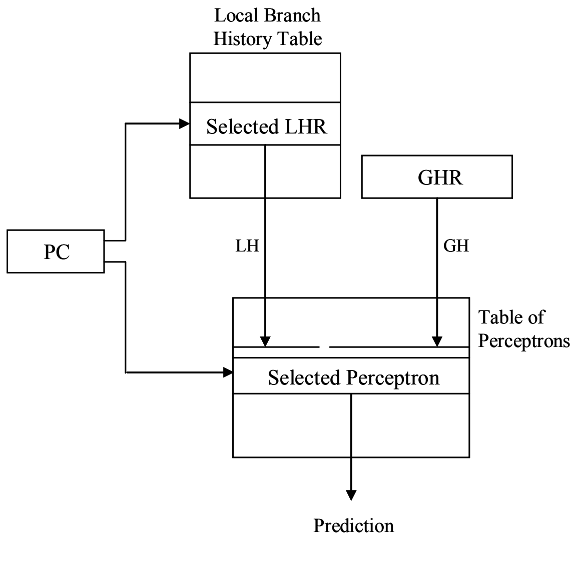
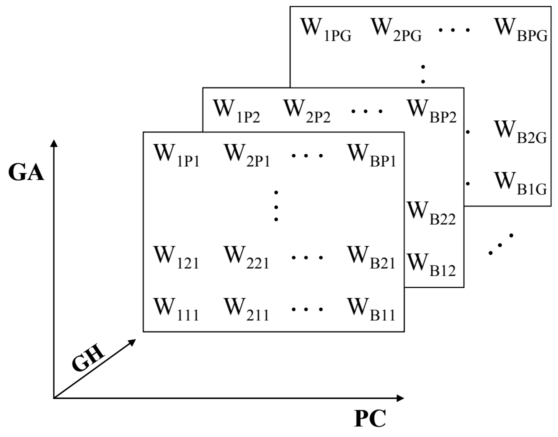


Figure 4.1. The perceptron-based branch predictor

The lower part of the branch address (PC) selects a perceptron in the table of perceptrons (weights' matrix) and a local history register in the local branch history table. Both local and global branch history are used as inputs for the selected perceptron in order to generate a prediction.

4.2.2. The Idealized Piecewise Linear Branch Predictor

The *piecewise linear branch predictor* (previously described in paragraph 4.1.2) has the same architecture as the perceptron-based branch predictor (see Figure 4.1). The weight selection mechanism of the idealized piecewise linear branch predictor is presented in Figure 4.2, where GH is the global history, PC is the branch's address and GA is the path – an array consisting in the addresses of the last executed branches. Thus, the weight W_{bpg} corresponds to branch b ($1 \le b \le B$), its global history bit g ($1 \le g \le G$) and the pth PC ($1 \le p \le P$) from its path.



W_{1PG} W_{2PG} ... W_{BPG}
W_{1P2} W_{2P2} ... W_{BP2}
W_{B2G}
W_{B1G}
W_{1P1} W_{2P1} ... W_{BP1}
W_{B22}
W_{121} W_{221} ... W_{B21}
W_{B12}
W_{111} W_{211} ... W_{B11}
GA
GH
PC

Figure 4.2. The weight selection mechanism of the idealized piecewise linear branch predictor

4.2.3. The Frankenpredictor

The Frankenpredictor [Loh05a] is a *gskew-agree* global history predictor combined with a path-based neural predictor. The prediction mechanism of the Frankenpredictor is presented in Figure 4.3.

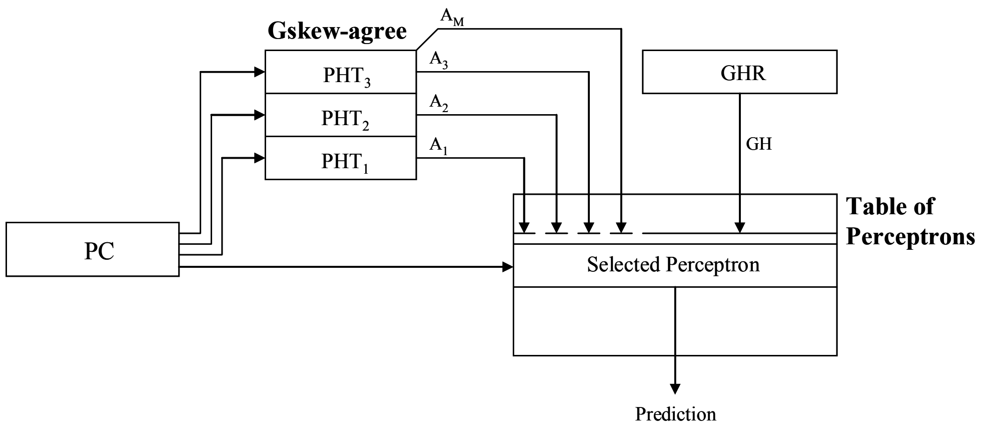


Figure 4.3. The Frankenpredictor's architecture

The *gskew-agree* predictor avoids interference by mapping potential conflicting branches to different entries from three different tables. Three different predictions are provided, the final prediction being furnished by taking majority vote. The agreement approach uses a default BTFNT (backward taken forward not taken) static prediction (bias) for each branch. The predictions (P_1, P_2 and P_3) generated by the selected pattern history table entries are further compared with the bias. The neural predictor provides the ability of working with long branch histories and it also provides the hybridization by including the predictions of the *gskew-agree* predictor as additional bits in the perceptron's input vector – the agreement bits (A_1, A_2 and A_3) provided by the three PHTs (A_i is 1 if P_i agrees with the bias and 0 otherwise, $1 \le i \le 3$) and the majority vote (A_M).

4.2.4. The O-GEHL Predictor

The *Optimized GEometric History Length* (O-GEHL) predictor [Sez05] uses *M* distinct prediction tables indexed with hash functions of the branch address and the global branch history. Distinct history lengths of up to 200 bits and a path history of up to 16 bits, consisting of one address bit per branch, are used to index the prediction tables. Table T_0 is indexed using the branch address. The history lengths used to index tables T_i, $1 \le i \le M$, form a geometric series:

$$L(i) = \alpha^{i-1} \cdot L(1) \qquad (4.1)$$

The prediction tables store predictions as signed counters. To compute a prediction, a single counter is read from each prediction table. The prediction is computed as the sign of the sum *S* of the *M* counters. The prediction is taken if *S* is positive and not taken otherwise. The final prediction mechanism of the O-GEHL predictor is presented in Figure 4.4.

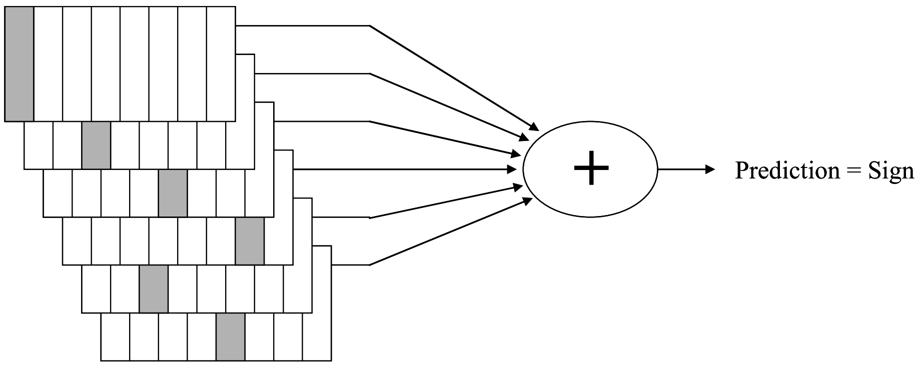


Figure 4.4. The O-GEHL prediction aggregation mechanism

4.3. Value-History-Based Branch Prediction with Markov Models

The context-based predictor predicts the next value based on a particular stored pattern (context) that is repetitively generated in the value sequence. Theoretically they can predict any stochastic repetitive sequences. A context predictor is of order *k* if its context information includes the last *k* values, and, therefore, the search is done using this pattern of *k* values length. In fact, in this case the prediction process is based on a simple Markov model [Vin07]. A first order discrete Markov process may be described at any time as being in one of a set of *N* distinct states $S = \{S_1, S_2, ..., S_N\}$, as illustrated in Figure 4.5.

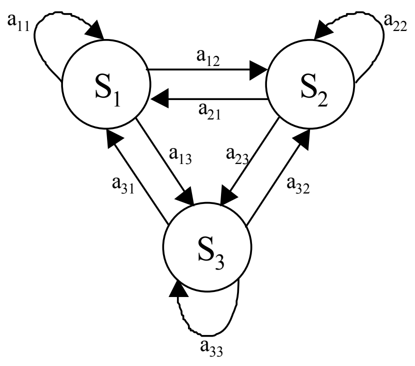


Figure 4.5. A Markov chain with 3 states

A full probabilistic description of discrete Markov chain requires specification of the current state as well as all the predecessor states (the current state in a sequence depends on all the previous states). For the special case of a discrete, first order, Markov chain, this probabilistic description is truncated to just the current and predecessor state (the current state depends only on the previous state):

$$P[q_t = S_j | q_{t-1} = S_i,\ q_{t-2} = S_k,\ ...] = P[q_t = S_j | q_{t-1} = S_i] \tag{4.2}$$

where q_t is the state at time *t*. Thus, for a first order Markov chain with *N* states, the set of transition probabilities between states S_i and S_j is $A = \{a_{ij}\}$, where $a_{ij} = P[q_t = S_j | q_{t-1} = S_i]$, $1 \le i,\ j \le N$, having the properties $a_{ij} \ge 0$ and $\sum_{j=1}^{N} a_{ij} = 1$.

For a Markov chain of order *R* the probabilistic description is truncated to the current and *R* previous states (the current state depends on *R* previous states). The following example shows the necessity of using superior order Markov models. If the sequence of states is AAABCAAABCAAA, the Markov models of order 1 and 2 mispredict A, and only a Markov model of order 3 predicts correctly the next state B. This example is also presented in Figure 4.6.

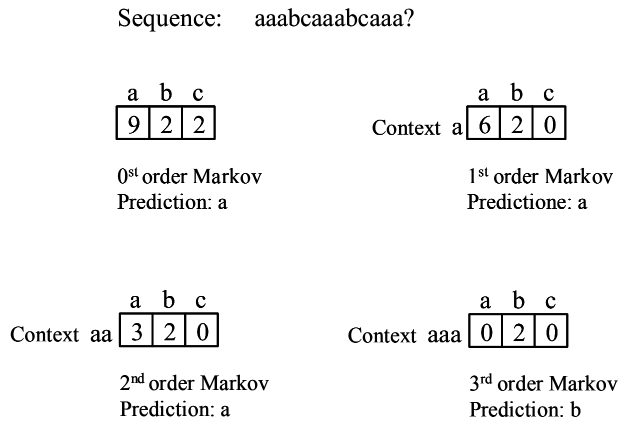


Figure 4.6. Markov predictors of different orders

Value predictors that implement the "*Prediction by Partial Matching*" algorithm (PPM) [Saz97, Jos97] represent an important class of context-based predictors. Mudge et al. [Mud96] demonstrates that all two-level adaptive predictors implement special cases of the PPM algorithm that is widely used in data compression. It seems that PPM provides the ultimate predictability limit of two-level predictors. The PPM-based predictor contains a set of simple

Markov predictors, each one predicting the value that followed the corresponding context with the highest frequency, as it can be seen in Figure 4.6. In a complete-PPM predictor, if a prediction cannot be furnished by the Markov predictor of order *k*, then the pattern length is shortened and the Markov predictor of order $k-1$ is used to furnish the prediction and so on until either a prediction is furnished or the Markov predictor is of the order *0*.

Our second idea in order to reduce the number of unbiased branches, after the feature set length extension (presented in Chapter 3), was to find new relevant information that could reduce their entropy making them more predictable. Representing the problem in a superior feature space dimension is a general well-known method in solving many Computer Science classification/prediction problems. Therefore, we predict the condition of the current branch (B_0) based on the conditions of the previous branches (B_1, B_2, ..., B_h), with different PPM predictors. We use each branch condition as the value or the sign of the difference between the operand values (two approaches). Regarding the approach that uses only the signs of the input differences, a value of 1 indicates that the corresponding branch difference is positive, a -1 indicates a negative difference, while a 0 indicates equality between the branch inputs. The outcome of the current branch B_0 is determined speculatively based on its predicted condition (difference).

But is it better to use only the signs of differences as history information instead of the values of differences? Is this compressed branch condition history more efficient than the most complete value history? The number of distinct symbols that can occur in a value history is huge reported to only three symbols that can appear in a sign history. Thus, the frequency of symbols in a value history is very low. In the following example only a Markov predictor of order 1 can be used for the value history, and it generates a misprediction, while in the case of the sign history, even a Markov predictor of order 5 can be used, which achieves the correct prediction:

Value history: -126, -34, 7, -42, -28, 75, -829, -7982, 102, -542, -42, ?
Sign history: -1, -1, 1, -1, -1, 1, -1, -1, 1, -1, -1, ?

Obviously, through a sign history much deeper correlations can be exploited than with a value history. A natural question is: are the sign histories better than the simplest branch outcome histories (taken / not taken)? The difference-sign history can be more efficient because, due to its additional information, it can efficiently exploit shorter contexts, too. The following example presents the situation for *bgez*:

Difference history: 138, 52, 47, 0, -591, 5783, 4, 702, 0, -35, 721, 5, 14, 0, ?
Sign history: +, +, +, 0, -, +, +, +, 0, -, +, +, +, 0, ?
Output history: T, T, T, T, NT, T, T, T, T, NT, T, T, T, T, ?

If after "0" statistically follows "-" (and, in the case of *bgez*, "0" is associated together with "+" to *taken*) a first order Markov can correctly predict in the case of sign history, while, in the case of outcome history, the Markov predictor must be of order 4 or higher for correct prediction. Anyway, the simulation results will decide which type of branch condition history is the most efficient.

4.3.1. Local Branch Difference Predictor

Figure 4.7 presents the speculative branch execution mechanism of our local PPM branch-difference predictor. The Branch Difference History Table (BDHT) maintains for each static branch the differences corresponding to the branch's last *h* dynamic instances (B_1, B_2, ..., B_h). The BDHT entry is selected by the branch address (PC of B_0). The branch differences from the selected BDHT entry are then used as inputs into the complete-PPM predictor. The PPM predictor of order *k* (where $k<h$) furnishes the predicted difference of the branch undergoing execution (B_0). Speculative execution of the branch B_0 based on its predicted difference only

occures in the case that the considered pattern of length *k* is repeated in the string of last *h* differences with a frequency greater than or equal to a certain threshold value.

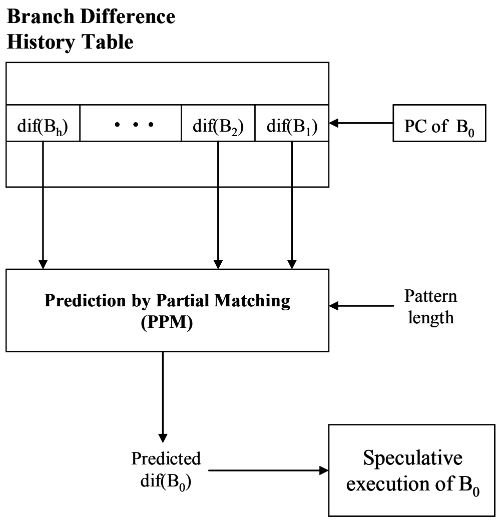

Branch Difference History Table
dif(B_h)
• • •
dif(B_2)
dif(B_1)
PC of B_0
Prediction by Partial Matching (PPM)
Pattern length
Predicted dif(B_0)
Speculative execution of B_0

Figure 4.7. A local PPM-based branch-difference predictor

4.3.2. Combined Global-Local Branch Difference Predictor

Figure 4.8 presents the speculative branch execution mechanism using a combined global and local PPM-based branch-difference predictor. The Global History Register (GHR) contains the global history: the global branch difference history or the global branch outcome history (two different approaches). For each global history pattern, a distinct BDHT is maintained. Thus, the BDHT is selected by the GHR. Each BDHT is configured as a local BDHT and is accessed as described in Section 4.3.1.

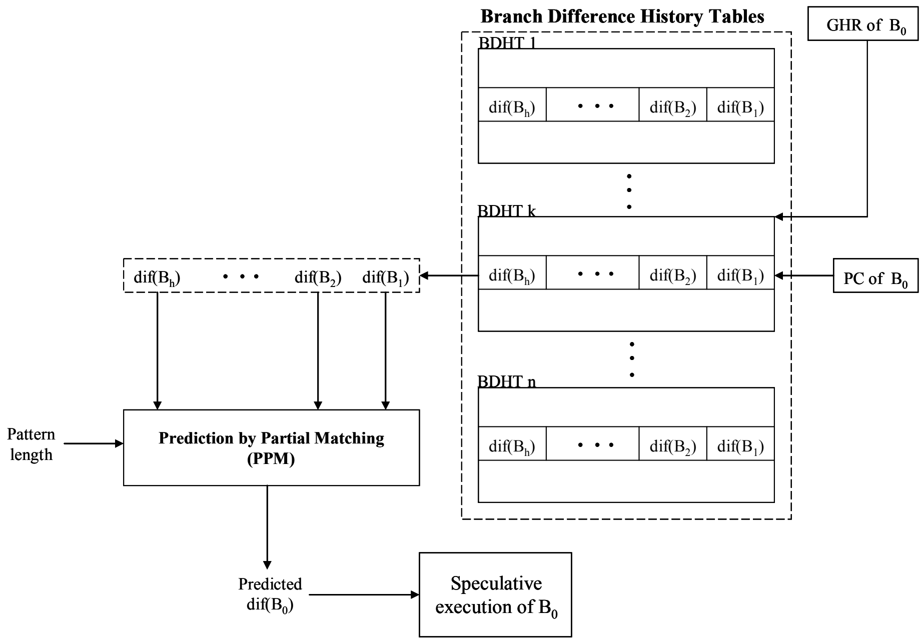


Figure 4.8. A global-local PPM-based branch-difference predictor

4.3.3. Branch Difference Prediction by Combining Multiple Partial Matches

Figure 4.9 presents the speculative branch execution mechanism using the *Branch-Difference Predicion by Combining Multiple Partial Matches* (BPCMP). An entry in the BDHT is accessed as described in Section 4.3.1, but now the *h* branch differences are used as inputs into multiple Markov predictors of different orders. Thus, the sign of the input difference (-1, 1, or 0) corresponding to the current branch (B_0) is predicted using multiple Markov predictors of orders ranging between [1, *n*], $n<h$ (see Figure 4.9). The final branch difference prediction is then furnished through majority vote.

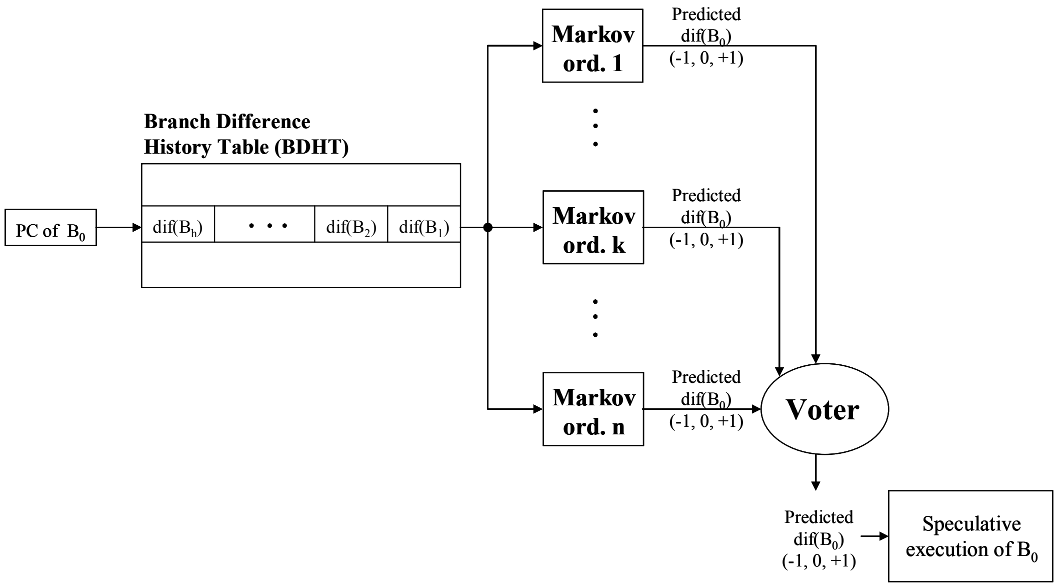

Branch Difference
History Table (BDHT)
PC of B_0
dif(B_h)
• • •
dif(B_2)
dif(B_1)
Markov ord. 1
Markov ord. k
Markov ord. n
Predicted dif(B_0) (-1, 0, +1)
Predicted dif(B_0) (-1, 0, +1)
Predicted dif(B_0) (-1, 0, +1)
Voter
Predicted dif(B_0) (-1, 0, +1)
Speculative execution of B_0

Figure 4.9. Branch-difference prediction by combining multiple Markov predictors

We have also investigated a confidence-based voting mechanism. In this case, each BDHT entry holds *n* saturated confidence counters, in the range [-4, 4], which are associated with the *n* Markov predictors. A certain Markov predictor of order *k* ($1 \le k \le n$) will furnish a value prediction if the corresponding pattern occures at least once in the history of *h* values. In the case of a correctly predicted branch, its confidence saturating counter is incremented and decremented in the case of a misprediction. Each Markov prediction is replicated as many times as the corresponding counter's value shows (only if this value is greater than zero). These multiple predictions are then passed to the voter, which furnishes the most frequent value.

4.4. Using Previous Branch Condition as Prediction Information

In this section we tried to use the value of previous branch condition (PBC) as prediction information, taking into account that it determines branch's behavior. A PBC value consists in the difference of the operand values involved in the previous branch condition. Using only one branch condition is in concordance with Heil's observation in [Hei99b] that majority of prediction accuracy improvement is gained by using a single branch difference. First we evaluated the percentage of unbiased context instances (having polarization *P* less than 0.95) using the PBC value together with the global histories of *p* bits ($1 \le p \le 24$). Figure 4.10 compares the percentages of unbiased branches using the global history (GH), the global history concatenated with the path (GH + PATH), and the global history concatenated with the value of the previous branch condition (GH + PBC).

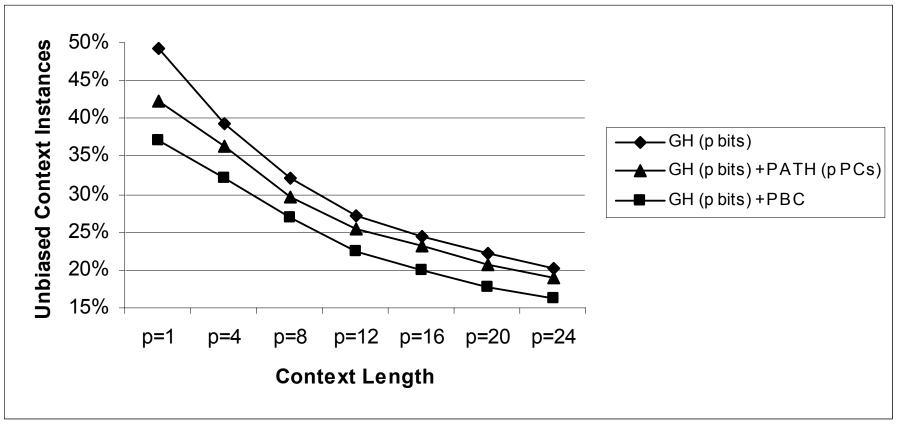


Figure 4.10. The gain introduced by the previous branch condition (PBC) vs. the path for different context lengths – SPEC 2000 benchmarks

The experimental results, presented in Figure 4.10, show that the PBC value is more efficient than the path information: it decreased the percentage of unbiased branches for all evaluated context lengths ($1 \le p \le 24$). Therefore we could use this new prediction information in some state-of-the-art branch predictors in order to increase prediction accuracy [Gel07a, Gel07b, Gel07c].

4.4.1. The GAg Predictor Using Global PBC Value

We first analyzed a GAg scheme that uses the previous branch condition (PBC) by XORing it with the GHR (as the Gshare XORed the PC with the GHR). The predictor's scheme is presented in Figure 4.11.

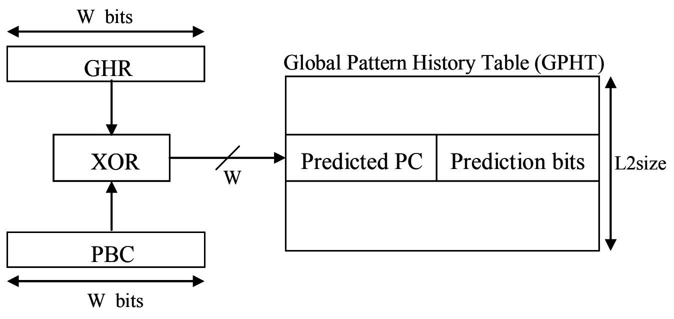


Figure 4.11. The GAg predictor using the previous branch condition (PBC)

4.4.2. The PAg Predictor Using Local PBC Value

We have also analyzed a PAg scheme that uses the local (per-address) PBC value (previous branch condition) by XORing it with the LHR (local history register). The Per-address Branch History Table (PBHT) maintains for each branch its own Local History (LH) and its Previous Branch Condition (PBC) value. The predictor is presented in Figure 4.12.

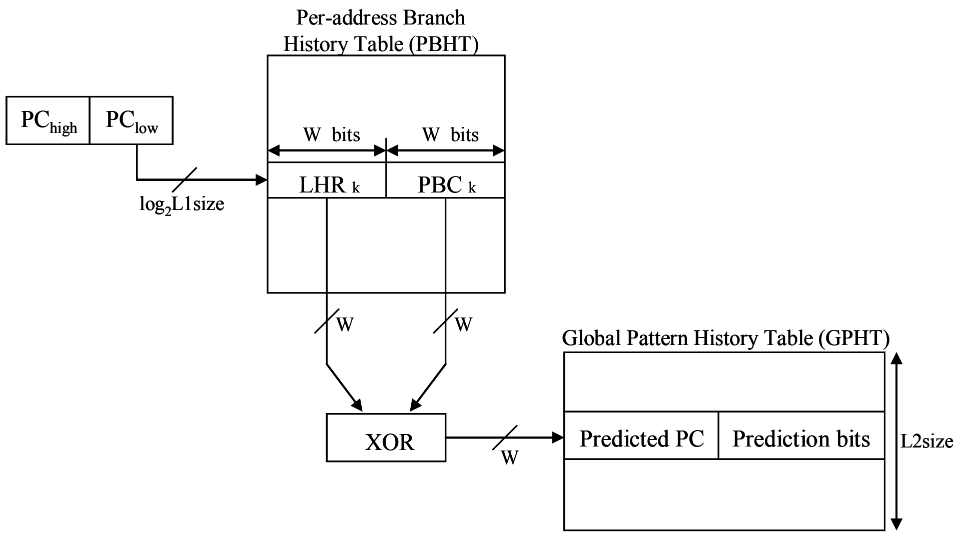


Figure 4.12. The PAg predictor using the local PBC value

4.4.3. The Piecewise Linear Branch Predictor Using PBC Value

Further, we propose some improved *idealized piecewise linear branch predictors* (see Figures 4.13 and 4.14) that use the previous global or local branch condition (PBC) as additional prediction information. The global history length is dynamically adjusted between 18 and 48 bits and the local history length between 1 and 16 bits, as in [Jim05, Gel07a, Gel07b]. In both schemes local and global branch histories together with the PBC value are used as inputs for the selected perceptron in order to generate a prediction. The three indexes used within the weight selection mechanism are obtained through a hash function that uses three prime numbers, as follows [Jim04]:

$$index_{GH}^{i} = [(PC \cdot 511387) \oplus (PC_{i-1} \cdot 660509) \oplus (i \cdot 1289381)] \bmod NW \qquad (4.3)$$

$$index_{LH}^{j} = [(PC \cdot 511387) \oplus (j \cdot 1289381)] \bmod NW \qquad (4.4)$$

$$index_{PBC}^{k} = [(PC \cdot 511387) \oplus (k \cdot 1289381)] \bmod NW \qquad (4.5)$$

where $i = \overline{1, GHlength}$, $j = \overline{1, LHlength}$, $k = \overline{LHlength + 1, LHlength + PBClength}$ ($PBClength$ is 32 in our case), and NW is the total number of weights (parameter varied in our simulations between 8590 and 30713). PC_{i-1} represents the previous (i-1)th branch's PC, belonging to the path of the current branch. Consequently, a certain prediction is generated using ($GHlength + LHlength + PBClength$) number of selected weights. These weights were selected from a table containing NW weights. The first two relations were used according to Jimenez's simulator proposals [Jim04] while the third one was introduced by us, according to the new introduced PBC information.

4.4.3.1 The Piecewise Linear Branch Predictor Using Global PBC Value

Figure 4.13 presents the scheme of the perceptron-based branch predictor that is using as additional prediction information the global previous branch condition (PBC). The lower part of the branch address (PC) selects a perceptron in the table of perceptrons and a local history register in the local branch history table.

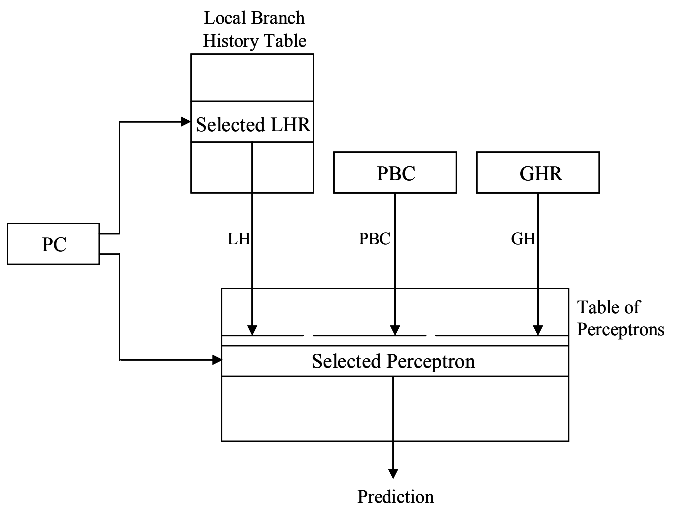


Figure 4.13. Perceptron-based branch predictor using the global PBC value

4.4.3.2 The Piecewise Linear Branch Predictor Using Local PBC Value

Figure 4.14 presents a possible scheme of the perceptron-based branch predictor that is using as prediction information local (per-address) previous branch condition (PBC).

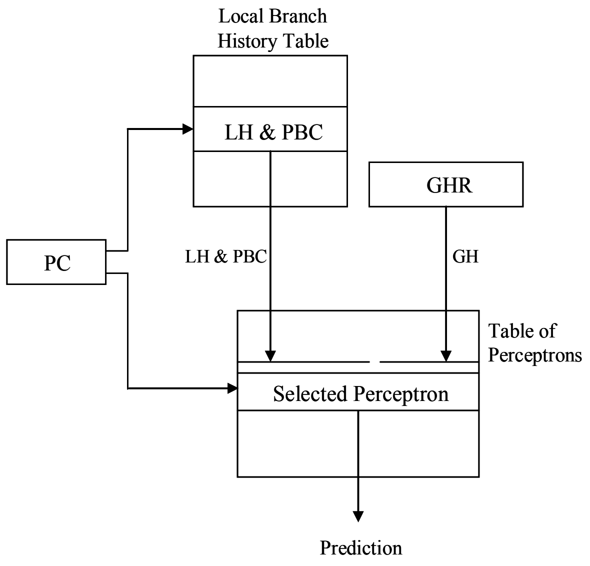


Figure 4.14. Perceptron-based branch predictor using the local PBC value

In Figure 4.14, the Local Branch History Table maintains for each branch its Local History (LH) and its the Previous Branch Condition (PBC) value.

4.5. Experimental Results

The perceptron and our branch difference predictors were implemented by extending the *sim-bpred* simulator provided in *SimpleSim-3.0* [Sim]. We also include the implementation of the unbiased branch selection mechanism and, thus, the predictors can be evaluated on unbiased branches, too. We have evaluated our predictors on SPEC 2000 benchmarks, especially those that indicated a high percentage of unbiased branches [Gel06a, Vin06]. The Championship Branch Prediction (CBP-1) simulators of the Frankenpredictor [Loh05a] and the Piecewise Linear Branch Predictor [Jim05] were extended to work with the same unbiased branch selection mechanism. In order to exploit these predictors we used the CBP-1 branch prediction framework which includes twenty traces (5 integer programs, 5 floating point, 5 multimedia applications and 5 server benchmarks) and a driver that reads the traces and calls the branch predictor [CBP04]. The traces are approximately 30 million instructions long and include both user and system codes. The two predictors were implemented within the constraints of a storage budget of (64K + 256) bits.

All simulation results are reported on 1 billion dynamic instructions skipping the first 300 million instructions from the SPEC 2000 benchmarks [SPEC] and on all instructions from the INTEL benchmarks [CBP04]. We note with LH(p)-GH(q) prediction information consisting in local history (LH) of *p* bits, and global history (GH) of *q* bits. We also note with *PPM(tdim, hlen, plen, thres, htype)* a complete-PPM branch-difference predictor using a Branch Difference History Table (BDHT) of *tdim* entries, a history length of *hlen* differences, a search pattern length of *plen* (specifying the current state), a threshold of *thres*, and considering a history of branch difference values or branch difference signs (*htype*=value/sign).

4.5.1. Evaluating State-of-the-Art Branch Predictors

In the first stage of this section, we have measured with present-day branch predictors the prediction accuracy on all branches and on the final list of unbiased branches identified in [Vin06], using different local and global history lengths.

4.5.1.1. Evaluating the Perceptron-Based Branch Predictor

Figure 4.15 shows comparatively the results obtained on the SPEC 2000 benchmarks with a simple perceptron-based predictor integrated into *Simplesim-3.0* [Sim]. We used a table of perceptrons with 256 entries.

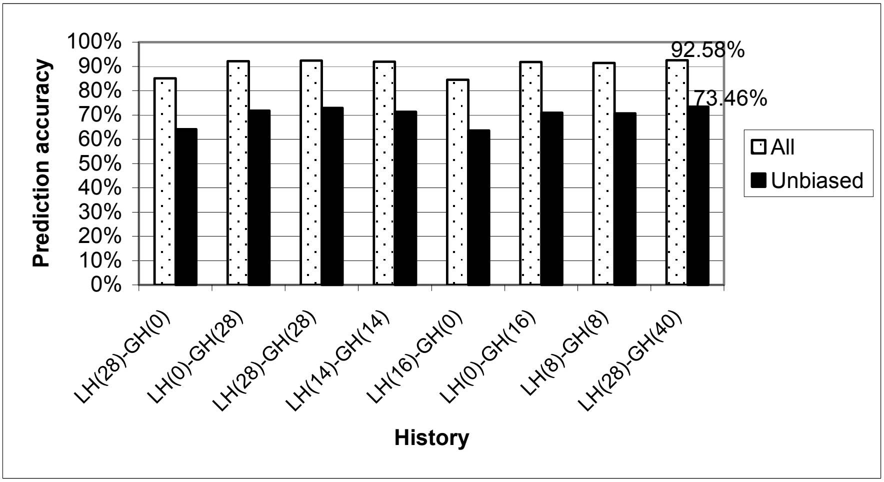


Figure 4.15. The average prediction accuracies obtained with the perceptron predictor using different prediction information on the SPEC 2000 benchmarks

Figure 4.15 intends to find an optimal LH(p)-GH(q) configuration within an enormous space of possible solutions. As Figure 4.15 shows, when we used the best configuration of the perceptron predictor (a local history of 28 bits and a global history of 40 bits – determined based on laborious simulations), we obtained an average prediction accuracy of 92.58% on all branches and of only 73.46% on the unbiased branches.

4.5.1.2. Evaluating the Idealized Piecewise Linear Branch Predictor

Figure 4.16 shows comparatively on the SPEC 2000 benchmarks the prediction accuracies obtained with the Idealized Piecewise Linear Branch Predictor (described in paragraph 4.2.2) on all branches and on the final list of unbiased branches identified in [Gel06a, Vin06, Oan06] using the XOR between the global history of 32 bits and the path of 32 PCs.

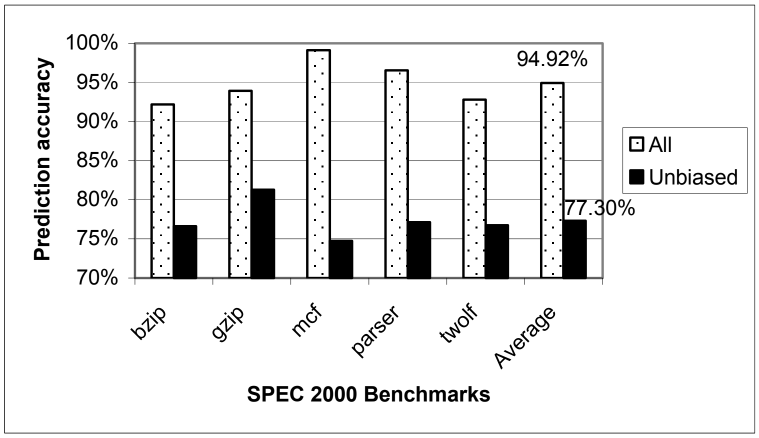


Figure 4.16. The average prediction accuracies obtained with the Idealized Piecewise Linear Branch Predictor on the SPEC 2000 benchmarks

We used the original Idealized Piecewise Linear Branch Predictor [Jim05] whose global history length is dynamically adjusted between 18 and 48 bits and its local history length between 1 and 16 bits. Even if the Idealized Piecewise Linear Branch Predictor doesn't solve satisfactory the unbiased branches problem, it predicts them with an average accuracy of 77.3% that is better than all the other simulated branch prediction schemes.

Figure 4.17 shows comparatively on the CBP-1 Intel benchmarks [CBP04] the prediction accuracies obtained with the Idealized Piecewise Linear Branch Predictor [Jim05] on all branches and on the final list of unbiased branches. We used the same configuration as on the SPEC 2000 benchmarks.

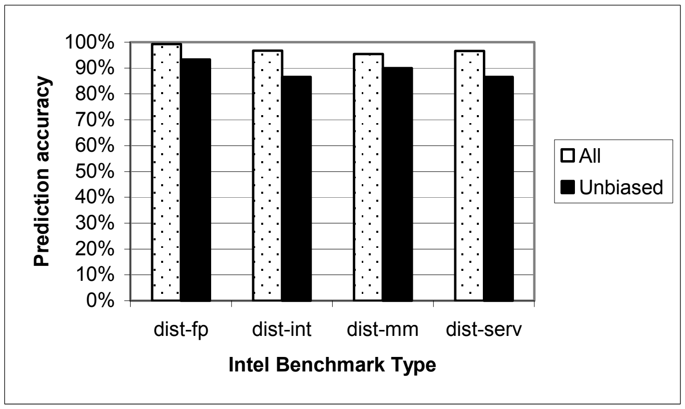


Figure 4.17. Average prediction accuracies obtained with the Idealized Piecewise Linear Branch Predictor on the Intel benchmarks

The Idealized Piecewise Linear Branch Predictor provides a prediction accuracy of 89.1% on the unbiased branches from the Intel benchmarks. Although the CBP-1 Intel benchmark suite includes integer, floating-point, multimedia and server applications, we are reserved regarding them due to their shortness. Furthermore, the *Second World Championship Branch Prediction Competition* (CBP-2) [CBP06] has used all the twelve CPUintSPEC2000 benchmarks and eight JavaSPECjvm98 benchmarks, which shows the weakness of the CBP-1 benchmark suite.

In [Flo07b] we have also evaluated on the SPEC JVM98 benchmarks the *fast path-based neural branch predictor* [Jim03c] – a particular configuration of the *piecewise linear branch predictor* – which uses a single global piecewise-linear function to predict all branches. As Figures 3.6 and 4.18 show, the lower percentage of unbiased branches within the object-oriented Java applications has a lower impact on the global prediction accuracy (98.57%) and even unbiased branches are predicted more accurately (80.51%).

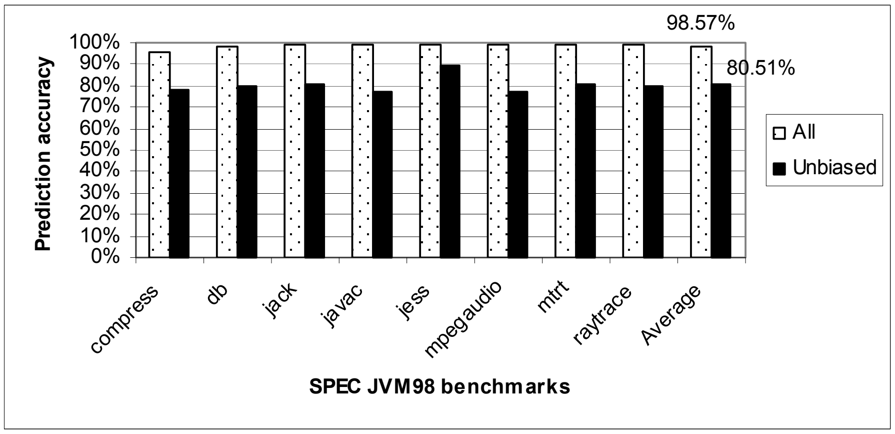


Figure 4.18. Average prediction accuracies obtained with the Fast Path-Based Neural Branch Predictor on the SPEC JVM98 benchmarks

4.5.1.3. Evaluating the Frankenpredictor

Figure 4.19 shows comparatively on the SPEC 2000 benchmarks the prediction accuracies obtained with the Frankenpredictor (described in paragraph 4.2.3) on all branches and on the unbiased branches identified in [Gel06a, Vin06, Oan06]. For the Frankenpredictor we used a global history of 59 bits [Loh05a].

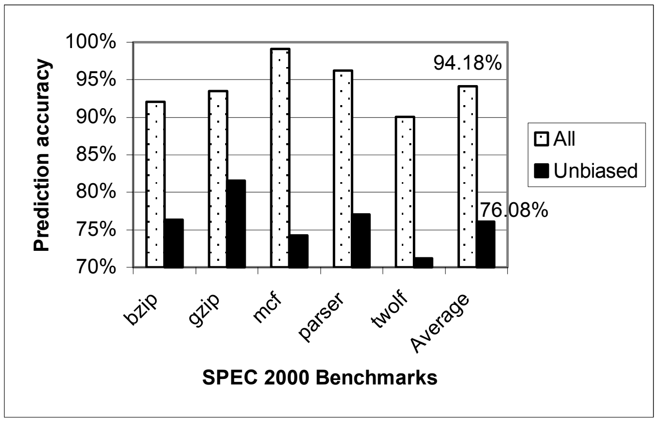


Figure 4.19. The average prediction accuracies obtained with the Frankenpredictor on the SPEC 2000 benchmarks

Figure 4.20 shows comparatively on the CBP-1 Intel benchmarks [CBP04] the prediction accuracies obtained with the Frankenpredictor on all branches and on the final list of unbiased branches, using the same configuration as on the SPEC 2000 benchmarks.

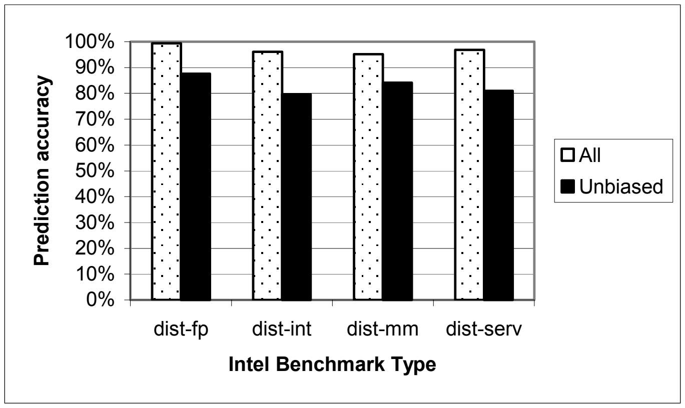


Figure 4.20. The average prediction accuracies obtained with the Frankenpredictor on the Intel benchmarks

We empirically found out that the behavior of difficult branches – as we defined them – cannot be sufficiently learned neither by neural predictors. Figures 4.16 – 4.20 confirm us again, that the unbiased branches, identified in our previous work [Vin06, Gel06a], are hard-to-predict with present-day branch predictors.

4.5.1.4. Evaluating the O-GEHL Predictor

We have also evaluated the Optimized GEometric History Length (O-GEHL) predictor [Sez05], described in paragraph 4.2.4 (see Figure 4.4). We used an 8-table O-GEHL predictor. The experimental results obtained on the SPEC 2000 benchmarks are presented in Figure 4.21.

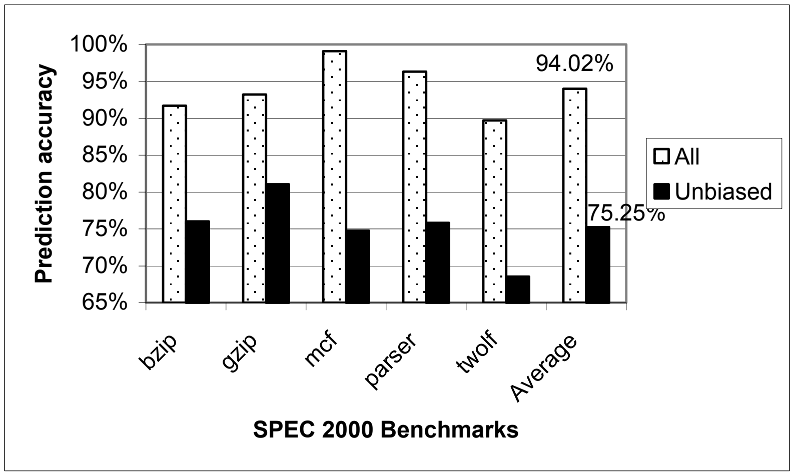


Figure 4.21. The average prediction accuracies obtained with the O-GEHL predictor on the SPEC 2000 benchmarks

As it can be observed, the neural branch predictors provided higher prediction accuracy then the O-GEHL predictor (see comparatively Figures 4.16, 4.19 and 4.21).

4.5.2. Evaluating Markovian Value-History-Based Branch Predictors

In this section we evaluate the Markovian value-history-based branch predictors proposed and described in Section 4.3. We emphasize that our investigation is about the impact that unbiased branches have on dynamic branch prediction and therefore realistic hardware costs and timings are out of scope.

4.5.2.1. Evaluating Local Branch Difference Predictors

We set out to determine the optimal local branch difference predictor. We asked ourselves five questions. Would the operand sign value difference algorithm achieve better prediction accuracy than the operand value difference? Which local history register length would provide the best prediction accuracy? Which pattern length would achieve the best prediction accuracy? What is the most suitable threshold value? What is the ideal number of local BDHT entries?

In Figure 4.22 we answer the first two questions: What would be the most suitable operand difference algorithm to use and, which history register length achieves the best prediction accuracy? We evaluated the impact of the unbiased branches (identified in [Vin06]) from the SPEC 2000 benchmarks using a complete PPM predictor with a local BDHT. We considered an unlimited BDHT which ensured that every static branch had its own entry thereby eliminating any possibility of collisions. The pattern length was set to 3, the threshold value was set to 1, and the local history register length was varied from 8 differences to 64.

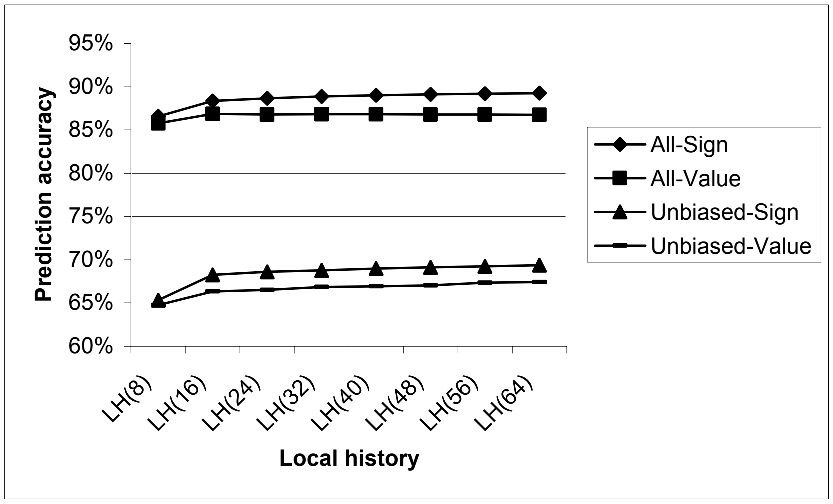


Figure 4.22. The average prediction accuracies on the SPEC 2000 benchmarks, using a *PPM(tdim=unlimited, hlen=varied, plen=3, thres=1, htype=value and sign)* branch difference predictor with different local history lengths

Our results show that better prediction accuracy is achieved by the difference signs rather than the difference values and that beyond a local history register length of 24 differences there is only marginal improvement in prediction accuracy. Consequently, the sign of the current branch difference is better correlated with the signs of its previous differences rather than with the values of those differences. But why is better to use only the signs of differences as history information instead of the values of differences? The number of distinct symbols that can occur in a value history is huge reported to only three symbols that can appear in a sign history. Thus, the frequency of symbols in a value history is very low. Therefore, as we have shown in Section 4.3 based on examples, through a sign history much deeper correlations can be exploited than with a value history.

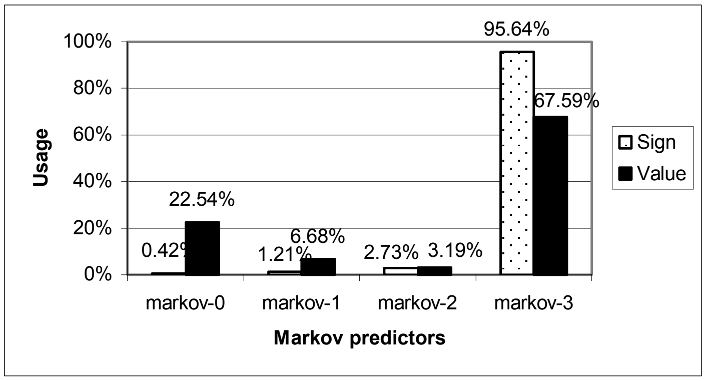


Figure 4.23. The average usage rates of Markov predictors using *PPM(tdim=unlimited, hlen=24, plen=3, thres=1, htype=sign and value)* branch difference predictors on all branches

Figure 4.23 compares the sign history with the value history in terms of usage rate for different order Markov predictors. We used the optimal history length 24 and a pattern length of 3, and therefore, we evaluated the usage rates corresponding to Markov predictors of orders 0, 1, 2 and 3. As Figure 4.23 shows, more often are used superior order Markov predictors by using a sign history, and thus, deeper correlations can be exploited. Therefore, we continued by evaluating different pattern lengths using an unlimited BDHT, a sign history of 24 branch difference signs, and a threshold of 1. In Figure 4.24 we answer the third question: Which pattern length would achieve the best prediction accuracy? Our results confirm that our original pattern length of 3 achieves the best prediction accuracy, considering the optimal local history of 24 branch difference signs.

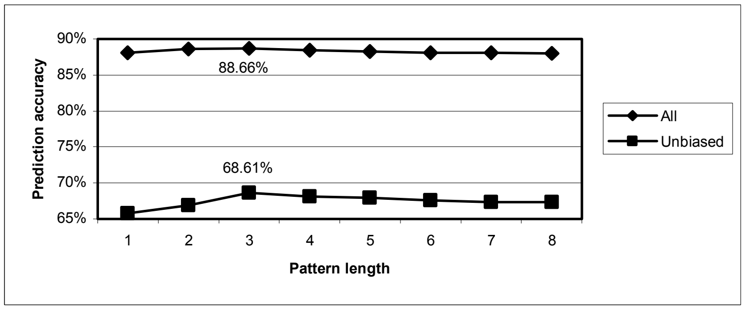


Figure 4.24. Average prediction accuracies on SPEC 2000 benchmarks, using a *PPM(tdim=unlimited, hlen=24, plen=varied, thres=1, htype=sign)* branch difference predictor with different pattern lengths

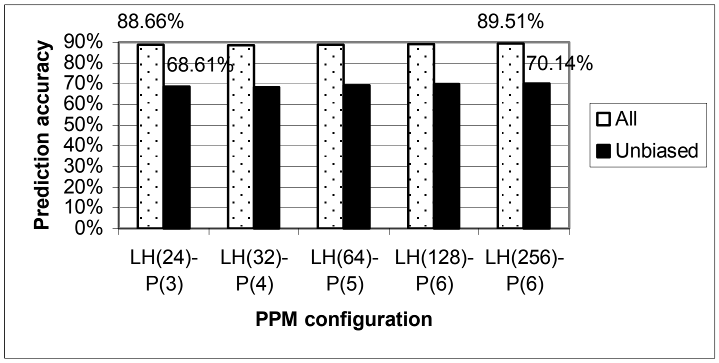


Figure 4.25. Average prediction accuracies on SPEC 2000 benchmarks using a *PPM(tdim=unlimited, hlen=varied, plen=varied, thres=1, htype=sign)* branch difference predictor exploring different local history lengths and pattern lengths

Figure 4.25 explores the space of local history lengths and pattern lengths using a threshold of 1 and confirms that an acceptable choice (taking into account a good accuracy/complexity trade-off report) is to use a history of 24 branch difference signs with a pattern length of 3.

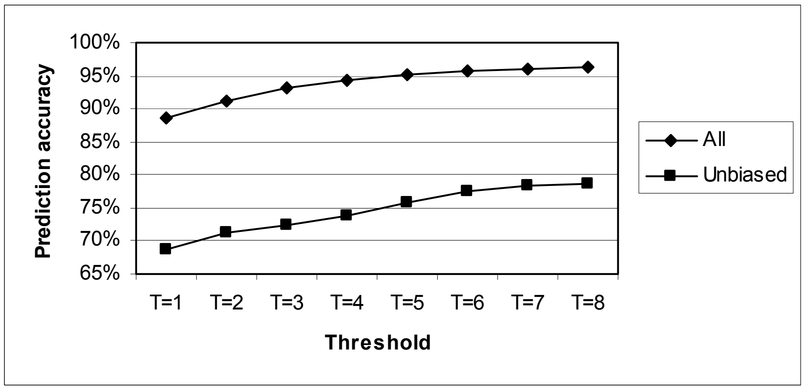


Figure 4.26. Average prediction accuracies on SPEC 2000 benchmarks, using a *PPM(tdim=unlimited, hlen=24, plen=3, thres=varied, htype=sign)* branch difference predictor with different threshold values

Threshold	Lost predictions [%]
T=1	0.00
T=2	7.59
T=3	13.37
T=4	17.31
T=5	20.50
T=6	23.40
T=7	25.13
T=8	26.98

Table 4.1. Average percentages of predictions lost with different thresholds

In Figure 4.26 we answer the fourth question: What is the most suitable threshold value? We used an unlimited BDHT, a local history of 24 branch difference signs, the pattern length was now set to 3 and the threshold value varied. The threshold's value means how many times the current search pattern must be found in the history string in order to generate a prediction, implementing thus a confidence degree (otherwise, no prediction is generated). Our results show that prediction accuracy improves with an increasing threshold value, but there is marginal, if any, benefit of increasing the threshold value beyond 7. Strictly considering the confidence metric, the experimental results presented in Figure 4.26 show that the optimal threshold value is 7. However, in this case, the total number of predictions decreases at average with 25.13% (see Table 4.1). Considering T=1, the global prediction accuracy on unbiased branches A(T=1) is 68.61%. In contrast, considering T=7, the global accuracy A(T=7) is $74.87\% \cdot 78.33\% = 58.64\%$ whereas for T=2, A(T=2) it is $92.41\% \cdot 71.16\% = 65.75\%$. Therefore, from the global accuracy point of view T=1 is optimal.

In Figure 4.27 we answer the final question: What would be the optimal number of entries in the local BDHT? We used the same parameters as Figure 4.26, and the number of entries in the local BDHT was varied from 64 entries to 256 entries in increments of 64. We have also included an unlimited local BDHT. Our experimental results show that the impact of the so called 3Cs (capacity, collisions and cold-start) to be minimal with a 256 entry local BDHT and that there is minimal prediction accuracy gain by increasing the number of entries beyond 256 entries where the increased number of cold-start mispredictions may impact on prediction accuracy.

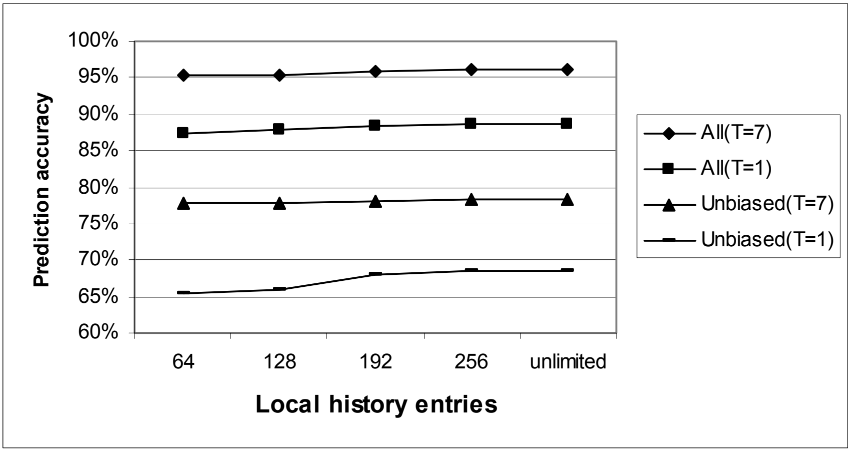


Figure 4.27. Average prediction accuracies obtained on the SPEC 2000 benchmarks using a *PPM(tdim=varied, hlen=24, plen=3, thres=1 and 7, htype=sign)* branch difference predictor with different BDHT sizes

The next step consists in speculatively executing branches based on their predicted input differences. We investigated the branch prediction accuracies of the individual SPEC 2000 benchmarks using our optimal local branch difference predictor. We used the operand sign difference algorithm, with a local history register length of 24-signs, a pattern length of 3, and we used a local 256 entry BDHT. In our results we compare two threshold values, 1 and 7. When the threshold value is 1, we achieve an average branch prediction accuracy of 90.55% and the unbiased branches have an average branch prediction accuracy of 71.76%. When the threshold value is increased to 7, we achieve an average branch prediction accuracy of 96.43% and the unbiased branches have a prediction accuracy of 79.69%. These results show the significance of the threshold value on prediction accuracy and the impact of unbiased branches. Consequently, unbiased branches in this local context remain difficult-to-predict.

4.5.2.2. Evaluating Combined Global and Local Branch Difference Predictors

We consider the high number of unbiased branches and their impact on prediction accuracy to be due to their high degree of shuffling. To alleviate the problem of shuffled branch behaviour of unbiased branches we have developed a combined global and local branch difference predictor which would convert an unbiased branch in a local context into a biased branch in a global context, and therefore a difficult-to-predict branch in a local context would be an easy-to-predict branch in a global context.

In our global and local branch difference predictor, each global history pattern is used to point to its own local BDHT as described in paragraph 4.3.2 and shown in Figure 4.8. Consequently, we restrict the global history register length to a maximum of 4 differences. The selected BDHT is indexed by the PC, as in the local approach. First, we evaluated the predictor by maintaining in the GHR (see Figure 4.8) the global branch difference history: the signs of the inputs' differences corresponding to the previous *h* branches. The parameters of each of the local BDHTs were the same as those of the optimal local BDHT determined in paragraph 4.5.2.1. In Figure 4.28 the global history register length of 0 represents the optimal local branch difference predictor whose results are provided in Figure 4.27, with a 256 entry BDHT. With the combined global and local difference predictor, as the global history register length is increased there is a marginal improvement in prediction accuracy.

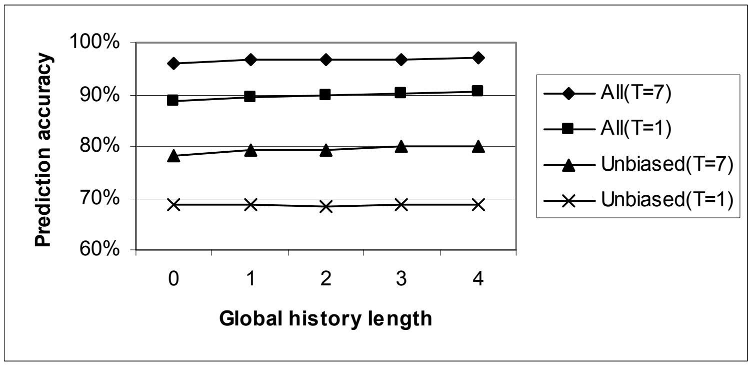


Figure 4.28. Average prediction accuracies on SPEC 2000 using a *PPM(tdim=256, hlen=24, plen=3, thres=1 and 7, htype=sign)* branch difference predictor varying the global branch difference history

The next step consists in investigating the branch prediction accuracies by speculatively executing branches based on their predicted input differences. With a global history register length of 4 signs and a threshold value of 1, the combined global and local branch difference predictor achieves an average prediction accuracy of 92.33%, but the unbiased branches only achieve an average prediction accuracy of 71.54% showing a marginal improvement over the local branch difference predictor. When the threshold value is increased to 7, the average prediction accuracy improves to 97.44% and the average prediction accuracy of unbiased branches is significantly better at 81.25%. Even though there is some improvement in prediction accuracy, these results show that the impact of unbiased branches still remains significant and therefore implies that alternative approaches are required.

We also evaluated the predictor by maintaining in the GHR the global branch outcome history (taken / not taken). Our simulation results show that the confidence is slightly better on unbiased branches if we use the global difference-sign history. As we have shown through an example in Section 4.3, the difference-sign history can be more efficient because, due to its additional information, it can efficiently exploit shorter contexts, too.

4.5.2.3. Branch Difference Prediction by Combining Multiple Partial Matches

Branch differences are predicted by five Markov predictors of orders ranging between 1 and 5, the final prediction being provided through majority voting (as described in Section 4.3.3 and shown in Figure 4.9). Again, we use a 256 entry BDHT, a local branch difference history of 24 values, and we compare the prediction accuracy of two voting algorithms, a simple voting algorithm and a confidence voting algorithm.

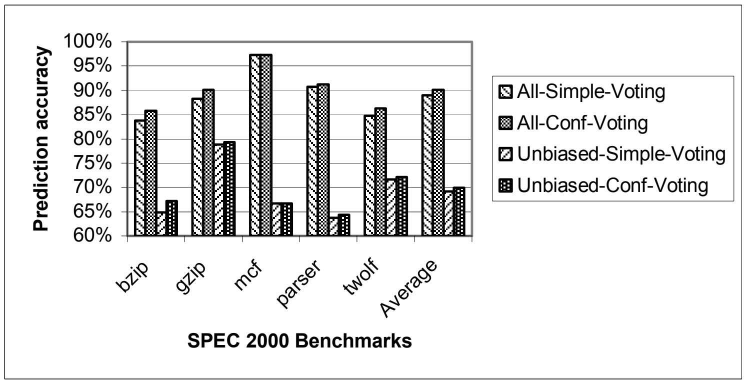


Figure 4.29. Branch difference prediction accuracies by combining multiple partial matches through simple voting and confidence-based voting

Our results show that the average prediction accuracy of the confidence voting algorithm is marginally better than the simple voting algorithm, as shown in Figure 4.29. The final branch prediction accuracy, obtained using the speculative branch differences generated by combining multiple partial matches through confidence-based voting, was 91.59% on all branches and only 72.24% on unbiased branches.

We also studied the influence of the threshold's value over the prediction accuracy by combining multiple partial matches through confidence-based voting, using a BDHT with 256 entries, and a local history of 24 branch difference signs. In this case, the confidence-based voting takes the majority, considering only Markov predictions found in the history string after the considered pattern at least *T* (threshold) times.

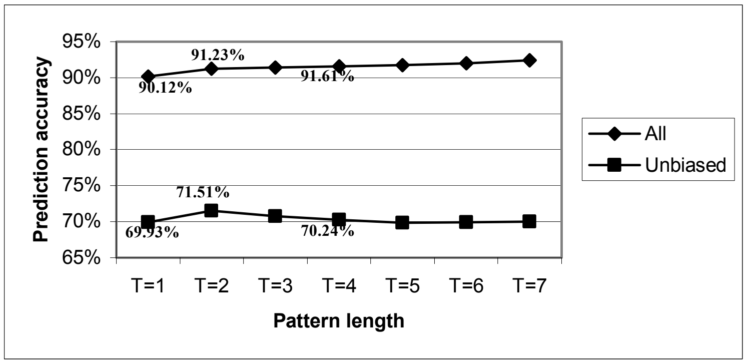


Figure 4.30. Branch difference prediction accuracies by combining multiple partial matches through confidence-based voting with different thresholds

Threshold	Lost predictions [%]
T=1	2,25
T=2	5,20
T=3	6,62
T=4	8,06
T=5	9,40
T=6	10,78
T=7	13,02
T=8	2,25

Table 4.2. Average percentages of predictions lost by using different thresholds

The experimental results presented in Figure 4.30 and Table 4.2 show that the optimal threshold value is 2. Thus, the final branch prediction accuracy by combining multiple partial matches through confidence-based voting with a threshold of 2 is 73.05% on unbiased branches and 92.42% on all branches.

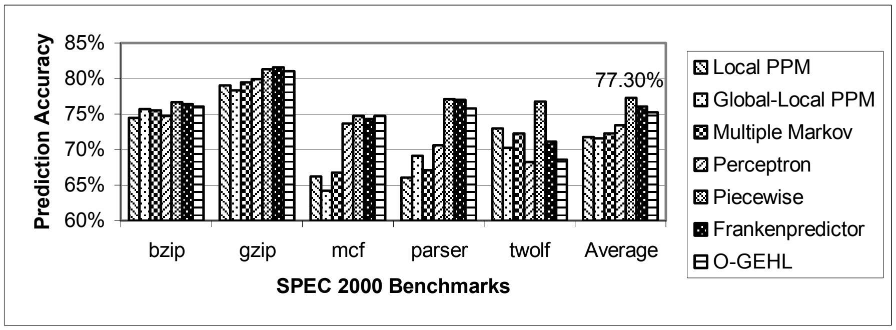


Figure 4.31. Branch prediction accuracies obtained using the perceptron-based predictors, the O-GEHL predictor and the PPM-based predictors, only on unbiased branches

Figure 4.31 shows again, that the unbiased branches identified in [Gel06a, Oan06, Vin06] cannot be accurately predicted even with condition-history-based Markov predictors. The highest average prediction accuracy on the unbiased branches, of 77.30%, was provided by the piecewise linear branch predictor.

4.5.3. Evaluating PBC-Based Branch Predictors

4.5.3.1 Evaluating the Global-PBC-Based GAg Predictor

Figure 4.32 presents the prediction accuracies obtained with the modified GAg predictor on unbiased branches.

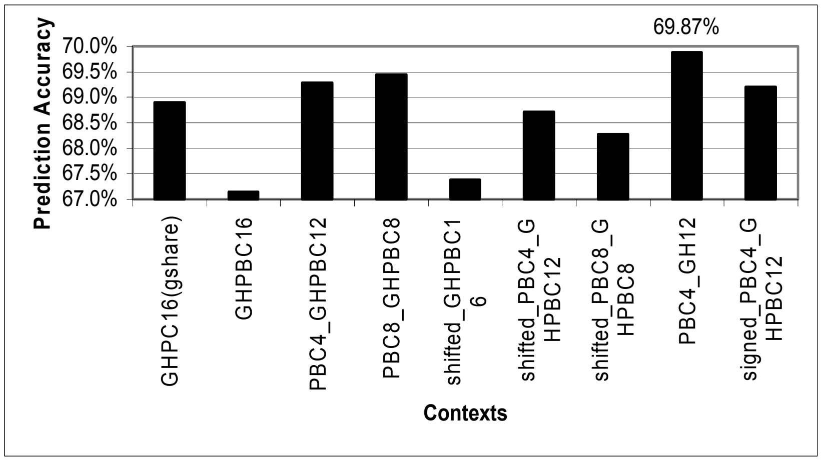


Figure 4.32. Average prediction accuracies of the modified GAg predictor on unbiased branches

The following contexts have been used with the modified GAg predictor (in Figure 4.32):

- GHPC16: the 16 least significant bits of the branch PC (shifted to right by 3 bits) XORed with 16 bits of global history (*gshare* predictor);
- GHPBC16: 16 least significant bits of PBC value XORed with 16 bits of global branch history;
- PBC4-GHPBC12: 4 least significant bits of PBC value concatenated with the XOR between 12 least significant bits of PBC value and 12 bits of global branch history;
- PBC8-GHPBC8: 8 least significant bits of PBC value concatenated with the XOR between 8 least significant bits of PBC value and 8 bits of global branch history;
- Shifted-GHPBC16: the 16 least significant bits of PBC value (shifted to right by 3 bits) XORed with 16 bits of global history;
- Shifted-PBC4-GHPBC12: 4 least significant bits of PBC value (shifted to right by 3 bits) concatenated with the XOR between 12 least significant bits of PBC value (shifted to right by 3 bits) and 12 bits of global branch history;
- Shifted-PBC8-GHPBC8: 8 least significant bits of PBC value (shifted to right by 3 bits) concatenated with the XOR between 8 least significant bits of PBC value (shifted to right by 3 bits) and 8 bits of global branch history;
- PBC4-GH12: 4 least significant bits of PBC value concatenated with 12 bits of global branch history;
- Signed-PBC4-GHPBC12: sign bit of PBC value (0 if positive, 1 if negative) concatenated with 3 least significant bits of PBC value and with the XOR between 12 least significant bits of PBC value and 12 bits of global branch history.

4.5.3.2 Evaluating the Local-PBC-Based PAg Predictor

Figure 4.33 presents the prediction accuracies obtained with the modified PAg predictor on unbiased branches.

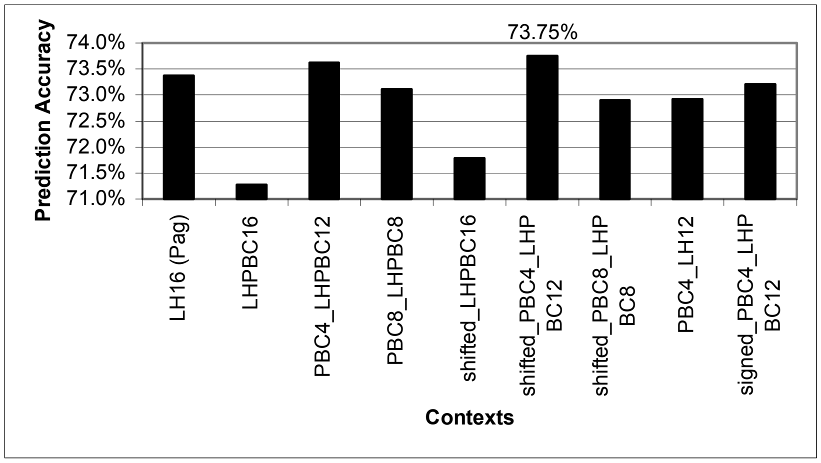


Figure 4.33. Average prediction accuracies of the modified PAg predictor on unbiased branches

The second level (GPHT) is indexed, depending on the used context, as follows:

- LH16: the second level is indexed by 16 bits of local branch history (PAg predictor);
- LHPBC16: 16 least significant bits of PBC value XORed with 16 bits of local branch history;
- PBC4-LHPBC12: 4 least significant bits of PBC value concatenated with the XOR between 12 least significant bits of PBC value and 12 bits of local branch history;
- PBC8-LHPBC8: 8 least significant bits of PBC value concatenated with the XOR between 8 least significant bits of PBC value and 8 bits of local branch history;
- Shifted-LHPBC16: 16 least significant bits of PBC value (shifted to right by 3 bits) XORed with 16 bits of local history;
- Shifted-PBC4-LHPBC12: 4 least significant bits of PBC value (shifted to right by 3 bits) concatenated with the XOR between 12 least significant bits of PBC value (shifted to right by 3 bits) and 12 bits of local branch history;
- Shifted-PBC8-LHPBC8: 8 least significant bits of PBC value (shifted to right by 3 bits) concatenated with the XOR between 8 least significant bits of PBC value (shifted to right by 3 bits) and 8 bits of local branch history;
- PBC4-LH12: 4 least significant bits of PBC value concatenated with 12 bits of local branch history;
- Signed-PBC4-LHPBC12: sign bit of PBC value (0 if positive, 1 if negative) concatenated with 3 least significant bits of PBC value and with the XOR between 12 least significant bits of PBC value and 12 bits of local branch history.

4.5.3.3 Evaluating the Global-PBC-Based Piecewise Linear Branch Predictor

Figure 4.34 presents the prediction accuracies obtained on all branches and on the unbiased branches with our best proposed and implemented predictor: the idealized piecewise linear branch predictor using the global PBC value as additional prediction information. The first two bars represent the prediction accuracies on all branches and on unbiased branches, obtained with the idealized piecewise linear branch predictor (PW). The rest of the bars were obtained using the PBC value (32 bits) as additional prediction information, varying the number of weights (from 8590 up to 30713).

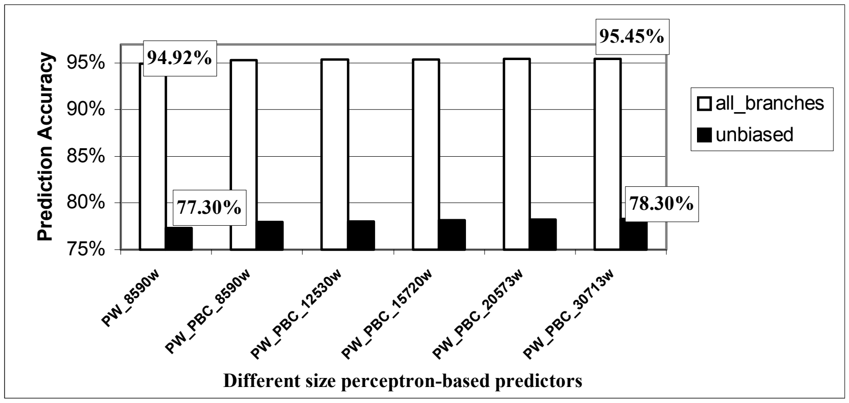


Figure 4.34. Average prediction accuracies obtained with *piecewise linear branch predictor* on unbiased branches versus all branches, using the global PBC value as additional prediction information

With the modified *piecewise linear branch predictor* we obtained a prediction accuracy of 78.30% (see Figure 4.34) opposite to those obtained with the modified GAg, 69.87% (see Figure 4.32) and the modified PAg, 73.75% (see Figure 4.33). This gain was probably obtained because both the modified GAg and PAg predictors use a hashing between PBC value and global/local branch history, while the modified *piecewise linear branch predictor* uses the branch history and PBC value without hashing (by concatenating them).

Analyzing comparatively the results presented in Figures 4.31 and 4.34 it can be observed how the PBC value determines the improvement of unbiased branch prediction accuracy, overcoming with at least 1% the best state of the art predictor's performance. Even if the improvement seems less significant, it is very clear how this small percentage contributes to the global prediction accuracy (value that overcomes with more than 0.53% the best state of the art predictor's performance).

4.5.3.4 Evaluating the Local-PBC-Based Piecewise Linear Branch Predictor

Figure 4.35 presents the prediction accuracies obtained with the perceptron-based branch predictor that is using as prediction information local (per-address) previous branch condition (PBC). Unfortunately, we have not obtained any improvement with the local PBC approach opposite to the global PBC approach, the accuracies being even lower.

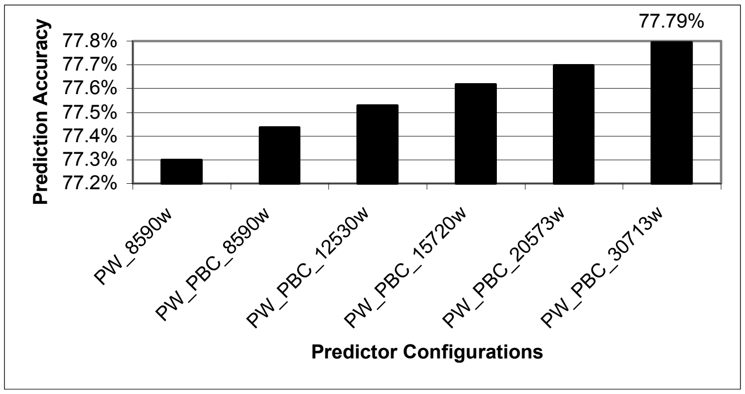


Figure 4.35. Average prediction accuracies of the *piecewise linear branch predictor* on unbiased branches, using the local (per-address) PBC value as additional prediction information

Consequently, based on laborious simulations we showed that the percentages of difficult branches are quite significant, depending on the different used contexts and their lengths, giving a new research challenge and a useful niche for branch prediction research. We showed that these difficult predictable branches cannot be well-predicted using state of the art predictors. They need some specific efficient predictors that are using some new more relevant prediction information. Finding a new relevant context to aggressively reduce the number of unbiased shuffled branches remains an open problem. Computer Architects cannot therefore continue to expect a prediction accuracy improvement with conventional predictors and alternative approaches are necessary.

4.5.3.5 Prediction Accuracy Improvements with PBC

Figure 4.36 shows comparatively the prediction accuracies obtained on unbiased branches using predictors with and without PBC. All the evaluated branch predictors that are using the PBC as additional prediction information are more accurate than the original versions (without PBC).

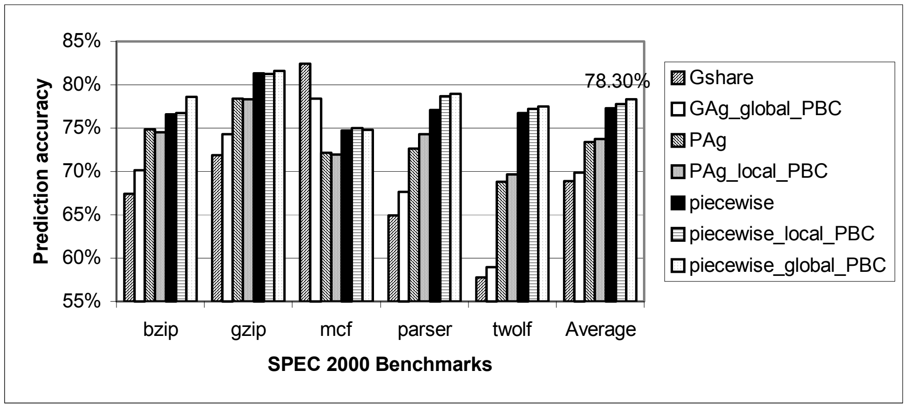


Figure 4.36. Prediction accuracy on unbiased branches using predictors with and without PBC

4.6. Summary

We showed that the best state of the art branch predictors [CBP04, CBP06] are obtaining very low prediction accuracies on unbiased branches, at average about 70% [Gel07b, Gel07c]. The same predictors are predicting a "normal" branch with accuracies ranging between 95% and 99%. These predictors are usually hybrid: Markovian, PPM-based, and neural. The unbiased branches cannot be accurately predicted even with the actual most powerful branch predictors. This fact is perfectly normal taking into account that the problem consists in better representing the unbiased branches in a new efficient feature space rather in finding better prediction structures. The highest average prediction accuracy on the unbiased branches, of 77.30%, was provided by the idealized piecewise linear branch predictor [Jim05]. This low prediction rate is understandable taking into account that even a neural predictor cannot effectively learn unbiased branches. As a comparison, the same predictor obtained far better average prediction accuracy, of 94.92%, on all branches.

We have also used the value of previous branch condition (PBC) as additional prediction information in some state-of-the-art branch predictors in order to increase their prediction accuracy. Our evaluations showed that the PBC value improves the accuracy of idealized piecewise linear branch predictor on unbiased branches with at least 1%. Even if the improvement seems less significant, it is very clear how this small percentage contributes to the global prediction accuracy, which increased with 0.53%.

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"

John von Neumann

5. Better Understanding Unbiased Branches Using Random Degrees

As we stated out in the previous chapter, the unbiased branches behavior is practically unpredictable. Why is this? Are these special branches unpredictable due to some relevant information misses or are they "random"? However, they were obtained by compiling some deterministic programs; therefore they were not randomly generated. But... what is random? During this chapter we try to understand random strings of symbols from a mathematical point of view in order to practically propose some concrete metrics characterizing them. These metrics could help us to better understand and analyze the unbiased branches behavior and their potential predictability.

A pragmatic aim consists in finding some deterministic hidden information that could reduce the unbiased branches' entropy. This is extremely difficult at least from two reasons: first, due to the enormous complexity of the benchmarks' dynamic behavior and, second, due to the fact that the simulated object code obviously has far less semantics comparing with the high-level language program. However, we consider that our developed random degrees could indicate the chance for uncovering this new relevant infomation. A high random degree might indicate a huge complexity and therefore, small chances to discover the right useful information.

5.1. Related Work

This section presents a brief related work on characterizing random sequences from a fundamental mathematical point of view and on applications of Hidden Markov Models (HMM) in different Computer Science areas.

5.1.1. What is a Random Sequence?

The questions are: is it possible to give an intrinsic or ontological definition of a random string of symbols? Could generate a deterministic program a "random" sequence? Mathematicians show that for strings it is only possible to develop a notion of randomness degree, the difference between random and nonrandom being therefore quite fuzzy [Vin08b].

There is a strong logic connection between the concept of randomness and computability theory. It is natural to consider that any string of symbols generated by an algorithm is not random (it could be perfectly predicted through a predictor implementing that algorithm). For an in-depth rigorous definition of randomness it is necessary to use the fertile Turing Machine (TM) concept. Any binary input sequence in a *TM* belongs to the so-called Finite Binary set (FB) and it codifies the input data. The *TM* can be in one state belonging to the set $Q = \{q_0 q_1 q_2 \ldots q_f\}$, where q_0 is the initial state (start) and q_f is the final state (stop). Depending on the current input symbol $s(t)$ and the current state $q(t)$, the *TM* generates the new symbol $s(t+1)$ and transits to a new state $q(t+1)$. Thus, each step can be described by $TM_t(x) = \{q(t),\ s(t),\ s(t+1),\ q(t+1),\ m\}$ where $m \in M = \{L, R\}$. $TM_t(x)$ is also called instruction, and the entire instruction sequence represents the program executed by the *TM*. For $\forall x \in FB$, if the *TM* reaches its final state, it generates the corresponding output sequence *TM*(*x*). Therefore, from a formal point of view, a *TM* is a function $f : Q \times S \to Q \times S \times M$.

The set of all *TM*s is a countable (infinite) set; therefore the *TM* set can be put in a one-to-one correspondence with the natural number set (N). This fact can be easily justified. Considering all the *TM*s with *k* instructions (noted TM^k) it means that one can then order the machines by the increasing size of the instruction set (TM^{k+1}), involving that the (TM^k) set is countable $\forall k = 2, 3, 4, 5...$ Therefore, taking into account that each certain TM^k set contains a finite number of *TM*s it involves that the set of all *TM*s is countable, too $(TM_1, TM_2, ..., TM_n, ...)$. Thus, each *TM* defines a partial function from the set of *FB* strings to itself. Alternatively, if the function is defined for all strings in *FB*, then the function is said to be total.

Now each string in *FB* is a binary representation of a positive integer through an encoding function $c : FB \to N$. A partial function $f : N \to N$ is Turing computable if there is a *TM* such that, for every *n* in the domain of *f*, there is an input $x \in FB$ with $n = c(x)$ for which the machine eventually stops and such that the output $TM(x)$ satisfies $f(n) = c(TM(x))$. From the countability of the collection of all *TM*s, it follows immediately that the set of partial Turing computable functions is a countable subset of the much larger uncountable set of all partial functions from *N* to *N*. In this sense, very few functions are Turing computable even if the number of these computable functions is infinite (noted *Alef 0* in G. Cantor's mathematical theory of actual infinites). This means that the Turing non-computable function set is uncountable. It is well-known the Church-Turing thesis saying that for any algorithm (finite steps procedure) there is an equivalent *TM* [Vol02].

Returning to the random binary string's definition problem, as we already stated, it is obvious that such a string must not be generated through an algorithm (i.e., Turing computable function). Thus, any random binary string is generated by a non-computable Turing function. (Reciprocal, mathematicians showed that there are some non-computable sequences that are not random for sure!) Taking into account that the non-computable Turing function set is an uncountable set it involves that the random binary strings set is uncountable, too. Unfortunately this rigorous definition of a random sequence is useless because it cannot effectively generate any concrete random string. This is equivalent to say that majority of real numbers are random even if it is not possible to generate at least one example. Rigorously defining and effectively generating random sequences seems to be an open problem for the actual mathematics and also for the general studies related to cognitive and noetic behavior.

The practical idea of randomness as incompressibility was proposed independently in the sixties of previous century by R. Solomonoff, A. Kolmogorov, and G. Chaitin. The main intuition is that a string is random if it cannot be "described" more efficiently than by giving the whole string itself. Thus, a string is random if it is algorithmically incompressible or irreducible. According to this view, a string is random if no computer program of size substantially smaller than the string itself can generate or describe it. This is the notion of program size algorithmic or Kolmogorov complexity. Obviously, its concrete value depends on the particular formal language that implements the generator algorithm. As a consequence of this practical approach of randomness we propose the compression rate as a random degree of a string of symbols.

5.1.2. Prediction with Hidden Markov Models

Rabiner in his work [Rab89] shows how HMMs can be applied to selected problems in speech recognition. His paper presents the theory of HMMs from the simplest concepts (discrete Markov chains) to the most sophisticated models (variable duration, continuous density models, etc.). He also illustrated some applications of the theory of HMMs to simple problems in speech recognition, and pointed out how the techniques have been applied to more advanced speech recognition problems.

Liu et al. in their work [Liu03], describe a HMM based framework for hand gesture detection and recognition. The goal of gesture interpretation is to improve human-machine communication and to bring human-machine interaction closer to human-human interaction, making possible new applications such as sign language translation. They present an efficient method for extracting the observation sequence using the feature model and Vector Quantization, and demonstrate that, compared to the classic template-based methods, the HMM-based approach offers a more flexible framework for recognition.

Machine Learning techniques based on HMMs have been also applied to problems in computational biology and they can be used as mathematical models of molecular processes and biological sequences. The goal of computational biology is to elucidate additional information required for drug design, medical diagnosis and medical treatment. The majority of molecular data used in computational biology consists in sequences of nucleotides corresponding to the primary structure of DNA and RNA, or sequences of amino acids corresponding to the primary structure of proteins. Birney in his work [Bir01] reviews gene-prediction HMMs and protein family HMMs. The role of gene-prediction in DNA is to discover the location of genes on the genome. HMMs have also been used in protein profiling to discriminate between different protein families and predict a new protein-family or subfamily. Yoon et al. in their work [Yoo04], proposed a new method based on context-sensitive HMMs, which can be used for predicting RNA secondary structure. The RNA secondary structure results from the base pairs formed by the nucleotides of RNA. The context-sensitive HMM can be viewed as an extension of the traditional HMM, where some of the states are equipped with auxiliary memory. Symbols that are emitted at certain states are stored in the memory, and they serve as the context that affects the emission and transition probabilities of the model. They demonstrated that the proposed model predicts the secondary structure very accurately, at a low computational cost.

In our previous work [Gel06c] we focused on a Hidden Markov Model (HMM) approach for context prediction in a ubiquitous computing application. Our application predicts the next room based on the history of rooms, visited by a certain person moving within an office building. We introduced the HMM-based predictors and compared them with simple Markov and neural predictors [Vin04b, Vin04c]. We evaluated these predictors by some movement sequences of real persons, acquired from the Smart Doorplates project developed at Augsburg University [Pet04]. The experimental results show that HMMs outperform other implemented prediction techniques such as Neural Networks and Markov predictors. Predicting from all rooms excepting own room and using a HMM with 4-state confidence automata, we obtained an average prediction accuracy of 84.81%, but the prediction accuracy measured on some local predictors grew up to over than 92%.

5.2. Random Degree Metrics for Characterizing Unbiased Branches Behavior

This section presents, based on our bibliographical research [Rab89, Gam99, Cor01, and Vol02], some practical ideas proposed in [Vin08b] for characterizing sequences generated by unbiased branches from the random degree viewpoint.

5.2.1. Random Degree Metric Based on Hidden Markov Models

New relevant information could reduce the string's entropy and thus its random degree. Unfortunately this information might be very difficult or even impossible to be found. As a consequence we think it would be interesting trying to predict a sequence using HMMs like those developed in [Rab89, Gel06c]. A HMM is a doubly embedded stochastic process with a hidden stochastic process that can only be observed through another set of stochastic processes that generate the sequence of observable symbols. A generic HMM is illustrated in Figure 5.1, where q_t is the hidden state at time *t*, O_t is the observation at time *t*, *A* is the matrix of transition probabilities between hidden states, and *B* is the matrix of observation probabilities within each hidden state.

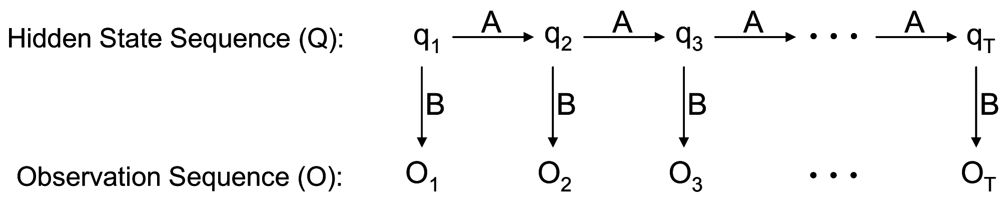

Hidden State Sequence (Q):
Observation Sequence (O):
$q_1 \xrightarrow{A} q_2 \xrightarrow{A} q_3 \xrightarrow{A} \cdots \xrightarrow{A} q_T$
B
O_1 O_2 O_3 $\cdots$ O_T

Figure 5.1. Hidden Markov Model

HMM predictors are very powerful adaptive stochastic models. Our hypothesis is that HMMs could compensate relevant information miss-knowledge through its underlying stochastic process that is not observable. HMM's prediction accuracy might be considered as an ultimate prediction limit. Therefore, we propose HMM prediction accuracy as another practical metric for calculating the random degree associated with a sequence of symbols. Of course, all these random degree metrics will be applied to our unbiased branches behaviors in order to estimate how much random they are.

5.2.1.1. First Order HMMs

Elements of a First Order HMM

1. N – the number of hidden states, with $S = \{S_0, S_1, ..., S_{N-1}\}$ the set of hidden states, and q_t the hidden state at time *t*. N will be varied in order to obtain the optimal value.
2. M – the number of observable states, with $V = \{V_0, V_1, ..., V_{M-1}\}$ the set of observable states (symbols), and O_t the observable state at time *t*.
3. A = $\{a_{ij}\}$ – the transition probabilities between the hidden states S_i and S_j, where $a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \quad 0 \le i, j \le N-1$.
4. B = $\{b_j(k)\}$ – the probabilities of the observable states V_k in hidden states S_j, where $b_j(k) = P[O_t = V_k | q_t = S_j], \quad 0 \le j \le N-1, \quad 0 \le k \le M-1$.
5. π = $\{\pi_i\}$ – the initial hidden state probabilities, where $\pi_i = P[q_1 = S_i], \quad 0 \le i \le N-1$.

We also defined the following variables:

- $\alpha_t(i) = P(O_1 O_2 ... O_t, q_t = S_i | \lambda)$ – the forward variable [Rab89], representing the probability of the partial observation sequence until time *t*, and hidden state S_i at time *t*, given the model $\lambda = (A, B, \pi)$.
- $\beta_t(i) = P(O_{t+1} O_{t+2} ... O_T | q_t = S_i, \lambda)$ – the backward variable [Rab89], representing the probability of the partial observation sequence from *t*+1 to the end *T*, given hidden state S_i at time *t* and the model $\lambda = (A, B, \pi)$.
- $\xi_t(i, j) = P(q_t = S_i, q_{t+1} = S_j | O_1 O_2 ... O_T, \lambda)$ – the probability of being in hidden state S_i at time *t*, and hidden state S_j at time *t*+1, given the model $\lambda = (A, B, \pi)$ and the observation sequence.

- $\gamma_t(i) = P(q_t = S_i | O_1\, O_2\, ...\, O_T,\, \lambda)$ – the probability of being in hidden state S_i at time *t*, given the model $\lambda = (A, B, \pi)$ and the observation sequence.
- H – the history (the number of observations used in the prediction process). In [Rab89] and [Sta04] the entire observation sequence is used in the prediction process (*H*=*T*), but in some practical applications the observation sequence increases continuously, therefore its limitation is necessary. Thus, the last *H* observations can be stored in a left shift register.
- I – the maximum number of iterations in the adjustment process. Usually the adjustment process ends when the probability of the observation sequence does not increase anymore, but for a faster adjustment, the number of iterations is limited.

Adjustment Process of a First Order HMM

1. Initialize $\lambda = (A, B, \pi)$;
2. Compute $\alpha_t(i),\ \beta_t(i),\ \xi_t(i, j),\ \gamma_t(i),\ \ t = 1, ..., T, \quad i = 0, ..., N-1, \ \ j = 0, ..., N-1$;
3. Adjust the model $\lambda = (A, B, \pi)$;
4. If $P(O|\lambda)$ increases, go to 2.

The model parameters (A, B, π) are adjusted in order to maximize the probability of the observation sequence. The model $\lambda = (A, B, \pi)$ can be chosen such that $P(O|\lambda)$ is locally maximized using an iterative procedure, or using gradient techniques. In this work we use the Baum-Welch iterative method introduced by Baum et al. [Bau72]. The Baum-Welch algorithm – identical to the Expectation Maximization (EM) method for this particular problem – improves iteratively an initial model. If we define the current model as $\lambda = (A, B, \pi)$ and use it to compute the reestimated model $\overline{\lambda} = (\overline{A}, \overline{B}, \overline{\pi})$ – through steps 3.5, 3.6 and 3.7 from the prediction algorithm –, then, as it has been proven by Baum, the model $\overline{\lambda}$ is more likely than model λ in the sense that $P(O|\overline{\lambda}) \geq P(O|\lambda)$. Thus, if $\overline{\lambda}$ is used iteratively in place of λ repeating the reestimation calculation, the probability of the observation sequence can be improved until some limiting point is reached. Rabiner show in [Rab89] that the same reestimation formulas can be obtained using the techniques of Lagrange multipliers.

Initialization of the First Order Model

- The transition probabilities between the hidden states $A(N \times N) = \{a_{ij}\}$, are randomly initialized to approximately 1/N; the sum of each row's elements must be 1.
- The probabilities of the observable states $B(N \times M) = \{b_j(k)\}$, are randomly initialized to approximately 1/M; the sum of each row's elements must be 1.
- The initial hidden state probabilities $\pi(1 \times N) = \{\pi_i\}$ are randomly set to approximately 1/N, their sum being 1.

Prediction Algorithm Using a First Order HMM

1.) T=H (T is the length of the observation sequence);
2.) c=0 (c is the number of current iteration, its maximum value is given by I);
3.) The model $\lambda = (A, B, \pi)$ is repeatedly adjusted based on the last H observations $O_{T-H+1}, O_{T-H+2}, ..., O_T$ (the entire observation sequence if *H*=*T*), in order to increase the

probability of the observation sequence $P(O_{T-H+1}\, O_{T-H+2} \,...\, O_T | \lambda)$. In 3.1, 3.2 and 3.3 steps the denominators are used in order to obtain a probability measure, and to avoid underflow. As Stamp showed in [Sta04], underflow is inevitable without scaling, since the probabilities tend to 0 exponentially as T increases.

3.1. Compute the forward variable α in a recursive manner:

$$\alpha_{T-H+1}(i) = \frac{\pi_i \cdot b_i(O_{T-H+1})}{\sum_{i=0}^{N-1} \pi_i \cdot b_i(O_{T-H+1})},\ i = 0,...,N-1,$$ where $\alpha_{T-H+1}(i)$ is the probability of

observation symbol O_{T-H+1} and initial hidden state S_i, given the model $\lambda = (A, B, \pi)$;

$$\alpha_t(j) = \frac{\sum_{i=0}^{N-1} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)}{\sum_{j=0}^{N-1}\sum_{i=0}^{N-1} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)},\ t = T-H+2,...,T,\quad j = 0,...,N-1,$$ where $\alpha_t(j)$

is the probability of the partial observation sequence until time t ($O_{T-H+1} \,...\, O_t$), and hidden state S_j at time t, given the model $\lambda = (A, B, \pi)$. Since, by definition,

$\alpha_T(j) = P(O_{T-H+1}\, O_{T-H+2} \,...\, O_T,\, q_T = S_j | \lambda)$,

the sum of the terminal forward variables $\alpha_T(j)$ gives the probability of the observation sequence:

$P(O_{T-H+1}\, O_{T-H+2} \,...\, O_T | \lambda) = \sum_{j=0}^{N-1} \alpha_T(j)$.

3.2. Compute the backward variable β in a recursive manner:

$$\beta_T(i) = \frac{1}{\sum_{j=0}^{N-1}\sum_{i=0}^{N-1} \alpha_{T-1}(i) \cdot a_{ij} \cdot b_j(O_T)},\ i = 0,...N-1;$$

$$\beta_t(i) = \frac{\sum_{j=0}^{N-1} a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)},\quad t = T-1,...,T-H+1,\quad i = 0,...,N-1,$$ where

$\beta_t(i)$ is the probability of the partial observation sequence from t+1 to the end T ($O_{t+1}O_{t+2}...O_T$), given hidden state S_i at time t and the model $\lambda = (A, B, \pi)$.

3.3. Compute ξ:

$$\xi_t(i,j) = \frac{\alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} \alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1})\beta_{t+1}(j)},\quad t = T-H+1,...,T-1,\quad i = 0,...,N-1,$$

$$j = 0,...,N-1$$

where $\xi_t(i,j)$ is the probability of being in hidden state S_i at time t and in S_j at time t+1, given the observation sequence $O_{T-H+1}\, O_{T-H+2} \,...\, O_T$ and the model $\lambda = (A, B, \pi)$.

3.4. Compute γ:

$\gamma_t(i) = \sum_{j=0}^{N-1} \xi_t(i,j), \ \ t = T-H+1,...,T-1, \ \ i = 0,...,N-1$, where $\gamma_t(i)$ is the probability of being in the hidden state S_i at time *t*, given the model $\lambda = (A,B,\pi)$ and the observation sequence $O_{T-H+1}\, O_{T-H+2} \ldots O_T$.

3.5. Adjust π:

$\overline{\pi_i} = \gamma_{T-H+1}(i)$ – represents the expected number of times the hidden state is S_i at the initial time $t = T-H+1$.

3.6. Adjust *A*:

$$\overline{a_{ij}} = \frac{\sum_{t=T-H+1}^{T-1} \xi_t(i,j)}{\sum_{t=T-H+1}^{T-1} \gamma_t(i)}$$ – represents the probability of transition from hidden state S_i to S_j.

The numerator is the expected number of transitions from state S_i to S_j, while the denominator is the expected number of transitions from state S_i to any state.

3.7. Adjust *B*:

$$\overline{b_j(k)} = \frac{\sum_{\substack{t=T-H+1 \\ O_t = V_k}}^{T-1} \gamma_t(j)}{\sum_{t=T-H+1}^{T-1} \gamma_t(j)}$$ – the probability of observation symbol V_k given that the model is in hidden state S_j. The numerator is the expected number of times the model is in hidden state S_j and the observation symbol is V_k, while the denominator is the expected number of times the model is in hidden state S_j.

3.8. c=c+1;

if $\log[P(O_{T-H+1} \ldots O_T | \overline{\lambda})] > \log[P(O_{T-H+1} \ldots O_T | \lambda)]$ and c<I then go to 3.).

Since *P* would be out of the dynamic range of the machine [Rab89], we compute the logarithm of *P*, using the following formula [Sta04]:

$$\log[P(O_{T-H+1} \ldots O_T | \overline{\lambda})] = -\log\left(\frac{1}{\sum_{i=0}^{N-1} \pi_i \cdot b_i(O_{T-H+1})}\right) - \sum_{t=T-H+2}^{T} \log\left(\frac{1}{\sum_{j=0}^{N-1}\sum_{i=0}^{N-1} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)}\right)$$

4.) At current time *T*, the next observation symbol O_{T+1} is predicted, using the adjusted model $\overline{\lambda} = (\overline{A}, \overline{B}, \overline{\pi})$:

4.1. choose hidden state S_i at time *T*, $i = 0, \ldots, N-1$, maximizing $\alpha_T(i)$;

4.2. choose next hidden state S_j (at time $T+1$), $j = 0, \ldots, N-1$, maximizing $\overline{a_{ij}}$;

4.3. predict next symbol V_k (at time $T+1$), $k = 0, \ldots, M-1$, maximizing $\overline{b_j(k)}$.

If the process continues, then $T = T+1$ and go to 2.).

5.2.1.2. A Possible Generalization: Superior Order HMMs

In this paragraph we present a Hidden Markov Model of order *R*, $R \geq 1$, based on our work published in [Gel06c]. There are multiple possibilities for doing this but we present here only one we considered the most appropriate due to its simplicity. The key of our proposed model is represented by the so-called hidden super-states, a combination of *R* primitive hidden states. Therefore, the main difference, comparing with a first order HMM, consists in the fact that the stochastic hidden Markov model is of order *R* instead of order one. This new model is justified because we suppose that in some specific applications, there are longer correlations within the hidden state model. In other words, we suppose that the next hidden state is better determined by the current super-state rather than by the current primitive state. As it can be further seen, the new proposed model is similar with the well-known HMM of order one, excepting the fact that the generic primitive hidden state becomes now a generic super-state.

Elements of a Superior Order HMM

1. R – the order of HMM (a combination of *R* primitive hidden states form a so called super-state).
2. N – the number of primitive hidden states (belonging to a HMM of order 1), with $S = \{S_0, S_1, ..., S_{N^R-1}\}$ being the set of hidden super-states and $q_t \in S$ the hidden super-state at time *t*. The current super-state determines the transition into the next one based on a super-state transition matrix with restrictions (this transition matrix involve a non-ergodic model, see example of Table 5.1). *N* will be varied in order to obtain the optimal value.
3. M – the number of observable states, with $V = \{V_0, V_1, ..., V_{M-1}\}$ the set of observable states (symbols), and O_t the observable state at time *t*.
4. A = $\{a_{ij}\}$ – the transition probabilities between the hidden super-states S_i and S_j, where $a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \quad 0 \leq i, j \leq N^R - 1$.
5. B = $\{b_j(k)\}$ – the probabilities of the observable states V_k, considering the current hidden super-state S_j, where $b_j(k) = P[O_t = V_k | q_t = S_j], \quad 0 \leq j \leq N^R - 1, \quad 0 \leq k \leq M - 1$.
6. π = $\{\pi_i\}$ – the initial hidden super-state probabilities, where $\pi_i = P[q_1 = S_i]$, $0 \leq i \leq N^R - 1$.

In order to simplify the terminology, in the rest of this chapter we'll refer to the hidden super-states as simply hidden states belonging to the HMM of order *R*.

We also define the following variables:

- $\alpha_t(i) = P(O_1 O_2 ... O_t, q_t = S_i | \lambda)$ – the forward variable [Rab89], representing the probability of the partial observation sequence until time *t*, and hidden state S_i at time *t*, given the model $\lambda = (A, B, \pi)$.
- $\beta_t(i) = P(O_{t+1} O_{t+2} ... O_T | q_t = S_i, \lambda)$ – the backward variable [Rab89], representing the probability of the partial observation sequence from *t*+1 to the end *T*, given hidden state S_i at time *t* and the model $\lambda = (A, B, \pi)$.
- $\xi_t(i, j) = P(q_t = S_i, q_{t+1} = S_j | O_1 O_2 ... O_T, \lambda)$ – the probability of being in hidden state S_i at time *t*, and hidden state S_j at time *t*+1, given the model $\lambda = (A, B, \pi)$ and the observation sequence.

- $\gamma_t(i) = P(q_t = S_i | O_1 O_2 \dots O_T, \lambda)$ – the probability of being in hidden state S_i at time *t*, given the model $\lambda = (A, B, \pi)$ and the observation sequence.
- H – the history (the number of observations used in the prediction process). In [Rab89] and [Sta04] the entire observation sequence is used in the prediction process (*H*=*T*), but in some practical applications the observation sequence increases continuously, therefore its limitation is necessary. Thus, the last *H* observations can be stored in a left shift register having a certain length.
- I – the maximum number of iterations in the adjustment process. Usually the adjustment process ends when the probability of the last *H* observations does not increase anymore, but for a faster adjustment, the number of iterations is limited.

For a HMM of order *R* with *N* primitive hidden states, the transition probabilities between the hidden states $A(N^R \times N^R) = \{a_{ij}\}$, are stored in a table with N^R rows and N^R columns but not all cells of the table are used; there are only *N* consistent (possible) transitions from each state involving a non-ergodic model. The following table, for example, corresponds to a HMM of order 3 (*R*=3) with 2 primitive hidden states (*N*=2):

States			**j**							
			0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
			AAA	**AAB**	**ABA**	**ABB**	**BAA**	**BAB**	**BBA**	**BBB**
i	**0**	**AAA**	X	X						
	1	**AAB**			X	X				
	2	**ABA**					X	X		
	3	**ABB**							X	X
	4	**BAA**	X	X						
	5	**BAB**			X	X				
	6	**BBA**					X	X		
	7	**BBB**							X	X

Table 5.1. Consistent transitions for a HMM of order 3 (*R*=3), with 2 primitive hidden states (*N*=2)

Only the consistent cells marked with "X" are used, because transitions are possible only between states which end and start with the same $(R-1)$ primitive hidden states. The consistent cells of the transition table are given by the following formulas:

- For next hidden states (columns) $j = 0, \dots, N^R - 1$, are consistent only the current hidden states (rows) $i = \left[\frac{j}{N}\right] + 0 \cdot N^{R-1}, \dots, \left[\frac{j}{N}\right] + (N-1) \cdot N^{R-1}$;
- For current hidden states (rows) $i = 0, \dots, N^R - 1$, are consistent only the next hidden states (columns) $j = (i \bmod N^{R-1}) \cdot N, \dots, (i \bmod N^{R-1}) \cdot N + N - 1$.

The HMM of order *R* is similar with a first order HMM with the above state transition restrictions.

Adjustment Process of a Superior Order HMM

1. Initialize $\lambda = (A, B, \pi)$;
2. Compute $\alpha_t(i),\ \beta_t(i),\ \xi_t(i,j),\ \gamma_t(i),\quad t = 1, \dots, T,\quad i = 0, \dots, N^R - 1,\quad j = 0, \dots, N^R - 1$;
3. Adjust the model $\lambda = (A, B, \pi)$;
4. If $P(O|\lambda)$ increases, go to 2.

Initialization of the Superior Order Model

- The transition probabilities between the hidden states $A(N^R \times N^R) = \{a_{ij}\}$, are randomly initialized to approximately 1/N; the sum of each row's elements must be 1. The hidden state transition probabilities are initialized for $i = 0,..., N^R - 1$ and $j = (i \bmod N^{R-1}) \cdot N,...,(i \bmod N^{R-1}) \cdot N + N - 1$.
- The probabilities of the observable states $B(N^R \times M) = \{b_j(k)\}$, are randomly initialized to approximately 1/M; the sum of each row's elements must be 1.
- The initial hidden state probabilities $\pi(1 \times N^R) = \{\pi_i\}$ are randomly set to approximately $1/N^R$, their sum being 1.

Prediction Algorithm Using a Superior Order HMM

1.) T=H (T is the length of the observation sequence);
2.) c=0 (c is the number of current iteration, its maximum value is given by I);
3.) The model $\lambda = (A, B, \pi)$ is repeatedly adjusted based on the last H observations $O_{T-H+1}, O_{T-H+2},..., O_T$ (the entire observation sequence if *H=T*), in order to increase the probability of the observation sequence $P(O_{T-H+1}\, O_{T-H+2}\, ...\, O_T | \lambda)$. In 3.1, 3.2 and 3.3 the denominators are used in order to obtain a probability measure, and to avoid underflow. As Stamp showed in [Sta04], underflow is inevitable without scaling, since the probabilities tend to 0 exponentially as *T* increases.

3.1. Compute the forward variable α in a recursive manner:

$\alpha_{T-H+1}(i) = \dfrac{\pi_i \cdot b_i(O_{T-H+1})}{\sum_{i=0}^{N^R-1} \pi_i \cdot b_i(O_{T-H+1})}$, $i = 0,..., N^R - 1$, where $\alpha_{T-H+1}(i)$ is the probability of observation symbol O_{T-H+1} and initial hidden state S_i, given the model $\lambda = (A, B, \pi)$;

$$\alpha_t(j) = \frac{\sum_{i=\left[\frac{j}{N}\right]+0\cdot N^{R-1}}^{\left[\frac{j}{N}\right]+(N-1)\cdot N^{R-1}} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)}{\sum_{j=0}^{N^R-1} \sum_{i=\left[\frac{j}{N}\right]+0\cdot N^{R-1}}^{\left[\frac{j}{N}\right]+(N-1)\cdot N^{R-1}} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)}, \quad t = T - H + 2,..., T, \quad j = 0,..., N^R - 1,$$

where $\alpha_t(j)$ is the probability of the partial observation sequence until time *t* ($O_{T-H+1}\, ...\, O_t$), and hidden state S_j at time *t*, given the model $\lambda = (A, B, \pi)$. Since, by definition,

$$\alpha_T(j) = P(O_{T-H+1}\, O_{T-H+2}\, ...\, O_T,\, q_T = S_j | \lambda),$$

the sum of the terminal forward variables $\alpha_T(j)$ gives the probability of the observation sequence:

$$P(O_{T-H+1}\, O_{T-H+2}\, ...\, O_T | \lambda) = \sum_{j=0}^{N^R-1} \alpha_T(j).$$

3.2. Compute the backward variable β in a recursive manner:

$$\beta_T(i) = \frac{1}{\sum_{j=0}^{N^R-1} \sum_{i=\left[\frac{j}{N}\right]+0\cdot N^{R-1}}^{\left[\frac{j}{N}\right]+(N-1)\cdot N^{R-1}} \alpha_{T-1}(i)\cdot a_{ij}\cdot b_j(O_T)}, \; i = 0,\ldots N^R-1;$$

$$\beta_t(i) = \frac{\sum_{j=(i \bmod N^{R-1})\cdot N}^{(i \bmod N^{R-1})\cdot N+N-1} a_{ij}\cdot b_j(O_{t+1})\cdot \beta_{t+1}(j)}{\sum_{i=0}^{N^R-1} \sum_{j=(i \bmod N^{R-1})\cdot N}^{(i \bmod N^{R-1})\cdot N+N-1} a_{ij}\cdot b_j(O_{t+1})\cdot \beta_{t+1}(j)}, \; t = T-1,\ldots,T-H+1, \; i = 0,\ldots,N^R-1,$$

where $\beta_t(i)$ is the probability of the partial observation sequence from *t*+1 to the end *T* ($O_{t+1}O_{t+2}\ldots O_T$), given hidden state S_i at time *t* and the model $\lambda = (A,B,\pi)$.

3.3. Compute ξ:

$$\xi_t(i,j) = \frac{\alpha_t(i)\cdot a_{ij}\cdot b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=0}^{N^R-1} \sum_{j=(i \bmod N^{R-1})\cdot N}^{(i \bmod N^{R-1})\cdot N+N-1} \alpha_t(i)\cdot a_{ij}\cdot b_j(O_{t+1})\beta_{t+1}(j)}, \; t = T-H+1,\ldots,T-1,$$

$i = 0,\ldots,N^R-1, j = (i \text{ mod } N^{R-1})\cdot N,\ldots,(i \text{ mod } N^{R-1})\cdot N+N-1$, where $\xi_t(i,j)$ is the probability of being in hidden state S_i at time *t* and in S_j at time *t*+1, given the observation sequence $O_{T-H+1}\, O_{T-H+2}\ldots O_T$ and the model $\lambda = (A,B,\pi)$.

3.4. Compute γ:

$\gamma_t(i) = \sum_{j=(i \bmod N^{R-1})\cdot N}^{(i \bmod N^{R-1})\cdot N+N-1} \xi_t(i,j), \; t = T-H+1,\ldots,T-1, \; i = 0,\ldots,N^R-1$, where $\gamma_t(i)$ is the probability of being in hidden state S_i at time *t*, given the model $\lambda = (A,B,\pi)$ and the observation sequence $O_{T-H+1}\, O_{T-H+2}\ldots O_T$.

3.5. Adjust π:
$\overline{\pi_i} = \gamma_{T-H+1}(i)$ – represents the expected number of times the hidden state is S_i $(i = 0,\ldots,N^R-1)$ at the initial time $t = T-H+1$.

3.6. Adjust A:

$\overline{a_{ij}} = \frac{\sum_{t=T-H+1}^{T-1}\xi_t(i,j)}{\sum_{t=T-H+1}^{T-1}\gamma_t(i)}$ – the probability of transition from hidden state S_i to S_j,

where $i = 0,\ldots,N^R-1$ and $j = (i \text{ mod } N^{R-1})\cdot N,\ldots,(i \text{ mod } N^{R-1})\cdot N+N-1$.
The numerator is the expected number of transitions from state S_i to S_j, while the denominator is the expected number of transitions from state S_i to any state.

3.7. Adjust *B*:

$\overline{b_j(k)} = \dfrac{\sum\limits_{\substack{t=T-H+1 \\ O_t=V_k}}^{T-1} \gamma_t(j)}{\sum\limits_{t=T-H+1}^{T-1} \gamma_t(j)}$ – the probability of observation symbol V_k $(k = 0,...,M-1)$ given that the model is in hidden state S_j $(j = 0,...,N^R - 1)$. The numerator is the expected number of times the model is in hidden state S_j and the observation symbol is V_k, while the denominator is the expected number of times the model is in hidden state S_j.

3.8. c=c+1;

if $\log[P(O_{T-H+1}...O_T|\overline{\lambda})] > \log[P(O_{T-H+1}...O_T|\lambda)]$ and c<I then go to 3.).

Since *P* would be out of the dynamic range of the machine [Rab89], we compute the log of *P*, using the following formula [Sta04]:

$$\log[P(O_{T-H+1}\,...\,O_T|\overline{\lambda})] = -\log\left(\frac{1}{\sum\limits_{i=0}^{N^R-1} \pi_i \cdot b_i(O_{T-H+1})}\right) - \sum_{t=T-H+2}^{T} \log\left(\frac{1}{\sum\limits_{j=0}^{N^R-1} \sum\limits_{i=\left[\frac{j}{N}\right]+0\cdot N^{R-1}}^{\left[\frac{j}{N}\right]+(N-1)\cdot N^{R-1}} \alpha_{t-1}(i)\cdot a_{ij}\cdot b_j(O_t)}\right)$$

4.) At time *T*, the next observation symbol O_{T+1} is predicted, using the adjusted model $\overline{\lambda} = (\overline{A}, \overline{B}, \overline{\pi})$:

4.1. choose hidden state S_i at time *T*, $i = 0,...,N^R - 1$, maximizing $\alpha_T(i)$;

4.2. choose next hidden state S_j (at time $T+1$), $j = (i \bmod N^{R-1})\cdot N,...,(i \bmod N^{R-1})\cdot N + N - 1$, maximizing $\overline{a_{ij}}$;

4.3. predict next symbol V_k (at time $T+1$), $k = 0,...,M-1$, maximizing $\overline{b_j(k)}$.

If the process continues, then $T = T+1$ and go to 2.).

As we previously emphasized, the prediction accuracy of a symbols sequence provided by a HMM predictor could define the random degree of that sequence. Obviously, it requires modifying the number of hidden states for the HMM predictor in order to maximize the prediction accuracy. Particularly, it is interesting to see whether this idealized powerful predictor would successfully predict the sequences generated by unbiased branches. An affirmative answer would mean that the relevant prediction information exists but is hard to identify it, differing from one branch to another. Otherwise, if the answer is negative, the intrinsic random degree (determinist chaos) of these branches would be very significant.

5.2.2. Random Degree Metric Based on Discrete Entropy

Considering a sequence *S* of symbols belonging to the set $X = \{X_1 X_2\ ...\ X_k\}$, another practical approach for characterizing the randomness of *S* might be based on its entropy:

$$E(S) = -\sum_{i=1}^{k} P(Xi)\log_2 P(Xi) \geq 0 \qquad (5.1)$$

Obviously its maximum ($\log_2 k$) is obtained for symbols of equal probabilities in *S*. Therefore, we propose a random degree (RD) for a branch's binary output sequence given by the formula

$$RD(S) = D(S) \cdot E(S) \in [0, \log_2 k] \tag{5.2}$$

where *D*(*S*) represents the shuffle degree (distribution index) and it was defined in formula (3.2). A high *RD* value might involve a high random degree. Of course, our proposed *RD*(*S*) is not theoretically perfect. As an example, the sequence 01010101010101... maximizes both *D* and *E* but despite of this fact it is very deterministic and, therefore, very predictable.

5.2.3. Random Degree Metric Based on Compression Rate

The compression rate of a symbols sequence (or the space savings due to its compression), provided by the well-known lossless compression algorithms such as *Huffman* and *Gzip*, could represent another effective metric for characterizing the random degree of that sequence.

Huffman proposes an entropic encoding greedy algorithm, effective and very useful in lossless compression, commonly used as final compression stage. The basic idea is to map an alphabet to a representation for that alphabet, composed of variable length strings, so that symbols with a higher occurance probability have a smaller representation than those that occur less often.

The kernel of the Gzip utility is the *DEFLATE* algorithm [Deu96], that represents a combination between the *LZ77* algorithm [Ziv77] (dictionary encoding technique) and the Huffman algorithm (statistical encoding technique). The compression is performed in two successive stages: i) the identification and replacement of duplicate strings with pointers (LZ77) and ii) replacement of the previously obtained symbols with new, weighted symbols based on frequency of use (Huffman).

In order to evaluate the compression rate of the sequences generated by biased and unbiased branches behavior, we used the following two metrics:

$$Compression\,Rate = \frac{Uncompressed\ Size}{Compressed\ Size} \cdot 100\% \tag{5.3}$$

$$Space\,Savings = \left(1 - \frac{Compressed\ Size}{Uncompressed\ Size}\right) \cdot 100\% \tag{5.4}$$

In our opinion, the compression rate and obviously, the space savings of sequences generated by unbiased branches behavior should be lower than those obtained for sequences generated by biased branches.

5.2.4. Random Degree Metric Based on Kolmogorov Complexity

The Kolmogorov-Chaitin complexity (or program size algorithmic complexity) of code sequence that generates unbiased branches could be a useful metric for describing the random degree. According to this metric, the length of the shortest program for a universal Turing Machine that correctly reproduces the observed data is a measure of complexity [Kol65]. A sequence *X* has Kolmogorov complexity *K*(*X*) equal to the length of the shortest program *p* for a (prefix) universal Turing Machine *U* that produces *X* and then halts:

$$K(X) = \min_{p:U(p)=X} l(p) \tag{5.5}$$

where *l*(*p*) is the length of *p* in bits. Kolmogorov complexity identifies a sequence *X* as random if $l(X) - K(X)$ is small: random sequences are those that are irreducibly complex. Thus, the unbiased branches complexity should be higher than the other conditional branches complexity. Nevertheless, the Kolmogorov complexity has a static nature while it tries to characterize the dynamic behavior of a certain branch. On the other hand, this metric is the single one that emphasizes the semantic complexity of the generator code sequence.

5.3. Evaluation Results

Like in the previous chapters, we used six difficult predictable SPEC 2000 benchmarks and simulated one billion dynamic instructions for each one, skipping the first 300 million instructions. It was considered a 16-bit global history (GH) context for each branch. We selected from each benchmark strongly unbiased contexts having low polarization indexes ($P(S) \in [0.501, 0.565]$) and strongly biased contexts with high polarization indexes ($P(S) \in [0.979, 0.997]$) that were very frequently processed (hundreds of thousands instances per a certain context). The polarization index was defined in formula (3.1). As an example, for the *gzip* benchmark we selected the unbiased context {PC= 4198960, GH=5904, P =0.565, 135533 instances} and the biased context {PC= 4195032, GH= 8135, P =0.980, 140396 instances}. Each context has associated a binary string representing its behavior (taken / not taken). This binary string represents the input sequence for the HMM predictor used by us in paragraph 5.3.1. During the paragraph 5.3.2 we calculated the random degrees associated to the same binary strings. In paragraph 5.3.3 we calculated the compression rates corresponding to the same branches behaviors.

5.3.1. Random Degree Evaluation with HMMs

During this paragraph we considered a per branch local history of 64 bits. Using a longer history significantly complicated our developed HMM predictors and grew up the computing time. Anyway, our proposed metric is quantitatively very relevant. Figure 5.2 presents the prediction accuracies obtained on strongly unbiased branches using a first order HMM predictor (R=1) for different numbers of possible hidden states (N). For the majority of the benchmarks considering two hidden states generate the best accuracies. Figure 5.3 is similar but for a second order HMM predictor (R=2). Only on the *gcc* benchmark the prediction accuracy grows as far as N grows. All our developed second order HMM predictors are worser, at average, than a first order HMM with two hidden states (R=1, N=2), which is the best evaluated configuration. As it can be seen, the average prediction accuracy obtained using the optimal HMM (R=1, N=2) is far greater on biased contexts than on unbiased contexts.

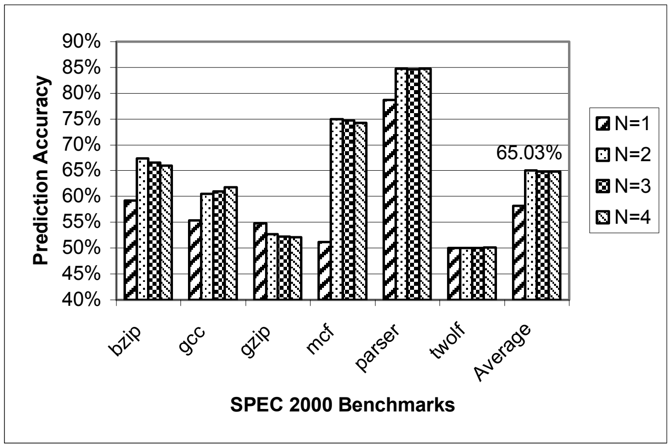


Figure 5.2. Prediction accuracy on unbiased branches using a first order HMM

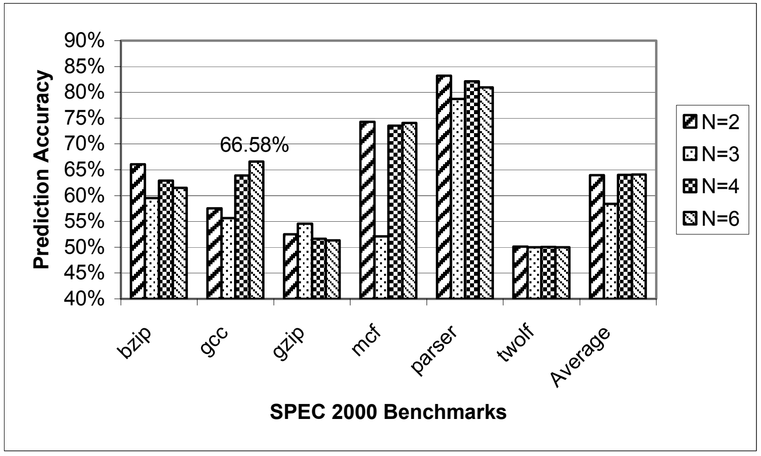


Figure 5.3. Prediction accuracy on unbiased branches using a second order HMM

Figure 5.4 comparatively presents, for unbiased and biased branches, the average prediction accuracies obtained by our determined quasi-optimal HMM (R=1, N=2). There is a significant difference between the average prediction accuracy on biased branches (98.43%) and on unbiased branches (65.03%).

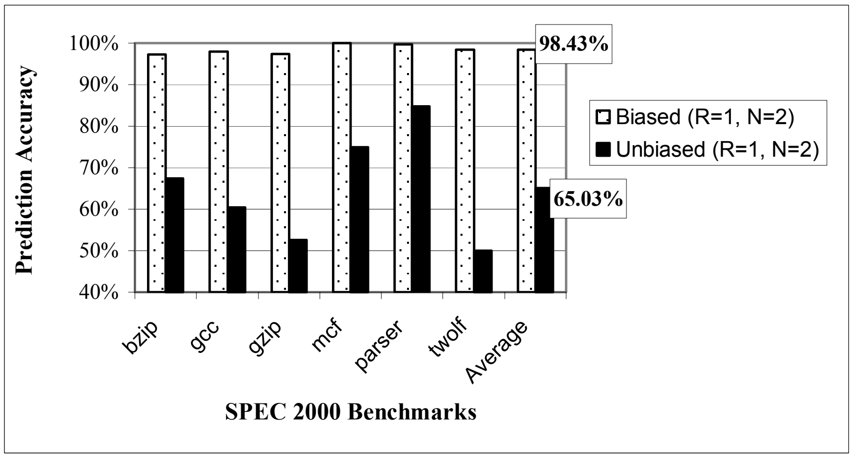


Figure 5.4. Prediction accuracies using the best evaluated HMM (R=1, N=2)

As we expected, the HMM predictor obtains an excellent average prediction accuracy on biased branches showing its high prediction power. As far as we know, we are the first researchers investigating HMMs as an ultimate branch prediction limit. Unfortunately even these powerful predictors cannot accurately predict unbiased branches. This fact suggests that unbiased branches are “intrinsic random” in some way, being generated by very complex program structures as we will further show.

5.3.2. Random Degree Evaluation Based on Discrete Entropy

In this paragraph we considered as the random degree of a binary sequence $RD(S)$, the product between discrete entropy $E(S)$ and shuffle degree $D(S)$ associated to S. Thus, $RD(S) = D(S) \cdot E(S)$. Figures 5.5, 5.6 and 5.7 show statistical results concerning the entropy, shuffle degree and random degree of the biased and unbiased binary sequences obtained through the previously exposed methodology.

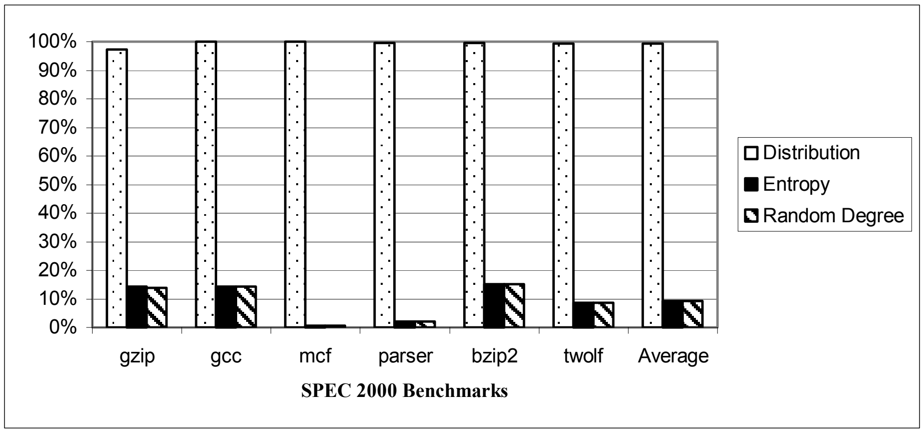


Figure 5.5. Characterizing biased sequences from entropy, shuffle degree and random degree viewpoint

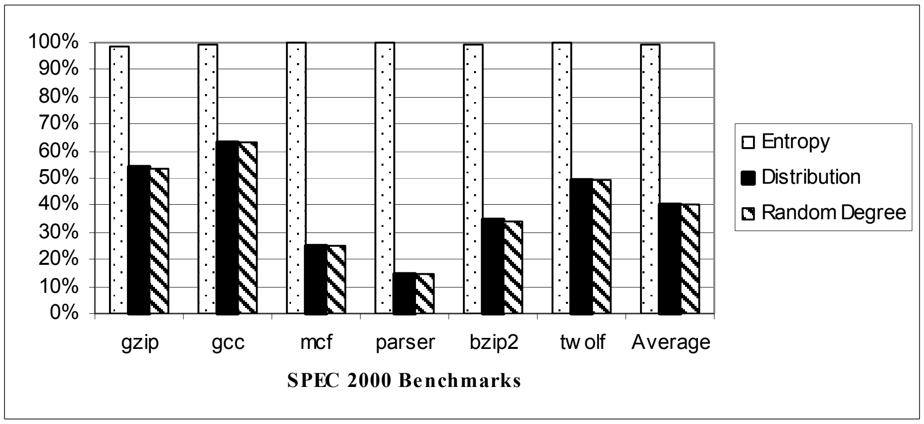


Figure 5.6. Characterizing unbiased sequences from entropy, shuffle degree and random degree perspectives

Regarding Figure 5.6 we notice that the entropy is mainly responsible for the higher random degree of unbiased branches, the role of shuffle degree being minor in this case.

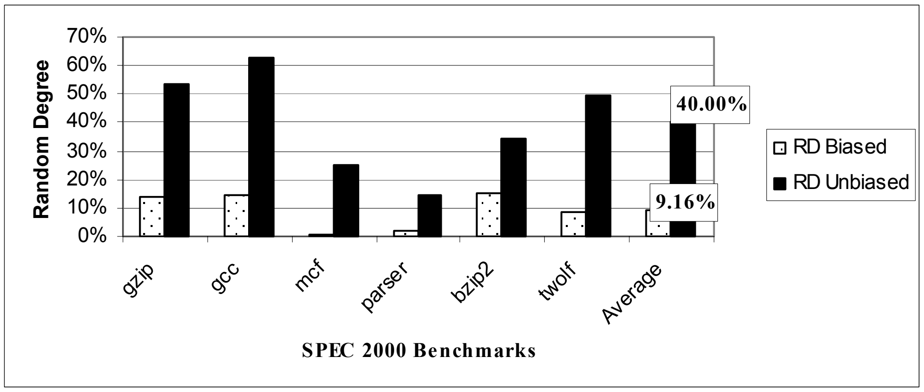


Figure 5.7. The random degree of biased and unbiased branches

Since our initial supposition was that biased branch sequences should have a lower random degree, the simulation results confirm that the considered *RD*(*S*) metric represents a good measure for random degree of binary sequences. A random degree around 40% shows that respective unbiased branch is difficult or, practically, even impossible to be accurately predicted.

5.3.3. Random Degree Evaluation Based on Compression Rate

Further we transformed into extended ASCII files the binary behavior sequences generated by unbiased and biased branches, obtained through the methodology exposed in paragraph 5.3. We grouped 8-bit sequences and generated the corresponding ASCII codes. We compressed these files using the *Gzip* utility [Gzip] and an own developed application that implements the *Huffman* encoding [Cor01].

We based our statistics on two commonly used metrics in data compression, presented in paragraph 5.2.3. In Figure 5.8, we illustrate the space savings obtained by compressing biased and unbiased branches using the previously described algorithms (*Gzip* and *Huffman*).

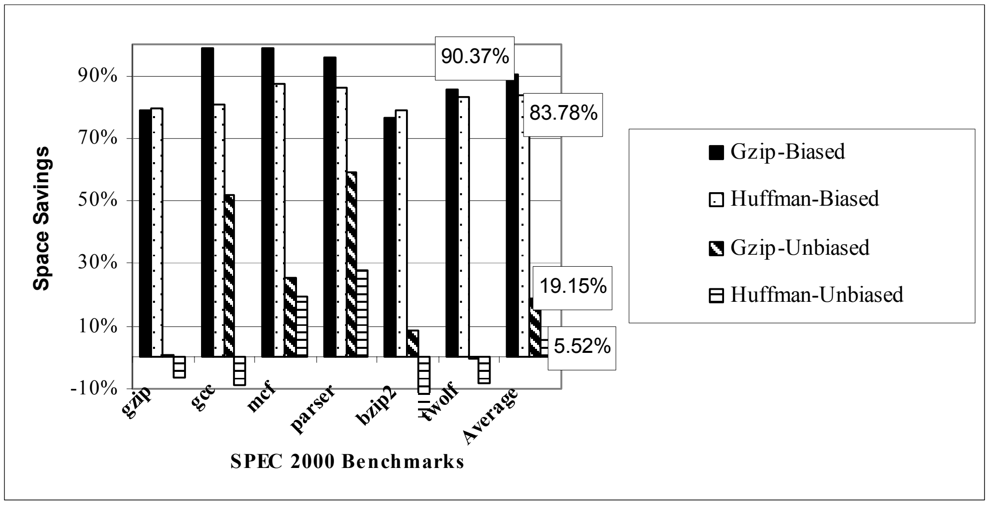


Figure 5.8. Space savings using the *Gzip* and *Huffman* algorithms

From the previous chart we can extract the following conclusions: first, the space savings obtained through unbiased branches compression (19.15% with *Gzip*) are significantly lower than those obtained through biased branches compression (90.37% with *Gzip*). The second conclusion refers to the ascendancy of the *Gzip* algorithm toward the *Huffman* algorithm that is understandable taking into account that the *Huffman* encoding represents the final stage of the *Gzip* compression. However, it can be observed that the space saving on the *twolf* benchmark becomes negative (-0.29%) even if the Gzip compression algorithm is used. The LZ77 algorithm's influence is almost inexistent leading to the conclusion that is impossible to find many repetitive patterns. Actually, we obtained similar results in [Gel07b], where we have shown that using some hybrid Markov predictors, the unbiased branches prediction accuracy is very low.

Since the Huffman encoding is very effective for strings characterized by low entropy symbols, the negative values of space savings on four SPEC benchmarks also illustrates the lack of repetitive pattern from unbiased sequences and the impossibility to predict them with higher accuracy using Markov predictors. The negative compression is caused by the necessity to store the encoding and decoding information in addition to the encoded sequence (header that contains the mapping of each distinct symbol from the input sequence into the new result symbol).

5.3.4. Random Degree Evaluation Based on Kolmogorov Complexity

Starting from several computationally intensive and heavily recursive Stanford benchmarks [Ste96], we give a code sequence example that will generate after execution some unpredictable sequences of unbiased branches [Rad07, Flo07a]. Further we partially present the C and Hatfield

Superscalar Architecture (HSA) assembly code of the *Perm* benchmark that generates a suite of permutations. First, we focused on the most important unbiased branch from the *Perm* benchmark (having PC=58) that exhibits an unpredictable behavior even if its context length is very long (53 bits of global history). Actually, the percentage of unbiased branches (1.53%) from the whole *Perm* program is exclusively due to the branch from PC=58.

```
Permute (int n){
  int k;
  pctr = pctr+1;
  if(n != 1)              // the first branch instruction analyzed (PC=35)
  {
    Permute(n-1);
    for( k = n-1; k >= 1; k--)          // the second branch instruction analyzed (PC=58)
    {
            Swap(&permarray[n], &permarray[k]);
            Permute(n-1);
            Swap(&permarray[n], &permarray[k]);
    };
  }
}

_Permute:
          SUB SP, SP, #128
          ST 0(SP), RA
          ST 8(SP), R17
          ST 12(SP), R18
          ST 16(SP), R19
          ST 20(SP), R20
          MOV R20, R5
          LD R13, _pctr
          ADD R13, R13, #1
          ST _pctr, R13
          EQ B1, R20, #1
          BT B1, L8 (#0)      # after compiling process this branch has the address 35 (PC=35)
          ADD R17, R20, #-1
          MOV R5, R17
          BSR RA, _Permute (#0)
          MOV R18, R17
          LES B1, R18, #0
          BT B1, L8 (#0)
          ASL R13, R20, #2
          MOV R7, #_permarray
          ADD R19, R13, R7
          ASL R13, R18, #2
          ADD R17, R13, R7
L12:      MOV R5, R19
          MOV R6, R17
          BSR RA, _Swap (#0)
          ADD R5, R20, #-1
          BSR RA, _Permute (#0)
          MOV R5, R19
          MOV R6, R17
          BSR RA, _Swap (#0)
          ADD R17, R17, #-4
          ADD R18, R18, #-1
          GTS B1, R18, #0
          BT B1, L12 (#0)     # after compiling process this branch has the address 58 (PC=58)
L8:       LD R17, 8(SP)
          LD R18, 12(SP)
          LD R19, 16(SP)
          LD R20, 20(SP)
          LD RA, 0(SP)
          ADD SP, SP, #128
          MOV PC, RA (#0)
```

We developed a particular fast path-based perceptron (FPBP) predictor [Rad07] with a global history length of 53 bits and 100 entries. FPBP predicted the branch 58, in its unbiased contexts, with 65.91% accuracy. The number of FPBP mispredictions was 286. The complete PPM predictor exploits the recursive character of *Perm* benchmark. The prediction accuracy (PA) obtained by our developed PPM using a global context length of 500 bits and a search pattern of 30 bits, on the branch 58, is 94.30%. As far as this solution is unfeasible for hardware implementation, we tried a simplified PPM, but the result was dissatisfactory (PA=79.85%). The global prediction accuracy provided by the complete PPM was 98.41%, lower than that generated by the FPBP predictor (99.04%). Actually, from 869 PPM mispredictions, the branch 58 generates 287. Thus, we can conclude that both PPM and FPBP predictors do not succeed to accurately predict an unbiased branch. The high prediction accuracy (94.30%) on the branch 58 provided by the PPM is actually centered on the whole behavior of the branch and not only on its unbiased context.

As we have already pointed out, the length of the shortest program for a universal Turing machine that correctly reproduces the observed data is a measure of complexity [Gam99]. Thus, analyzing the behavior of the branch 58 from the Kolmogorov complexity perspective (we noted it $K(58)$), it can be observed that the minimal length of machine-code that generates this unbiased branch is equal with the *Permute* routine length (measured in instructions). This happens because, in order to reach the branch 58, the Permute routine should completely execute at least once (due to recursive call).

Thus, $K(58)$=42 HSA instructions or 8 C instructions. We must mention that the whole assembly program has 108 instructions and the rest of subroutines are not recursive and consist of array initialization, variables interchange, and simple repetitive program structures. Among the other conditional branches only one (PC=35) proved to be unbiased for shorter global history length (≤32 bits). However, increasing the global history length to 53 bits the branch 35 became fully biased, and, therefore predictable. Analyzing the Kolmogorov complexity of branch 35 we calculated $K(35)$=12 HSA instructions or 3 C instructions. It involves that $K(35)<K(58)$. This happens because the test of the branch 35 does not require the complete execution of the *Permute* routine. Therefore, the complexity of the code sequence that generates the unbiased branch (58) induces a determinist chaos, frequently occurred in many science domains. In addition, based on the analysis of many integer recursive benchmarks we have reasons to believe that recurrence combined with some certain conditional branches will generate branches with unbiased behavior and thus with high Kolmogorov complexity. Such examples occur in the link lists or trees cases where the address of an element is tested and followed by a recurrent call of the same function to test the next element in the tree.

5.4. Summary

Our experiments proved that all these four developed random degree metrics are converging at the same point. The unbiased branches are not quite "completely random". They are "almost random" due to programs complexity. They generate a deterministic chaos. For example, *RD* is 1 for a "completely random" branch, but as we pointed out in paragraph 5.2, it is about 0.40 for unbiased branches and 0.09 for biased branches (Figure 5.7). The *space saving* is 0 for a "completely random" branch, and in our experiments it was about 0.83 for biased branches and 0.05 for unbiased branches using the Huffman compression algorithm (Figure 5.8). The HMM predictor also obtains an excellent average prediction accuracy on biased branches (98.43%) showing its significant prediction power while the average prediction accuracy on unbiased branches is limited to 65.03% (Figure 5.4). Moreover, the Kolmogorov complexity of an unbiased branch is higher than the Kolmogorov complexity of any conditional branch belonging to the same programs. As a conclusion, using these random degree metrics, the computer architect would understand whether these difficult branches are or are not predictable.

"We learn as we do and we do as well as we have learned"

Vernon B. Brooks

6. Exploiting Selective Instruction Reuse and Value Prediction in a Superscalar Architecture

In the previous chapters we have shown that unbiased branches cannot be accurately predicted irrespective of the prediction information type used in the state-of-the-art branch predictors [Vin06, Gel07b]. Furthermore, the behavior sequences generated by these difficult branches are characterized by high random degrees. Since the overall performance of modern superscalar processors is seriously affected by misprediction recovery, these difficult branches represent a source of important performance penalties. As we pointed out in [Gel06b], 28.68% of branches are dependent on long-latency instructions (critical Loads, Multiply, Division), and 5.61% are unbiased and dependent on a previously committed long-latency instruction. Such hard-to-predict branches that depend on critical Loads (with miss in the L2 data cache) occur in pointer chasing applications based on linked list traversal:

```
while (node)            // Branch
  node = node→next      // Load
```

Since the branch from the above example depends on the Load, a branch misprediction cannot be solved until the Load returns the value. If the Load has a high L2 cache miss rate, the misprediction penalties of the branch will have significant impact on the overall performance. For example, the average misprediction penalty of such a branch, measured as the latency between fetching the branch instruction and resolving the misprediction, is about 540 cycles, considering a L2 cache miss penalty of 300 cycles [Gao08]. Thus, the aforementioned dependences involve high-penalty mispredictions becoming serious performance obstacles and causing significant performance degradation in executing instructions from wrong paths. Therefore, the negative impact of branches, and especially of unbiased branches, over global performance should be seriously attenuated by anticipating the results of long-latency instructions, including critical Loads. On the other hand, hiding instructions long latencies in a pipelined superscalar processor represents an important challenge itself. Therefore, in this chapter we present based on [Gel08b, Vin05a] some original anticipatory methods developed for superscalar architectures.

6.1. Related Work

The idea of dynamic instruction reuse was first introduced by Sodani and Sohi in [Sod97]. Dynamic instruction reuse is a non-speculative microarchitectural technique that exploits the repetition of dynamic instructions. The main idea is that if an instruction or an instruction chain is reexecuted with the same input values, its output value will be the same. The authors introduced different schemes that maintain the inputs and the results of previously executed instructions in a hardware structure called Reuse Buffer. With instruction reuse the number of executed dynamic instructions is reduced and the critical path might be compressed. According to the authors' simulations on the SPEC'95 benchmarks, at average 26% of dynamic instructions are reusable. This quite high reuse degree is understandable taking into account that less than 20% of the static instructions are generating more than 90% of dynamic instructions. These useful statistics are qualitatively justified due to the fact that programs are written in a compact (loops, recurrence, inheritance, etc.) and generic manner (the programs have to operate on a variety of data structures). There are some important differences between our approach and Sodani's. We reuse only Mul and Div instructions and, although we use the same S_v scheme that tracks operand values for each instruction, our scheme does not require all fields of Sodani's S_v scheme. Since we do not reuse Load instructions, we renounce to the *Address* and *Mem Valid* fields. This reduces the hardware cost with benefits on power consumption, too. Another difference refers to the moment when the instructions are reused: in contrast with Sodani's approach, the Reuse Buffer (RB) is accessed in our architecture during the *issue* stage, because most of the Mul/Div instructions found in the RB in the *dispatch* stage do not have their operands ready.

Richardson introduced *Instruction Memoization* [Ric93], a technique that consists in storing the inputs and outputs of long-latency operations and reusing the output if the same inputs are encountered again. The memo table is accessed in parallel with the first computation cycle, and the computation halts in the case of hit. Thus, memoing reduces a multi-cycle operation to one-cycle when there is a hit in the memo table. In [Bro00] the authors proposed a memoing technique in order to save power. Brooks et al. used memo tables in parallel with the floating-point and integer multipliers, the floating-point adder, and the floating-point divider. Their experimental results show an average speedup of 1.7% and an average power consumption improvement of 5.4%.

Citron and Feitelson in [Cit02] compare different instruction reuse techniques, including *Instruction Reuse* (IR) and *Instruction Memoization* (IM). The authors splat the Lookup Table into several smaller tables for floating-point instructions, Loads, multi-cycle integer instructions (Multiply and Division) and all other single-cycle instructions. Each table contained 256 entries. They used IM only for multi-cycle operations. The evaluation results (reuse degree and speedup) obtained on the SPEC'95 benchmarks show that only floating-point applications can benefit from instruction reuse.

Golander and Weiss present in [Gol07] different instruction reuse methods for Checkpoint Processors. In checkpoint microarchitectures a misspeculation initiates the rollback, in which the latest safe checkpoint preceding the point of misprediction is recovered, and after that performs the reexecution of the entire code segment between the recovered checkpoint and the mispredicting instruction (selective reissue). The authors proposed two instruction reuse methods for normal execution and other two methods for reexecution after a misprediction. The *Trivial* method identifies trivial arithmetic operations having one of the inputs a neutral element, or both operands with the same magnitude. The hardware for detecting trivial computations and selecting the result consists in comparators for the input operands and selectors for the writeback. In our simulator, we implemented the *Trivial* method proposed by Golander. The *SelReuse* method uses a small fully associative reuse cache for long latency arithmetic operations. As the authors are showing, an 8-entry cache is sufficient for reusing most of the available results. The *RbckReuse* method is used for all instruction types from reexecuted paths, excepting control-flow instructions. Finally, the *RbckBr* method is used for the branch instructions from reexecuted paths. The reuse structure maintains only the branch outcome and relies on the BTB for the branch target address. A reuse approach that combines all the four methods briefly presented above requires an area of 0.87 mm^2 and consumes 51.6 mW. It achieves an average IPC speedup of 2.5% for the SPEC 2000 integer benchmarks, of 5.9% for the floating point benchmarks, and an improvement in energy-delay product of 4.80% and 11.85%, respectively.

Based on the dynamic correlation between Load instruction addresses and the values the Loads produce, Lipasti et al. [Lip96a] proposed a new data-speculative micro-architectural technique entitled *Load Value Prediction* that can effectively exploit value locality to collapse true data dependencies (exceeding thus the dataflow limit and enhancing the instruction-level parallelism), reduce average memory latency and bandwidth requirement and provide measurable performance gains. Their Load Value Prediction Unit was presented in Chapter 2. In [Lip96b], Lipasti and Shen extended the prediction of Load values predicting all integer and floating point register values. An important difference between our value prediction approach and Lipasti's is that we selectively predict Load instructions generating a miss in L1 cache. Thus, we attenuate the misprediction cost and reduce the hardware cost of the speculative micro-architecture. Moreover, since less hardware is required, there is also less power consumption.

Tullsen and Seng in [Tul99] proposed a technique entitled register-value prediction that identifies instructions which produce values that are already in the register file. Therefore, the corresponding results are predicted using the values belonging to the register file. Mainly, this technique uses the previous value in the instruction's destination register as a prediction for the new result, in a static or dynamic manner. An important advantage of this prediction scheme is that it does not require storage for values. For dynamic prediction, only a table of confidence counters is used, which is indexed by the instruction PC. Thus, the confidence counters are associated with instructions indicating which of them have high register value reuse. This technique produced speedups of up to 11% for the SPECint95 benchmarks and up to 13% for SPECfp95 benchmarks. In contrast to our register-centric approach [Vin05a, Vin05b, Gel03], this approach is an instruction-centric one.

In [Sen04] the authors defined register value locality as the probability that the next value produced by an instruction to be the value already stored in the destination register. In contrast, in our work [Vin05a] we define it as the probability that the next value of the destination register belongs to the previous k values stored in that register. Therefore, our original register value prediction technique consists in predicting the next value of a register based on the previously seen values. In [Sen04] the authors used perceptron-based predictors to perform a limited form of register value prediction: their scheme predicts if the value written to a register will be the same as the current value. The proposed predictor uses a table of perceptrons. For a certain instruction the perceptron is selected with the lower bits of the instruction address. The input of the perceptron is the global history of the most recent committed instructions, where a value of 1 indicates that the corresponding instruction was redundant and a -1 indicates otherwise. They demonstrate that for a given size predictor, a perceptron based predictor performs better than a saturating counter based register value predictor [Tul99] – for an 8KB hardware budget the speedup is 8.1%.

A. Thomas and D. Kaeli in their work [Tho04] improve the two-level value prediction schemes, presented in [Wan97], by using perceptrons instead of confidence counters in the second level. For counter-based predictors, the number of counters grows exponentially with the value history length and, therefore, the history is limited. The main advantage of the perceptron-based predictor is that its size grows linearly with the value history length and, thus, longer value histories can be used for prediction. The perceptron-based predictors achieved considerably better prediction accuracy – 93.55% at average – but without IPC improvement due to their higher prediction and update latency.

R. Thomas et al. [Tho01] improved instruction-centric value prediction by using a *dynamic dataflow inherited speculative context* (DDISC) for hard-to-predict instructions. The DDISC consists in a compression of the PCs and the predicted values of the predictable source producer instructions. The context is determined by assigning a signature to each node in the dataflow graph. The signature of a predictable instruction is its value predicted by a conventional predictor. The signature of unpredictable non-Load instructions is inherited from the signatures of its operand producers. In the case of multiple operands, the signature of unpredictable non-Load instructions is the XOR of the signatures of their operand producers. The signature of unpredictable Load instructions is inherited from the signature of the preceding Store instruction that wrote the value into the same memory location. The DDISC for a certain instruction is obtained by rotating its calculated signature by a value determined by the PC (e.g. the last five bits of the PC). Their simulation results show that introducing dataflow-based contexts the prediction accuracy improvement ranges from 35% to 99%.

Mutlu et al. presented in [Mut06] a new hardware technique named *address-value delta (AVD) prediction*, able to parallelize dependent cache misses. They observed that some Load instructions exhibit stable relationships between their effective addresses and data values, due to the regularity of allocating structures in the memory by the program, which is sometimes accompanied by the regularity in the program's input data. In order to exploit these regular memory allocation patterns, the authors proposed an AVD structure that maintains the Load instructions having a stable address-value difference (delta). Each entry of the AVD table consists in the following fields: *Tag* (the upper bits of the Load's PC), *AVD* (the address-value delta corresponding to the last occurrence of that Load) and *Conf* (a saturating counter that records the confidence of AVD). The *Conf* field is used to avoid predictions for Loads with an unstable AVD. If a Load instruction having a stable AVD occurs with a cache miss, its data value is predicted by subtracting the stable delta from its effective address. This prediction enables the preexecution of dependent instructions, including Loads with cache miss. The experimental results show that integrating a 16-entry AVD predictor into a *runahead* processor improves the average execution time of pointer-intensive applications by 14.3%.

Liao and Shieh proposed in [Lia02] a new scheme that combines value prediction and instruction reuse. The main idea consists in predicting operand values if they are not available and speculatively reusing instructions if the predicted operands match the values from the Reuse Buffer (RB). Obviously, instructions must be correctly reexecuted in the case of misprediction. If the operands of an instruction are ready and their values match the value fields of the corresponding RB entry, the result is guaranteed to be correct, and therefore the execution is non-speculative. The simulations on the SPEC'95 benchmarks showed that this scheme provides an average speedup of 8.9%.

In [Cha08] the authors proposed a hardware-based method, called *Early Load*, in order to hide the Load-to-Use latency (the latency that instructions wait for their operands produced by Load instructions) with little additional hardware costs. The key idea is to make use of the time that instructions are waiting in the instruction queue to load the data early, before the Loads are effectively executed, by pre-decoding instructions during the *fetch* stage. Thus, instead of using previous instances (values) of the current Load instruction Chang et al. are using an earlier executed-instance (value) of the current Load instance. In this way, the chance to be a correct value seems to increase. They use a small table, called Early Load Queue (ELQ) that records Load instructions and the early loaded data. The proposed scheme allows Load instructions to load data from memory before the *execution* stage. Obviously, a detection method assures the correctness of the early operation before the Load enters into the *execution* stage. If the corresponding ELQ entry is valid in the Load's *dispatch* stage, the execution of the Load instruction is completely avoided and all dependent instructions get the data from the ELQ. Unfortunately this method does not work for out-of-order speculative architectures whereas our technique does. Also, it works only for very small instruction queues. The experimental results showed that this scheme can achieve a performance improvement of 11.64% on the *Dhrystone* benchmark and of 4.97% on the *MiBench* benchmark suite.

6.2. Anticipating Long-Latency Instructions Results

Our main objective is to develop a superscalar architecture that selectively anticipates the values produced by high-latency instructions. We will focus on Multiply, Division and Loads with miss in the L1 data cache. The reusability degree of Mul and Div instructions, measured with an unlimited Reuse Table, was 28.9% on the integer benchmarks and 61.9% on the floating-point benchmarks [Gel08a]. These instructions would be solved by a Dynamic Instruction Reuse scheme. The reusability degree of Load values was 77.4% on the integer benchmarks and 76.4% on the floating-point benchmarks [Gel08a]. However, an additional Reuse Buffer for Load Value (Data) Reuse is not necessary, because a similar reuse mechanism is already provided by the existing L1 and L2 data caches. Therefore, the Load instructions with miss in the L1 data cache (selective approach) would be solved through value prediction.

6.2.1. Selective Dynamic Instruction Reuse

For the Mul and Div instructions we will use the S_v reuse scheme. The information about instructions is maintained in a direct mapped Reuse Buffer (RB). The RB is accessed during the *issue* stage, because most of the Mul/Div instructions found in the RB during the *dispatch* stage do not have their operands ready (91.5% on the integer benchmarks and 64.6% on the floating-point benchmarks). An additional RB access in the *dispatch* stage does not have sense due to the insignificant expected performance gain obtained with supplementary costs. Each RB entry has the following fields: *Tag* (the higher part of the PC), *SV1* and *SV2* (the source values of the Mul/Div instruction), *Result* (the output value of the Mul/Div instruction). Since we do not reuse Loads with this scheme, the *Address* and *Mem Valid* fields used in [Sod97] are unnecessary. In this way, our implemented structure is simpler and more cost effective (from hardware budget and power consumption point of view) than the initial scheme proposed by Sodani and Sohi.

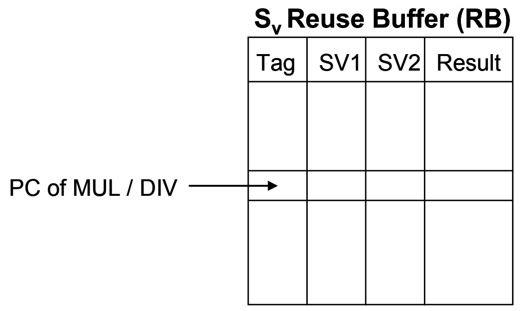


Figure 6.1. Reuse scheme for Mul & Div instructions

If a certain Mul/Div instruction is found in the RB, a reuse test is generated. If the actual operand values, taken from the ROB, match the SV1 and SV2 fields from the selected RB entry, the instruction is not sent to a functional unit, its result value being already available for dependent instructions. Every non-reused Mul/Div instruction updates the RB in the *commit* stage: writes the tag, the source values and the result into the corresponding RB entry. From the power consumption point of view, the Reuse Buffer was modeled as a cache array structure using the same power models as the other array structures are using. Obviously, the main benefit of reusing long-latency instructions consists in unlocking dependent instructions (see Figure 6.2). In Figures 6.2, 6.4 and 6.9, all stages except the *execute* stage are a single cycle length; the *execute* stage has variable length, depending upon the latency of the executing instruction (see Table 6.1).

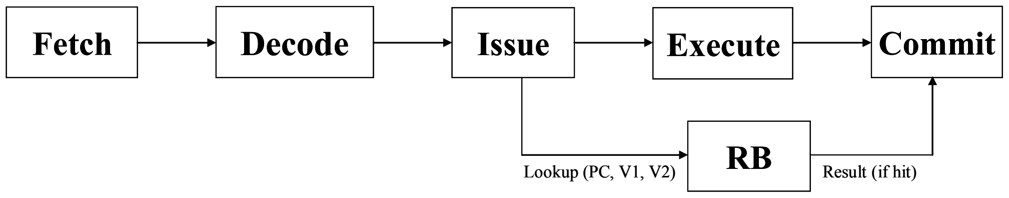


Figure 6.2. Pipeline with Reuse Buffer (RB)

We also detected trivial operations implementing a technique first introduced in [Ric93] by Richardson. We considered the following operations: V*0, V*1, 0/V, V/1 and V/V. A simple hardware scheme for detecting trivial computations and selecting the result is presented in [Gol07] and consists in comparators for the input operands and selectors for the write-back. If during the *dispatch* stage, a Mul instruction is detected with an operand value of 0 or 1, the result is provided by the detector, avoiding the functional unit allocation and execution. In the same manner, if a Div instruction is detected with the first operand being 0, the second operand 1, or with identical operands, the result is provided by the detector being thus available at the end of the *dispatch* stage. The Reuse Buffer is accessed during the *issue* stage for the reuse test only if the Mul/Div operation is not detected in the *dispatch* stage as being trivial.

6.2.2. Selective Load Value Prediction

We will integrate into our architecture a simple Last Value Predictor used only for Loads with miss in the L1 data cache (selective approach). In this way, the implemented structure is more efficiently used; the collisions number will be lower against the approach that predicts all Load instructions, having tables of the same size. The information about Load instructions is maintained in a direct mapped Load Value Prediction Table (LVPT). The LVPT is accessed during the *issue* stage, only if the current Load instruction involves a miss in the L1 data cache (critical Load). Each LVPT entry has the following fields: *Tag* (the higher part of the PC), *Counter* (a 2-bit saturating confidence counter with two *unpredictable* and two *predictable* states), and *Value* (the Load instruction's result).

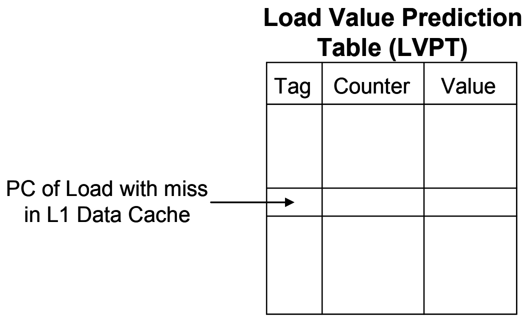

Load Value Prediction Table (LVPT)
Tag
Counter
Value
PC of Load with miss in L1 Data Cache

Figure 6.3. The Last Value Predictor architecture

In the case of a hit in the LVPT, the corresponding *Counter* is evaluated. If the confidence counter is in an unpredictable state, the Load is executed without prediction. Otherwise the *Value* from the selected LVPT entry is speculatively forwarded to the dependent instructions. In the *commit* stage, when the real value is available, in the case of misprediction, a recovery is necessary in order to squash speculative results and selectively re-execute the dependent instructions with the correct values (see Figure 6.4). We considered in our simulations a value prediction latency of one cycle and, in the misprediction case, a recovery taking 7 cycles.

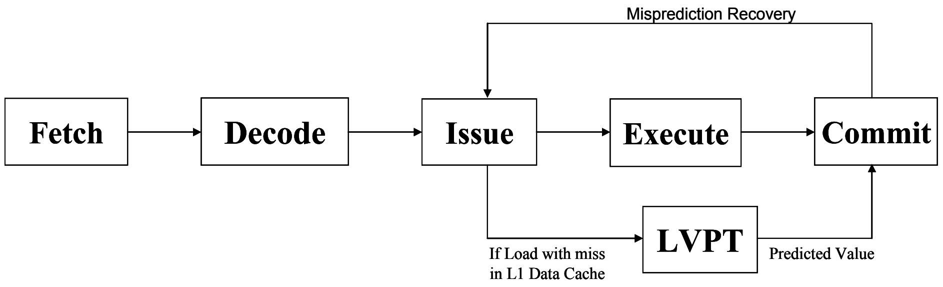


Figure 6.4. Pipeline with Load Value Predictor

During the *commit* stage, every critical Load updates the LVPT: only the *Counter* field in the case of correct prediction or the *Value* and the *Counter* fields in the case of misprediction. In the case of miss in the LVPT, the *Tag* and the *Value* are inserted into the selected entry, and the *Counter* is reset (strongly unpredictable state).

6.2.3. Simulation Methodology

We developed a cycle-accurate execution driven simulator derived from the M-SIM simulator [Sha05] supporting the unmodified, statically linked Alpha AXP binaries as well as the power estimation as supplied by the Wattch framework [Bro00]. M-SIM extends the SimpleScalar toolset [Bur97] with accurate models of the pipeline structures, including explicit register renaming, and support for the concurrent execution of multiple threads. We modified M-SIM to incorporate our superscalar architecture with selective instruction reuse and value prediction in order to measure the relative IPC speedup and relative energy-delay product gain when the results of long-latency instructions are anticipated.

All simulation results are generated on the SPEC 2000 benchmarks [SPEC] and are reported on 1 billion dynamic instructions, skipping the first 300 million instructions. We evaluated seven integer benchmarks (*bzip*, *gcc*, *gzip*, *mcf*, *parser*, *twolf*, *vpr*) and six floating-point benchmarks (*applu*, *equake*, *galgel*, *lucas*, *mesa*, *mgrid*). Table 6.1 presents some important parameters of the simulated architecture:

Execution Latencies	**Execution unit**	**Number of units**	**Operation latency**
	intALU	4	1
	intMULT / intDIV	1	3 / 20
	fpALU	4	2
	fpMULT / fpDIV	1	4 / 12
Superscalarity	Fetch / Decode / Issue / Commit width = 4		
Branch predictor	bimodal predictor with 2048 entries		
Caches and Memory	**Memory unit**		**Access Latency**
	4-way associative L1 data cache, 32 KB		1 cycle
	8-way associative unified L2 data cache, 512 KB		6 cycles
	Memory		100 cycles
Resources	**Register File:** 32 INT / 32 FP		
	Reorder Buffer (ROB): 128 entries		
	Load/Store Queue (LSQ): 48 entries		

Table 6.1. Parameters of the simulated architecture

For the relative IPC speedup calculation we used the following formula:

$$IPC\ Speedup = \frac{IPC_{improved} - IPC_{base}}{IPC_{base}} \cdot 100\% \qquad (6.1)$$

where IPC_{base} and $IPC_{improved}$ are the instructions executed per cycle with the baseline and improved architectures, respectively.

The power consumption measurements are generated using an 80 nm CMOS technology. Figure 6.5 presents the structure of the simulator.

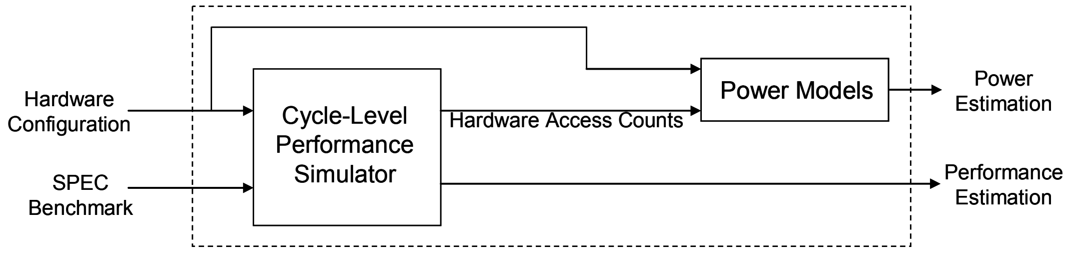


Figure 6.5. The structure of the simulator

As Figure 6.5 shows, the simulator generates both performance and power consumption estimation. The detailed power modeling methodology, used in the simulator, is presented in [Bro00]. The dynamic power consumption in CMOS microprocessors is defined as:

$$P_d = C \cdot V_{dd}^2 \cdot a \cdot f \tag{6.2}$$

where C is the capacitance, generated using *Cacti* [Shi01], V_{dd} is the supply voltage, and f is the clock frequency. V_{dd} and f depend on the assumed process technology. The activity factor a indicates how often clock ticks lead to switching activity on average. The power consumption of the modeled units highly depends on the internal capacitances of the circuits. From the capacitance point of view, there are three categories of architectural structures: array structures, content-associate memories, and complex logic blocks. The first two categories are used to model the caches, branch predictors, the reorder buffer, the register renaming table, and the register file, while the last category is used to model functional units.

For the energy measurements, we used the Energy-Delay Product, a widely used metric [Gon96, Bro00, Gol07]:

$$EDP = \frac{Total\ Power}{IPC^2} \tag{6.3}$$

The Energy-Delay Product (EDP) represents the processor's total power, divided by the squared IPC. In other words, the EDP is the energy consumption relative to the processor's global performance (IPC). The *EDP Gain* represents the relative energy-delay product improvement. After each architectural improvement we determined the *EDP Gain* based on:

$$EDP\ Gain = \frac{EDP_{base} - EDP_{improved}}{EDP_{base}} \cdot 100\% \tag{6.4}$$

where, EDP_{base} is the energy-delay product of the baseline architecture, whereas $EDP_{improved}$ is the energy-delay product of the improved architecture. Thus, a positive value of the *EDP Gain* means an improvement of the relative energy consumption.

6.2.4. Experimental Results

Figure 6.6 presents the reuse degrees obtained with and without detecting trivial operations. An RB of 1024 entries provides on the integer benchmarks a reuse degree of 17.2%, compared with the reusability degree of 28.9% (the upper limit obtained with an unlimited RB). It was more efficient for the floating-point benchmarks, where we obtained a reuse degree of 54.8% with an RB of 2048 entries, compared with the reusability degree of 61.9% (through an unlimited RB). As Figure 6.6 shows, trivial operations detection improves significantly the reuse degree.

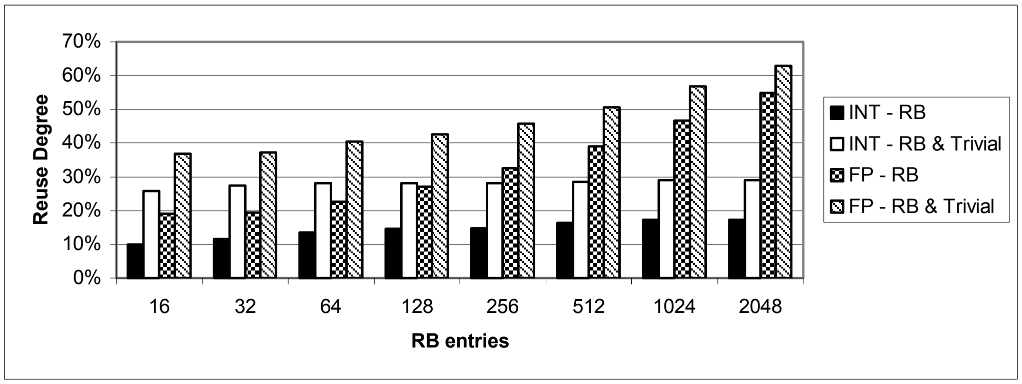


Figure 6.6. Reuse degrees obtained for different RB sizes with and without trivial operation detection

Table 6.2 presents the reuse degrees, the IPC, and the power consumption obtained, on the integer and floating-point SPEC 2000 benchmarks, by using the S_v reuse scheme together with the Trivial Operation Detector for the Mul and Div instructions. The *Reuse Degree* columns represent the percentage of reused Mul and Div instructions across all the evaluated integer and floating-point benchmarks. The *IPC* represents the average executed instructions per cycle. The *RB Power* column shows the additional dynamic power dissipated by the RB for each evaluated size in *mW* and in percentages reported to the total processor power.

RB entries	**SPEC 2000 integer**		**SPEC 2000 floating-point**		**RB Power**	
	Reuse Degree [%]	**IPC**	**Reuse Degree [%]**	**IPC**	**[mW]**	**[%]**
0 (no RB)	–	1.6857	–	2.0410	0	0.000
16	25.8	1.6881	36.8	2.0612	7.2	0.008
32	27.4	1.6862	37.3	2.0613	12.7	0.014
64	28.1	1.6862	40.5	2.0747	16.3	0.018
128	28.2	1.6862	42.5	2.0752	28.8	0.031
256	28.2	1.6862	45.8	2.0787	38.4	0.042
512	28.5	1.6862	50.6	2.0828	70.2	0.077
1024	29.0	1.6862	56.9	2.0863	99.6	0.109
2048	29.0	1.6862	62.8	2.0888	178.8	0.195

Table 6.2. Reuse degree, IPC and power consumption obtained with the RB and Trivial Operation Detector on the SPEC 2000 benchmarks

The very low IPC gain measured on the integer benchmarks is justified because only about 11 million instructions were reused from a total of 7 billion across all the integer benchmarks. Moreover, reusing Mul/Div instructions belonging to wrong speculated paths frequently involves issuing some long latency Loads. These critical instructions would not be executed without successful reuse.

Although the RB structure dissipates additional dynamic power, reusing long-latency instructions increases the IPC and therefore lowers the relative energy consumption (see Figure 6.7). We determined the energy-delay product for the architecture without RB and for the architecture with RB of different sizes, based on relation (6.3). The *EDP Gain* represents the relative energy-delay product improvement determined based on relation (6.4) for each RB size.

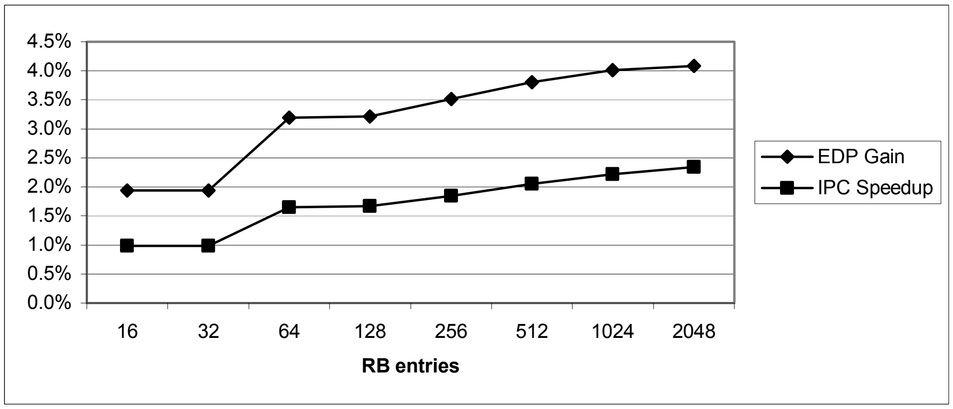


Figure 6.7. Relative IPC speedup and relative energy-delay product gain on the SPEC 2000 floating-point benchmarks with RB and Trivial Operation Detection

The speedup is insignificant in the case of the integer benchmarks, due to the significantly lower number of Mul and Div instructions. Consequently, the energy-delay product is better only for RB sizes between 16 and 128 entries, but the improvement is insignificant. These results are in concordance with Citron [Cit02] who also remarked the poor evaluation results (reuse degrees and speedups) obtained on the SPEC'95 integer benchmarks. Therefore a significant benefit of Mul/Div instructions reuse is achieved only for floating-point applications.

Table 6.3 presents the prediction accuracy, the IPC, and the power consumption obtained by evaluating our developed architecture with Mul/Div Reuse Buffer of 1024 entries and Trivial Operation Detector for the Mul and Div instructions and with Last Value Predictor for critical Load instructions. The *PA* columns represent the prediction accuracy of critical Loads. The *IPC* represents the average instructions per cycle. The *LVPT Power* column shows the additional dynamic power dissipated by the LVPT for each evaluated size in *mW* and in percentages reported to the total processor power.

LVPT entries	**SPEC 2000 integer**		**SPEC 2000 floating-point**		**LVPT Power**	
	PA	**IPC**	**PA**	**IPC**	**[mW]**	**[%]**
0 (no RB, LVP)	–	1.6857	–	2.0410	0	0.000
16	94.0	1.7066	99.7	2.1873	6.4	0.007
32	93.5	1.7094	99.8	2.2333	8.7	0.009
64	92.6	1.7245	99.8	2.3533	14.6	0.016
128	91.0	1.7318	99.7	2.3915	19.9	0.022
256	88.7	1.7351	99.5	2.4378	33.6	0.037
512	88.1	1.7387	99.3	2.4484	48.0	0.052
1024	87.1	1.7456	99.2	2.5241	84.9	0.092
2048	87.2	1.7460	99.1	2.5320	128.1	0.139

Table 6.3. Prediction accuracy, IPC and power consumption obtained with an RB of 1024 entries, the Trivial Operation Detector and the LVPT

Figure 6.8 presents the relative IPC speedup and the relative energy-delay product improvement for the integer and floating-point benchmarks. We determined the energy-delay product for the architecture without RB and LVPT and for the architecture with an RB of 1024 entries and LVPTs of different sizes, based on relation (6.3). The *EDP Gain* represents the relative energy-delay product improvement determined based on relation (6.4) for each LVPT size. As it can be observed, the optimal LVPT size is 1024.

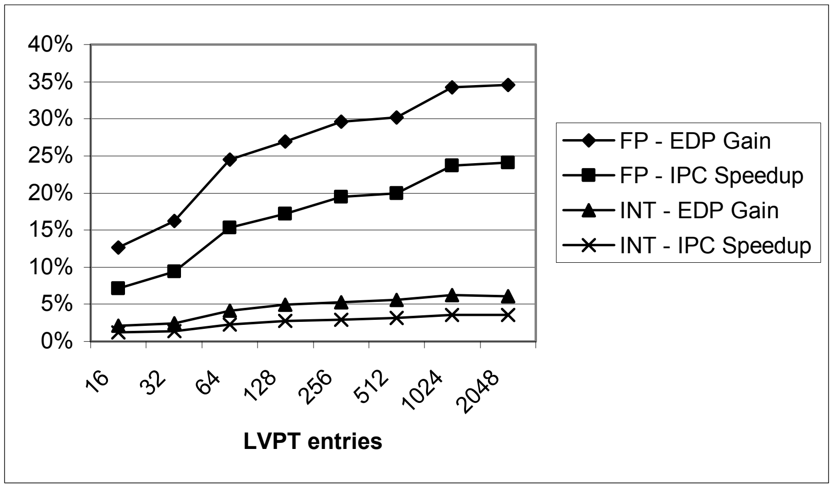


Figure 6.8. Relative IPC speedup and relative energy-delay product gain with a Reuse Buffer of 1024 entries, the Trivial Operation Detector, and the Load Value Predictor

Both IPC speedup and EDP gain are significantly higher on the floating-point benchmarks compared to the integer benchmarks (see Figure 6.8). This difference occurs because the number of critical Loads is more than twice higher in the floating-point benchmarks. The difference is further accentuated by the percentage of predicted critical Loads (classified as predictable by

LVPT confidence counters) which is 85% on the floating-point benchmarks and only 40% on the integer benchmarks [Gel08a]. Finally, the difference is also slightly increased by the higher prediction accuracy obtained on the floating-point benchmarks.

We also measured the memory traffic reduction as the percentage of correctly predicted Loads reported to the total number of memory accesses. Our evaluations show an average memory traffic reduction of 1.58% on the integer benchmarks and of 10.93% on the floating-point benchmarks, which are in concordance with our energy consumption estimations.

The selective instruction reuse approach proposed by Golander and Weiss (presented in paragraph 6.1) achieves an average IPC speedup of 2.5% on the SPEC 2000 integer benchmarks, of 5.9% on the floating point benchmarks, and an improvement in energy-delay product of 4.80% and 11.85%, respectively. In comparison, our improved superscalar architecture achieves an average IPC speedup of 3.5% on the integer SPEC benchmarks, 23.6% on the SPEC floating-point benchmarks, and an improvement in energy-delay product of 6.2% and 34.5%, respectively.

6.3. Contributions to Dynamic Value Prediction: CPU Context Prediction

The main aim of this section consists in focusing dynamic value prediction to the CPU context [Vin05a, Vin05b]. The idea of attaching a value predictor to each CPU register (register-centric predictor) instead of an instruction or memory-centric predictor is original and could involve new architectural techniques for improving performance and reducing the hardware cost of speculative microarchitectures. In an earlier work [Flo02], Florea et al. performed several experiments to evaluate the value locality exhibited by MIPS general-purpose integer registers. The results obtained on some special registers ($at, $sp, $fp, $ra) were quite remarkable (≈90% value locality degree) leading to the conclusion that value prediction might be successfully applied at least on these favorable registers.

Whether the prediction process has been instruction (producer) or memory-centered with great complexity and timing costs, by implementing the well known value prediction schemes [Lip96a, Saz99] centered on the CPU's registers will reduce the hardware cost. However, there are some disadvantages. Addressing the prediction tables with the instructions' destination register name (during the *decode* stage) instead of the Program Counter will cause some interference. However, we have proved that, with a sufficiently large history a hybrid predictor could eliminate this problem and achieve very high prediction accuracy (85.44% at average on eight MIPS registers using SPEC'95 benchmarks and 73.52% on 16 MIPS registers using SPEC 2000 benchmarks). The main benefit of the proposed VP technique consists in unlocking the subsequent dependent instructions.

6.3.1. Register Value Predictors

Statistical results based on simulation have proved that commonly used programs are characterized by strong value repetitions [Lip96a, Sod00]. The main causes for this phenomenon are: data and code redundancy, program constants, and the compiler routines that resolve virtual function calls, memory aliases, etc. The register value locality is frequently met in programs and shows the number of times each register is written with a value that was previously seen in the same register and dividing by the total number of dynamic instructions having this register as their destination field [Flo02, Gel03].

As we observed in [Vin05a, Gel03], the value locality on some registers is remarkable high (90%), and this predictability naturally leads us to the idea of implementing value prediction on these favorable registers. Dynamic value prediction on registers represents a new technique that allows the speculative execution of the read after write dependent instructions by predicting the values of the destination registers during second half of the instruction's *decode* stage (see Figure 6.9). The Value Prediction Table (VPT) is accessed with the name of the destination register. The register's next value is predicted based on the last values belonging to that register. In the case of a valid prediction, the VPT will forward the predicted value to the subsequent corresponding RAW dependent instructions. After execution, when the real value is known, it is compared with the predicted value. If the value was correctly predicted the critical path might be reduced. In the case of a misprediction the speculatively executed dependent instructions are re-issued for execution (recovery).

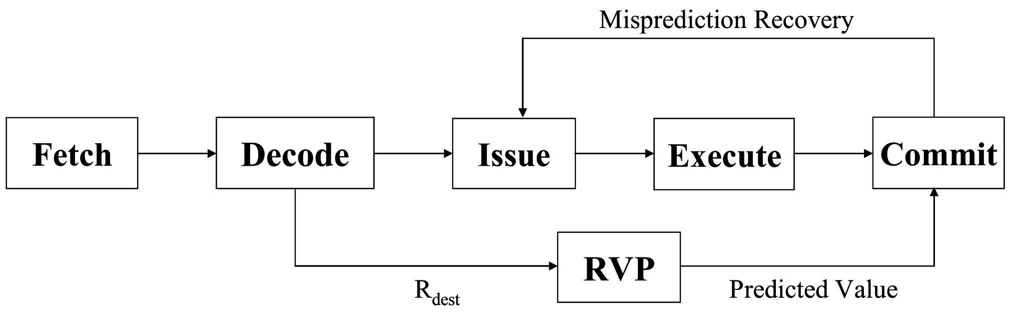


Figure 6.9. The implementation of the register value prediction mechanism in the pipeline structure of a general microarchitecture

In [Vin05a, Gel03] we developed and simulated several different basic value predictors, such as the last value predictor, the stride value predictor, the context-based predictor and hybrid value predictors to capture certain type of value predictabilities from the SPEC benchmarks and to obtain higher prediction accuracy. All these predictors were adapted to our proposed prediction model.

6.3.1.1. Last Value Predictors

The *last value predictors* (see Figure 6.10) predict the next value as the same as the last value stored in the corresponding register. Exploiting the correlation between register names and the values stored in those registers will decrease instruction latencies. Each register used in the prediction mechanism has an entry in the VHT. In this way the number of entries in the prediction table is the same as the number of logical registers.

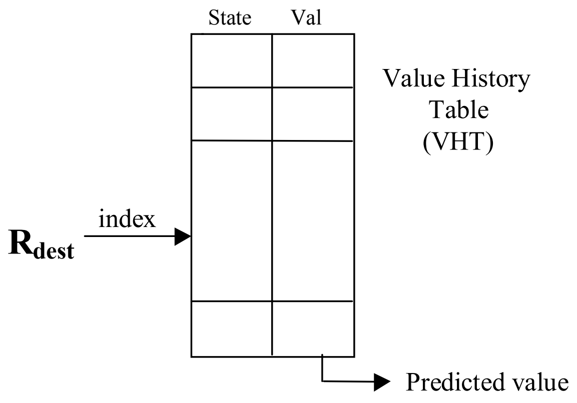


Figure 6.10. Last value predictor

Each entry of the prediction table has its own automaton in the *State* field (a 2-bit saturating confidence counter with two *unpredictable* and two *predictable* states). The last value from the *Val* field is predicted only if the automaton is in a *predictable* state. Obviously, it is necessary to verify the value generated by the value history table (VHT). The automaton's state will be changed according to the comparison between the predicted and actual values. The *Val* field is also updated.

6.3.1.2. Stride Predictors

In this case, considering that v_{n-1} and v_{n-2} are the most recent values, the new value v_n will be calculated using the recurrence formula: $v_n = v_{n-1} + (v_{n-1} - v_{n-2})$, where $(v_{n-1} - v_{n-2})$ is the stride of the sequence. Figure 6.11 shows the structure of this predictor.

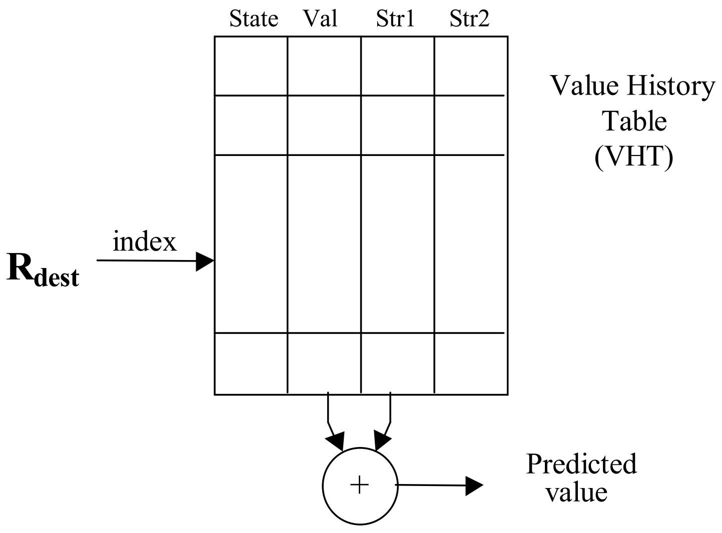


Figure 6.11. Stride predictor

The *Str1* and *Str2* fields keep the last two strides. Each time a register is used as destination, its current stride is computed: $Str = V - Val$, where *V* is the actual value of that register and *Val* is its last value stored in the VHT. The automaton is incremented if the prediction is correct otherwise it is decremented. If $Str_1 = Str_2$, the predicted value is calculated adding the stride Str_2 to the value stored in the VHT's *Val* field. If the automaton is in the predictable state, the prediction is furnished.

6.3.1.3. Context-Based Predictors

The context-based predictors predict the value that will be stored in a register based on the last values stored in that register. A context is a finite sequence of values with repeated appearance as in a Markov chain. The *Prediction by Partial Matching* (PPM) algorithm has been already presented in Section 4.3. A PPM-based predictor furnishes the value that followed the considered context with the highest frequency. Obviously, the predicted value depends on the context length. A longer context frequently drives to higher prediction accuracy but sometimes it can behave as noise.

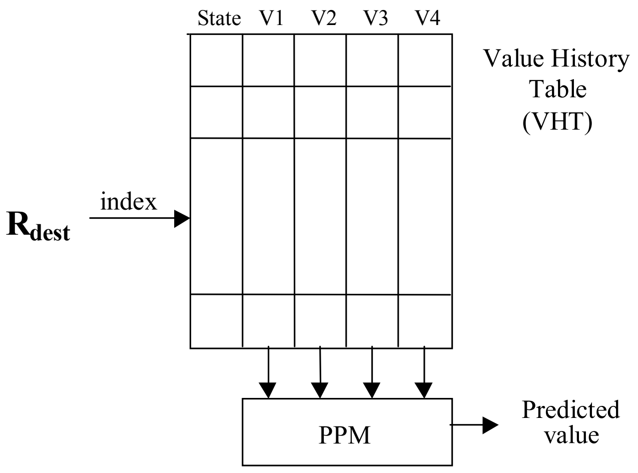


Figure 6.12. Structure of a context-based PPM predictor

Figure 6.12 shows the structure of the context-based predictor. Each entry from the VHT has an associated automaton that is incremented when the prediction is correct and is decremented in the case of a misprediction. The fields V_1, V_2, …, V_4 store the last four values associated with each register (considering that the predictor works with a history of four values). If the automaton is in the predictable state, it predicts the value that follows the context with the highest frequency.

6.3.1.4. Hybrid Predictors

It has been shown that a single type of predictor does not offer the best results. Some types of value sequences generated in programs are better predicted with a certain predictor, and others, with another type of predictor [Wan97]. Therefore, it is natural to consider the idea of hybrid prediction: two or more value predictors working together dynamically in the prediction process. Figure 6.13 shows a hybrid predictor composed of a context-based PPM predictor and a stride predictor. The context-based predictor always has priority, as in [Wan97]. In this way the value generated by the stride predictor is only used if the context-based predictor cannot generate a prediction.

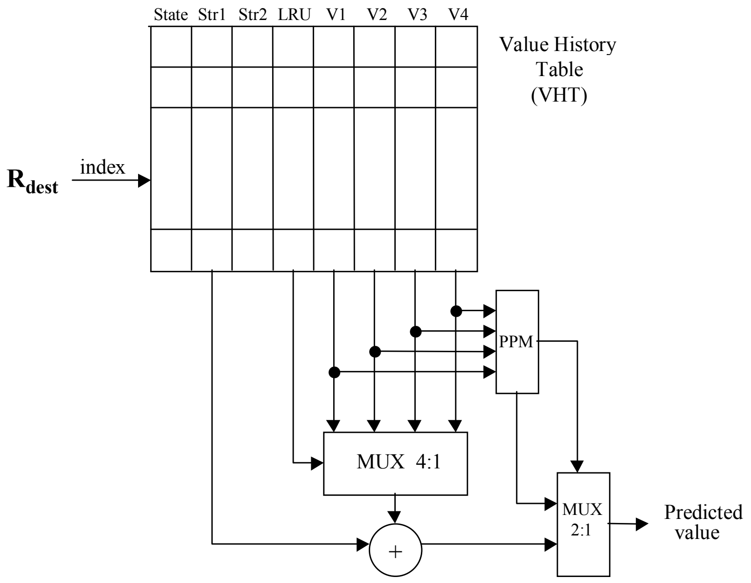


Figure 6.13. Hybrid predictor (PPM & stride)

Figure 6.14 presents the hybrid predictor composed of a 2-Level predictor and a Stride predictor adapted for register-centric prediction. It has the same functionality as the instruction-centric approach [Wan97] presented in Section 2.3, but it is indexed with the destination register name instead of the PC. This fixed prioritization used in Figures 6.13 and 6.14 seems not to be optimal. Probably a dynamic prioritization based on some confidences should be better (the predictor having the highest confidence degree will have priority).

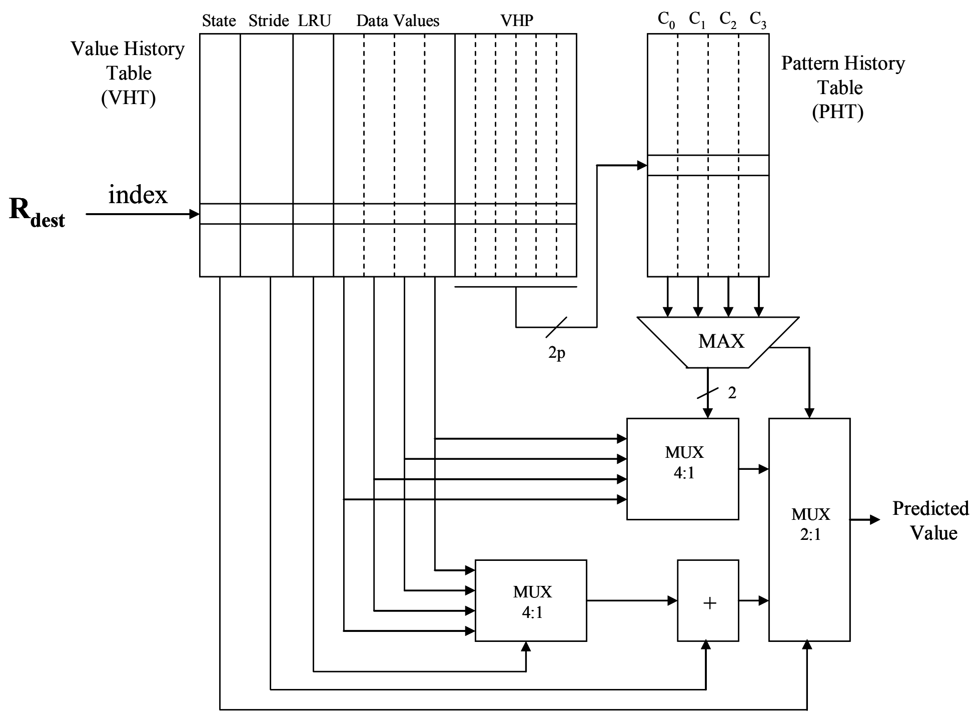


Figure 6.14. Hybrid predictor (two-level & stride) with fixed prioritization

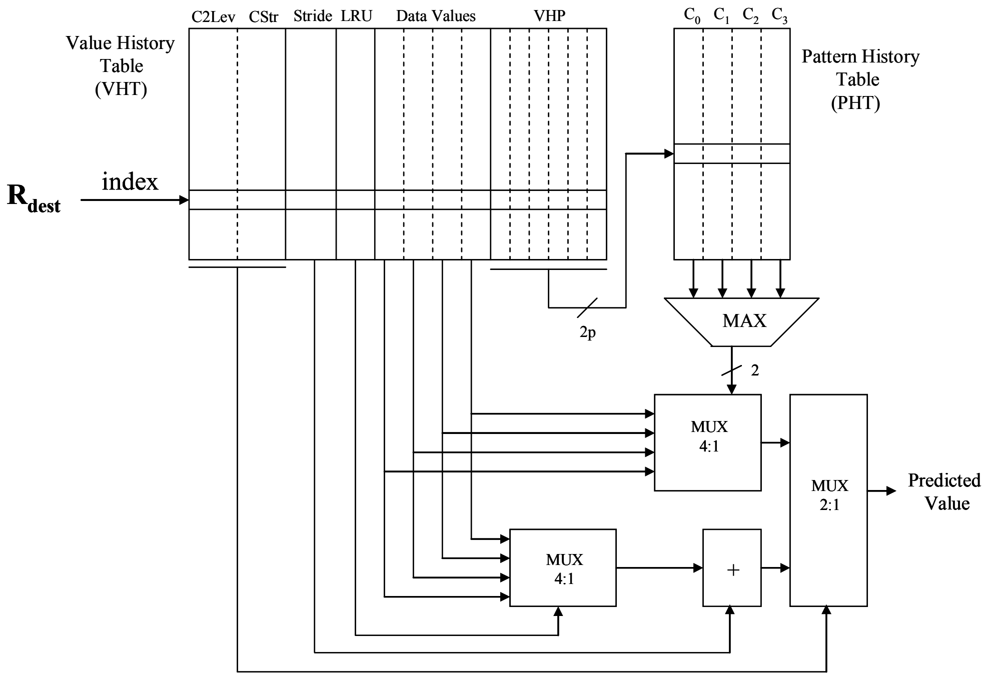


Figure 6.15. Hybrid predictor (two-level & stride) with adaptive prioritization

The adaptive hybrid predictor presented in Figure 6.15 uses a saturating confidence counter for each component predictor: *C2Lev* for the 2-Level predictor and *CStr* for the Stride predictor. Thus, it dynamically selects the most confident predictor. Other adaptive neural metapredictors have been proposed and evaluated in [Vin04a], but with less efficiency mainly due to the complexity of the backpropagation learning algorithm. Some simplified perceptron-based metapredictors might be more efficient and feasible for hardware implementation

6.3.2. Simulation Methodology

We developed a cycle-accurate execution driven simulator derived from the *sim-outorder* simulator of the *SimpleScalar* toolset [Sim]. The baseline superscalar processor supports out-of-order instruction issue and execution. We modified it to incorporate our proposed register value predictors. Table 6.4 shows the configuration of the baseline processor used to obtain the results.

To perform our evaluation, we collected results from different versions of SPEC benchmarks: five integer (*li, go, perl, ijpeg, compress*) and three floating-point (*swim, hydro, wave5*) SPEC'95 benchmarks. We simulated seven benchmarks (*gzip, b2zip, parser, crafty, gcc, twolf* and *mcf*) from the CINT SPEC 2000 set.

The number of instructions fast forwarded through before starting our simulations is 400 million. We used the `–fastfwd` option in SimpleScalar / PISA 3.0 to skip over the initial part of execution in order to concentrate on the main body of the programs. Results are then reported by simulating each program for 500 million committed instructions.

Processor Core	Fetch / Decode / Issue Width	8 instruction / cycle
	Reorder Buffer Size	128 entries
	Load-Store Queue	64 entries
	Integer ALUs	8 units, 1-cycle latency
	Integer Multiply / Divide	4 units, 3 / 12-cycle latency
Predictors	Hybrid branch predictor	*gshare* with 16K entries, 14 bit history, *bimodal* with 16K entries.
	Branch and Value misprediction	7-cycle latency
Memory	Memory Access	60-cycles latency
	Memory Width	32 bytes
Caches	Level-one data cache	4-way set associative, 64 KB, 1-cycle hit latency
	Level-one instruction cache	direct mapped, 128 KB, 1-cycle hit latency
	Level-two cache (unified)	4-way set associative, 1024 KB, 10-cycle hit latency

Table 6.4. Machine configuration for baseline architecture

6.3.3. Experimental Results

Starting with a minimal superscalar architecture, we studied how the simulator's performance will be affected by the variation of its parameters. We now present the results obtained with a hybrid of PPM and stride register value predictor. Each register has associated a 4-state confidence automaton. A prediction is made only if the automaton is in one of the two predictable states. In Figures 6.16 and 6.17, respectively, each bar represents the average of register value prediction accuracy obtained for eight SPEC'95 benchmarks and for seven integer SPEC 2000 benchmarks, respectively.

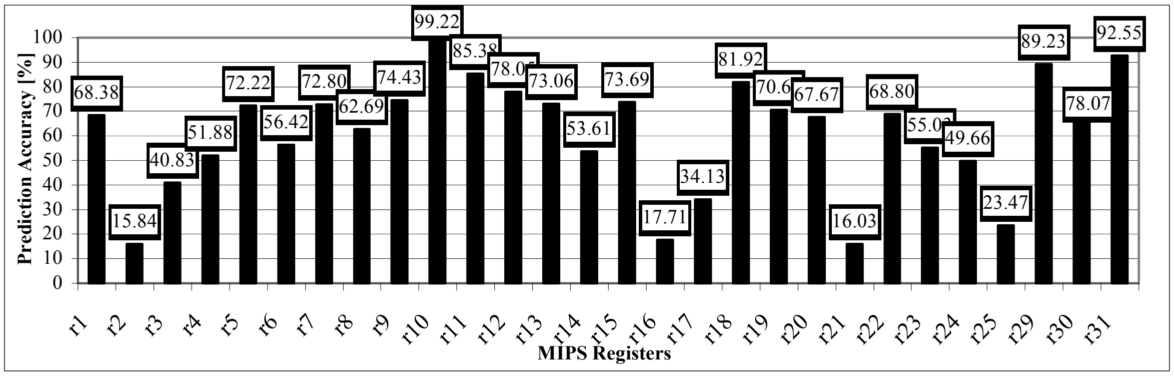


Figure 6.16. Register value prediction using a hybrid predictor (PPM, stride), a history of 256 values, and a pattern of 4 values (SPEC'95 simulation results)

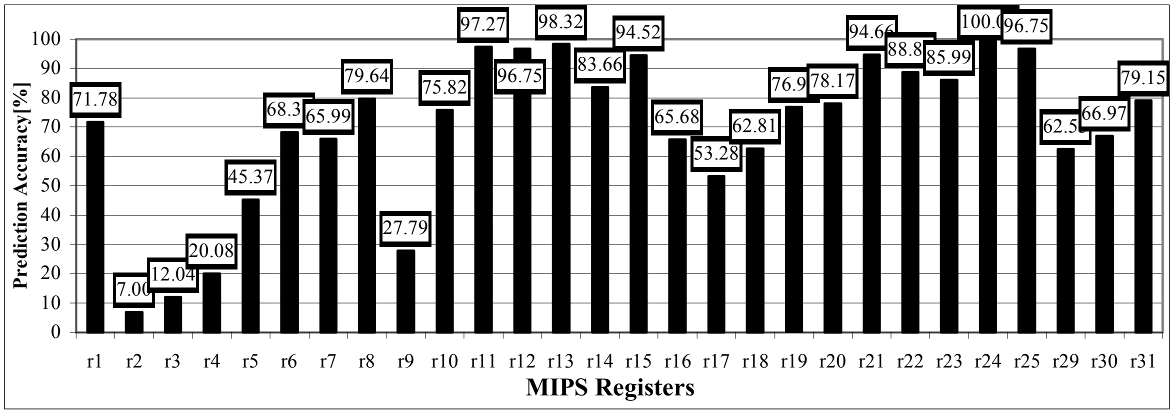


Figure 6.17. Register value prediction using a hybrid predictor (PPM, stride), a history of 256 values, and a pattern of 4 values (SPEC 2000 simulation results)

In Figures 6.16 and 6.17 we calculated the prediction accuracy (PA) using the following formula:

$$PA(R_k) = \frac{\sum_{i=1}^{n} CPV^k(i)}{\sum_{i=1}^{n} VRef^k(i)} \qquad (6.5)$$

where n = number of benchmarks (8 for SPEC'95 and 7 for SPEC 2000), k = register number, $CPV^k(i)$ = number of correctly predicted values for register R_k (on benchmark i), and $VRef^k(i)$ = the total number of dynamic instructions that have register R_k as their destination (on benchmark i).

In the next investigations, we are focusing only on the predictable registers which have prediction accuracy higher than a certain threshold (60% and 80%, respectively), measured using the PPM-based hybrid predictor on the SPEC benchmarks. As it can be seen in Figures 6.16 and 6.17 the registers having a prediction accuracy higher than 60% are: 1, 5, 7–13, 15, 18–20, 22, 29–31 on SPEC'95, and, 1, 6–8, 10–16, 18–25, 29–31 on SPEC 2000. The statistic results on the SPEC'95 benchmarks exhibit a using degree of 19.36% for these 17 registers. This means that 19.36% of instructions use one of these registers as a destination. The equivalent average result on SPEC 2000 is 13.24% using 22 general purpose registers.

In Figures 6.18 and 6.19 we compared the previously presented value prediction techniques: last value prediction (Figure 6.10), stride prediction (Figure 6.11), PPM prediction (Figure 6.12) and PPM-based hybrid prediction (Figure 6.13). We used in the prediction process only the 17 favorable registers on the SPEC'95 benchmarks and 22 favorable registers on the SPEC 2000 benchmarks. The PPM and the hybrid predictors use a history of 256 values and a search pattern of 4 values.

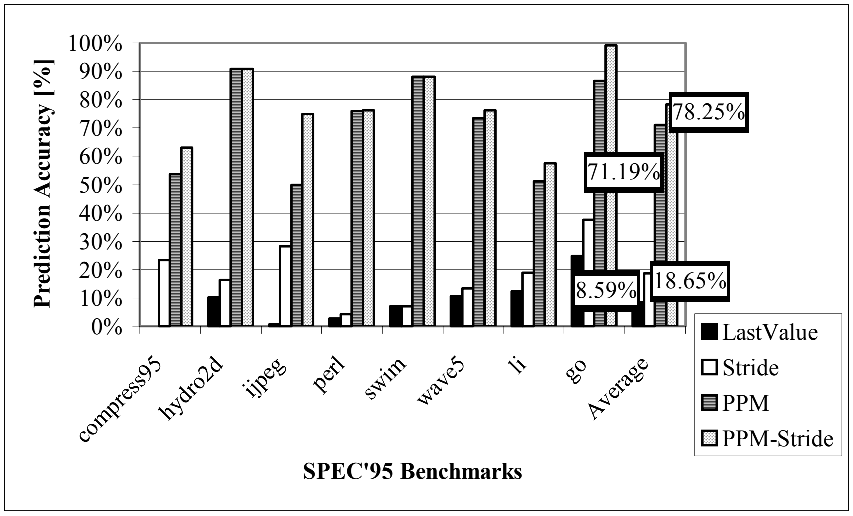


Figure 6.18. Prediction accuracy using 17 favorable registers (PA>60%) on the SPEC'95 benchmarks

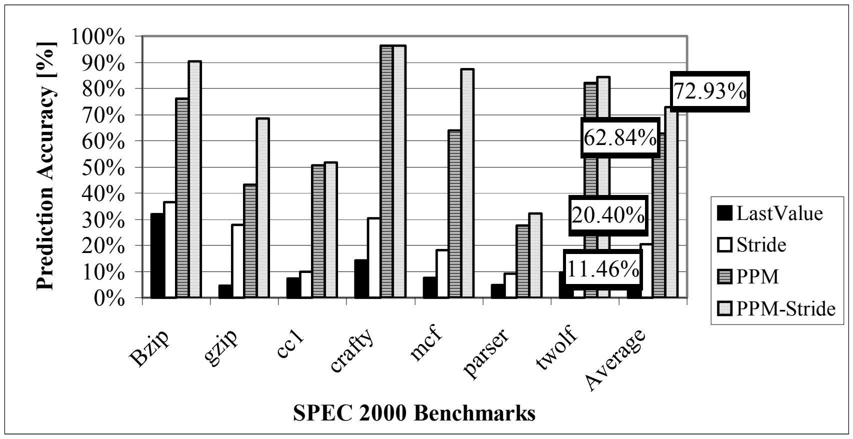


Figure 6.19. Prediction accuracy using 22 favorable registers (PA>60%) on the SPEC 2000 benchmarks

These results (see Figures 6.18 and 6.19) represent the global prediction accuracies of the favorable registers for each benchmark. The hybrid predictor synergy can be observed. It involves an average prediction accuracy of 78.25% on the SPEC'95 benchmarks and 72.93% on the SPEC 2000 benchmarks.

Now we will try a more elitist selection considering only the registers with prediction accuracy higher than 80% (see Figures 6.20 and 6.21). The selection is again based on Figures 6.16 and 6.17. We can observe that there are 8 registers that fulfill this condition (1, 10–12, 18, 29–31) on the SPEC'95 benchmarks and 16 registers (1, 8, 11–15, 20–25, 29–31) on the SPEC 2000 benchmarks (registers 1, 29–31 are included even if they do not fulfill this condition because they exhibit a high degree of value locality [Vin05a] and they also have special functions). The global using rate of these registers is 10.58% on the SPEC'95 benchmarks, and 9.01% on the SPEC 2000 benchmarks.

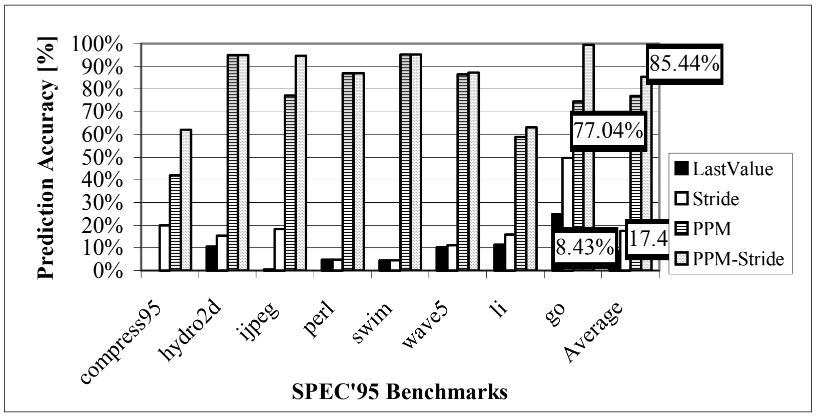


Figure 6.20. Prediction accuracy using 8 favorable registers (PA>80%) on the SPEC'95 benchmarks

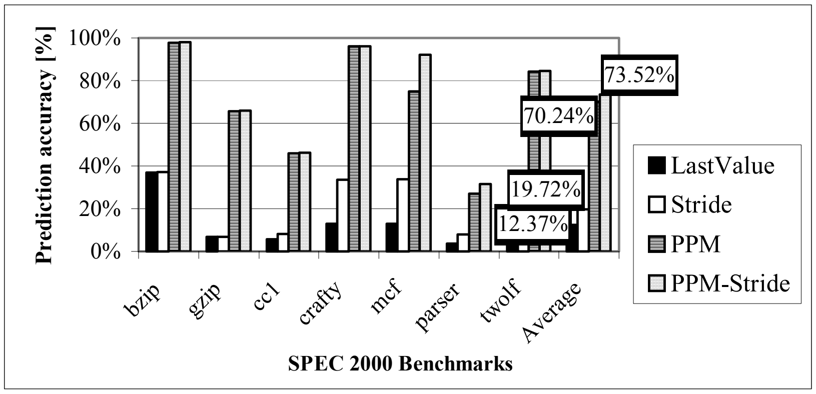


Figure 6.21. Prediction accuracy using 16 favorable registers (PA>80%) on the SPEC 2000 benchmarks

Figures 6.20 and 6.21 emphasize, for each benchmark, the global prediction accuracy obtained with the implemented predictors using 8 and 16 selected registers, respectively (threshold over 80%, according to the previous explanations). Each bar represents the prediction accuracy for a certain benchmark, measured by counting the number of times when prediction is accurate for any of the favorable registers and dividing by the total number when these registers are written. The simulation results offered by the last value predictor are relatively close to the stride predictor's results. The best average prediction accuracy was obtained with the hybrid predictor 85.44%, which was quite remarkable (on some benchmarks over 96%).

Figures 6.22 and 6.23 show the speedup obtained compared to the baseline processor when using each register value predictor.

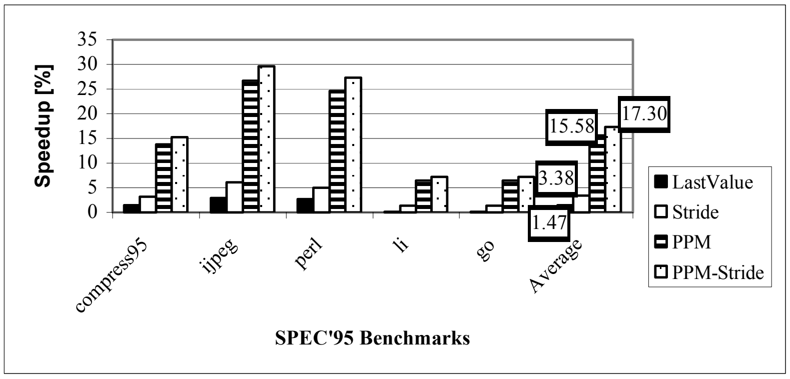


Figure 6.22. Speedup over baseline machine using 8 favorable registers (SPEC'95)

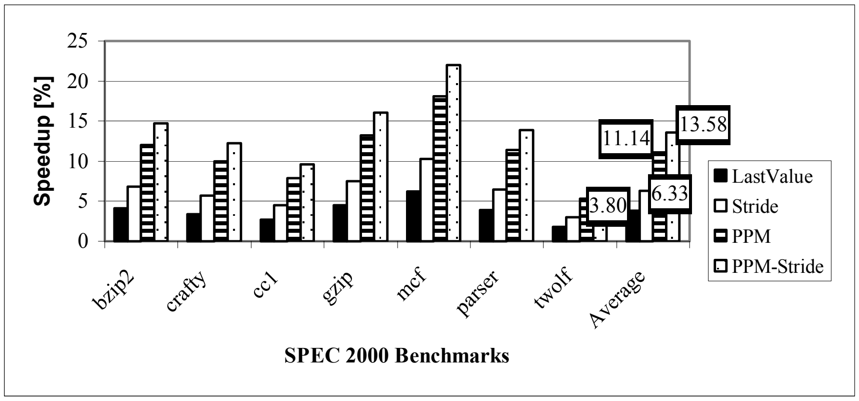


Figure 6.23. Speedup over baseline machine using 16 favorable registers (SPEC 2000)

Finally, in Figures 6.24 and 6.25 we have compared the PPM-based hybrid predictor *(PPM-Stride)* with the two-level-based hybrid predictors: *2Lev-Stride* with fixed prioritization (presented in Figure 6.14) and *2Lev+Stride* with adaptive prioritization (presented in Figure 6.15), both using a history of 32 values and a pattern of 4 values.

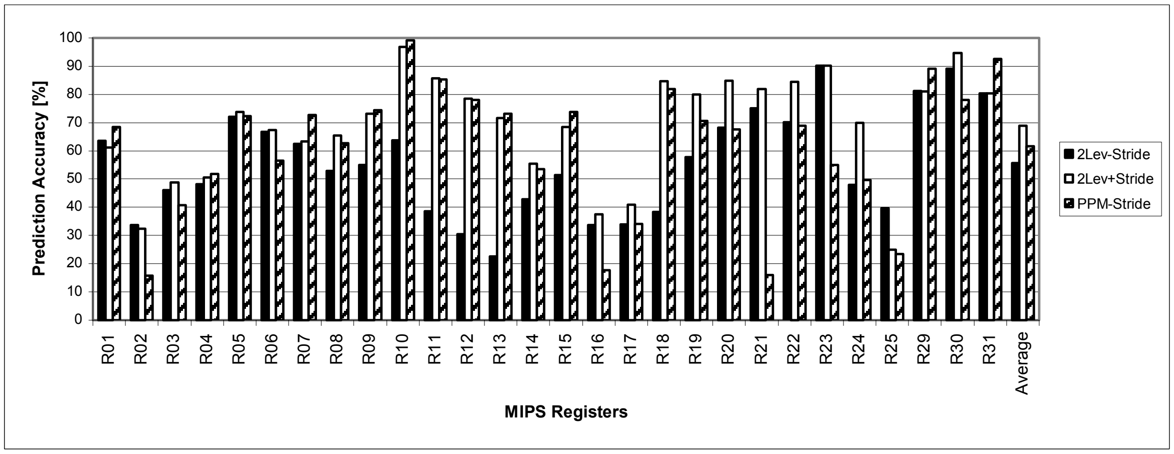


Figure 6.24. Comparing the hybrid predictors on the SPEC'95 benchmarks

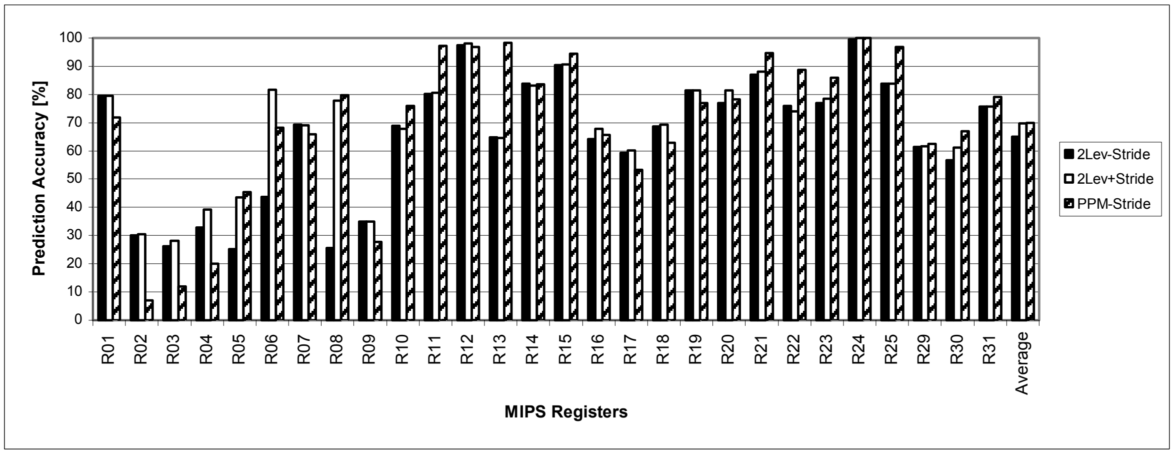


Figure 6.25. Comparing the hybrid predictors on the SPEC 2000 benchmarks

Figures 6.24 and 6.25 show that the hybrid predictor with adaptive prioritization composed of a two-level and a stride predictor is comparable to or even outperforms the PPM-based hybrid predictor, at significantly lower implementation cost and complexity.

6.4. Summary

In this chapter we have presented and evaluated a superscalar architecture that selectively anticipates the values produced by high-latency instructions. As we pointed out, about 28% of branches (more than 5% being unbiased) are dependent on long-latency instructions. Therefore, our goal was to attenuate the negative impact of branches, and especially of unbiased branches, over global performance. We developed a Reuse Buffer and a Trivial Operation Detector for Mul and Div instructions and a simple Last Value Predictor for critical Load instructions, and we integrated all these structures into the M-SIM simulator [Sha05].

The experimental results, performed on the SPEC 2000 benchmarks, show a significant speedup and improved energy consumption for the proposed architecture. Using a Reuse Buffer of 1024 entries together with the Trivial Operation Detector improves the IPC with 2.2% and reduces the relative energy consumption with 4% on the floating-point benchmarks. Predicting critical Load instructions through an additional Last Value Predictor, improves the IPC with 3.5% on the integer benchmarks and with 23.6% on the floating-point benchmarks. This significant speedup lowers the relative energy consumption (EDP) with 6.2% on the integer benchmarks and with 34.5% on the floating-point benchmarks. Consequently, applying some well-known techniques selectively on long-latency instructions provides serious performance gain and significantly reduces energy consumption within the simulated architecture.

Finally, we have introduced and studied the register value prediction concept. As we discussed, the intention of the register value prediction is to reduce the unfavorable effect of the RAW dependencies, by reducing the wait times of the subsequent dependent instructions. Also, the prediction focused on registers instead of instructions is advantageous because fewer predictors are needed, thus significantly saving complexity and costs. We proposed to exploit the value locality on registers using different prediction techniques. We used the hybrid predictor presented in Figure 6.13 to select the favorable registers. We continued after that with the evaluation of the predictors using registers with prediction accuracy higher than 60%. The best results were obtained with the hybrid predictor: an average prediction accuracy of 78.25% and a using rate of 19.36%. We then tried a more elitist selection of the registers and we continued the evaluation of the predictors using only the registers with prediction accuracy higher than 80%. The best results were obtained again with the hybrid predictor: an average prediction accuracy of 85.44% (on some benchmarks with over 96%) and a using rate of 10.58%. Also, considering an 8-issue out-of-order superscalar processor, simulations show that register-centric value prediction produces average speedups of 17.30% for the SPECint95 benchmarks and 13.58% for the SPECint2000 benchmarks. We also showed that the PPM-based hybrid predictor is outperformed by the less complex but adaptive two-level-based hybrid predictor.

"Entities should not be multiplied unnecessarily"

William of Ockham

7. Enhancing the Simultaneous Multithreading Paradigm Through Selective Instruction Reuse and Value Prediction

In the previous chapter we improved a superscalar microarchitecture with selective instruction reuse and value prediction techniques focused on long-latency instructions. We obtained significant IPC speedups and energy-delay product gains, proving the necessity of these techniques for higher instruction-level parallelism. A very important question is: would these techniques improve even multithreading architectures? Additionally a multithreaded processor would naturally hide the long instructions latencies, including the memory-wall, and also some of the branches' problems. This chapter answers the question by evaluating a simultaneous multithreaded architecture enhanced with selective instruction reuse and value prediction to anticipate the results of long-latency instructions.

7.1. Related Work

This section presents an overview of several multithreading approaches, focusing then on Simultaneous Multithreading architectures, used and enhanced during this chapter. With multithreading multiple threads can share the functional units of a single processor [Hen03]. To support multithreading, the processor must be able to maintain the independent state of each thread in separate resource copies. The hardware also must support quick context switches between threads.

7.1.1. Multithreading Architectures

There are two different multithreaded architecture designs [Ung02, Ung03]: implicit- and explicit multithreaded processors. Implicit multithreaded superscalar processors aim at a low execution time of a single program, while explicit multithreaded processors aim at a low execution time of a multithreaded workload.

Implicit multithreaded processors concurrently execute several threads from a single sequential program. The threads in such architectures represent contiguous regions of the static or dynamic instruction sequence that can be obtained with or without the help of the compiler. *Multiscalar processors*, first introduced by Gurindar Sohi, divide a single-threaded program into tasks that are distributed to different parallel processing units. The multiscalar model supports control speculation and data dependence speculation. If a control speculation turns out to be incorrect, the speculative thread and all its successor threads are discarded. Data dependence speculation occurs when a thread loads data from memory with the expectation that the predecessor threads will not store a value to the same memory location. *Trace processors* partition a processor into distinct cores and divide the program into traces that are collected by a trace cache. They solved the so called fetch bottleneck limitation [Vin07]. One core of the processor executes the current trace while the other cores execute future traces speculatively.

Explicit multithreaded processors are able to execute threads of several processes concurrently. A classification of explicit multithreaded processors that issue instructions from a single thread per cycle, distinguishes between *fine-grained* and *coarse-grained multithreading* [Hen03], while an explicit multithreading technique that issue instructions from multiple threads per cycle is *simultaneous multithreading*. In fine-grained multithreading (interleaved multithreading) threads are switched after each instruction fetch, interleaving in this way their execution. More exactly, an instruction from a certain thread enters in the pipeline after the retirement of the previous instruction of that thread. Thus, the processor must be able to switch threads every clock cycle. Interleaved multithreading partially eliminates control and data dependences between instructions in the pipeline, leading to a simple and fast pipeline. Memory latency is tolerated by not scheduling a thread until the memory operation has completed. In order to completely hide pipeline hazards this model requires at least as many threads as many stages are in the pipeline. The key disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads because they are delayed by instructions from the other active threads. This deficiency can be overcome with the *dependence lookahead technique* and the *interleaving technique* [Ung02]. The dependence lookahead technique, by using additional opcode bits, allows the compiler to state the number of instructions directly following in program order that are not data- or control-dependent on the instruction being executed. Thus, the instruction scheduler can feed non-data- and non-control-dependent instructions of the same thread successively into the pipeline. The interleaving technique uses caching and full pipeline interlocks. With caching not all memory references are long latency operations. Using full pipeline interlocks, a certain context is not limited to only one instruction in the pipeline. Instructions are issued switching each cycle between available contexts. Contexts become unavailable when they encounter a long-latency operation and become available again when that operation completes. Thus, even a single context is supported in the pipeline.

In the case of coarse-grained multithreading (blocked multithreading), threads are switched only when stalls occur, and thus, it does not slow down the execution of the individual threads (single-thread performance is similar to that of superscalar processors). Instructions are issued from a single thread and the pipeline is emptied if a stall occurs. The disadvantage of coarse-grained multithreading consists in the start-up pipeline costs, since the thread that is executed after the stall must fill again the pipeline. Coarse-grained multithreading can be classified based on the event that triggers a context switch into static and dynamic models [Ung02]. In static models the context switch is encoded by the compiler and occurs each time the same instruction is executed in the instruction stream. The advantage of static models is that context switching can be triggered in the *fetch* stage of the pipeline. The static model with explicit switching uses an additional instruction for triggering context switches. In static models with implicit switching, a context switch decision depends on the class of the fetched instruction. Instruction classes that cause context switch include Load, Store and, obviously, branch instructions. In dynamic models the context switch is triggered by dynamic events. Usually, all instructions between the *fetch* stage and the stage that triggers the context switch are discarded, leading to a higher context switch overhead. Several dynamic models are presented in [Ung02]. The *switch-on-cache-miss* dynamic model switches the context if a Load or Store instruction misses in the cache. These switches are detected in a late stage of the pipeline and, therefore, a large number of subsequent instructions that are already in the pipeline must be discarded, increasing the context switch overhead. The *switch-on-signal* dynamic model switches the context if a specific signal occurs, such as interrupt request, trap or message arrival. The *switch-on-use dynamic* model switches the context when an instruction tries to use the still missing value of a Load. This model is implemented by adding a *valid bit* to each register, the bit being cleared when a Load to the corresponding register is issued and set when the result is available. A context switch occurs only if a thread needs a value from a register whose valid bit is still cleared. The *conditional-switch* dynamic model couples an explicit instruction with a condition, the context being switched only if the condition is fulfilled. Such conditional-switch instruction can be used after a group of Load instructions, the switch being ignored if all Loads hit the cache and performed otherwise.

In [Ung02] the nanothreading and microthreading coarse-grained techniques are also presented. The nanothreading approach uses a nanothread that executes in the same register set and the same page as the main thread. When a stall occurs in the main thread, the processor automatically begins fetching instructions from the nanothread. Usually the nanothread focuses on simple tasks that can be done asynchronously to the main thread, such as prefetching data into a buffer. In the DanSoft processor, nanothreading is used to fetch both sides of a branch. A static three-bit branch prediction scheme is used. In the case of the states with low branch prediction confidence (the middle four of the eight states) the processor fetches instructions from both paths. If the branch is mispredicced in the main thread, the path executed by the nanothread is used, generating a misprediction penalty of only one to two cycles. The microthreading technique is similar to the nanothreading, but the number of threads is not restricted to only two. The threads share the same register set and the same run-time stack. A disadvantage of both nanothreading and microthreading techniques is that the compiler has to schedule registers to each active thread, since all threads share the same register set.

7.1.2. Simultaneous Multithreading

Combining the superscalar instruction issue with the multithreading approach, naturally leads to the idea of issuing instructions from several active threads in parallel. Latencies that occur in the execution of single threads are bridged by issuing operations of the remaining threads. Simultaneous multithreading (SMT) uses the resources of a multiple-issue processor to simultaneously exploit both thread-level parallelism (TLP) and instruction-level parallelism (ILP). In SMT processors [Egg97], TLP can come from either multithreaded programs or independent programs within a workload, whereas ILP comes from each single program or thread. SMT is motivated because it successfully exploits both types of parallelism and therefore uses resources more efficiently, increasing instruction throughput and speedup. Thus, instructions from multiple threads are issued simultaneously in a single clock cycle. In the case of out-of-order processors with dynamic scheduling, register renaming provides a large set of virtual registers that can be used to hold the register sets of multiple threads. The independent commitment of instructions from different threads can be supported if a separate reorder buffer is used for each thread.

A classification of SMT processors can be made based on their resource organization [Ung02]. In architectures with *resource sharing*, instructions from different threads share all resources: the fetch buffer, the physical registers that provide the renaming function for all register sets, the instruction window and the reorder buffer. The architectures with *resource replication* actually replicate all internal buffers of a superscalar processor, each buffer being associated to a specific thread. The issue unit is able to issue instructions from different instruction windows simultaneously to the execution units. The *threaded multipath execution* model, which exploits existing hardware of a SMT processor to execute simultaneously alternate paths of a conditional branch, is also presented in [Ung02]. Therefore, additional hardware is introduced into SMT processors to test for unused resources (hardware threads). If the hardware detects processor threads that are not processing useful instructions, the prediction confidence estimator is used to decide if only one path of a conditional branch should be followed (high prediction confidence), or both paths should be followed simultaneously (low prediction confidence).

In SMT architectures some processor structures (i.e. instruction queue, physical register files, execution units, caches) are shared among the threads, and others (ROBs, Load/Store Queues, branch predictors) are private to each thread [Bar08]. The different characteristics and requirements of every thread within a SMT environment can unbalance resource allocation and some threads will consume more resources than others. The overall performance of a SMT processor depends on how shared resources are distributed among threads. There are several possible policies to distribute the resource entries (distribution policies) and to select the instructions that will leave the resource at each cycle (scheduling policies). The most flexible scheme for distributing the entries of a resource is the *dynamic distribution* policy under which any instruction from any thread can compete for any free entry. The distribution of resources can also be *static*: each resource is partitioned and each thread has a private access to one partition. This completely prevents starvation and ensures a fair access to the common resources for all threads. However, the performance may not be optimal in this case because some threads may be slowed down due to a lack of resources whereas other threads might underuse their allocated partition. Static partitioning is widely used to share instruction queues among threads, due to its easier implementation. Besides distribution policy, shared resources are also controlled by a scheduling policy that arbitrates between threads to select the instructions that can leave the resource. The most common scheme is the very simple *Round-Robin* policy, which switches between threads in a circular way, regardless of their behavior. The ICOUNT policy is another possible strategy, which is based on dynamic priorities reevaluated at each cycle to reflect the number of instructions per thread present in the pre-issue pipeline stages. Instructions of several threads can be fetched simultaneously, with the constraint that the thread with the highest priority is satisfied first.

Liu and Gaudiot proposed in [Liu08] a resource sharing control technique on both the Instruction Fetch Queue (IFQ) and Reorder Buffer (ROB) structures in order to improve the performance of SMT processors. The research is important taking into account that the power of SMT lies in its ability to issue and execute instructions from different threads at every clock cycle. The authors are developing four distinct sharing control schemes, built on the well-known ICOUNT policy. They observed that controlling the resource sharing of either IFQ or ROB alone can only provide very limited performance improvement, even leading to performance degradation in some cases. On the other hand, controlling the resource sharing of both IFQ and ROB together could achieve significant performance gain.

Marcuello et al. in [Mar99] analyzed the performance of speculative multithreaded processors with different value predictors. The thread speculation logic of a Clustered Speculative Multithreaded processor is responsible for detecting those parts of a sequential program that can be executed by different threads. This architecture considers the beginning of loops as quasi-independent control points. Thus, each speculative thread corresponds to a different iteration of a loop, called *loop trace*. The value prediction is focused on trace input or output values, since these values flow through inter-thread dependences. The instruction-based predictors correlate their predictions with previous values of the same instruction whereas trace-based predictors correlate predictions with previous values of the same instruction within the same trace. The authors proposed a value predictor, called increment predictor, and evaluated its performance within a particular microarchitecture that implements the speculative multithreading paradigm. The increment predictor predicts every trace output value as the value of that storage location at the beginning of the trace plus an increment. This increment is computed as the difference between the values at the end and at the beginning of the trace. The predicted increment is updated when a new increment has been seen twice in a row. Their 1 KB trace-oriented increment predictor, with its prediction accuracy of 73%, clearly outperforms the trace-adapted versions of the last value, stride and context-based predictors.

Martin et al. show in their work [Mar01] that multithreaded pointer manipulation can generate erroneous results when value prediction is implemented without considering memory consistency correctness. Therefore, only verifying prediction correctness by comparing the predicted and actual values is not always sufficient. In a TLP system, unlike in a single-threaded uniprocessor, it is possible for a value prediction to be incorrect at the time of the prediction but "correct" by the time the predicted value is verified, since another thread or processor could have modified the value between prediction and verification. When multiple threads or processors concurrently access a logically shared memory, the definition of correctness becomes more complicated. Thus, speculative TLP implementations must ensure that value prediction does not cause consistency model violations.

An important impediment in developing large-scale SMT architectures is the register file size required by a large number of contexts. In [Red03] Redstone et al. introduce and evaluate the mini-thread concept, a simple extension of SMT that increases thread-level parallelism without register file size increase. A mini-threaded SMT architecture adds additional per-thread state to each hardware context. Using this hardware, an application can exploit more thread-level parallelism within a context, by creating multiple mini-threads that use their own per-thread state, but share the context's architectural register set. Their experimental results show that adding mini-threads improves performance by an average of 38% (and a maximum of 66%) on a 2-context SMT.

Ramírez et al. proposed in [Ram08] *runahead threads* to exploit memory-level parallelism while reducing resource contention in SMT processors. Runahead execution is a mechanism whose goal is to bring speculative data and instructions into the caches, and it was also used in [Mut03] within checkpointing architectures (see paragraph 2.1.3). The technique presented in [Ram08] applies runahead execution to any running thread when a long-latency Load is pending. Thus, when a thread undergoes a long-latency Load, it turns into a runahead thread and operates in speculative mode. With runahead threads, memory-bound threads can advance speculatively (instead of stalling) by using different resources for short times without disturbing the other threads. Their evaluations show that runahead threads improve throughput by 83% over static fetch policies.

In [Sub08] Subramaniam et al. studied the interaction between long-latency stalls caused by ambiguous memory dependences and SMT processing. A thread that encounters a stalling condition (e.g. a cache miss) can potentially tie up many of the shared resources for the entire latency of the stall. This effectively reduces the number of critical resources available to the non-stalled threads. If the stall timings are predictable, then this information can be directly exploited by the SMT fetch unit to better manage the shared processor resources. Therefore, the authors proposed a technique called *proactive exclusion* which stops the SMT to fetch from a thread (avoiding thus resource allocation) when a memory dependence is predicted, before the stalling condition has been occurred. In order to mitigate delaying such threads, they introduced the so called *early parole* mechanism that exploits the predictability of dependence delays and restarts fetching from an excluded thread in an anticipatory manner such that the instructions arrive to the out-of-order execution units just as the original dependence resolves. Their simulations show that a fetch policy which combines these two techniques yields a 16.9% throughput improvement on a 4-way SMT processor that supports speculative memory disambiguation.

7.2. Selective Instruction Reuse and Value Prediction in SMT Architectures

As a final objective of our research, we quantified the impact of our developed Selective Instruction Reuse and Load Value Prediction techniques in a simultaneous multithreaded architecture (SMT) that involves per thread Reuse Buffers and LVP tables [Vin08a].

We developed a cycle-accurate execution driven simulator derived from the M-SIM simulator [Sha05] supporting the unmodified, statically linked Alpha AXP binaries as well as the power estimation as supplied by the Wattch framework [Bro00]. M-SIM supports single threaded execution (superscalar mode) as well as the multithreaded mode in which multiple threads of control are executed simultaneously, according to the Simultaneous Multithreaded (SMT) model [Egg 97]. In the SMT mode, some processor structures (i.e. issue queue, physical register files, functional units, caches) are shared among the threads, and others (rename tables, ROBs, Load/Store Queues, branch predictors) are private to each thread. Figure 7.1 presents a SMT architecture enhanced with our selective instruction reuse and value prediction methods proposed in Section 6.2.

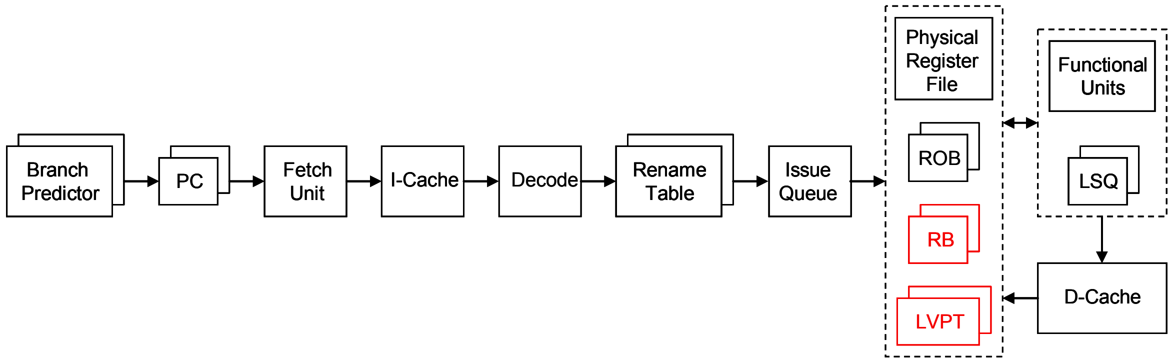


Figure 7.1. SMT architecture enhanced with selective instruction reuse and value prediction

Threads maintain separate PC counters, but share the fetch unit and I-Cache. Threads also share the available bandwidth in the front end, including fetch, decode and renaming. The M-SIM implements the well known ICOUNT fetch policy (briefly described in paragraph 7.1.2), by default, fetching from up to two threads per cycle. The M-SIM has implemented separate branch predictors per thread, which was shown in [Ram03] as providing the best performance for multithreaded processors. The Reorder Buffers (ROB) as well as our Reuse Buffers (RB) and Load Value Prediction Tables (LVPT) are private. Each thread maintains its own rename table because it has its own set of architectural registers. After renaming, instructions from all threads are dispatched into the shared Issue Queue. In the Issue Queue, instructions from all threads participate in instruction wakeup and compete for the issue bandwidth in selection. Instructions that are selected for issue continue to register file access. There are separate integer and floating-point physical register files, both being shared among threads. After register file access is complete, instructions begin execution on the functional units, which are also shared. Loads and Stores access the shared data cache. In order to maintain the correct ordering of memory accesses, the Load/Store Queue (LSQ) is used. The M-SIM uses separate LSQs per thread, so that an unresolved address from one thread does not prevent Loads in other threads from issuing. After execution, instructions write back to the register files. Commitment is done in order for each thread.

Our Reuse Buffers and Load Value Prediction Tables have the same structures as in Section 6.2 (see Figures 6.1 and 6.3). The RB and LVPT were implemented in *sim-outorder.c* within the M-SIM through the following structures:

```
struct RBLocation
{
        md_addr_t tag;  // the higher part of the buffered instruction's PC
        qword_t srcval1;        // first source value
        qword_t srcval2;        // second source value
        qword_t res;            // result value
};

struct RBLocation rbuff[10][10000];        // maximum 10 threads and 10,000 entries per thread

struct LVPTLocation
{
        md_addr_t tag;  // the higher part of the Load instruction's PC
        byte_t counter;  // 2-bit saturating counter
        qword_t value;  // Load value
};

struct LVPTLocation lvpt[10][10000];        // maximum 10 threads and 10,000 entries per thread
```

The context identifier (*context_id*) is maintained for each instruction in the Reorder Buffer (*ROB_entry*). The private resources are selected for instructions belonging to each thread based on this identifier.

7.3. Simulation Methodology

All simulation results are generated on the SPEC 2000 benchmarks [SPEC] and are reported on 1 billion dynamic instructions, skipping the first 300 million instructions. For the superscalar architecture we evaluated seven integer benchmarks (*bzip*, *gcc*, *gzip*, *mcf*, *parser*, *twolf*, *vpr*) and six floating-point benchmarks (*applu*, *equake*, *galgel*, *lucas*, *mesa*, *mgrid*). In SMT mode, the M-SIM runs multiple benchmarks as different threads in parallel. Therefore, we combined benchmarks into groups of 2, 3 or 6 depending on the simulated SMT architecture. Thus, we used {*bzip*, *gcc*}, {*gzip*, *parser*}, {*twolf*, *vpr*}, {*applu*, *equake*}, {*galgel*, *lucas*}, {*mesa*, *mgrid*} for our 2-way SMT, {*bzip*, *gcc*, *gzip*}, {*parser*, *twolf*, *vpr*}, {*applu*, *equake*, *galgel*}, {*lucas*, *mesa*, *mgrid*} for the 3-way SMT, and {*bzip*, *gcc*, *gzip*, *parser*, *twolf*, *vpr*}, {*applu*, *equake*, *galgel*, *lucas*, *mesa*, *mgrid*} for the 6-way SMT. Table 7.1 presents some important parameters of the simulated architecture:

Execution Latencies	**Execution unit**	**Number of units**	**Operation latency**
	intALU	4	1
	intMULT / intDIV	1	3 / 20
	fpALU	4	2
	fpMULT / fpDIV	1	4 / 12
Superscalarity	Fetch / Decode / Issue / Commit width = 4		
Branch predictor	bimodal predictor with 2048 entries		
Caches and Memory	**Memory unit**		**Access Latency**
	4-way associative L1 data cache, 32 KB		1 cycle
	8-way associative unified L2 data cache, 512 KB		6 cycles
	Memory		100 cycles
Resources	**Register File:** 32 INT / 32 FP		
	Reorder Buffer (ROB): 128 entries		
	Load/Store Queue (LSQ): 48 entries		

Table 7.1. Parameters of the simulated architecture

The dynamic power consumption measurements are generated using an 80 nm CMOS technology:

$$P_d = C \cdot V_{dd}^2 \cdot a \cdot f \tag{7.1}$$

where *C* is the capacitance, generated using *Cacti* [Shi01], V_{dd} is the supply voltage, and *f* is the clock frequency. V_{dd} and *f* depend on the assumed process technology. The activity factor *a* indicates how often clock ticks lead to switching activity on average. For the energy measurements, we used the Energy-Delay Product, a widely used metric [Gon96, Bro00, Gol07]:

$$EDP = \frac{Total\ Power}{IPC^2} \tag{7.2}$$

The Energy-Delay Product (EDP) represents the processor's total power, divided by the squared IPC.

7.4. Experimental Results

We measured the IPC and the dynamic power consumption of the proposed SMT architecture by varying the number of threads. Figures 7.2 and 7.3 present the IPC obtained by evaluating our developed superscalar and SMT architectures with and without Reuse Buffer and Load Value Predictor. According to our previous results obtained with the enhanced superscalar architecture (presented in paragraph 6.2.4), we optimally sized the RB and the LVPT to 1024 entries.

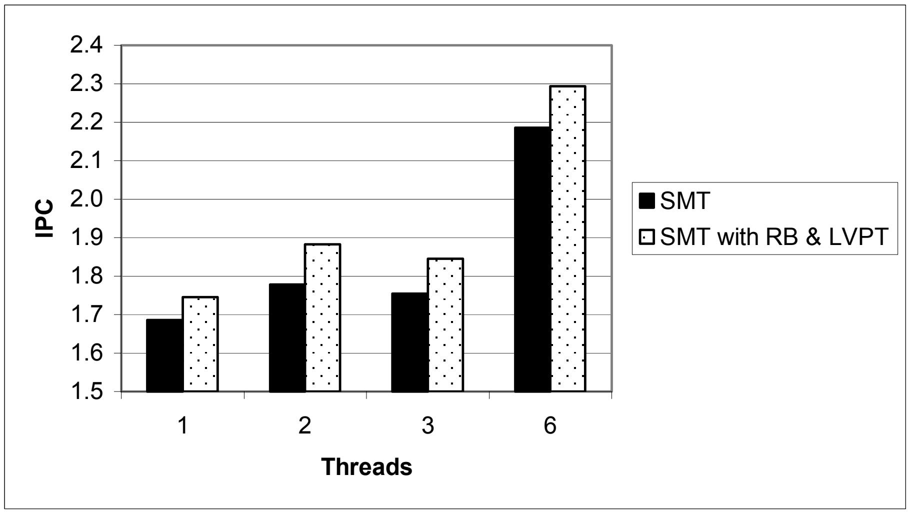


Figure 7.2. IPC obtained with and without RB & LVPT on the integer SPEC 2000 benchmarks

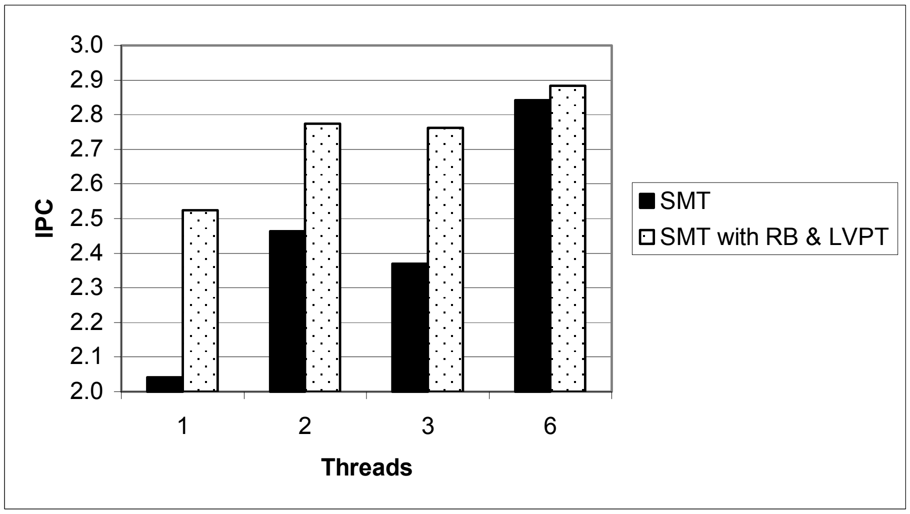


Figure 7.3. IPC obtained with and without RB & LVPT on the floating-point SPEC 2000 benchmarks

Figures 7.2 and 7.3 show that the RB and LVPT structures improve the IPC on all evaluated architectural configurations (superscalar and SMT). As far as concern floating-point benchmarks, the highest improvement was obtained with one thread, and as the number of threads grows, the IPC improvement becomes lower (see Figure 7.3). With fewer threads, the ten shared functional units (see Table 7.1) are underused and therefore the selective instruction reuse and value prediction techniques have an important improvement potential. With a higher number of threads, the same ten functional units are highly used by the SMT engine, thus both the instruction reuse and value prediction mechanisms becoming less important. Therefore, especially on floating-point benchmarks, with six threads we obtained the best IPC but the lowest relative IPC speedup (see Figures 7.3 and 7.4).

Finally, we evaluated, for different number of threads, the IPC speedup and the EDP gain of a SMT architecture enhanced with Selective Instruction Reuse and Value Prediction against a classical SMT architecture. The IPC speedups obtained with our superscalar (one thread) and SMT architecture (2, 3 and 6 threads) are presented in Figure 7.4, whereas Figure 7.5 presents the EDP gains achieved with the same architectures.

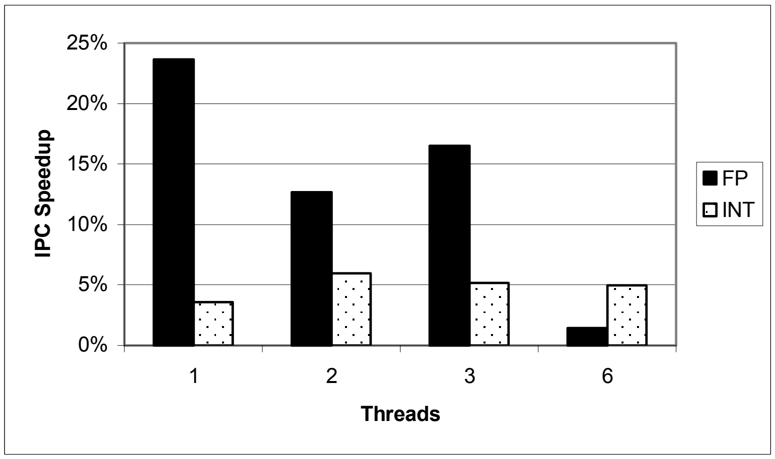


Figure 7.4. Relative IPC speedup (enhanced SMT vs. classical SMT) by varying the number of threads

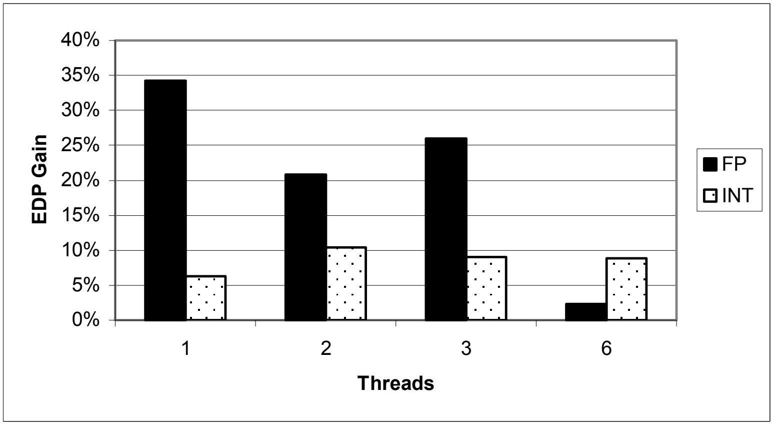


Figure 7.5. Relative energy-delay product gain (enhanced SMT vs. classical SMT) for different number of threads

As Figures 7.4 and 7.5 depict, the RB and LVPT structures achieved IPC speedups and EDP gains on all the simulated configurations. The best improvements on the integer benchmarks have been obtained with 2 threads: an IPC speedup of 5.95% and an EDP gain of 10.44%. Although, on the floating-point benchmarks, we obtained the highest improvements with the enhanced (LVP+Reuse) superscalar architecture, the SMT with 3 threads also provides an important IPC speedup of 16.51% and an EDP gain of 25.94%. Analyzing Figures 7.2 and 7.3 we can observe the advantage of SMT architectures against the superscalar architecture irrespective these are enhanced or not with selective instruction reuse and value prediction mechanisms.

7.5. Summary

In this chapter we have studied the impact of selective instruction reuse and value prediction in a Simultaneous Multithreaded architecture. We used these methods to anticipate the results of long-latency instructions (Mul, Div, Load), as we did in Chapter 6 within a superscalar architecture. Thus, we integrated the Reuse Buffer and Last Value Predictor structures into the M-SIM simulator [Sha05]. We implemented private RBs and LVPTs for each thread. Our simulation results, performed on the SPEC 2000 benchmarks, show that the IPC is better on all evaluated SMT configurations, when the RB and LVPT structures are used. However, as the number of threads grows, the IPC speedup becomes less significant, because the shared functional units are better exploited by the SMT engine even without RB and LVPT. We measured the highest IPC with the six-threaded enhanced SMT architecture: 2.29 on the integer SPEC 2000 benchmarks and 2.88 on the floating-point benchmarks.

"The strongest arguments prove nothing so long as the conclusions are not verified by experience"

Roger Bacon

8. Conclusions and Further Work

The main contributions of this work can be summarized as follows: a systematic methodology of identifying difficult-to-predict branches, dedicated predictors designed to improve the prediction accuracy of unbiased branches, random degree metrics developed to characterize the randomness of sequences produced by unbiased branches, and selective dynamic value prediction and instruction reuse methods integrated into superscalar and simultaneous multithreaded architectures. This chapter presents some quantitative and qualitative conclusions regarding the important experimental results obtained within this thesis and emphasizes some possible further work directions.

First, we have shown that unbiased branches are hard to predict if their outcomes, in the considered prediction contexts (branch address, local or global branch history, path), tend to chaotically shuffle between *taken* and *not taken*. We identified through laborious simulations these difficult-to-predict branches in the SPEC 2000 benchmarks, and partially solved them through context length extension. However, about 6% of branches could not be solved even with the longest evaluated correlation information (28 bits), their polarization degrees remaining still unacceptably low (less than 0.95). Despite some branches are path-correlated, a global branch history of more than 12 bits approximates very well the longer path information. Thus, the path is useful only in the case of short contexts, for longer contexts its gain being insignificant. In other words, a sufficiently long branch history might be viewed as a good "compression" of the most complete path information. We also concluded that current state-of-the-art branch predictors correlate either insufficient information or wrong information in the prediction of unbiased branches. Even one of the most effective predictors, the *idealized piecewise linear branch predictor* developed by Jiménez, only achieved a prediction accuracy of 77.3% on the unbiased branches, leading us to consider alternative approaches. Therefore, we improved several state-of-the-art branch predictors with additional prediction information. Thus, we developed and evaluated some PPM-based value predictors that are using a compressed branch condition history whose digits were -1, 0, or 1, depending on the sign of the difference between the operand values implied in each considered past branch. Unfortunately, even these idealistic predictors, able to exploit the correlation between branch outcome and branch condition history, could not improve the predictability of unbiased branches.

We have analyzed comparatively the percentages of unbiased branches obtained using the global history, the global history concatenated with the path, and the global history concatenated with a new prediction information, namely, the previous branch condition (PBC) represented as a 32-bit difference between the operand values of the previous dynamic branch. The evaluations showed that the previous branch condition is more efficient than the path information: it decreased the percentage of unbiased branches for all the evaluated context lengths. Therefore we additionally used local (per-address) or global PBC value, hashed together with the local/global branch history, integrated in some conventional branch predictors like the GAg and PAg, and in some state-of-the-art neural branch predictors. The *piecewise linear branch predictor* improved with the global PBC value was the most efficient, according to our evaluations. Nevertheless, even this powerful predictor achieved a modest 78.3% average prediction accuracy on the unbiased branches, whereas its global average prediction accuracy was 95.45% overcoming the original *piecewise linear branch predictor* (the best state of the art branch predictor) with 0.53%. However, this modified *piecewise linear branch predictor* significantly outperformed the modified GAg and PAg predictors. This gain was probably obtained because both the improved GAg and PAg predictors used a hashing between the PBC value and the global/local branch history, whereas the modified *piecewise linear branch predictor* used the branch history and PBC value without hashing (by concatenating them).

Other very powerful general predictors like our developed HMM, have predicted unbiased branches with an average accuracy of only 65.03%. Since the impact of unbiased branches significantly restricts the global accuracy, predicting them still represents a hard challenge for computer architects. This means that accurate prediction of unbiased branches remains an open problem and such branches will continue to limit the ceiling of dynamic branch prediction. Moreover, taking into account that these difficult branches are generated by very complex program structures, we expect that their negative influence will be even more significant in the future.

At this moment there is not a universally accepted paradigm for effectively defining random strings of symbols. Not surprisingly, understanding randomness is closely related with strong mathematical concepts like computability and algorithms, information theory and complexity, actual infinites theory, etc. The problem is therefore open and of great interests in many fields of science. We showed that unbiased branches could be understandable in more depth using this interdisciplinary methodological frame. We developed four metrics that are defining the random degree of a string of symbols. These metrics are based on: HMM-based predictability, discrete entropy, compression rate and Kolmogorov complexity associated to the code sequence that generates unbiased branches. The proposed random degree metrics could practically help the computer architect to better understand if a certain branch predictor should be improved. All these four developed metrics are converging at the same point. They are showing how much “intrinsic randomness” a string of symbols and, particularly, the sequences produced by unbiased branches contain. If some difficult-to-predict branches are not intrinsic random with our metrics, according to our experience, their prediction accuracy could be further improved by the researcher. Unfortunately, if these branches are intrinsic random, the answer is a pessimistic one, generating a strong limitation in Computer Architecture. Since the future applications complexity will increase (object oriented programs, design patterns, complex project management, virtual machines, etc.), we expect that also the number and therefore the influence of unbiased branches will further increase.

Our statistics show that about 28% of branches are dependent on long-latency instructions. Moreover, 5.61% of branches are unbiased and depend on long-latency instructions, too. These dependences involve high-penalty mispredictions becoming serious performance obstacles and causing significant performance degradation in executing instructions from wrong paths. Therefore, the negative impact of (unbiased) branches over global performance should be seriously attenuated by anticipating the results of long-latency instructions, including critical Loads. On the other hand, hiding long execution latencies in a pipelined superscalar processor represents an important challenge itself. Therefore, we developed a superscalar architecture that selectively anticipates the values produced by high-latency instructions. We have focused on Multiply, Division and Loads with miss in L1 data cache, implementing a Dynamic Instruction Reuse scheme for the Mul/Div instructions and a simple Last Value Predictor for the critical Load instructions. Our improved architecture achieved an average IPC speedup of 3.5% on the integer SPEC 2000 benchmarks, of 23.6% on the floating-point benchmarks, and an improvement in energy-delay product of 6.2% and 34.5%, respectively. Actually, this lower energy consumption shows the efficiency of our anticipatory techniques in a superscalar architecture. We have also demonstrated that there is a dynamic correlation between the names of the destination registers and the values stored in these registers. Therefore we extended dynamic value prediction by introducing the register-centric prediction concept instead of instruction-centric prediction. This register-centric approach is advantageous because fewer predictors are needed, thus reducing complexity and costs. We developed several different basic value predictors, such as the last value predictor, the stride value predictor, context-based predictors and hybrid value predictors to capture certain type of value predictabilities from the

SPECint95 and the SPECint2000 benchmarks. All these predictors were adapted to our proposed prediction model. The evaluations showed that the hybrid predictors have best exploited the value locality concept. Moreover, the hybrid predictor with counter-based adaptive prioritization composed of a two-level and a stride predictor outperformed the PPM-based hybrid predictor, at significantly lower implementation cost and complexity. Considering an 8-issue out-of-order superscalar processor, the register centric value prediction achieves average speedups of 17.30% on the SPECint95 benchmarks and 13.58% on the SPECint2000 benchmarks.

After we have shown the utility of selectively anticipating long-latency instructions in superscalar architectures, it was natural to analyze the efficiency of these methods in multithreaded environments. Thus, we have studied the impact of dynamic instruction reuse and value prediction, applied selectively on Mul/Div instructions and on critical Loads, in a Simultaneous Multithreaded (SMT) architecture. We implemented private Mul/Div Reuse Buffers (RB) and Load Value Prediction Tables (LVPT) for each thread. Our simulations performed on the SPEC 2000 benchmarks showed higher IPC on all evaluated SMT configurations, when the RB and LVPT structures were used. With fewer threads, the shared functional units are underused and therefore the selective instruction reuse and value prediction techniques have an important improvement potential. However, as the number of threads grows the IPC speedup decreases, because the shared functional units are better exploited due to the higher thread-level parallelism (TLP) and therefore the RB and LVPT structures become less important. We measured the highest IPC of 2.29 on the integer and 2.88 on the floating-point benchmarks with our six-threaded enhanced SMT architecture. However, the best improvements on the SPEC integer applications have been obtained with 2 threads: an IPC speedup of 5.95% and an EDP gain of 10.44%. Although, on the SPEC floating-point programs, we obtained the highest improvements with the enhanced superscalar architecture, the SMT with 3 threads also provides an important IPC speedup of 16.51% and an EDP gain of 25.94%. As a conclusion, applying some well-known anticipatory techniques selectively on long-latency instructions provides serious performance gain and significantly reduces energy consumption in superscalar and even in multithreaded architectures.

Finally, we highlight some interesting research topics that need to be further investigated in the future. Since accurate prediction of unbiased branches still remains an open problem, we consider that the use of more prediction contexts (some relevant high-level language information) is required to further improve prediction accuracies. Perhaps an alternative mechanism might be to hand-shake scheduler support with dynamic branch prediction. The idea of the scheduler would be to remove as many branch instructions (especially unbiased branches) from the static code as possible and leave the remaining branches to be dynamically predicted. Yet another alternative could be to pursue the concepts of micro-threading where small fragments of code (e.g. both branch paths) are executed concurrently and the branch problem is no longer a major concern. It would be also useful to quantify the unbiased branch ceiling in multicore architectures. Also, understanding and exploring instruction reuse and value prediction benefits in a multicore architecture might be another very important challenge.

"Experimental science is the queen of sciences
and the goal of all speculation"

Roger Bacon

References

[Aam03] Aamer M., Lux K., Mistry R., Mulholland B., *Efficiency of Pre-Computed Branches*, Technical Report, University of Pennsylvania, USA, 2003.

[Akk03a] Akkary H., Rajwar R., Srinivasan S.T., *Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors*, Proceedings of the 36th International Symposium on Microarchitecture, ACM Press, 2003.

[Akk03b] Akkary H., Rajwar R., Srinivasan S.T., *Checkpoint Processing and Recovery: An Efficient, Scalable Alternative to Reorder Buffers*, IEEE Micro, Vol. 23, No. 6, 2003.

[Ara01] Aragón J.L., González J., García J.M., González A., *Selective Branch Prediction Reversal by Correlating with Data Values and Control Flow*, Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors, 2001.

[Bar08] Barre J., Rochange C., Sainrat P., *A Predictable Simultaneous Multithreading Scheme for Hard Real-Time*, The 21st International Conference on Architecture of Computing Systems, TU Dresden, Germany, February 2008.

[Bau72] Baum L.E., *An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of Markov Processes*, Inequalities, Vol. 3, 1972.

[Bir01] Birney E., *Hidden Markov Models in Biological Sequence Analysis*, IBM Journal of Research and Development, Volume 45, Numbers 3/4, 2001.

[Bro00] Brooks D., Tiwari V., Martonosi M., *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*, Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, June 2000.

[Bur97] Burger D., Austin T., *The SimpleScalar Tool Set*, Version 2.0, (ftp://ftp.cs.wisc.edu/pub/sohi/Code/simplescalar), Technical Report, University of Wisconsin, Madison, USA, June 1997.

[Cal99] Calder B., Reinman G. and Tullsen D., *Selective Value Prediction*, Proceedings of the 26th International Symposium on Computer Architecture, pages 64-74, May 1999.

[CBP04] *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, http://www.jilp.org/cbp, 2004.

[CBP06] *The 2nd Journal of Instruction-Level Parallelism Championship Branch Prediction Competition* (CBP-2), Orlando, Florida, USA, (2006), http://camino.rutgers.edu/cbp2/.

[Cha94] Chang P.-Y., Hao E., Yeh T.-Y., Patt Y.N., *Branch Classification: a New Mechanism for Improving Branch Predictor Performance*, Proceedings of the 27th International Symposium on Microarchitecture, San Jose, California, 1994.

[Cha02a] Chang M.-C., Chou Y.-W., *Branch Prediction using Both Global and Local Branch History Information*, IEE Proceedings – Computer and Digital Techniques, Vol. 149, No. 2, United Kingdom, March 2002.

[Cha02b] Chappell R., Tseng F., Yoaz A., Patt Y., *Difficult-Path Branch Prediction Using Subordinate Microthreads*, The 29th Annual International Symposia on Computer Architecture, Alaska, USA, May 2002.

[Cha03] Chaver D., Pinuel L., Prieto M., Tirado F., Huang M., *Branch Prediction On Demand: An Energy-Efficient Solution*, Proceedings of the International Symposium on Low Power Electronics and Design, pages 390-395*,* Seoul, Korea, August 2003.

[Cha08] Chang S.C., Li W.Y.H., Kuo Y.J., Chung C.P., *Early Load: Hiding Load Latency in Deep Pipeline Processor*, Proceedings of the Asia-Pacific Computer Systems Architecture Conference, Taiwan, August 2008.

[Che03] Chen L., Dropsho S., Albonesi D.H., *Dynamic Data Dependence Tracking and its Application to Branch Prediction*, The 9th International Symposium on High-Performance Computer Architecture, February 2003.

[Cit02] Citron D., Feitelson D., *Revisiting Instruction Level Reuse*, Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD), May 2002.

[Con04] Constantinides K., Sazeides Y., *A Hardware-Based Method for Dynamically Detecting Instruction-Isomorphism and its Application to Branch Prediction*, The 2nd Value Prediction and Value-Based Optimization Workshop, Boston, Massachusetts, October 2004.

[Cor01] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Introduction to Algorithms*, Section 16.3, pages 385–392, Second Edition, MIT Press and McGraw-Hill, 2001

[Cri02] Cristal A., Valero M., Gonzalez A., Llosa J., *Large Virtual ROBs by Processor Checkpointing*, Technical Report, Computer Architecture Department, University Politècnica of Catalunya, Barcelona, Spain, 2002.

[Cri04a] Cristal A., Ortega D., Llosa J., Valero M., *Out-of-order Commit Processors*, Proceedings of the 10th International Symposium on High Performance Computer Architecture, February 2004.

[Cri04b] Cristal A., Santana O., Valero M., *Towards Kilo-instruction Processors*, ACM Transactions on Architecture and Code Optimization, Vol. 1, No. 4, December 2004.

[Cri05] Cristal A., Santana O., Cazorla F., Galluzzi M., Ramírez T., Pericàs M., Valero M., *Kilo-instruction Processors: Overcoming the Memory Wall*, IEEE Micro, Vol. 25, No. 3, 2005.

[Des02] Desmet V., Goeman B., Bosschere K., *Independent Hashing as Confidence Mechanism for Value Predictors in Microprocessors*, Proceedings of the 8th International EuroPar Conference on Parallel Processing, Augsburg, Germany, August 2002.

[Des04] Desmet V., Eeckhout L., De Bosschere K., *Evaluation of the Gini-index for Studying Branch Prediction Features.* Proceedings of the 6th International Conference on Computing Anticipatory Systems (CASYS), AIP Conference Proceedings, Vol. 718, 2004.

[Des06] Desmet V., *On the Systematic Design of Cost-Effective Branch Prediction*, PhD Thesis, Ghent University, Belgium, 2006.

[Deu96] Deutsch P., *DEFLATE Compressed Data Format Specification version 1.3,* Aladdin Enterprises, Network Working Group, RFC 1951, pages 1-15, 1996.

[Ega03] Egan C., Steven G., Quick P., Anguera R., Vintan L., *Two-Level Branch Prediction using Neural Networks*, Journal of Systems Architecture, Vol. 49, Elsevier, December 2003.

[Egg97] Eggers S. Emer J., Levy H., Lo J., Stamm R., Tullsen D., *Simultaneous Multithreading: A Platform for Next-Generation Processors*, IEEE Micro, Vol 17, Issue 5, September 1997.

[Fal04] Falcón A., Stark J., Ramirez A., Lai K., Valero M., *Prophet/Critic Hybrid Branch Prediction*, Proceedings of the 31st Annual International Symposium on Computer Architecture, München, Germany, June 2004.

[Fer04] Fern A., Givan R., Falsafi B.,Vijaykumar T.N., *Dynamic Feature Selection for Hardware Prediction*, Journal of Systems Architecture, Vol. XX, Elsevier, 2004.

[Flo02] Florea A., Vintan L., Sima D., *Understanding Value Prediction through Complex Simulations*, Proceedings of the 5th International Conference on Technical Informatics, University "Politehnica" of Timisoara, Romania, October, 2002.

[Flo04] Florea A., Vintan L., Mihu Z.I., *Understanding and Predicting Indirect Branch Behavior,* Studies in Informatics and Control, Vol.13, No. 1, National Institute for Research and Development in Informatics, Bucharest, March 2004.

[Flo05a] Florea A., *The dynamic values prediction in the next generation microprocessors*, MatrixRom Publishing House, Bucharest, 2005.

[Flo05b] Florea A., Vintan L., *Advanced techniques for improving indirect branch prediction accuracy*, Proceedings of 19th European Conference on Modelling and Simulation, Riga, Latvia, June 2005.

[Flo06] Florea A., **Gellert A.**, *Memory Wall — A Critical Factor in Current High-Performance Microprocessors*, Science and Supercomputing in Europe, ISBN 978-88-86037-19-8, Barcelona, Spain, 2006.

[Flo07a] Florea A., Radu C., Calborean H., Crapciu A., **Gellert A.**, Vintan L., *Designing an Advanced Simulator for Unbiased Branches Prediction*, Proceedings of 9th International Symposium on Automatic Control and Computer Science, ISSN 1843-665X, Iasi, 2007.

[Flo07b] Florea A., Radu C., Calborean H., Crapciu A., **Gellert A.**, Vintan L., *Understanding and Predicting Unbiased Branches in General-Purpose Applications*, Bulletin of the Polytechnic Institute of Iasi, Tom LIII (LVII), Fasc. 1-4, Section IV, ISSN 1220-2169, 2007.

[Gab98] Gabbay F., Mendelsohn A., *Using Value Prediction To Increase The Power Of Speculative Execution Hardware*, ACM Transactions On Computer Systems, Vol 16, Nr. 3, 1998.

[Gam99] Gammerman A., Vovk V., *Kolmogorov Complexity: Sources, Theories and Applications*, The Computer Journal, Vol.42, No. 4, pages 252-255, 1999.

[Gao06] Gao H., Zhou H., *PMPM: Prediction by Combining Multiple Partial Matches*, The 2nd Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2), Orlando, Florida, USA, December 2006.

[Gao08] Gao H., Ma Y., Dimitrov M., Zhou H., *Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches*, Proceedings of the 14th International Symposium on High-Performance Computer Architecture, Salt Lake City, Utah, February 2008.

[Gel03] **Gellert A.**, *Contributions to speculative execution of instructions by dynamic register value prediction*, MSc Thesis, University "Lucian Blaga" of Sibiu, Computer Science Department, 2003 (in Romanian, supervisor Prof. L. Vintan).

[Gel06a] **Gellert A.**, *Prediction Methods Integrated into Advanced Architectures*, 1st PhD Report, Computer Science Department, "Lucian Blaga" University of Sibiu, January 2006.

[Gel06b] **Gellert A.**, Florea A., *Finding and Solving Difficult Predictable Branches*, Science and Supercomputing in Europe, ISBN 978-88-86037-19-8, Barcelona, Spain, 2006.

[Gel06c] **Gellert A.**, Vintan L., *Person Movement Prediction Using Hidden Markov Models*, Studies in Informatics and Control, Vol. 15, No. 1, ISSN 1220-1766 (**IEE INSPEC**), National Institute for Research and Development in Informatics, Bucharest, March 2006.

[Gel07a] **Gellert A.**, *Integration of Some Advanced Prediction Methods into Speculative Computing Systems*, 2nd PhD Report, Computer Science Department, "Lucian Blaga" University of Sibiu, March 2007.

[Gel07b] **Gellert A.**, Florea A., Vintan M., Egan C., Vintan L., *Unbiased Branches: An Open Problem*, Twelfth Asia-Pacific Computer Systems Architecture Conference (ACSAC'07), Seoul, Korea, August 2007; Lecture Notes in Computer Science, Advances in Computer Systems Architecture, vol. 4697, pp. 16-27, ISSN 0302-9743 (Print) 1611-3349 (Online), Springer-Verlag Berlin / Heidelberg, 2007 (**ISI Thomson Journals**).

[Gel07c] **Gellert A.**, Vintan L., Florea A., *A Systematic Approach to Predict Unbiased Branches*, ISBN 978-973-739-516-0, "Lucian Blaga" University Press, Sibiu, Romania, 2007 (http://webspace.ulbsibiu.ro/arpad.gellert/html/Unb_Br_Book.pdf).

[Gel08a] **Gellert A.**, *Developing and Improving the Performances of Some Predictive Architectures*, 3rd PhD Report, Computer Science Department, "Lucian Blaga" University of Sibiu, April 2008.

[Gel08b] **Gellert A.**, Florea A., Vintan L., *Exploiting Selective Instruction Reuse and Value Prediction in a Superscalar Architecture*, Journal of Systems Architecture, ISSN 1383-7621, 2008 (**ISI Thomson Journals**).

[Gol07] Golander A., Weiss S., *Reexecution and Selective Reuse in Checkpoint Processors*, HiPEAC Journal, 2007.

[Gon96] Gonzalez R., Horowitz M., *Energy Dissipation in General Purpose Microprocessors*, IEEE Journal of Solid State Circuits, Vol. 31, No. 9, September 1996.

[Gon99] González J., González A., *Control-Flow Speculation through Value Prediction for Superscalar Processors*, International Conference on Parallel Architecture and Compilation Techniques, 1999.

[Gon01] González J., González A., *Control-Flow Speculation through Value Prediction*, IEEE Transactions on Computers, Vol. 50, No. 12, December 2001.

[Gzip] http://www.gzip.org/.

[Hei99a] Heil T., Smith Z., Smith J.E., *Using Data Values to Predict Branches*, Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999.

[Hei99b] Heil T.H., Smith Z., Smith J.E., *Improving Branch Predictors by Correlating on Data Values*, Proceedings of the 32nd International Symposium on Microarchitecture, November 1999.

[Hen03] Hennessy J., Patterson D., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Third Edition, 2003.

[Hun03] Hunt S.P., Egan C., Shafarenko A., *A Simple Yet Accurate Neural Branch Predictor*, Proceedings of the IASTED International Conference on Artificial Intelligence and Application (AIA), Malaga, Spain, September 2003.

[Jim01a] Jiménez D., Lin C., *Dynamic Branch Prediction with Perceptrons*, In Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7), January 2001.

[Jim01b] Jiménez D., Lin C., *Perceptron Learning for Predicting the Behavior of Conditional Branches*, Proceedings of the INNS-IEEE International Joint Conference on Neural Networks (IJCNN), Washington DC, July 2001.

[Jim02] Jiménez D., Lin C., *Neural Methods for Dynamic Branch Prediction*, ACM Transactions on Computer Systems, Vol. 20, New York, USA, November 2002.

[Jim03a] Jiménez D., Lin C., *Dynamic Branch Prediction with Perceptrons*, Proceedings of the 7th International Symposium on High Performance Computer Architecture, January 2001.

[Jim03b] Jiménez D., *Reconsidering Complex Branch Predictors*, Proceedings of the 9th International Symposium on High Performance Computer Architecture, February 2003.

[Jim03c] Jiménez D., *Fast Path-Based Neural Branch Prediction*, Proceedings of the 36th Annual International Symposium on Microarchitecture, December 2003.

[Jim04] Jiménez D., *Idealized Piecewise Linear Branch Prediction*, Championship Branch Prediction (CBP-1), 2004, http://www.jilp.org/cbp/Agenda-and-Results.htm.

[Jim05] Jiménez D., *Idealized Piecewise Linear Branch Prediction*, Journal of Instruction-Level Parallelism, April 2005.

[Jos97] Joseph D., Grunwald D., *Prefetching using Markov Predictors*, Proceedings of the 24th International Symposium on Computer Architecture, pages 252-263, June 1997.

[Ken07] Kennedy M., *Design of Double Precision IEEE-754 Floating-Point Units*, MSc Thesis, Griffith University, March 2007.

[Kim03] Kim S., *Branch Prediction using Advanced Neural Methods*, Technical Report, University of California, Berkeley, 2003.

[Kim07] Kim H., Joao J., Mutlu O., Lee C.J., Patt Y.N., Cohn R., *VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization*, Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA07), San Diego, CA, June 2007.

[Kol65] Kolmogorov A.N., *Three Approaches to the Quantitative Definition of Information*, Problems of Information Transmission, 1965.

[Lep00a] Lepak K.M., Lipasti M.H., *On the Value Locality of Store Instructions*, Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, June 2000.

[Lep00b] Lepak K.M., Lipasti M.H., *Silent Stores for Free*, Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO33), California, USA, 2000.

[Lia02] Liao C.H., Shieh J.J., *Exploiting Speculative Value Reuse Using Value Prediction*, Seventh Asia-Pacific Computer Systems Architecture Conference, Melbourne, Australia, February 2002.

[Lip96a] Lipasti M.H., Wilkerson C.B., Shen J.P., *Value Locality and Load Value Prediction*, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 138-147, October 1996.

[Lip96b] Lipasti M. H., Shen J.P., *Exceeding the Dataflow Limit via Value Prediction*, Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, December 1996.

[Liu03] Liu N., Lovell B.C., *Gesture Classification Using Hidden Markov Models and Viterbi Path Counting*, Proceedings of the Seventh International Conference on Digital Image Computing: Techniques and Applications, Sydney, Australia, December 2003.

[Liu08] Liu C., Gaudiot J.L., *Resource Sharing Control in Simultaneous MultiThreading Microarchitectures*, Proceedings of the Asia-Pacific Computer Systems Architecture Conference, Taiwan, August 2008.

[Loh05a] Loh G.H., *Deconstructing the Frankenpredictor for Implementable Branch Predictors*, Journal of Instruction-Level Parallelism, April 2005.

[Loh05b] Loh G.H., Jiménez D., *A Simple Divide-and-Conquer Approach for Neural-Class Branch Prediction*, Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT), St. Louis, MO, USA, September 2005.

[Loh05c] Loh G.H., Jiménez D., *Reducing the Power and Complexity of Path-Based Neural Branch Prediction*, 5th Workshop on Complexity Effective Design (WCED5), Madison, WI, USA, June 2005.

[Mah94] Mahlke S.A., Hank R.E., Bringmann R.A., Gyllenhaal J.C., Gallagher D.M., Hwu W.-M.W., *Characterizing the Impact of Predicated Execution on Branch Prediction*, Proceedings of the 27th International Symposium on Microarchitecture, San Jose, California, December 1994.

[Mar99] Marcuello P., Tubella J., González A., *Value Prediction for Speculative Mutithreaded Architectures*, Proceedings of the 32nd International Symposium on Microarchitecture, November 1999.

[Mar01] Martin M., Sorin D., Cain H., Hill M., Lipasti M., *Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing*, Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, Austin, Texas, December 2001.

[McFar93] McFarling S., *Combining Branch Predictors*, WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[Mit97] Mitchell T., *Machine Learning*, McGraw-Hill, 1997.

[Mud96] Mudge T.N., Chen I.K., Coffey J.T., *Limits to Branch Prediction*, Technical Report, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, Michigan, USA, January 1996.

[Mut03] Mutlu O., Stark J., Wilkerson C., Patt Y.N., *Runahead Execution: An Effective Alternative to Large Instruction Windows*, IEEE Micro, Vol. 23, No. 6, 2003.

[Mut06] Mutlu O., Kim H., Patt Y.N., *Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses*, IEEE Transactions on Computers, Vol. 55, No. 12, December 2006.

[Nair95] Nair R., *Dynamic Path-Based Branch Correlation*, IEEE Proceedings of MICRO-28, 1995.

[Oan06] Oancea M, **Gellert A.**, Florea A., Vintan L., *Analyzing Branch Prediction Contexts Influence*, Advanced Computer Architecture and Compilation for Embedded Systems, (ACACES 2006), ISBN 90 382 0981 9, pages 5-8, L'Aquila, Italy, July 2006.

[Pan92] Pan S., So K., Rahmeh J.T., *Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation*, ASPLOS-V International Conference, Boston, October 1992.

[Per06] Pericàs M., Cristal A., González R., Jiménez D., Valero M., *A Decoupled Kilo-Instruction Processor*, Proceedings of the 12th International Symposium on High Performance Computer Architecture, February 2006.

[Per07] Pericàs M., Cristal A., Cazorla F., González R., Jiménez D., Valero M., *A Flexible Heterogeneous Multi-Core Architecture*, Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques, Brasov, Romania, September 2007.

[Pet04] Petzold J., *Augsburg Indoor Location Tracking Benchmarks*, Technical Report 2004-9, Institute of Computer Science, University of Augsburg, Germany, 2004, http://www.informatik.uni-augsburg.de/skripts/techreports/.

[Rab89] Rabiner L.R., *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, Proceedings of the IEEE, Vol 77, No. 2, February 1989.

[Rad07] Radu C., Calborean H., Crapciu A., **Gellert A.**, Florea A., *An Interactive Graphical Trace-Driven Simulator for Teaching Branch Prediction in Computer Architecture*, The 6th EuroSim Congress on Modeling and Simulation, 2007, Ljubljana, Slovenia.

[Ram03] Ramsay M., Feucht C., Lipasti M., *Exploring Efficient SMT Branch Predictor Design*, Workshop on Complexity Effective Design, 2003.

[Ram08] Ramírez T., Pajuelo A., Santana O., Valero M., *Runahead Threads to Improve SMT Performance*, Proceedings of the International Symposium on High Performance Computer Architecture, 2008.

[Red03] Redstone J., Eggers S., Levy H., *Mini-threads: Increasing TLP on Small-Scale SMT Processors*, Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9), 2003.

[Ric93] Richardson S., *Exploiting trivial and redundant computation*, Proceedings of the 11th Symposium on Computer Arithmetic, July 1993.

[Rot99] Roth A., Moshovos A., Sohi G., *Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation*, Proceedings of the International Conference on Supercomputing, 1999.

[Ryc98] Rychlik B., Faistl J., Krug B., Kurland A., Jung J., Velev M. and Shen J., *Efficient and Accurate Value Prediction Using Dynamic Classification*, Technical Report, Department of Electrical and Computer Engineering, Carnegie Mellon Univ., 1998.

[Saz97] Sazeides Y., Smith J.E., *The Predictability of Data* Values, Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997.

[Saz99] Sazeides Y., *An analysis of value predictability and its application to a superscalar processor*, PhD Thesis, University of Wisconsin-Madison, 1999.

[Sen04] Seng J.S., Hamerly G., *Exploring Perceptron-Based Register Value Prediction*, The 2nd Value-Prediction and Value-Based Optimization Workshop (in conjunction with ASPLOS 11 Conference), Boston, USA, 2004.

[Sez02] Seznec A., Felix S., Krishnan V., Sazeides Y., *Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor*, Proceedings of the 29th International Symposium on Computer Architecture, Anchorage, AK, USA, May 2002.

[Sez04] Seznec A., *Revisiting the Perceptron Predictor*, Technical Report, IRISA, May 2004.

[Sez05] Seznec A., *Genesis of the O-GEHL branch predictor*, Journal of Instruction-Level Parallelism, April 2005.

[Sez07a] Seznec A., *The Idealistic GTL Predictor*, Journal of Instruction-Level Parallelism, No. 9, May, 2007.

[Sez07b] Seznec A., *The L-TAGE Branch Predictor*, Journal of Instruction-Level Parallelism, No. 9, May, 2007.

[Sha05] Sharkey J., Ponomarev D., Ghose K., *M-SIM: A Flexible, Multithreaded Architectural Simulation Environment*, Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, October 2005.

[She03] Shen J.P., Lipasti M.H., *Modern Processor Design. Fundamental of Superscalar Processors*, Beta Edition, McGraw-Hill Co, 2003.

[Shi01] Shivakumar P., Jouppi N.P., *Cacti 3.0: An Integrated Timing, Power, and Area Model*, WRL Research Report, Aug 2001, USA.

[Sim] *The SimpleSim Tool Set*, ftp://ftp.cs.wisc.edu/pub/sohi/Code/simplescalar.

[Smi95] Smith J., Sohi G., *The Microarchitecture of Superscalar Processors*, Proceedings of the IEEE, Vol. 83, December 1995.

[Smi98] Smith Z., *Using Data Values to Aid Branch-Prediction*, MSc Thesis, Wisconsin-Madison, USA, December 1998.

[Sod97] Sodani A., Sohi G., *Dynamic Instruction Reuse*, Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97), Denver, 1997.

[Sod00] Sodani A., *Dynamic Instruction Reuse*, PhD Thesis, University of Wisconsin-Madison, USA, 2000.

[SPEC] SPEC 2000, *The SPEC benchmark programs*, http://www.spec.org.

[Spr02] Sprangle E., Carmean D., *Increasing Processor Performance by Implementing Deeper Pipelines*, Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, Alaska, May 2002.

[Sri06] Srinivasan R., Frachtenberg E., Lubeck O., Pakin S., Cook J., *Neuro-PPM Branch Prediction*, The 2nd Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2), Orlando, Florida, USA, December 2006.

[Sta04] Stamp M., *A Revealing Introduction to Hidden Markov Models*, January 2004, http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf.

[Ste96] Steven G., Collins R., *A Superscalar Architecture to Exploit Instruction Level Parallelism*, Proceedings of the Euromicro Conference, Prague, 1996.

[Ste01] Steven G., Egan C., Anguera R., Vintan L., *Dynamic Branch Prediction using Neural Networks*, Proceedings of International Euromicro Conference DSD '2001, pages 178-185, Warsaw, Poland, September 2001.

[Sub08] Subramaniam S., Prvulovic M., Loh G., *PEEP: Exploiting Predictability of Memory Dependences in SMT Processors*, International Symposium on High Performance Computer Architecture 2008.

[Tar05] Tarjan D., Skadron K., *Merging Path and GshareIndexing in Perceptron Branch Prediction*, ACM Transactions on Architecture and Code Optimization, Vol. 2, No. 3, September 2005.

[Tho01] Thomas R., Franklin M., *Using Dataflow Based Context for Accurate Value Prediction*, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2001.

[Tho03] Thomas R., Franklin M., Wilkerson C., Stark J., *Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History*, Proceedings of the 30th International Symposium on Computer Architecture, June 2003.

[Tho04] Thomas A., Kaeli D., *Value Prediction with Perceptrons*, The Second Value Prediction and Value-Based Optimization Workshop, Boston, USA, October 2004.

[Tom67] Tomasulo R., *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal, Vol. 11, 1967.

[Tul99] Tullsen D.M., Seng J.S., *Storageless Value Prediction using Prior Register Values*, Proceedings of the 26th International Symposium on Computer Architecture, May 1999.

[Ung02] Ungerer T., Robic B., Silc J., *Multithreaded Processors*, The Computer Journal, Vol. 45, No. 3, 2002.

[Ung03] Ungerer T., Robic B., Silc J., *A Survey of Processors with Explicit Multithreading*, ACM Computing Surveys, Vol. 35, No. 1, March 2003.

[Vin99a] Vintan L., Iridon M., *Towards a High Performance Neural Branch Predictor*, Proceedings of the International Joint Conference on Neural Networks, Washington DC, USA, July 1999.

[Vin99b] Vintan L., Egan C., *Extending Correlation in Branch Prediction Schemes*, International Euromicro'99 Conference, Milano, Italy, September 1999.

[Vin00a] Vintan L., *Instruction Level Parallel Architectures* (in Romanian), Romanian Academy Publishing House, Bucharest, 2000.

[Vin00b] Vintan L., *Towards a Powerful Dynamic Branch Predictor*, Romanian Journal of Information Science and Technology, Romanian Academy Publishing House, Bucharest, 2000.

[Vin03] Vintan L., Sbera M., Mihu I.Z., Florea A., *An Alternative to Branch Prediction: Pre-Computed Branches*, ACM SIGARCH Computer Architecture News, Vol.31, Issue 3, ACM Press, NY, USA, June 2003.

[Vin04a] Vintan L., **Gellert A.**, Florea A., *Register value prediction using metapredictors*, Proceedings of the 8th International Symposium on Automatic Control and Computer Science, Iasi, October 2004.

[Vin04b] Vintan L., **Gellert A.**, Petzold J., Ungerer T., *Person movement prediction using neural networks*, Technical Report 2004-10, Institute of Computer Science, University of Augsburg, Germany, April 2004, (http://www.informatik.uniaugsburg.de/skripts/techreports/)

[Vin04c] Vintan L., **Gellert A.**, Petzold J., Ungerer T., *Person Movement Prediction Using Neural Networks*, Proceedings of the KI2004 International Workshop on Modeling and Retrieval of Context (MRC 2004), Vol-114, ISSN 1613-0073, Ulm, Germany, September 2004.

[Vin05a] Vintan L., Florea A., **Gellert A.**, *Focalising Dynamic Value Prediction to CPU's Context*, IEE Proceedings. Computers & Digital Techniques, Vol. 152, No. 4, Stevenage, UK, July 2005 (**ISI Thomson Journals**).

[Vin05b] Vintan L., **Gellert A.**, Florea A., *Value prediction focalized on CPU registers*, Advanced Computer Architecture and Compilation for Embedded Systems, (ACACES 2005), Academia Press, ISBN 90 382 0802 2, pages 181-184, Ghent, Belgium, July 2005.

[Vin06] Vintan L., **Gellert A.**, Florea A., Oancea M., Egan C., *Understanding Prediction Limits through Unbiased Branches*, Eleventh Asia-Pacific Computer Systems Architecture Conference (ACSAC'06), Shanghai, China, September 2006; Lecture Notes in Computer Science, Advances in Computer Systems Architecture, vol. 4186, pp. 480-487, ISSN 0302-9743, ISBN-13 978-3-540-40056, Springer-Verlag Berlin / Heidelberg, 2006 (**ISI Thomson Journals**).

[Vin07] Vintan L., *Prediction Techniques in Advanced Computing Architectures* (in English), MatrixRom Publishing House, Bucharest, 2007.

[Vin08a] Vintan L., Florea A., **Gellert A.**, *Forcing Some Architectural Ceilings of the Actual Processor Paradigm*, Invited Paper, The 3rd Conference of The Academy of Technical Sciences from Romania (ASTR), Cluj-Napoca, November 2008.

[Vin08b] Vintan L., Florea A., **Gellert A.**, *Random Degrees of Unbiased Branches*, Proceedings of the Romanian Academy, Series A, No. 3, 2008 (**ISI Thomson Journals**).

[Vol02] Volchan S.B., *What Is a Random Sequence?,* The American Mathematical Monthly, 109, January 2002.

[Wan97] Wang K., Franklin M., *Highly Accurate Data Value Prediction using Hybrid Predictors*, Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December 1997.

[Wan99] Wang Y., Lee S., and Yew P. *Decoupling Value Prediction on Trace Processors*, Proceedings of the 6th International Symposium on High performance Computer Architecture, 1999.

[Yeh92] Yeh T.-Y., Patt Y.N., *Alternative Implementations of Two-Level Adaptive Branch Prediction*, Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

[Yeh93] Yeh T.-Y., Patt Y.N., *A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History*, Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, California, May 1993.

[Yi06] Yi J.J., Lilja D.J., *Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies and Recommendations*, IEEE Transactions on Computers, Vol. 55, No. 3, pages 268-280, March 2006.

[Yok08] Yokota T., Ootsu K., Baba T., *Potentials of Branch Predictors – from Entropy Viewpoints*, Proceedings of the 21st International Conference on Architecture of Computing Systems, TU Dresden, Germany, February 2008.

[Yoo04] Yoon B., Vaidynathan P.P., *RNA Secondary Structure Prediction Using Context-Sensitive Hidden Markov Models*, Proceedings of International Workshop on Biomedical Circuits and Systems, Singapore, December 2004.

[Zho03] Zhou H., Flanagan J., Conte T., *Detecting Global Stride Locality in Value Streams*, Proceedings of the 30th Annual International Symposium on Computer Architecture, San Diego, California, June 2003.

[Ziv77] Ziv J., Lempel A., *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, Vol. IT-23, No. 3, pages 337-343, 1977.

“The Fast drives out the Slow even if the Fast is wrong”

William Kahan

Glossary

Benchmark: is a program (execution driven simulation) or a program’s trace (trace driven simulation) used for evaluations. In this work we used the SPEC 2000 benchmark suite, the SPEC JVM98 benchmarks, Stanford benchmarks, and the CBP-1 traces.

Basic Block: sequence of instructions that occur between two consecutive branches and are not the targets of branch instructions.

Biased branch: mostly always taken or mostly always not taken branch (mostly-one-direction branch). The behavior (taken / not taken) of a biased branch is polarized.

Biased branch context: the branch behavior (taken / not taken) is polarized for that certain *context* (*local branch history*, *global history*, *path*, etc.).

Branch difference: represents the value or the sign of the difference between the branch’s inputs. Regarding the sign of the inputs’ difference, a value of 1 indicates that the corresponding branch difference is positive, a value of -1 indicates a negative difference, while a 0 indicates equality between the branch’s inputs.

Branch difference predictor: the branch outcomes are predicted based on *branch difference* histories.

Branch polarization: measured through the *polarization index* (P).

Branch prediction: is the prediction of the direction (taken / not taken) and/or the target address (next PC) of a branch instruction.

Checkpointing architecture: allows speculative execution by saving or checkpointing the state of the processor at certain points in a history buffer or a checkpoint, respectively.

Chip-level multiprocessor (CMP): see *multicore architecture*.

Complete-PPM predictor: see *Prediction by Partial Matching (PPM)*.

Compression rate: a commonly used metric in data compression, representing the uncompressed size divided to the compressed size, as follows.

$$Compression\,Rate = \frac{Uncompressed\ Size}{Compressed\ Size} \cdot 100\%$$

Confidence automaton: saturated counter that indicates the confidence of a certain prediction. The prediction is generated only if the confidence automaton is in a predictable state.

Context: the context of length *p* represents the last *p* elements from the correlation information used in order to make a prediction. In the case of branch prediction the correlation information is the branch history (e.g. local or global branch history) or the path leading to the branch, and a context of length *p* consists in the last *p* bits from the branch history or in the last *p* PCs from the path.

Context instance: is a *dynamic branch* executed in the respective *context*.

Critical Load: a Load instruction with miss in both cache levels.

Distribution (index): the distribution index of a certain branch *context* is computed as follows.

$$D(S_i) = \begin{cases} 0, & n_t = 0 \\ \dfrac{n_t}{2 \cdot \min(NT,\, T)}, & n_t > 0 \end{cases} \quad \text{, where}$$

- n_t = the number of branch outcome transitions, from taken to not taken and vice-versa, in context S_i;
- $2 \cdot \min(NT,\, T)$ = maximum number of possible transitions;

- k = number of distinct contexts, $k \leq 2^p$, where p is the length of the binary context;
- if $D(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is "contradictory" (the most unfavorable case), and thus its learning is impossible;
- if $D(S_i) = 0, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is constant (the most favorable case), and it can be learned.

Dynamic branch: is an instance of a *static branch* during program's execution.

Dynamic branch prediction: the branches are predicted with hardware techniques.

Dynamic learning: is the run-time prediction process when the outputs of the predictor are used to generate predictions and to adjust the prediction structures.

Dynamic power consumption: see *power consumption*.

Energy-Delay Product (EDP): a widely used metric, representing the processor's total power, divided by the squared IPC, as follows:

$$EDP = \frac{Total\ Power}{IPC^2}$$

Entropy: considering a sequence S of symbols belonging to the set X={X$_1$X$_2$ ... X$_k$}, the entropy of S is $E(S) = -\sum_{i=1}^{k} P(Xi) \log_2 P(Xi) \geq 0$. Obviously its maximum ($\log_2 k$) is obtained for symbols of equal probabilities in S.

Feature (set): is the binary *context* on p bits of prediction information such as local history, global history or path. Each *static branch* finally has associated k dynamic contexts in which it can appear ($k \leq 2^p$).

Gain: is the factor which gives the improvement of the quality, with a certain metric.

Global branch history (GH): the outcome sequence (taken / not taken) generated by the previous *dynamic branches*.

Hidden Markov Model (HMM): is a doubly embedded stochastic process with an underlying stochastic process that is not observable (it is hidden), but it can be observed through another set of stochastic processes that produce the sequence of observations.

Hidden super-state: see *super-state*.

Hidden state: is a state in a *Hidden Markov Model (HMM)*.

Instruction-level parallelism (ILP): is a measure of how many instructions can be processed simultaneously in multiple instruction issue (MII) microarchitectures.

Kolmogorov complexity: a sequence X has Kolmogorov complexity $K(X)$ equal to the length of the shortest program p for a universal Turing Machine U that produces X and then halts:

$$K(X) = \min_{p:U(p)=X} l(p),$$

where $l(p)$ is the length of p in bits. Kolmogorov complexity identifies a sequence X as random if $l(X) - K(X)$ is small: random sequences are those that are irreducibly complex.

Local branch history (LH): the outcome sequence (taken / not taken) generated by the previous dynamic instances of a certain *static branch* instruction.

Markov chain: in the case of a first order Markov chain the probabilistic description is truncated to just the current and predecessor state.

$P[q_t = S_j | q_{t-1} = S_i,\ q_{t-2} = S_k,\ ...] = P[q_t = S_j | q_{t-1} = S_i]$, where q_t is the state at time t. Thus, for a first order Markov chain with N states, the set of transition probabilities between states S_i and S_j is $A = \{a_{ij}\}$, where $a_{ij} = P[q_t = S_j | q_{t-1} = S_i]$, $1 \leq i, j \leq N$, having the properties $a_{ij} \geq 0$ and $\sum_{j=1}^{N} a_{ij} = 1$. For a Markov chain of order R the probabilistic description is truncated to the current and R previous states.

Markov predictor: the prediction is generated based on the state transition probabilities of a *Markov chain*.

Memory wall: is the continuously increasing gap between processor and memory speeds. The *memory wall* produces a serious performance limitation of high-frequency microprocessors by main memory access latencies.

Multicore architecture: combines two or more independent cores into a die, or more dies packaged together.

Multithreaded processor: is a microarchitecture that exploits *thread-level parallelism (TLP)*, by executing instructions from multiple threads simultaneously or concurrently.

Observable state: an observation produced by the stochastic process of the corresponding hidden state in a *Hidden Markov Model*.

Path: is a prediction information consisting in the sequence of branch PCs or target PCs leading up to a certain *dynamic branch* instruction. The path can include all branch instruction types or exclusively conditional branches.

Path-based correlation: means using the *path* information leading up to a certain *dynamic branch* in order to determine (predict) the outcome of that branch.

Pattern-based correlation: means using branch outcome history (e.g. *local branch history*, *global branch history*) in order to determine (predict) the outcome of a certain *dynamic branch*.

Polarization (index): the polarization index (P) of a certain branch *context* is computed as follows.

$$P(S_i) = \max(f_0, f_1) = \begin{cases} f_0, & f_0 \geq 0.5 \\ f_1, & f_0 < 0.5 \end{cases} \quad \text{, where}$$

- $S = \{S_1, S_2, ..., S_k\}$ = set of distinct contexts that appear during all branch instances;
- k = number of distinct contexts, $k \leq 2^p$, where p is the length of the binary context;
- $f_0 = \frac{T}{T+NT}$, $f_1 = \frac{NT}{T+NT}$, NT = number of "not taken" branch instances corresponding to context S_i, T = number of "taken" branch instances corresponding to context S_i, $(\forall) i = 1, 2, ..., k$, and obviously $f_0 + f_1 = 1$;
- if $P(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the context S_i is completely biased (100%), and thus, the branch is highly predictable;
- if $P(S_i) = 0.5, (\forall) i = 1, 2, ..., k$, then the context S_i is totally unbiased, and thus, the branch is not predictable if the taken and not taken outcomes are shuffled.

Power Consumption: the dynamic power consumption is the main source of power dissipation in CMOS microprocessors and it is defined as $P_d = C \cdot V_{dd}^2 \cdot a \cdot f$ **,** where *C* is the capacitance, V_{dd} is the supply voltage, *f* is the clock frequency, and the activity factor *a* indicates how often clock ticks lead to switching activity on average.

Prediction accuracy: the percentage or ratio of correct predictions reported to the total number of predictions.

Prediction by Partial Matching (PPM): is a context-based prediction algorithm. The PPM predictor contains a set of simple Markov predictors. It predicts the value that followed the context with the highest frequency. In the case of complete-PPM predictor, if a prediction cannot be generated with the Markov predictor of order *k*, then the pattern length is shortened and the Markov predictor of order *k-1* tries to predict and so on.

Previous branch condition (PBC): the difference between the operand values implied in the previous branch condition. The global PBC is the previous branch condition difference. The local PBC is the previous per-address branch condition difference.

Primitive (hidden) state: is a (hidden) state in a first order *Hidden Markov Model*.

Program counter (PC): is a register in the processor (also called instruction pointer) which indicates the memory address of the next fetched instruction within a program.

Simultaneous multithreading (SMT): uses the resources of a multiple-issue processor to exploit both *thread-level parallelism (TLP)* and *instruction-level parallelism (ILP)*. Instructions from multiple threads are issued simultaneously in a single clock cycle, and thus, the available resources are better utilized.

Space savings: a commonly used data compression metric, computed as follows.

$$Space\,Savings = \left(1 - \frac{Compressed\ Size}{Uncompressed\ Size}\right) \cdot 100\%$$

Speculative architecture: is a microprocessor that allows *speculative execution* to reduce the execution time of conditional branches and long-latency instructions by predicting their results.

Speculative execution: instruction execution based on predicted values or predicted branch outcomes.

Static branch: a certain branch instruction from a program.

Static branch prediction: the branches are predicted statically by the compiler. Static branch predictors are used in processors where the expectation is that branch behavior is highly predictable at compile-time. It is especially useful for the global static scheduling methods.

Static learning: means that before effective run-time prediction process, the predictor is trained based on some patterns. In the static learning process the outputs of the predictor are used only to adjust the prediction structures.

Super-state: in the case of *Hidden Markov Models* of order R a combination of R primitive hidden states form a super-state.

Superscalar processor: is a microarchitecture that exploits *instruction-level parallelism (ILP)* by introducing more than one instruction at a time into multiple pipelines to be executed simultaneously.

Thread-level parallelism (TLP): means processing instructions from multiple threads simultaneously or concurrently within multithreaded microarchitectures. TLP can be extracted from either sequential programs or multithreaded workloads.

Unbiased branch: a branch whose behavior (taken / not taken) is not sufficiently polarized.

Unbiased branch context: the branch behavior (taken / not taken) is not sufficiently polarized for that certain *context* (local branch history, global history, path, etc.).