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

Advanced Prediction Methods Integrated Into Speculative Computer Architectures

PhD Thesis

Abstract

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

Rezumat

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

Conducători ştiinţifici:
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).

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 biți). Istoriile locale şi globale suficient de lungi aproximează foarte bine informația de cale.

În Capitolul 4 am arătat că pentru anumite instrucțiuni de ramificație, informațiile de predicție limitate (istorie locală, istorie globală şi calea spre branch-ul supus predicției) folosite de predictoarele actuale nu sunt întotdeauna suficient de relevante şi, din această cauză, ele nu pot fi predicționate cu acuratețe. Acuratețea cea mai ridicată pe branch-urile nepolarizate, de doar 77,30%, am obținut-o cu *piecewise linear branch predictor* [Jim05]. De aceea, este deosebit de importantă găsirea unor informații relevante care determină comportamentul instrucțiunilor de ramificație, pentru a fi utilizate de predictoare mai eficiente. Am dezvoltat diferite predictoare de valori Markoviene care folosesc istoria comprimată a precedentelor condiții de salt, ale cărei elemente pot fi -1, 0 sau 1, în funcție de semnul diferenței dintre operanzi. Nici aceste predictoare puternice, capabile să exploateze corelația dintre comportamentul branch-ului şi istoria condițiilor, n-au reuşit să îmbunătățească predictibilitatea acestor branch-uri dificile. De asemenea, am îmbunătățit predictoare convenționale (GAg, PAg) şi neuronale, prin utilizarea condiției de salt precedente – Previous Branch Condition (PBC) – sub forma unei diferențe pe 32 de biți dintre operanzi. Chiar şi predictorul *piecewise linear branch predictor* îmbunătățit, cel mai performant pe branch-urile nepolarizate, obține o acuratețe modestă de 78,3% pe branch-urile nepolarizate, în timp ce acuratețea globală a predicției este de 95,45%. Aşadar, branch-urile nepolarizate sunt caracterizate de acurateți de predicție scăzute, indiferent de informația de predicție folosită, reprezentând astfel o limitare fundamentală în domeniul predicției branch-urilor. Astfel, creşterea acurateții de predicție a acestor instrucțiuni de ramificație nepolarizate constituie o problemă deschisă, deoarece fiecare procent de astfel de instrucțiuni reduce decisiv acuratețea predicției şi implicit performanța de procesare.

Pornind de la această provocare tehnică, în Capitolul 5 am realizat un studiu comparativ privind gradul de aleatorism al secvențelor 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 secvenței de program care generează branch-ul, rata de compresie, entropia discretă respectiv acuratețea de predicție cu HMM (Hidden Markov Models) a secvenței 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 secvențe binare din punct de vedere al gradului de aleatorism asociat, converg în aceeaşi direcție. 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 îmbunătățire a predictorului aferent. În cazul aleatorismului ridicat, răspunsul este unul pesimist întrucât secvența este una intrinsec, şi deci iremediabil, aleatoare. De precizat că aleatorismul comportării acestor branch-uri este o consecință a complexității uriaşe a programelor care le generează, după cum arătăm în lucrare.

Instrucțiunile cu latență ridicată reprezintă o altă sursă de limitare a paralelismului la nivelul instrucțiunilor. În Capitolul 6 am arătat că 28,68% din instrucțiunile de ramificație (5,61% fiind chiar nepolarizate) sunt dependente de instrucțiuni cu latență ridicată (Load-uri critice, înmulțiri, împărțiri). Aceste dependențe reprezintă o sursă importantă de penalități datorate predicțiilor greşite, afectând serios performanța globală a procesorului. De aceea, impactul negativ al branch-urilor, în special al celor nepolarizate, asupra performanței globale poate fi atenuat anticipând rezultatele instrucțiunilor cu latență ridicată. Am dezvoltat un mecanism de anticipare selectivă a valorilor instrucțiunilor cu latență de execuție ridicată, care include o schemă de reutilizare pentru instrucțiunile Mul şi Div, respectiv un predictor de valori pentru instrucțiunile 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

Abstract

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

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.

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.

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. 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.

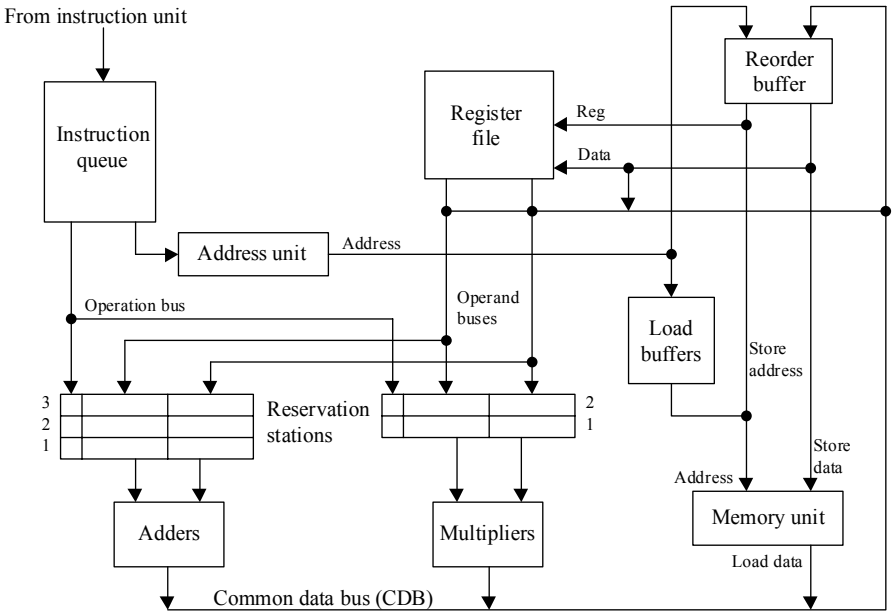

Figure 2.1. Tomasulo's architecture extended to support speculation

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 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.

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.

2.2. The Architecture of Sim-Outorder

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. The *sim-outorder* simulator (see Figure 2.2) from the *Simplesim-3.0* toolset [Bur97] 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.

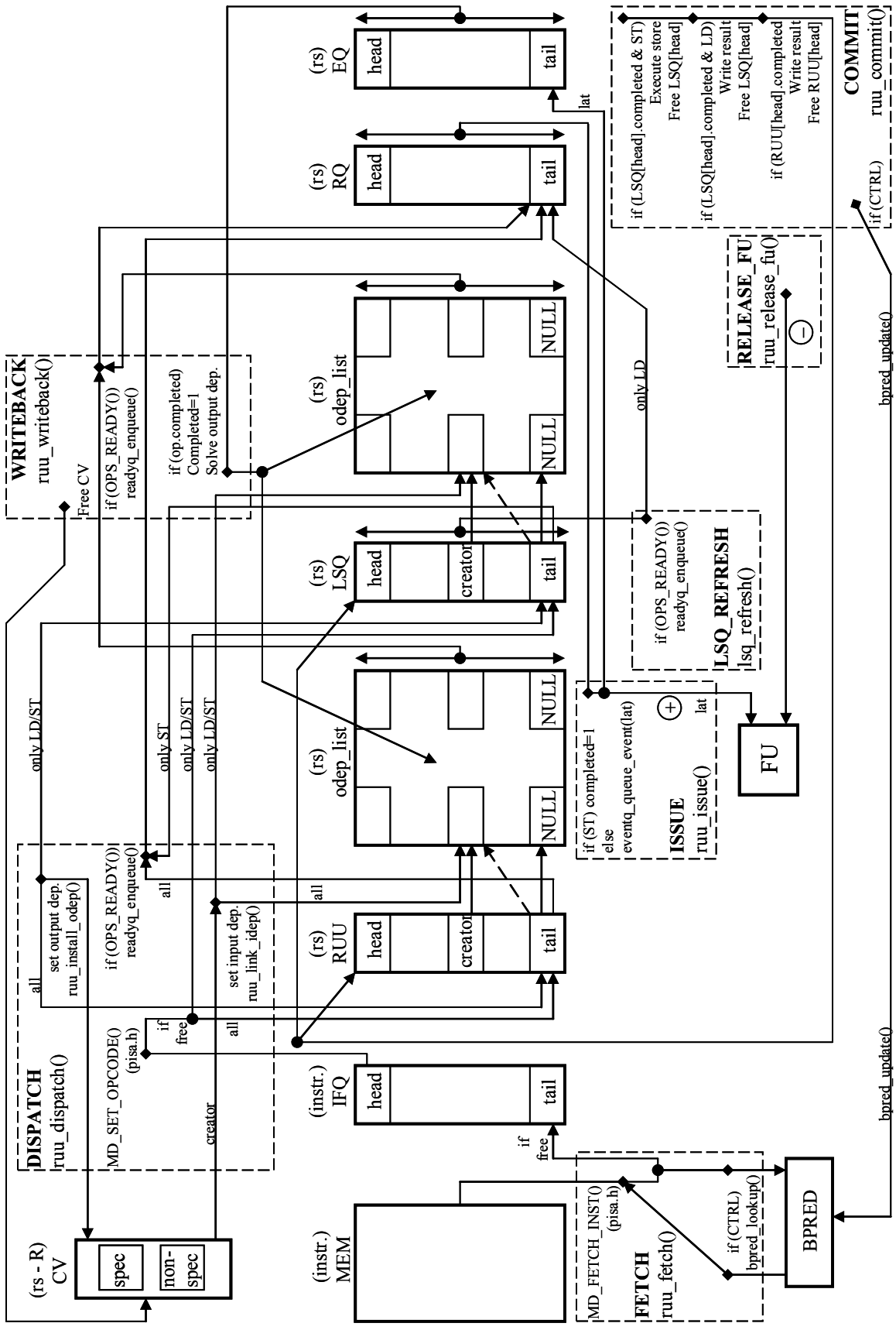

Figure 2.2. The architecture of *Sim-Outorder*

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). 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 afferent to 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.

3. Finding Difficult-to-Predict Branches

Since the performances of modern speculative architectures highly depend on branch prediction accuracy, we will further focalize 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. 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. 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 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 afferent 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 afferent 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 ($0 \to 1$ 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";
- if $D(S_i) \to 0, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is constant.

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).

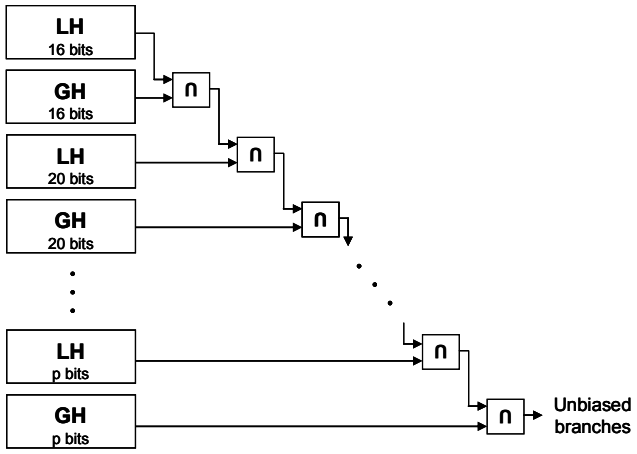

Figure 3.1. Reducing the number of unbiased branches through feature set extension

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.

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 (the total number of branch context instances).

3.2. Experimental Results

3.2.1. Pattern-Based Correlation

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).

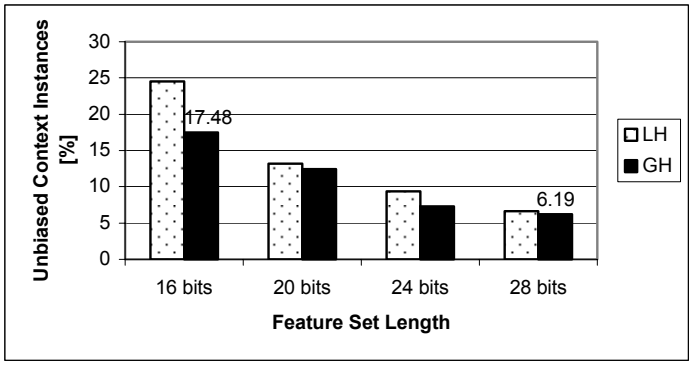

Figure 3.2. Reduction of average percentages of unbiased context instances (P<0.95) in the SPEC 2000 benchmarks by extending the lengths of feature sets

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.

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. As our evaluations show, in the case of unbiased branch contexts, the *taken* and *not taken* outcomes are not grouped separately, more, they are highly shuffled.

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 and 89.37% in the case of global history of 16 bits. 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.

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. Focalising 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 3.2). 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.

3.2.2. 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 evaluated – 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 3.3 where they are compared with the results obtained using only global history.

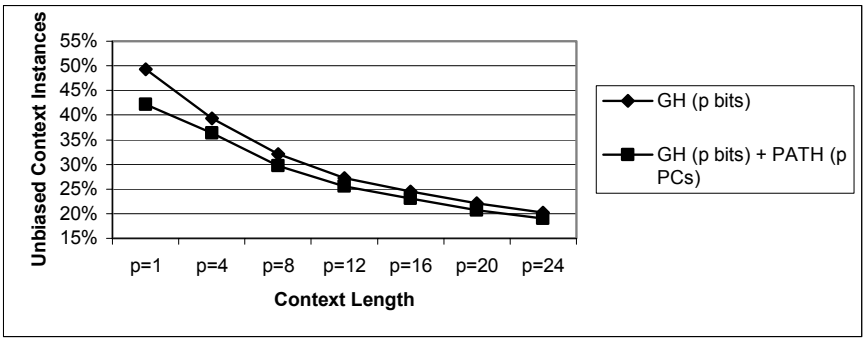

Figure 3.3. The gain introduced by the path for different context lengths – SPEC 2000 benchmarks

Figure 3.3 shows 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).

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. Value-History-Based Branch Prediction with Markov Models

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. 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.

4.1.1. Local Branch Difference Predictor

Figure 4.1 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.

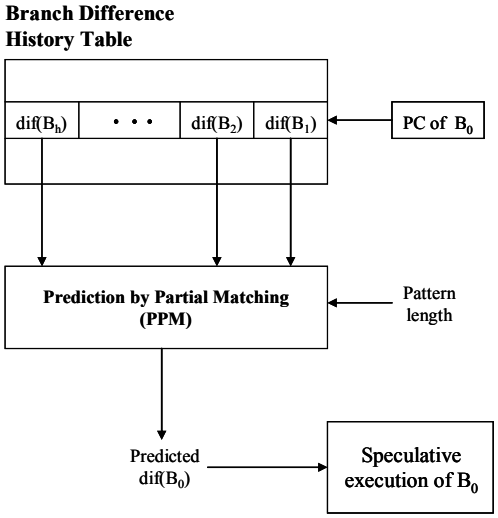

Figure 4.1. A local PPM-based branch-difference predictor

4.1.2. Combined Global-Local Branch Difference Predictor

Figure 4.2 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 4.1.1.

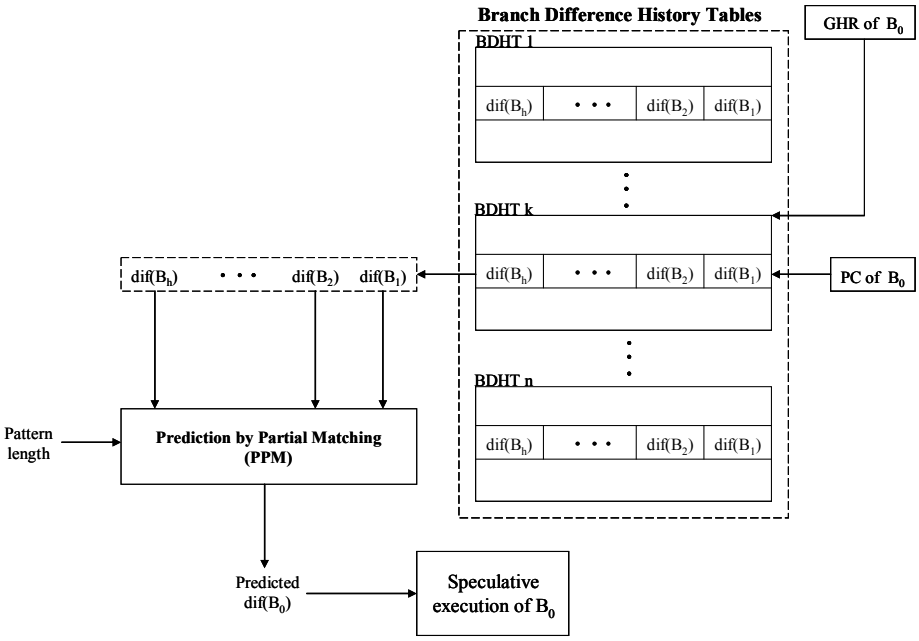

Figure 4.2. A global-local PPM-based branch-difference predictor

4.1.3. Branch Difference Prediction by Combining Multiple Partial Matches

Figure 4.3 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 4.1.1, 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 4.3). The final branch difference prediction is then furnished through majority vote.

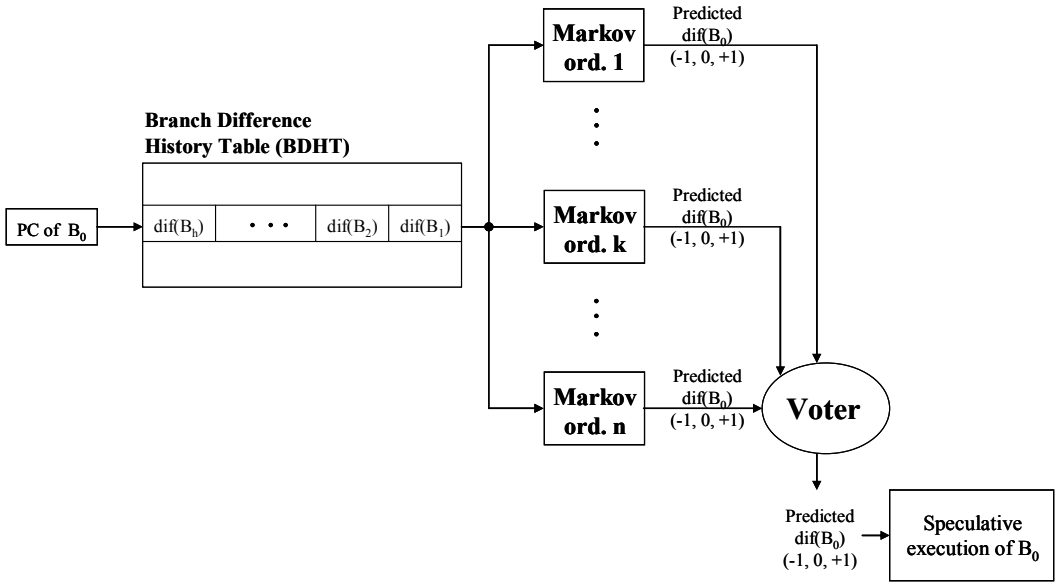

Figure 4.3. Branch-difference prediction by combining multiple Markov predictors

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.

4.2. 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 4.4 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).

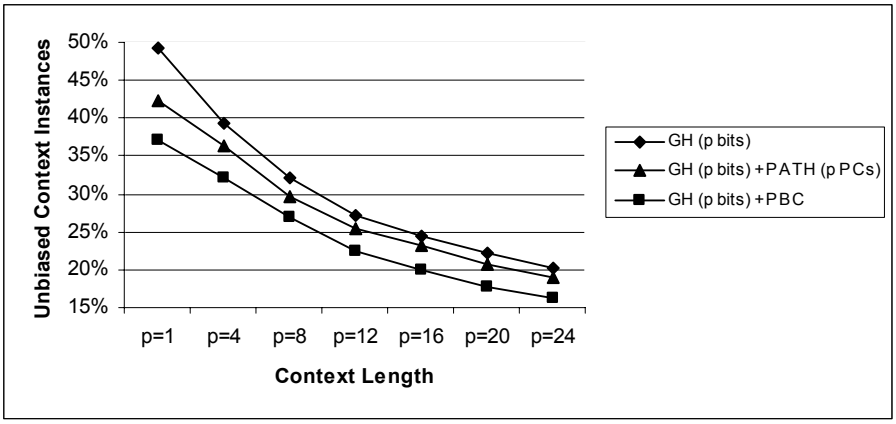

Figure 4.4. The gain introduced by the previous branch condition (PBC) vs. the path for different context lengths – SPEC 2000 benchmarks

The experimental results, presented in Figure 4.4, 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].

4.2.1. 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 4.5.

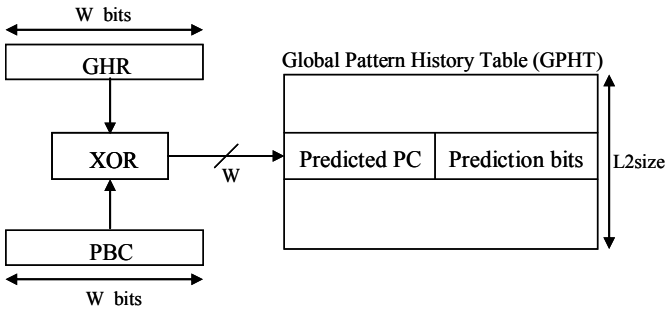

Figure 4.5. The GAg predictor using the previous branch condition (PBC)

4.2.2. 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 4.6.

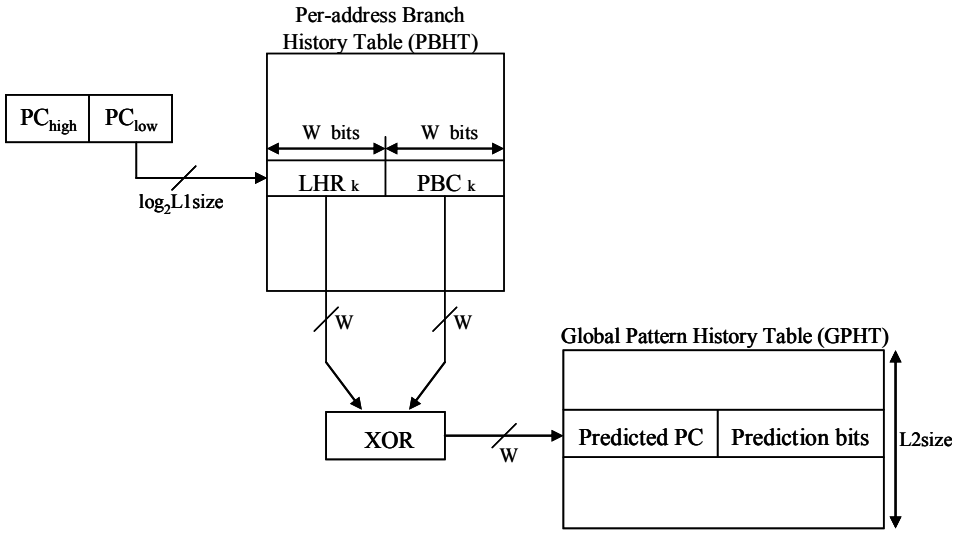

Figure 4.6. The PAg predictor using the local PBC value

4.2.3. The Piecewise Linear Branch Predictor Using PBC Value

Further, we propose some improved *idealized piecewise linear branch predictors* (see Figures 4.7 and 4.8) 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]:

$$index_{GH}^{i} = \left[\left(PC \cdot 511387\right) \oplus \left(PC_{i-1} \cdot 660509\right) \oplus \left(i \cdot 1289381\right)\right] \bmod NW \qquad (4.1)$$

$$index_{LH}^{j} = \left[\left(PC \cdot 511387\right) \oplus \left(j \cdot 1289381\right)\right] \bmod NW \qquad (4.2)$$

$$index_{PBC}^{k} = \left[\left(PC \cdot 511387\right) \oplus \left(k \cdot 1289381\right)\right] \bmod NW \qquad (4.3)$$

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.

4.2.3.1 The Piecewise Linear Branch Predictor Using Global PBC Value

Figure 4.7 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.

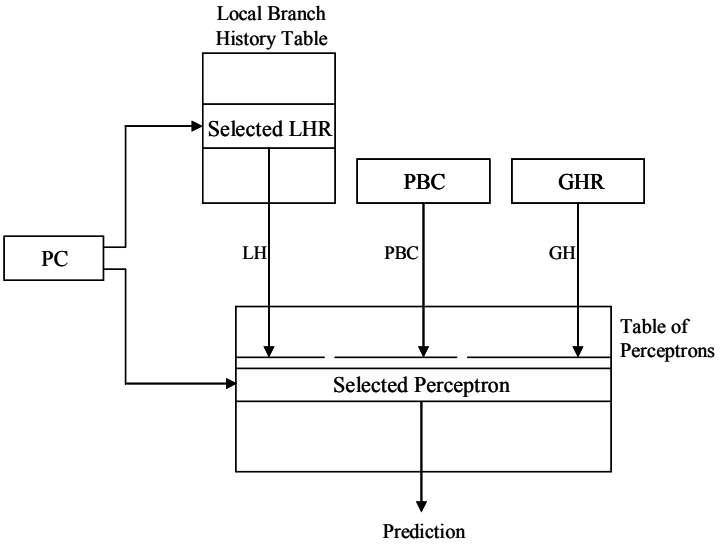

Figure 4.7. Perceptron-based branch predictor using the global PBC value

4.2.3.2 The Piecewise Linear Branch Predictor Using Local PBC Value

Figure 4.8 presents a possible scheme of the perceptron-based branch predictor that is using as prediction information local (per-address) previous branch condition (PBC). The Local Branch History Table maintains for each branch its Local History (LH) and its the Previous Branch Condition (PBC) value.

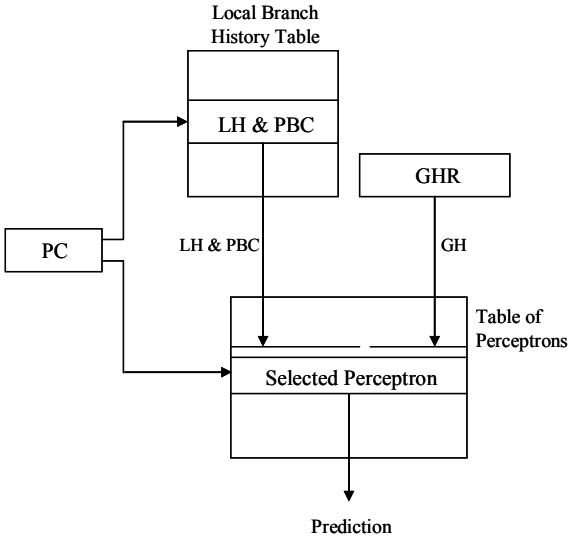

Figure 4.8. Perceptron-based branch predictor using the local PBC value

4.3. 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].

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

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.

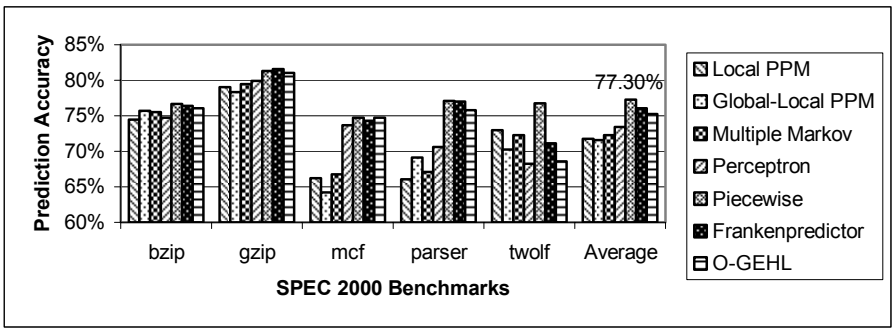

Figure 4.9. Branch prediction accuracies obtained using the perceptron-based predictors, the O-GEHL predictor and the PPM-based predictors, only on unbiased branches

As Figure 4.9 shows, 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.

4.3.2. Evaluating PBC-Based Branch Predictors

We evaluated our modified GAg, PAg and *piecewise linear branch predictor* on unbiased branches, using the global PBC value as additional prediction information. For the *piecewise linear branch predictor* we increased the number of weights from 8590 upto 30713, the higher weights number being justified by the long additional information.

With the modified *piecewise linear branch predictor* we obtained a prediction accuracy of 78.30% opposite to those obtained with the modified GAg, 69.87% and the modified PAg, 73.75%. 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).

Figure 4.10 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).

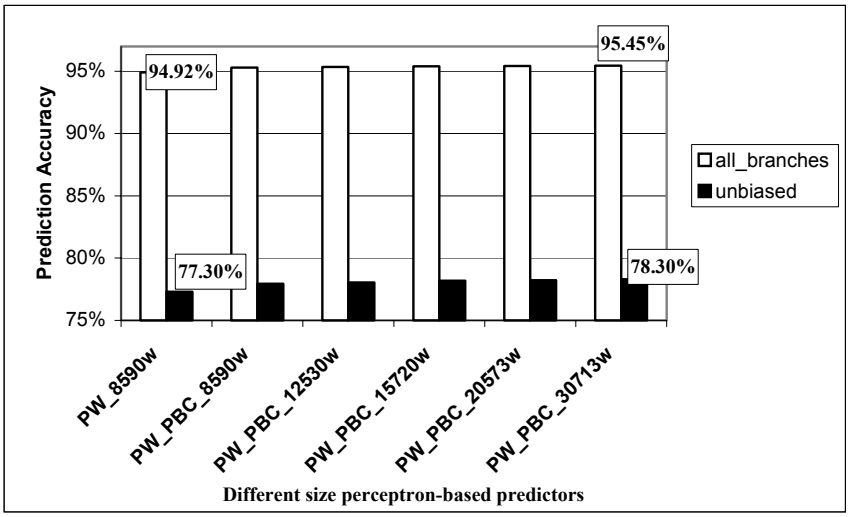

Figure 4.10. Average prediction accuracies obtained with *piecewise linear branch predictor* on unbiased branches versus all branches, using the global PBC value as additional prediction information

Analyzing comparatively the results presented in Figures 4.9 and 4.10 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).

5. Better Understanding Unbiased Branches Using Random Degrees

As we stated out in the previous chapter, the unbiased branches behavior is practically unpredictable. Why 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 HLL 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. 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.

5.1.1. 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.

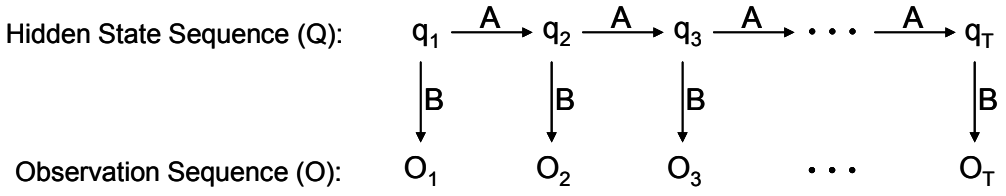

Figure 5.1. Hidden Markov Model

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.

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.

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.

5.1.2. 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

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

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.

5.1.3. 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.

5.1.4. 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.

5.2. Evaluation Results

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). 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 5.2.1. During the paragraph 5.2.2 we calculated the random degrees associated to the same binary strings. In paragraph 5.2.3 we calculated the compression rates corresponding to the same branches behaviors.

5.2.1. 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. We evaluated prediction accuracies on strongly unbiased branches using a HMM predictor of order one (R=1) and two (R=2) for different numbers of hidden states (N). For the majority of the benchmarks considering two hidden states generate the best accuracies. The average prediction accuracy obtained using the quasi-optimal HMM (R=1, N=2) is far greater on biased contexts than on unbiased contexts. Figure 5.2 comparatively presents, for unbiased and biased branches, the average prediction accuracies obtained by the quasi-optimal HMM (R=1, N=2).

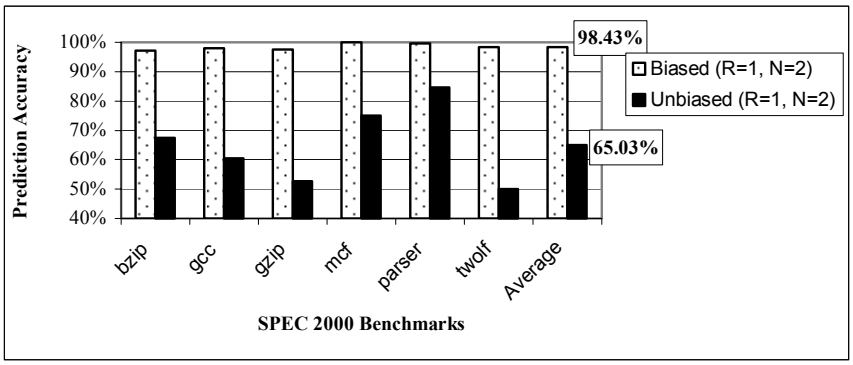

Figure 5.2. Prediction accuracies using the best evaluated HMM (R=1, N=2)

There is a significant difference between the average prediction accuracy on biased branches (98.43%) and on unbiased branches (65.03%). 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.

5.2.2. 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, $RD(S) = D(S) \cdot E(S)$. Figure 5.3 shows statistical results concerning the random degree of the biased and unbiased binary sequences obtained through the previously exposed methodology.

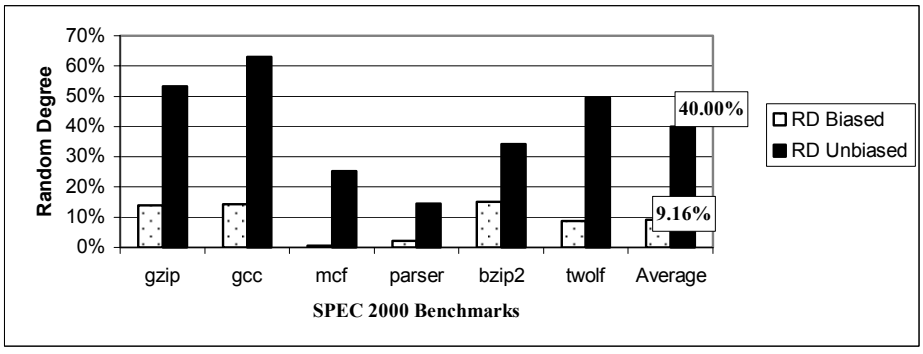

Figure 5.3. The random degree of biased and unbiased branches

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.

5.2.3. 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 5.2. 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 5.1.3. In Figure 5.4, we illustrate the space savings obtained by compressing biased and unbiased branches using the previously described algorithms (*Gzip* and *Huffman*).

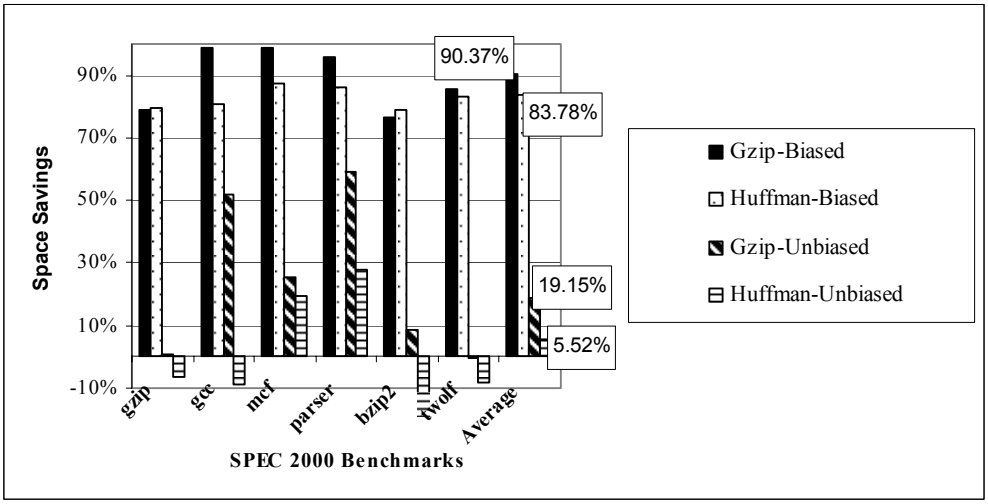

Figure 5.4. Space savings using the *Gzip* and *Huffman* algorithms

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).

5.2.4. Random Degree Evaluation Based on Kolmogorov Complexity

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.

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.

Among the other conditional branches only one (PC=35) proved to be unbiased for shorter global history length ($\leq$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.

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 forementioned 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. 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.

6.1.1. 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.

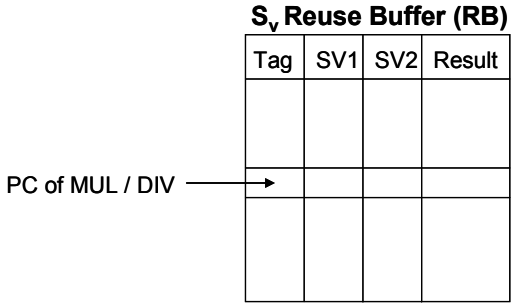

Figure 6.1. Reuse scheme for Mul & Div instructions

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 6.7, 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.

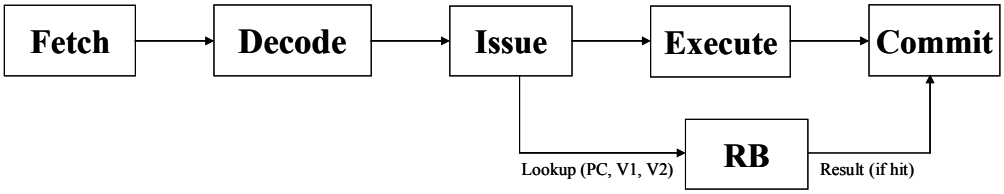

Figure 6.2. Pipeline with Reuse Buffer (RB)

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.

6.1.2. 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).

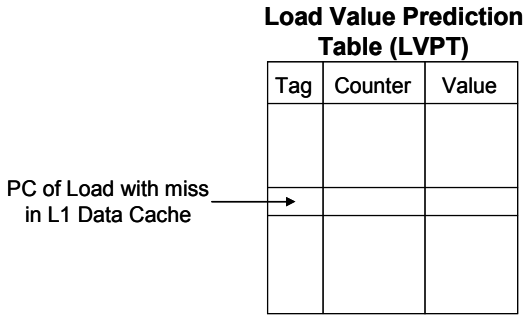

Figure 6.3. The Last Value Predictor architecture

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.

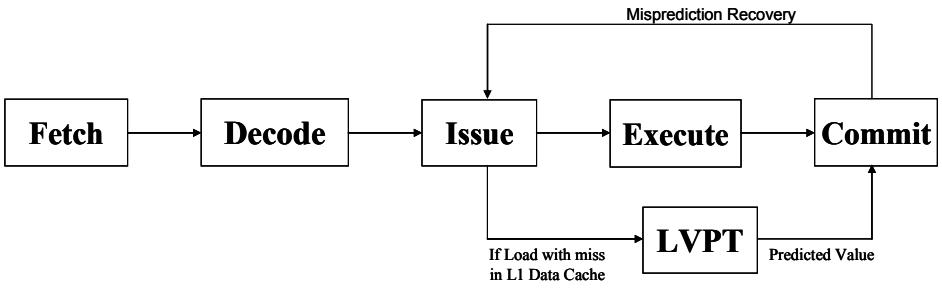

Figure 6.4. Pipeline with Load Value Predictor

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).

6.1.3. Experimental Results

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

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

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. 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.

We evaluated seven integer benchmarks (*bzip*, *gcc*, *gzip*, *mcf*, *parser*, *twolf*, *vpr*) and six floating-point benchmarks (*applu*, *equake*, *galgel*, *lucas*, *mesa*, *mgrid*). Although the RB structure dissipates additional dynamic power, reusing long-latency instructions increases the IPC and therefore lowers the relative energy consumption (see Figure 6.5). 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.

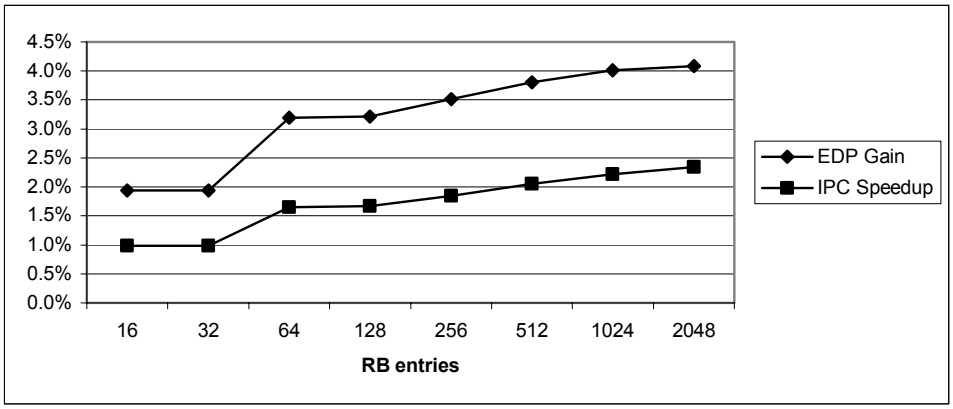

Figure 6.5. Relative IPC speedup and relative energy-delay product gain on the SPEC 2000 floating-point benchmarks with RB and Trivial Operation Detection

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.

Figure 6.6 presents the relative IPC speedup and the relative energy-delay product improvement obtained 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. 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.

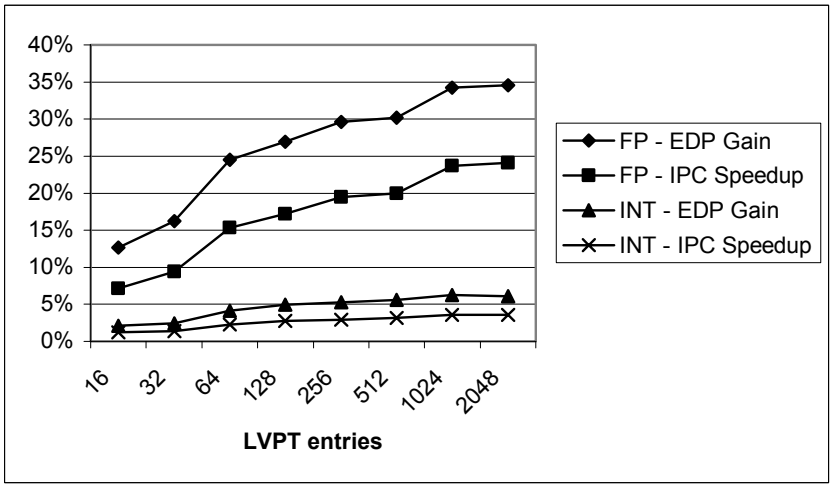

Figure 6.6. Relative IPC speedup and relative energy-delay product gain with a Reuse Buffer of 1024 entries, the Trivial Operation Detector, and the Load Value Predictor

Both IPC speedup and EDP gain are significantly higher on the floating-point benchmarks compared to the integer benchmarks (see Figure 6.6). 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.

The selective instruction reuse approach proposed by Golander and Weiss in [Gol07] 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.

6.2. Contributions to Dynamic Value Prediction: CPU Context Prediction

The main aim of this section consists in focalizing 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.

6.2.1. 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 6.7). 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).

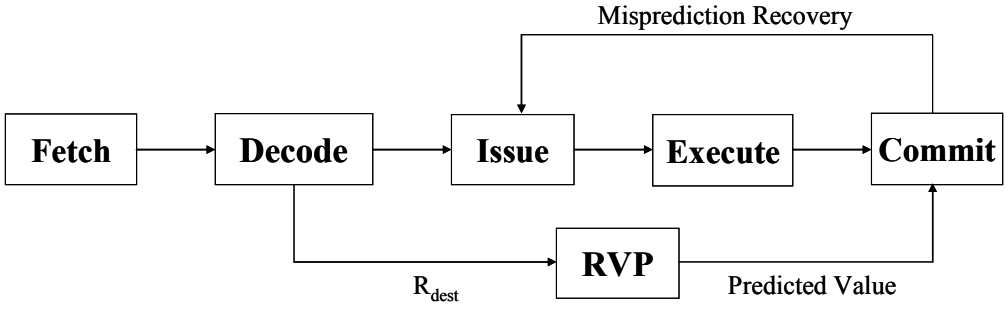

Figure 6.7. The implementation of the register value prediction mechanism in the pipeline structure of a general microarchitecture

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.

6.2.1.1. Last Value Predictors

The *last value predictors* (see Figure 6.8) 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.

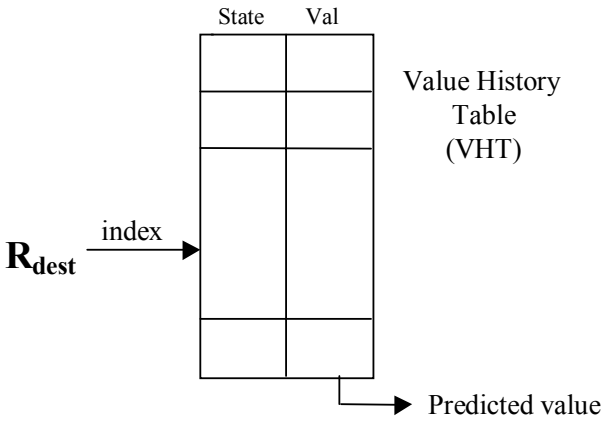

Figure 6.8. Last value predictor

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.

6.2.1.2. 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 6.9 shows the structure of this predictor.

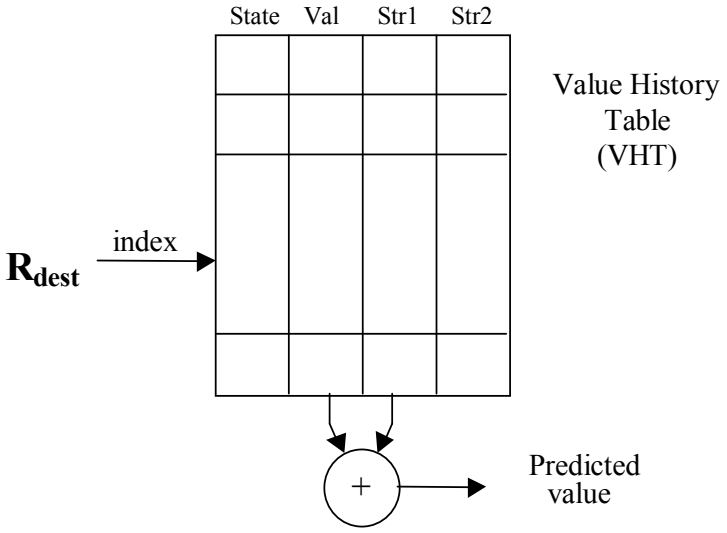

Figure 6.9. Stride predictor

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.

6.2.1.3. 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 4.1. 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.

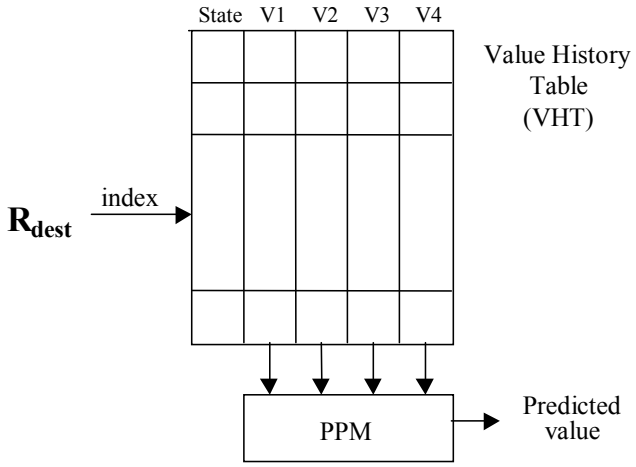

Figure 6.10. Structure of a context-based PPM predictor

Figure 6.10 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.

6.2.1.4. 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 6.11 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.

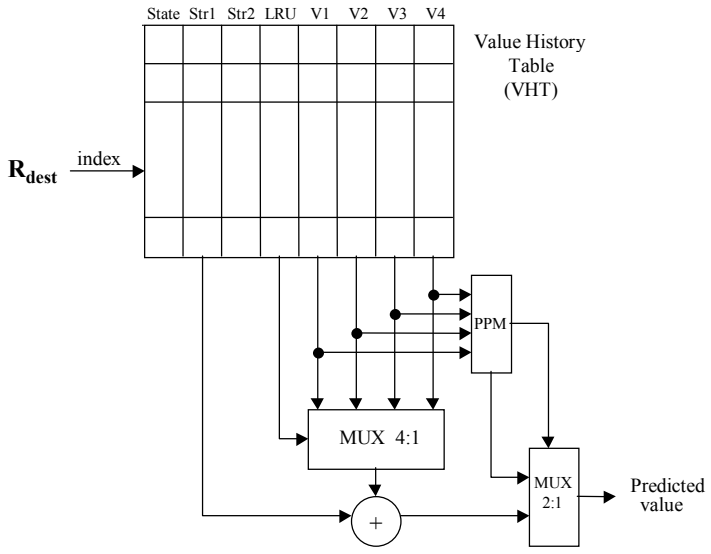

Figure 6.11. Hybrid predictor (PPM & stride)

Figure 6.12 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 proposed by Wang and Franklin in [Wan97], but it is indexed with the destination register name instead of the PC. This fixed prioritization used in Figures 6.11 and 6.12 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).

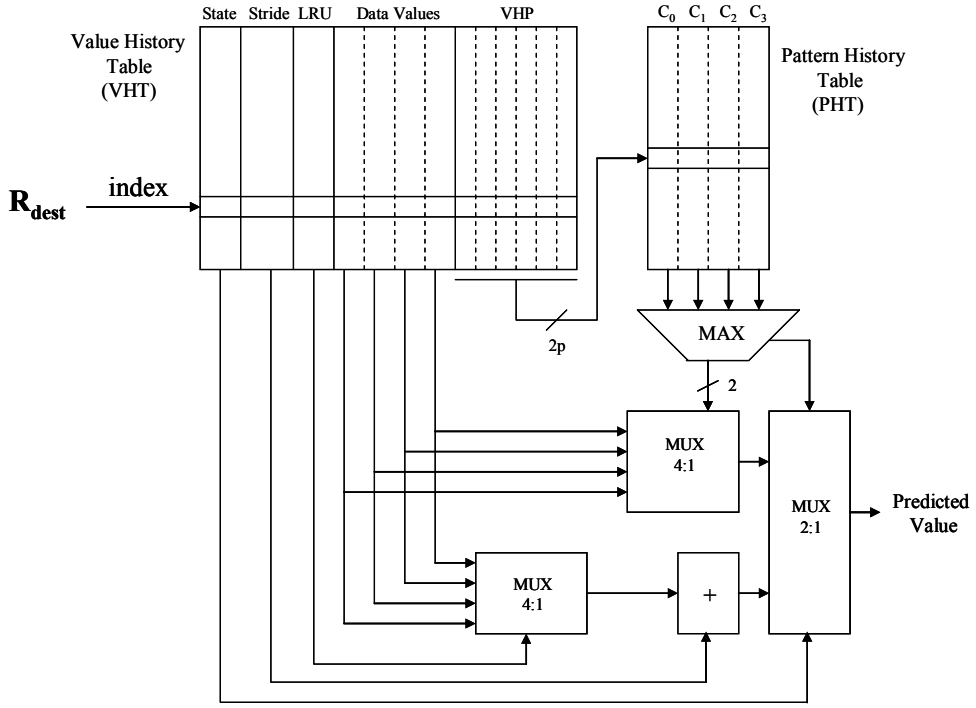

Figure 6.12. Hybrid predictor (two-level & stride) with fixed prioritization

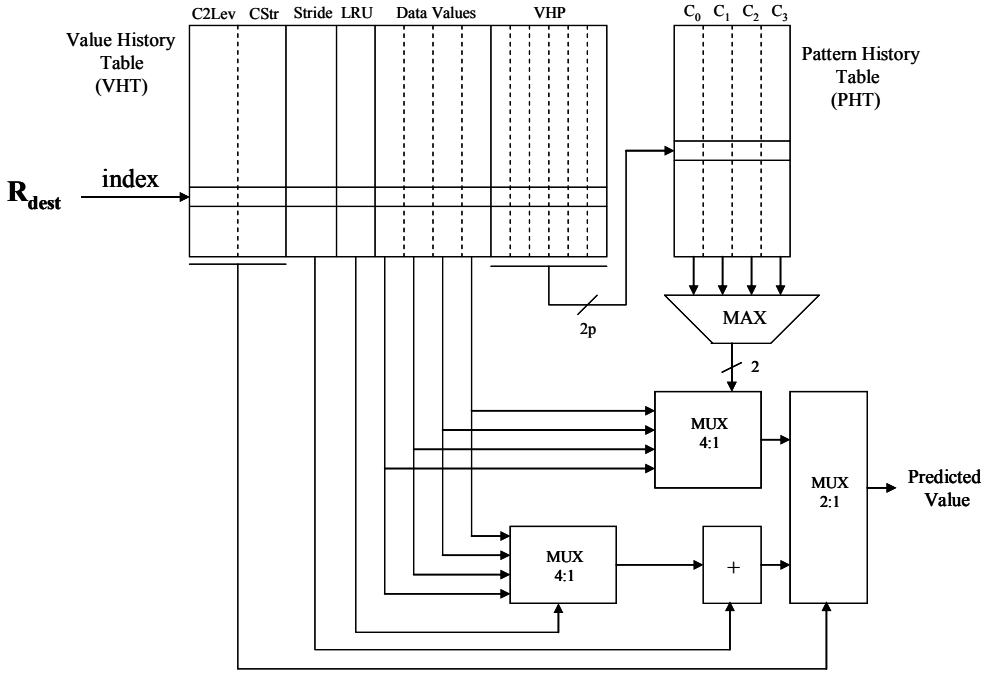

Figure 6.13. Hybrid predictor (two-level & stride) with adaptive prioritization

The adaptive hybrid predictor presented in Figure 6.13 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

6.2.2. Experimental Results

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. In this paragraph, 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. 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 6.14 and 6.15 we compared the previously presented value prediction techniques: last value prediction (Figure 6.8), stride prediction (Figure 6.9), PPM prediction (Figure 6.10) and PPM-based hybrid prediction (Figure 6.11). 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.

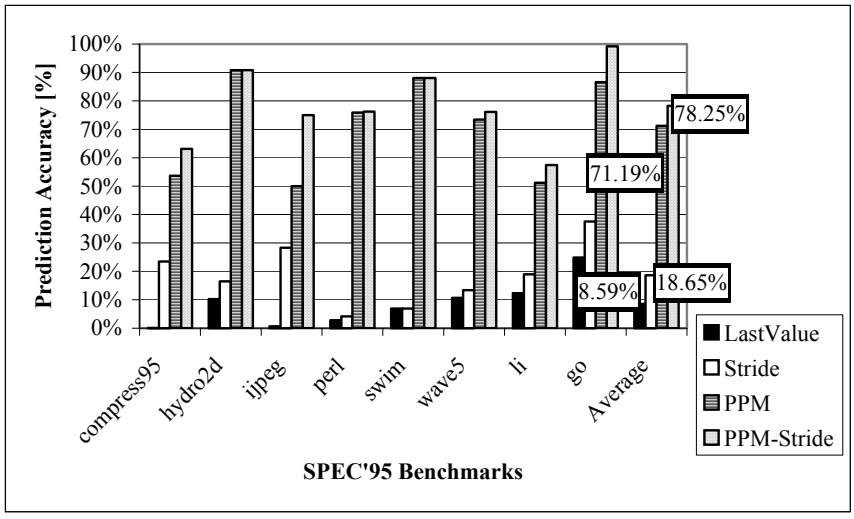

Figure 6.14. Prediction accuracy using 17 favorable registers (PA>60%) on the SPEC'95 benchmarks

These results (see Figures 6.14 and 6.15) 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.

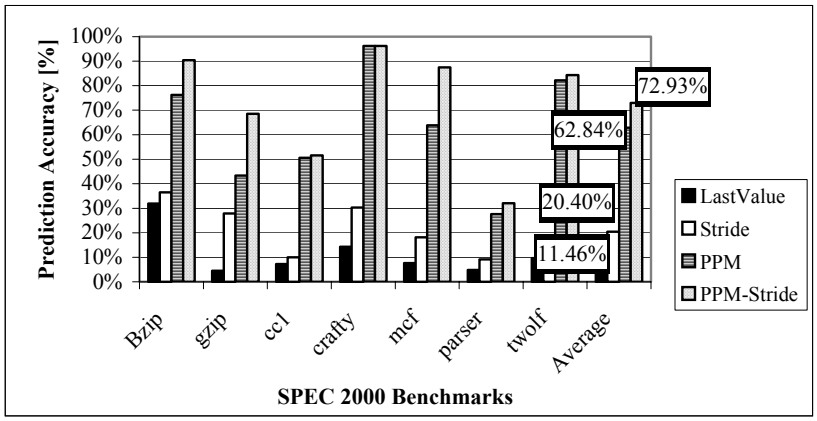

Figure 6.15. Prediction accuracy using 22 favorable registers (PA>60%) on the SPEC 2000 benchmarks

Now we will try a more elitist selection considering only the registers with prediction accuracy higher than 80% (see Figures 6.16 and 6.17). 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.

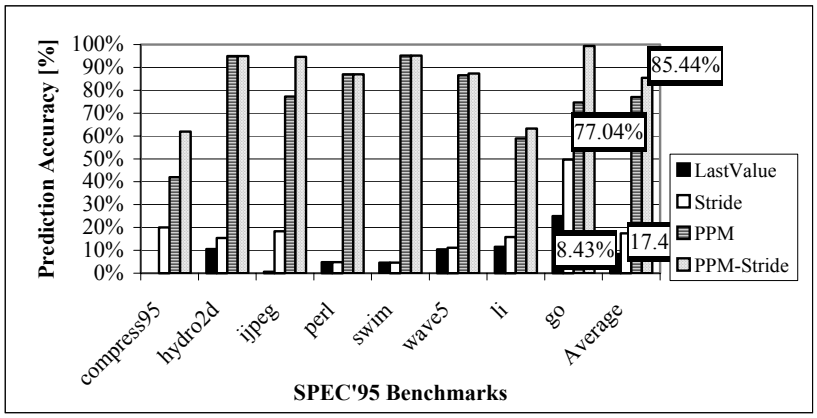

Figure 6.16. Prediction accuracy using 8 favorable registers (PA>80%) on the SPEC'95 benchmarks

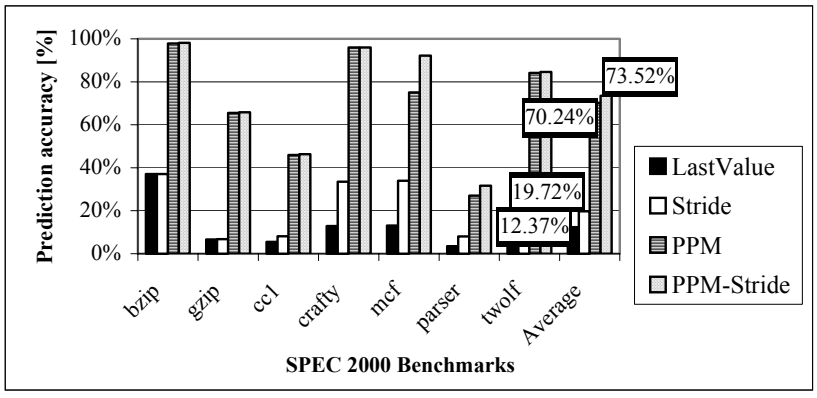

Figure 6.17. Prediction accuracy using 16 favorable registers (PA>80%) on the SPEC 2000 benchmarks

Figures 6.16 and 6.17 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%). Considering an 8-issue out-of-order superscalar processor simulations show that register centric value prediction produce average speedups of 17.30% for the SPECint95 benchmarks, respectively of 13.58% for the SPECint2000 benchmarks.

Finally, in Figures 6.18 and 6.19 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 6.12) and *2Lev+Stride* with adaptive prioritization (presented in Figure 6.13), both using a history of 32 values and a pattern of 4 values.

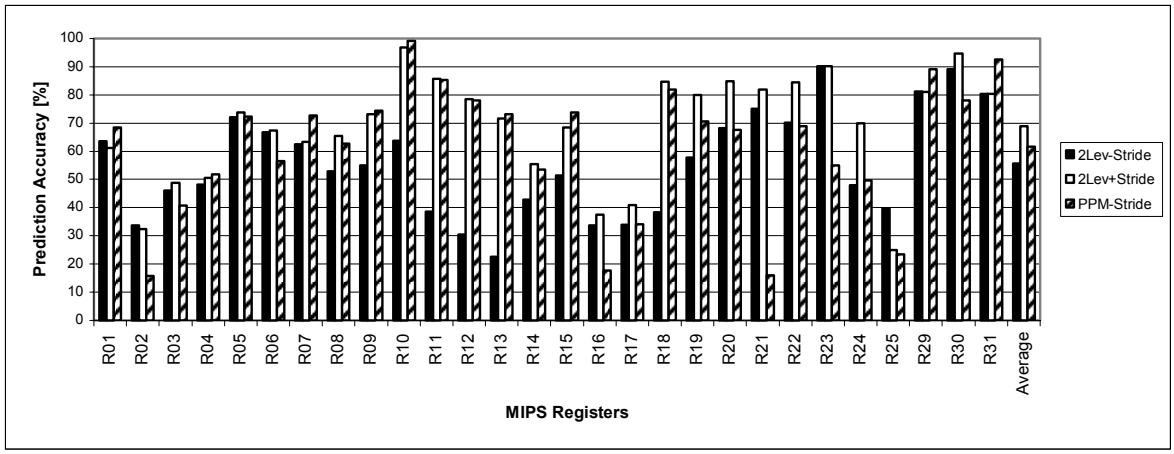

Figure 6.18. Comparing the hybrid predictors on the SPEC'95 benchmarks

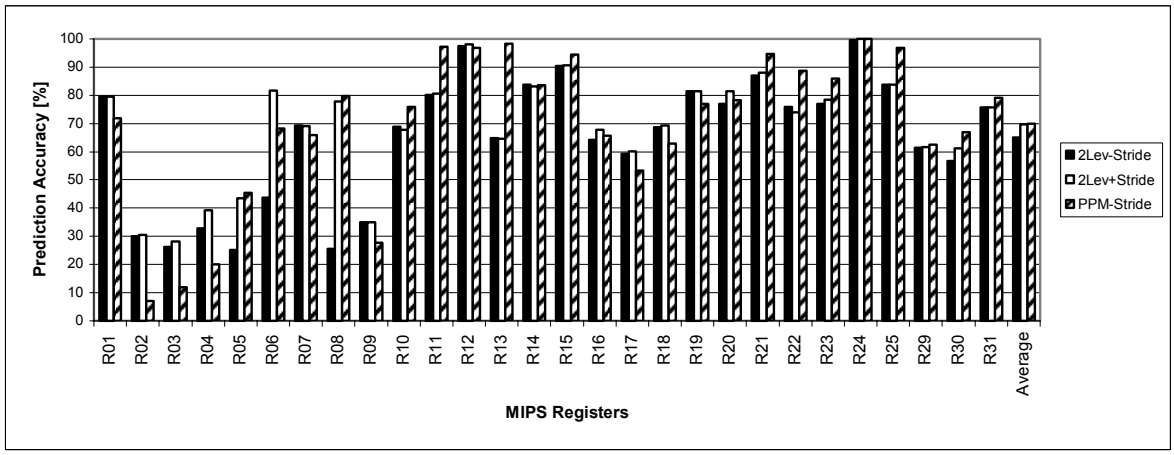

Figure 6.19. Comparing the hybrid predictors on the SPEC 2000 benchmarks

Figures 6.18 and 6.19 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.

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 focalized 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. 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 6.1.

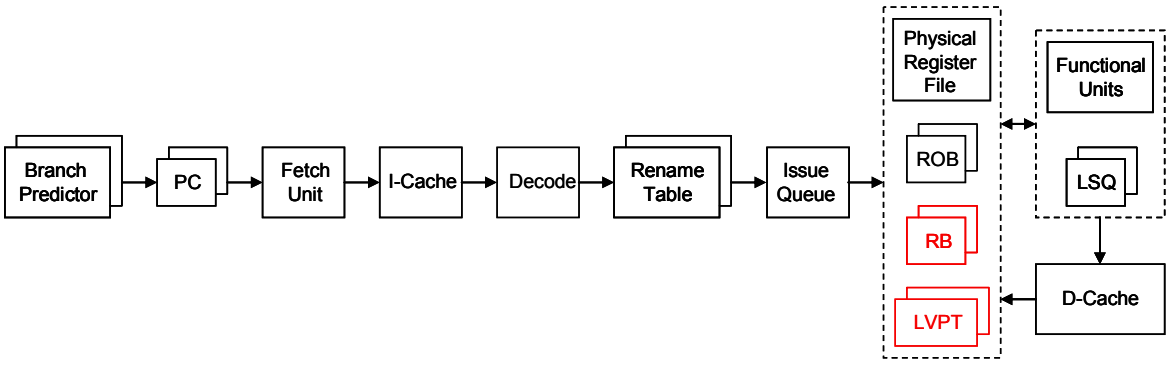

Figure 7.1. SMT architecture enhanced with selective instruction reuse and value prediction

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, 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.

7.2. Experimental Results

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.

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.

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 6.1.3), we optimally sized the RB and the LVPT to 1024 entries. 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).

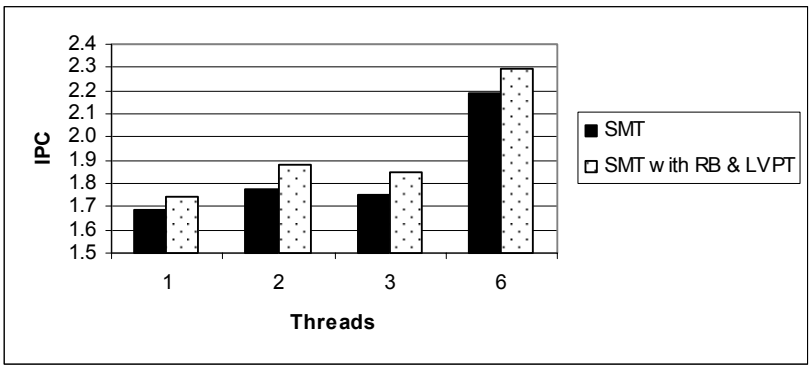

Figure 7.2. IPC obtained with and without RB & LVPT on the integer SPEC 2000 benchmarks

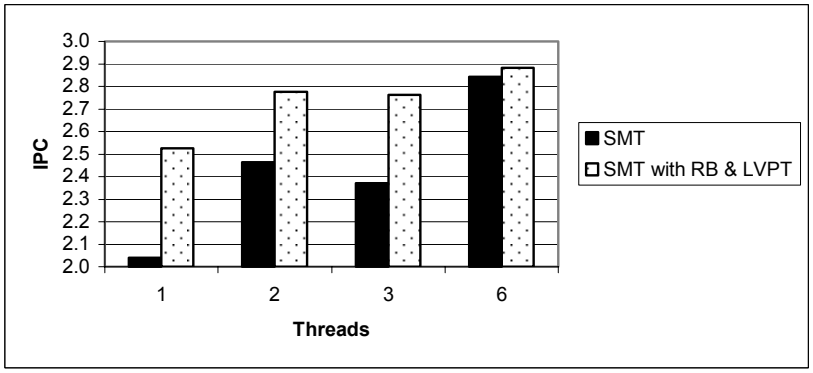

Figure 7.3. IPC obtained with and without RB & LVPT on the floating-point SPEC 2000 benchmarks

With fewer threads, the ten shared functional units 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.

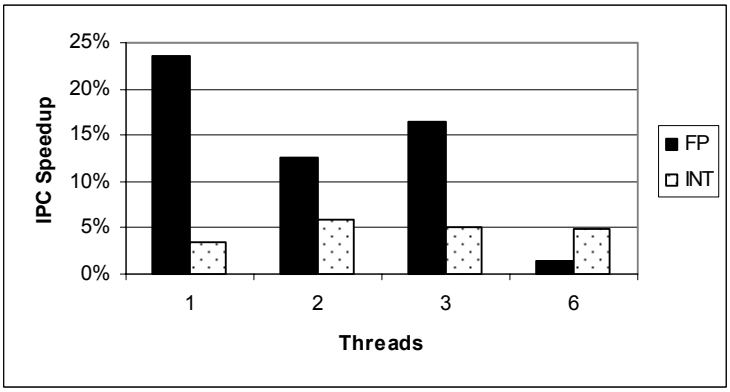

Figure 7.4. Relative IPC speedup (enhanced SMT vs. classical SMT) by varying the number of threads

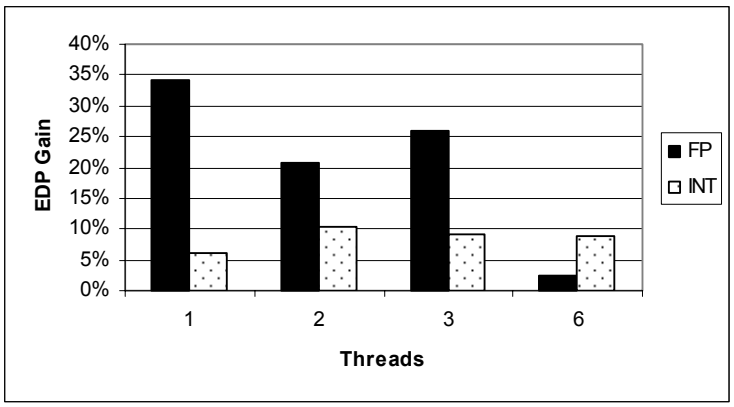

Figure 7.5. Relative energy-delay product gain (enhanced SMT vs. classical SMT) for different number of threads

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.

8. Conclusions and Further Work

This chapter presents some quantitative and qualitative conclusions regarding the important experimental results obtained within this thesis and emphasizes some possible further work directions. The main contributions of this work can be summarized as follows:

- **A systematic methodology of identifying difficult-to-predict branches:** 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.
- **Dedicated predictors designed to improve the prediction accuracy of unbiased branches:** we 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). 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.

- **Random degree metrics developed to characterize the randomness of sequences produced by unbiased branches:** 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.
- **Selective anticipatory methods integrated into superscalar architectures:** 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.

- **Selective anticipatory methods integrated into simultaneous multithreaded architectures:** 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 HLL code 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.

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*, Revised version submitted to Journal of Systems Architecture, July 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.