Lucian N. VINȚAN

Prediction Techniques in Advanced Computing Architectures


Matrix Rom Publishers, Bucharest
2007
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>Foreword</td>
<td>5</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2. Advanced Branch Prediction Methods</td>
<td>24</td>
</tr>
<tr>
<td>2.1. Dynamic Neural Branch Prediction. Genesis</td>
<td>24</td>
</tr>
<tr>
<td>2.1.1. Previous Work In The Field</td>
<td>28</td>
</tr>
<tr>
<td>2.1.2 First Neural Dynamic Branch Prediction Models</td>
<td>30</td>
</tr>
<tr>
<td>2.2. Trace Driven Simulation Results</td>
<td>32</td>
</tr>
<tr>
<td>2.3. A Static Branch Neural Predictor</td>
<td>38</td>
</tr>
<tr>
<td>2.4. Some Comments To Neural Branch Prediction Idea</td>
<td>44</td>
</tr>
<tr>
<td>2.5 Making It Feasible: Perceptron Branch Predictors</td>
<td>52</td>
</tr>
<tr>
<td>2.6 Towards Cross-Fertilization Between Computer Architecture And Other Computer Science Fields [Vin01]</td>
<td>63</td>
</tr>
<tr>
<td>2.6.1. Some Other Neural Approaches In Computer Architecture</td>
<td>65</td>
</tr>
<tr>
<td>2.6.2. A Markovian Branch Predictor</td>
<td>66</td>
</tr>
<tr>
<td>2.6.3. Genetic Branch Predictors</td>
<td>67</td>
</tr>
<tr>
<td>3. Understanding Some Present-Day Branch Prediction Limits</td>
<td>74</td>
</tr>
<tr>
<td>3.1. Unbiased Branches Problem And Related Works</td>
<td>74</td>
</tr>
<tr>
<td>3.2 Finding Unbiased Branches. Research Methodology</td>
<td>84</td>
</tr>
<tr>
<td>3.3 Some Experimental Results [27, 19]</td>
<td>89</td>
</tr>
<tr>
<td>3.4 Some Conclusions And Further Work</td>
<td>108</td>
</tr>
<tr>
<td>4. Pre-Computing Branches</td>
<td>113</td>
</tr>
<tr>
<td>4.1. Introduction</td>
<td>113</td>
</tr>
<tr>
<td>4.2 The Pre-Computed Branch Algorithm</td>
<td>114</td>
</tr>
<tr>
<td>4.3 Complexity And Costs Evaluations</td>
<td>120</td>
</tr>
<tr>
<td>4.4 Performance Evaluations Through Simulation</td>
<td>123</td>
</tr>
<tr>
<td>4.5 Some Conclusions And Further Work</td>
<td>126</td>
</tr>
<tr>
<td>4.6 Appendix. The Pcb Algorithm</td>
<td>127</td>
</tr>
<tr>
<td>5. Dynamic Instruction Reuse</td>
<td>132</td>
</tr>
<tr>
<td>5.1. Fetch Bottleneck. Dynamic Instruction Reuse Through Trace-Cache Processors</td>
<td>132</td>
</tr>
<tr>
<td>5.2 Issue Bottleneck. Dynamic Instruction Reuse. Function Reuse</td>
<td>138</td>
</tr>
<tr>
<td>6. Dynamic Instruction Value Prediction And Speculation</td>
<td>159</td>
</tr>
<tr>
<td>6.1. Introduction To Value Prediction</td>
<td>159</td>
</tr>
</tbody>
</table>
6.2 Predicting Load Instructions ___________________________ 165
6.3 Generalised Value Prediction _________________________ 172
6.3.1 Computational Predictors ___________________________ 172
6.3.2 Contextual Predictors__________________________________175
6.3.3 Hybrid Predictors______________________________________183
6.4 The Value Prediction Speedup. A Simple Analytical Model __ 189

7. Focalizing Dynamic Value Prediction To CPU’s Registers _195
7.1. The Register Value Predictor Concept____________________ 195
7.2 Register Value Predictors _______________________________ 197
7.3 Simulation Methodology And Experimental Results [26, 05] _ 201
7.4 Register Value Prediction Using Metapredictors [26]_________210
7.5 Metapredictors’ Simulations And Quantitative Evaluations___ 214
7.6 Conclusions And Some Further Work Ideas_______________ 219

8. Neural Networks Models With Applications In Ubiquitous Computing. Next Location Prediction___________________225
8.1. Introduction To Ubicom ______________________________ 225
8.2 A Related Concept: Autonomic Computing _______________ 226
8.3 The Next Location Prediction Problem In An Ubicom Environment ____________________________________________232
8.4 Person Movement Prediction Using Neural Networks _______ 244
8.5 Experimental Results. Analysis And Comments___________ 249
8.6 Some Conclusions ___________________________________ 263

9. Hidden Markov Models With Applications In Ubiquitous Systems. Next Location Prediction _____________________266
9.1. Applications Based On Hidden Markov Models _________ 266
9.2 Discrete Markov Processes ____________________________ 268
9.3 Hidden Markov Models Of Order 1 ______________________272
9.4 Prediction Algorithm Using A HMM Of Order 1 __________ 274
9.5 A Possible Generalization: HMM Of Order R>1 ___________278
9.6 Prediction Algorithm Using A HMM Of Order R>1 ________ 281
9.7 Methodology And Experimental Results__________________285
9.8 Some Conclusions ___________________________________ 290
The application of prediction techniques and speculative actions conquer many field in computer science. Excellent prediction techniques are crucial to speed up non-parallel computations by speculative execution. Most well-known and most intensely investigated are branch prediction techniques in processor architecture that allow to speculatively fetch, decode, issue, and execute instructions before the branch direction is determined. In case of a misprediction many processor cycles are lost before all instructions on the wrong speculation path are squashed from the pipeline and the pipeline is reloaded with instructions of the correct branch path. Fortunately branch prediction techniques are so well developed that mispredictions rarely occur. The performance of current superscalar processors would be magnitudes less without branch prediction techniques with excellent prediction accuracy. However, even today, a small increase in branch prediction accuracy significantly speeds up a processor’s performance. Therefore branch prediction is still a hot and vivid research topic.

Besides branch prediction researchers started in the 90ies to apply the same prediction techniques in different flavors to value prediction like load address prediction, load value prediction, and even register value prediction. The precondition for these techniques is the observation that many computations are repeated with the same values and that it might make sense to predict these values and perform loads and computations speculatively with speculative values. The observation electrified many researchers in processor architecture and led to vivid research up to the current time. However, except for speculative loads none of these techniques has made it into real processors.

Most of today’s branch prediction techniques are based on special kinds of Markov predictors called two-level adaptive schemes. Recent research successfully implements also neural network based techniques, in particular the perceptron predictors. Prof. Vințan of University “Lucian Blaga” in Sibiu, Romania, paved the way for neural network predictors by his 1999 paper on the use of neural networks in dynamic branch prediction and is one of the leading scientists in this domain. Neural network predictors can outperform the two-level adaptive schemes, but are harder to implement.
Vivid research in this direction tries to find solutions to the implementation problem.

Prediction is commonly used in many other application areas as e.g. in video streaming, speech recognition, hand-written character recognition, and in network routing. Research is also performed in prediction of coherence messages in directory-based multiprocessor cache protocols and in ubiquitous computing as context and next location prediction.

Prof. Vințan is one of the rare European scientists working in the research field of prediction techniques in computer architecture, but aiming also at „cross-fertilisation between computer architecture and other computer science fields”. Our common research interests concern prediction techniques in ubiquitous computing. We investigate methods to predict the next context (working environment) of a handheld or wearable computer in order to support and enhance human interaction with the surrounding environment. The investigated methods are neural networks, Bayes’ nets, Markov chains and the adaptation of branch prediction to context prediction. Moreover, Prof. Vințan performed theoretical investigations that show that prediction is a specific instance of the general Time Series Prediction problem.

This book exhaustively covers all kinds of branch prediction techniques and also next context prediction. It is the most comprehensive book written so far on prediction techniques in computer science. It will be mandatory as introduction as well as reference book for all researchers working on prediction techniques in computer science.

Prof. Dr. Theo Ungerer,
Chair of Systems and Networking Institute of Computer Science,
University of Augsburg, Germany
1. INTRODUCTION

This book is focused on the domain of Instruction Level Parallelism (ILP) advanced microarchitectures, especially on prediction (Latin origins: prae = before, dicere = to say) techniques implemented in order to permit speculative instructions’ processing. The essential characteristic of the domain consists in understanding and optimizing the complex architecture’s hardware-software interface. The base idea is that the hardware machine must be built for optimal execution of software applications usually written in High Level Languages (HLL), as diverse as possible (general-purpose architectures) or very dedicated (embedded architectures). Any such investigation finally leads to direct and subtle links between the HLL language or application and respectively the hardware architecture. The analysis and research instruments that must be developed are complex (execution-driven and trace-driven simulators, debuggers, compilers, software schedulers, etc.). At the level of scientific achievements and currently spent sums worldwide, the domain of ILP research is situated in the first 2-3 within computer science and engineering. In the last 20 years the computer architectures recorded important progresses and the microprocessors, representing their brain, are the main responsible for this high performance gain. The evolution of the microprocessors’ performance was exponential in the last 15 years, with an approximate rate of 55-60% per year. From Intel 486 to Intel Pentium IV the clock frequency increased about 50 times and the performance about 75 times. But the complexity of these actual microprocessors is very huge and becomes harder to be mastered (about one billion transistors per chip in 2006). According to the specialists’ opinion, about 65% of this performance increasement is due to architectural progresses and only 35% to technological progresses. Consequently, this book, containing some research “adventures”, proposes to contribute to the new architectural paradigm of the next commercial microprocessor generation, the fourth, which probably will appear on the market in the next 5-6 years.

The understanding and optimization of the program’s processing mode on the hardware machine is essential. Unfortunately, at this time, there is a semantic gap between the software systems and applications (object oriented programming, intelligent agents, information recovery, semantic
web, data mining, pattern recognition etc.) and the more rudimental hardware microarchitectures, which are still tributary to the von Neumann’s model. We would remember in this sense only that the object program loses through compilation many of the HLL and software application’s semantics, with negative performance repercussions. This microarchitectural processing loses many contacts with the real HLL application, being reduced to the execution of the machine-level instructions. Consequently, the international scientific community understood that the progresses of this domain are essential in computer engineering and there are numerous well-financed research programs. The domain of advanced microarchitectures is more dynamic in the last years due to the formidable necessities of the embedded systems and the enormous variety of their applications. In this sense, the advanced architectural ideas from the microprocessors’ domain migrate more and more into the embedded systems (the pipelining and superscalarity in 16 and 32 bit microcontrollers, the static instruction scheduling and SIMD-vector processing in multimedia processors, etc.). Particularly, branch prediction and other prediction techniques implemented in advanced architectures in order to reduce the critical path’s limitation are very important for improving actual microprocessors’ performance. Even a percent growth in prediction accuracy involves several percents improvement in the global performance (Instructions Per Cycle). Interesting to point out, all these micro-architectural prediction techniques are particular cases of some general prediction methods already implemented in many computer science intelligent applications. This was the reason that determines me to present during the last two chapters of this book some applications in context prediction in ubiquitous computing area, that are generalizing the branch predictors and value predictors analyzed during the previous chapters. These facts have determined huge financings of the academic research in the last 4 years, especially through FP6/FP7 programs in the European Union (e.g. the research projects: High Performance Embedded Architectures and Compilers- HiPEAC, see www.hipeac.net, FP6 STREP, SARC IP, ARTEMIS Technology Platform, etc.) respectively through the NSF, NASA, etc. organizations in the USA. In these institutionalized researches, beside academic partners also participate actively companies with direct interest such like Intel, IBM, Apple, Motorola, Compaq, HP, AMD, ARM (UK), Philips – now NXP Semiconductors, ST Microelectronics, Siemens, Kaiser (Italy), ACE-Dutch, Infineon, Simula, Xyratex, etc. These companies finance significantly researches in the domain of compilers, microprocessors and high performance microcontrollers. Also, important world-level research institutes are cooperating in these scientific projects (IBM TJ Watson
Research Center, Intel Research Microarchitecture Labs, Bell Labs, INRIA Futurs-France, CNRS-France, FORTH-Greece, etc.). In order to improve branch prediction research, very necessary in developing new microprocessors by companies, Intel Co. developed a very important world-level contests, called Branch Prediction World Championship, involving each year tens of professors, researchers, PhD students, computer engineers and architects, etc., from both academia and industry (see http://www.microarch.org/micro39/). It must be mentioned that, in my opinion, the significant architectural progresses came mostly from the academy, more available for fundamental researches in instruction and data processing.

In order to give an idea related to the domain’s axiology we further present only a few prestigious journals and conferences dedicated to computer microarchitectures having periodical apparition where researchers publish the most significant results of the domain. Journals (ISI Thomson Philadelphia): Journal of Instruction Level Parallelism; IEEE Transactions on Computers; IEEE Micro; ACM SIGARCH Computer Architecture News; ACM Transactions on Architecture and Code Optimization; ACM Transactions on Computer Systems; IEE Proceedings. Computers and Digital Techniques; Journal of Systems Architecture, Elsevier; HiPEAC Journal, Lecture Notes in Computer Science Springer-Verlag series, etc. Conferences: Annual International Symposium on Computer Architecture (ISCA); Annual International Symposium on Microarchitecture (MICRO); EuroPar; The JILP Championship Branch Predictors; International Symposium on High Performance Computer Architecture (HPCA); International Symposium on High Performance Computing (ISHPC); High Performance Embedded Architectures and Compilers (HiPEAC); EuroMicro; International Conference on Parallel Architectures and Compilation Techniques (PACT); ASPLOS etc. As the reader can see further, I have used some of these top reviews and conference proceedings for writing this book.

The present-day out of order issue superscalar microprocessor model (dynamic scheduling processors) is implemented as a speculative microarchitecture that actually fetches, issues and executes instructions based on branch prediction using R. Tomasulo’s or closely related algorithms and a special data structure called ReOrder Buffer (ROB). This model extends the idea of instructions dynamic scheduling by introducing the possibility of speculative execution. The hardware that implements Tomasulo’s algorithm 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 updates that might modify the CPU’s logical state. When the instruction is no longer speculative, thus after its write-back stage, it updates the logical registers or data memory; this last phase is called instruction’s *commit*. Separating the results’ bypassing from instructions’ completion makes possible to avoid imprecise exceptions during the out-of-order instructions’ processing.

Adding this last *commit* phase to the instruction execution sequence, an additional set of hardware buffers is required, which hold the instructions’ results that have finished execution but have not yet committed. The Reorder Buffer structure implements these buffers. The Reorder Buffer is also used to pass the results of the out of order executed instructions through the reservation stations. The reservation stations buffer instructions only between the time they issue and the time they begin execution. In this architecture the ROB provides the register renaming function, as we’ll further presented. Figure 1.1 shows the hardware structure of the processor including the ROB and Figure 1.2 presents ROB’s structure.

![Diagram](image)

*Figure 1.1. A typical out-of-order superscalar microarchitecture*
Each reservation station (SR in Figure 1.1) has the following fields:

- **Op** – instruction’s operation code (opcode);
- **Q_j, Q_k** – the ROB entries that will store the source operands, a value of zero (NULL) indicating that the source operand is already available in V_j, V_k, or that it is unnecessary; we called these Q fields “potential operands”.
- **V_j, V_k** – the values of the instruction’s source operands; for loads and stores the V_j field is used to hold the address’ offset; we called these V fields “actual operands”.
- **Addr** – holds the memory effective address for loads or stores;
- **Dest** – supply the corresponding ROB entry’s number representing the destination for the result produced by the execution unit.
- **Busy** – indicates if a reservation station (SR) is available or not.

The register file has also a field Q_i indicating the number of the ROB entry that contains the operation whose result should be stored into the register. Excepting instructions’ fetch, the other six phases involved in pipeline instruction processing are the following:

1. **Dispatch** – get the next instruction from the instruction’s queue. If all reservation stations are full or the ROB is full (for example due to a Load with miss in level two cache being in the ROB’s head), 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. Here is also implemented a routing algorithm to dispatch the instruction to the right
reservation station (attached to the right execution unit). The Busy bit of the allocated reservation station is set and the Ready field of the ROB entry is reset. ROB is associatively searched in all its Dest fields using the source registers’ name. In the case of multiple hits, it is considered the last entry 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 fields \( V_j / V_k \). If the operand value is not available (Ready=0), the number of ROB’s entry attached to the operand is written into the reservation station fields \( Q_j / Q_k \). In the case of a miss in the ROB the operand value is written from the register set into the reservation station fields \( V_j / V_k \). The number of ROB’s entry allocated for the value of the result is sent into the reservation station’s Dest field. The destination register number is written into the Dest field of the ROB entry.

2. Issue – if an operand is not yet available, the common data bus (CDB) is monitored and when the operand is available on the CDB it is placed into the corresponding reservation station. When both source operands are available, the instruction is issued to the appropriate execution unit. By delaying instruction’s execution until their source operands are available, RAW dependences are detected.

3. Execute – the corresponding functional unit executes the ALU instructions. In the case of loads and stores in this stage is calculated the data memory effective address. In the case of a taken branch, usually it is calculated the branch’s target address.

4. Memory – active only for load and store instructions. During this stage the result is written in the data-memory location (store) or loaded from the data-memory location into the destination register (store). The data-memory address was calculated during the previous stage.

5. Writeback – when the result is available, it is put on the Common Data Bus (together with the ROB’s entry number indicated by the reservation station’s Dest field) and from there into the Value field of the corresponding ROB entry (its Ready field is set to 1). The Busy field of the corresponding reservation station is reset. The result is also written into the fields \( V_j / V_k \) of all those 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 the data-value 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 and its result is available (Ready=1). In this case, the
result is written from the *Val* field of the ROB’s entry into the destination register indicated by the *Dest* field of the ROB’s entry or into a data-memory location and, after that, the instruction is squashed from the ROB. Thus, the in order *commit* is guaranteed by the in order *dispatch*, while the *issue*, *execute* and *writeback* stages can be processed out of order. When an incorrect predicted branch reaches the head of the ROB, the ROB is emptied and the execution is restarted with the correct successor of the branch.

As it can be observed, in the case of speculative architectures it is very important *when* is performed the updating. Using the ROB, speculative executions are possible because the register file or data memory are updated with the result of an instruction only when that instruction is no longer speculative. ROB capacity is largely determined by how many instructions may be executed in parallel between *Dispatch* and *Commit*. Of course branch prediction is crucial for exploiting instruction level parallelism between multiple basic-blocks.

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’s 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 (the value is ready). ROB completely replaces the store buffers. ROB is usually implemented as a circular FIFO queue having associative search facilities. For a better understanding of the ROB functions we are analyzing the following problems:

1. What is happen in the `<Tail_ROB>` after the current instruction decode?

If the `<Tail_ROB>` entry is free it will be allocated to the decoded instruction. After this ROB’s *Ready* field will be reset on 0. In the *Dest* field it will be written the destination register’s number.

2. When an ALU instruction will be introduced into a Reservation Station, what might be written in its *Q* fields?

In the *Q_i* and *Q_k* fields it will be written the ROB locations’ indexes that will generate operand’s value (after associative searches
Opcode[Source_Reg1,2]=Dest[ROB]?). Obviously, if an operand is available its value will be introduced in the reservation station’s V field. In this case the corresponding Q field will contain a NULL value. In the reservation station’s Dest fields it will be introduced ROB location’s index that was allocated to that concrete instruction (associative search Opcode[Dest_Reg]=Dest[ROB]?). If yes, k→Dest[SR]; the corresponding ROB entry’s index is noted with k). At the end of the execution’s stage the functional execution unit will write the result in ROB at the address Dest. Now, the question is: what will happen if after these associative searches we’ll obtain multiple hits in ROB? Answer: We’ll consider the last hit location from the ROB. Why? Because the instructions were in order allocated in the ROB during the decode stage. But if one search is with miss? Answer: Then the register file will be accessed.

3. What happen in ROB during the WriteBack stage?

If the instruction produced a result it is written in ROB’s Value field. Produced Data-Value→ROB at address Dest[SR](k) AND Ready=1. The result’s value is send also to the reservation stations (through the fields Vj/Vk). In the case of a Store, the value that must be written in the Commit stage is written in ROB’s Value field.

4. What conditions must be fulfilled to start the Commit phase?

Commit stage starts only if the instruction is already in the WriteBack stage and the corresponding ROB entry is in <Head_ROB> (in order commit). More precisely, Commit is doing if: <Instr.→Head_ROB> AND <Ready=1> AND <No_Exception> (If Exception=1 → Blocking writes in register file)

5. Considering Dispatch, Issue, Exec, WriteBack, Commit stages, what of them are processing Out of Order. Why?

Issue, Exec, and WriteBack because the operations’ results are written only in ROB and not in the CPU’s logical registers. After an exception event, branch misprediction, etc. the processor’s context might be recovered by evicting from the ROB all the instructions processed immediately after the exception’s event. The instruction’s result is written into the register file or data memory only during Commit stage. (Also see below the Commit phase’s comments)
6. Why Commit phase is In Order processed?

Its in order processing is guaranteed by the ROB’s FIFO circular queue organization. Obviously Dispatch stage must be in order in this processing model, otherwise we can’t guarantee that Commit will be in order.

7. What is the ROB’s Dest field role?

The Dest field contains destination register’s name if the instruction really has a destination register. During Commit stage the value from ROB is written in the register file at the address Dest. After Dispatch it is associatively searched in ROB if Source_Reg=Dest[ROB].

As an alternative to ROB is the dynamic register renaming technique. This is based on a set of extended physical registers instead of using a ROB and reservation stations. During the Dispatch phase the destination register is renamed with a free physical register belonging to the extended set of registers. In this way, WAR (Write After Read) and WAW (Write After Write) instructions data dependences are avoided. After Commit stage this register become a logical register and the physical register becomes free (if there aren’t instructions that use it as a source). The logic-physic conversion is done through a simple mapping table. As a register renaming advantage, the in order processes are simplified. As an alternative to this register de-allocation process, the CPU has to wait for de-allocation until another instruction will modify that register. In this case a register might be allocated more that it is strictly necessary (MIPS R10000 case). MIPS R10000/120000, Alpha 21264, Pentium III, IV implemented register renaming technique, adding up to 80 extended physical registers.

However, in the last 3-4 years there are some opinions considering that future processors must avoid a ROB centric design and move toward a checkpoint-oriented processing model. Check pointing means saving the CPU’s state at appropriate points of execution in order to repair the processor to a known previous architectural state. ROB limits the microarchitecture’s scalability, it does not scale efficiently as instruction window size increases. If an uncompleted instruction reaches the ROB’s head, it blocks the ROB and, in the nearest time, also the instructions’ issue process (because the ROB would be full). The checkpointing key idea is that sometimes reconstructing CPU’s logical state is more efficient than explicitly storing state like ROB-based approaches that record states every instruction. Checkpointing records state only at some selected execution
points (miss-predicted branches, L2 cache miss Loads – the so called memory wall problem, etc.) and will regenerate state for individual instructions only if it is necessary. Researchers show that such an approach is scalable and more efficient comparing with centralized ROB models. This solution will permit thousands of in-flight instructions, as Professor Mateo Valero’s research group from UPC Barcelona already proposed (Kilo-Instruction Processors Project). The main challenge is to design such an architecture without so much resource enlargements (instruction queues, load/store queues, physical register file, etc.) and therefore permitting a high clock frequency.

As the average instruction issue rate and depth of the pipeline in multiple-instruction-issue processors increase, accurate dynamic branch prediction becomes more and more essential. Taking into account that present-day microprocessors have a branch delay slot of one to several cycles (depending on the pipeline’s depth) and the instruction issue rate is about 6 instructions/cycle, while a basic block contains at average 6-7 instructions in non-numerical applications, the branch prediction necessity becomes obvious. In Chapter 2 we presented and analyzed some advanced neural branch predictors. In complete contrast to earlier work in this field, in 1999 we first explore and publish at the IJCNN Conference held in Washington DC the possibility of using neural networks to dynamically predict branch outcomes. This chapter starts by presenting the dynamic neural branch prediction genesis. One of our main research objectives was to use neural networks in order to identify new correlations that can be exploited by branch predictors. An important advantage is that, in contrast with conventional branch prediction, neural branch prediction can exploit deeper correlations at linear complexities rather than exponential complexities as the classical Two Level Adaptive schemes involve. We also wished to determine whether more accurate branch prediction is possible and to gain a deeper understanding of the underlying prediction mechanisms. These ideas were further efficiently developed by Dr. D. Jiménez from Rutgers University, USA, and other researchers, both from academy or industry (one of the Intel IA-64 Itanium’s simulator already contains a neural branch predictor!). Therefore we presented and analyzed in details the perceptron branch predictor and the fast-path perceptron branch predictor, both outperforming the classic branch predictors and feasible to be implemented in hardware. In the same chapter we also presented a genetic branch prediction approach and some other neural useful approaches in Computer Architecture, pointing out the importance of cross-fertilization between this field and other computer science domains.
In Chapter 3 we are focusing on finding and solving difficult predictable branches, meaning dynamic branches that are not predictable with the correlation information used by the actual prediction methods (local branch history, global history, and path). More precisely, we initially considered that a branch in a certain unbiased dynamic context would be unpredictable and further we showed through simulation that this fact is true. Focalising on these branches with high entropy – in order to further design some efficient specific predictors for them by using more suitable context information – the overall prediction accuracy should increase with some percents, that would be quite remarkable. More precisely we proved that a branch in a certain dynamic context is difficult predictable if it is unbiased and its outcomes are quasi-randomly (anyway non-deterministic) shuffled. Based on laborious simulations we showed that the percentages of difficult branches are quite significant (at average between 6% and 24%, depending on the different used contexts and their lengths), giving a new research challenge and a useful niche for further research. Even after cascading branches through a series of prediction contexts it remains a significant number of difficult-to-predict branches. Computer Architects cannot therefore continue to expect a prediction accuracy improvement with conventional predictors and thus alternative approaches are necessary. They need some specific efficient predictors using some new more relevant prediction information. In this sense we believe that a solution might be “branch prediction through value prediction”, as some researchers already proposed during the last years. In our used SimpleScalar (PISA-MIPS) architecture the branch’s condition is represented by the difference between two source operands. Obviously, this can be positive, negative or zero. Predicting this difference (because branch’s sources aren’t usually available at the prediction time) involve predicting the branch itself. Therefore branch condition’s prediction represents a value prediction problem and this could be approached based on the branch condition’s histories (a string of integer values). Based on the history condition’s values, the prediction problem is: what’s the next condition’s sign (positive, negative or zero) rather than its concrete value. This approach might be a useful one in our unbiased branches problem.

In our 4th Chapter we suggest an alternative approach to the present – day dynamic branch prediction schemes. Instead of predicting branches based on history information, we propose to pre - calculate the branch’s outcome. A pre - calculated branch (PCB) determines the outcome of a branch as soon as both of the branch’s operands (used for calculating the condition) are
known. This occurs when the last operand of the branch’s instruction is produced at execution. The instruction that produced the last operand may trigger supplementary branch condition estimation and, after this operation, it correspondingly computes the branch’s outcome (Taken/Not Taken). Similarly to branch history prediction schemes, branch information is cached into a specific table. Through this method, excepting the first one, every instance of a branch can be computed and therefore correctly “predicted” (in fact, computed), before its issue. In our PCB study we used MIPS-I microprocessor’s Instruction Set Architecture (ISA) since a branch instruction has addressing modes with two register operands and no immediate operands. The improvement in “prediction accuracy” brought by this scheme must be paid some way in timing, complexity and costs. Unfortunately, if the PCB’s timing can be partially reduced by hiding it through some overlapping processes, the structural costs cannot be reduced so easy. So, a PCB prediction scheme is about 105 KBits complex comparing with a full associative BTB scheme having only 34 KBits complexity at the same number (1024) of tables’ entries.

In Chapter 5 we are focusing on two fundamental limitations in the present-day superscalar paradigm: fetch bottleneck (Flynn’s bottleneck) and issue bottleneck. Fetch bottleneck means that the CPU can’t fetch more than a basic-block per cycle due to the final branch instruction. (If this branch is a taken one, the subsequent instructions will not be processed.) Therefore, fetch bottleneck is a limitation due to the programs’ structure rather than the microarchitecture’s structure. A solution to this problem consists in a Trace-Cache structure, representing a cache memory containing successive basic-blocks, as the processor previously executed them. In case of a basic-blocks sequence reuse, these basic-blocks will be fetched from the Trace-Cache, using a multiple branch predictor and solving fetch bottleneck. How to reduce Trace-Cache information redundancy is still an open problem. Another fundamental limitation in instructions’ processing is given by the so-called program’s critical path (issue bottleneck, data-flow bottleneck, critical path bottleneck). The critical path is defined as the longest chain of dependent RAW (Read after Write) instructions, thus having the longest latency. In some recent prestigious conferences dedicated to advanced microarchitectures and systems, the researchers proposed two new techniques in order to solve this bottleneck: dynamic instruction reuse and respectively dynamic instruction value prediction. These techniques are both revolutionary (critical path’s compression) and evolutionary (compatibility with actual superscalar paradigm). The idea of dynamic instruction reuse (DIR) was first introduced by Dr. A. Sodani and Prof. G.
Sohi from Wisconsin Madison University, USA, in 1997, at The International Symposia on Computer Architecture (ISCA). The DIR basic idea is the following: if an instruction or a dependent instructions chain is repeatedly executed with the same input values, its output values will be the same. In this way the number of dynamic executed instructions is reduced and the critical path might be compressed. It is shown that reusability is frequently due to the fact that programs are written in a compact manner (loops, recursion, etc.) and in a generic manner (processed data structures). Less than 20% of static instructions are generating over 90% of the dynamic instructions and about 26% of the dynamic instructions are reusable. Instructions’ reuse is efficiently exploited through Reuse Buffers that are integrated into a superscalar architecture. An open problem is how to reuse in the same microarchitecture different entities, from dynamic instructions to basic-blocks, traces and HLL functions (procedures). All these problems are presented and analyzed in this chapter.

In Chapter 6 we are presenting Dynamic Instruction Value Prediction (VP), a relatively recently proposed processing technique independently developed by four research groups (AMD NexGen – W. Widgen & A. Sowatski, Haifa University – F. Gabbay & A. Mendelson, Carnegie Melon University – M. Lipasti et al., Wisconsin-Madison University – Y. Sazeides & J. Smith). Just as an example, in 2004 there were published about 70 papers at the most prestigious conferences focused on VP. Its main aim is to early predict the instructions’ results, during their fetch or decode pipeline stages, and, therefore, to speculatively execute the instruction flow. The predicted value can then be used as an input to some subsequent RAW dependent instructions so that they can execute earlier. If the prediction is incorrect, recovery mechanism must be employed to squash speculative results and re-execute all instructions that already used the incorrectly predicted value. VP is based on the so-called value locality statistic principle. Value Locality means the statistical probability (frequency) to produce a value belonging to the $N$ previous value instruction’s instances. More generally, value locality consists in a dynamic correlation between a resource name (register, memory location, I/O port, etc.) and the value attached momentarily to that resource. Value locality is exploited through instruction value prediction followed by speculative execution of the program’s critical path. There are developed computational predictors (incremental, for non-repetitive values) and contextual predictors (Markov based, for repetitive instructions). An important challenge for VP technique is to compress the program’s dynamic critical path and therefore to solve Issue Bottleneck. Otherwise, if the predicted instructions do not belong to the critical path, the technique is not
efficient in reducing critical path’s limitation. Therefore, the VP technique tries to avoid a fundamental limitation in the present day computing model, the Read After Write (RAW) data dependences, thus the intrinsic program’s sequential execution. Anyway, until this moment (October 2006) no real microprocessors incorporate value prediction, excepting speculative Load instructions. Even if, together with dynamic instruction reuse, we believe they are powerful feasible techniques, only time will tell if their performance potential will compensate the additional complexity and power consumption required for effective implementation.

**Chapter 7**’s main aim consists in focalizing dynamic value prediction to CPU’s context rather than on the current fetched/processed instruction, as we already proceed in the previous chapter. The idea of attaching a value predictor for each CPU register – we called it *register centric prediction* - instead of instruction or memory centric prediction was proposed by my research group (2005) and could involve new architectural techniques for improving performance and reducing the hardware cost of speculative microarchitectures. We 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 on these registers) leading to the conclusion that value prediction might be successful applied at least on these favorable registers. By implementing the value prediction schemes presented in the previous chapter, centered now on CPU’s registers rather than on program’s instructions, will be mainly decreased the hardware complexity, power consumptions (both static and dynamic) and costs. However, there are some disadvantages. Addressing the prediction tables with instructions’ destination register name (during the decode stage) instead with the Program Counter, will cause some interference. As we proved in this chapter, with a sufficiently large history, a hybrid predictor could eliminate this problem and achieve high prediction accuracy.

Finally, in **Chapters 8 and 9** after a short introduction in the ubiquitous computing area and in the related autonomic computing concept we are presenting some joint researches between Professor Theo Ungerer’s Research Group from Augsburg University and author’s Research Group from University “L. Blaga” of Sibiu in the next location prediction problem. The concrete problem is to manage some smart doorplates that are able to direct visitors to the current location of an office owner based on a location-tracking system and predict if the office owner is soon coming back or not. In order to solve this challenging problem we developed and simulated
some predictors based on Markov Models, Hidden Markov Models (HMM) and Neural Networks. Through this chapter we want to emphasize that prediction techniques applications aren’t related only to microarchitectures. In our opinion prediction methods are requested for each pro-active intelligent application. This chapter shows that the prediction techniques used in microarchitectures and presented in the previous chapters might be generalized to other IT applications. Some interesting quantitative and qualitative obtained results are presented during these two chapters.

This book approaches all these important scientific challenges integrated into three distinct views:

1. **Hardware – software**, meaning optimizing hardware structures based on software applications characteristics.
2. Using **various advanced research tools** like simple and hidden Markov processes, neural networks, genetic algorithms, simulation and benchmarking, advanced processing algorithms, etc.
3. **Improving multiple instruction issue concepts** through dynamic instruction reuse, value prediction and speculative instructions processing.

As a prerequisite for the book’s in-depth understanding it is necessary for the reader to already know the fundamental problems in Computer Architecture, as they are usually presented in a basic academic course or textbook (instruction set architecture, pipelining, memory hierarchy, dynamic/static instruction scheduling, etc.). It is important to point out that this book presents only some computer engineering challenges where the author has his own research experience. As the reader can see, I wrote this book exclusively based on my own research and other research papers. As a consequence, each chapter contains many research investigations, developed by the author and his small research group in Advanced Computer Architecture & Processing Systems (ACAPS) belonging to “Lucian Blaga” University of Sibiu (http://acaps.ulbsibiu.ro/research.php), including some original methodologies and approaches. Of course, important worldwide researchers in Computer Engineering developed other researches presented by me in this book. As far as I know it is one of the first textbooks presenting together some advanced computing architecture topics like neural branch prediction, trace processors, dynamic instruction reuse, dynamic value prediction, neural and HMM predictors with applications, etc. From this point of view, the author hopes that it might be like a book of (technological) adventures, and
therefore interesting to be read (studied?) by computer engineers and researchers, from both academia and industry, students in Computer Science and Engineering, Informatics, Electrical Engineering and Electronics, Telecommunications, Automatic Control, etc. More than this, the book can be really useful for all “very amateurs” in Computers. Of course there are many other hot topics in computer architecture not analyzed in this book, due to our limited research experience in these fields like: multiprocessors on a chip, simultaneous multithreading processors, vector processing (SIMD), domain specific microarchitectures, program and power consumption optimizations, etc. Fortunately, there are many very good textbooks, especially Americans but not only, covering these interesting actual topics. In fact this book continues my Computer Architecture book series: first, two books focused on ILP architectures published by Romanian Academy Publishing House and Technical Publishing House Bucharest in 2000, second, my book about prediction and speculation in advanced microarchitectures published in 2002 by Matrix Rom Bucharest, and third, in 2003, the book written together with Dr. Adrian Florea presenting techniques for microarchitectures’ simulation and optimization, published by Matrix Rom, too. (This last one was distinguished with “Tudor Tănăsescu” Romanian Academy Prize, in 2005). This 5th book is the first one that I wrote in English language.

As it can be observed the proposed research domains are mainly conceptual, architectural rather than technological. (“Hardware engineers are concept-limited, not component-limited,” said E. A. Feigenbaum many years ago.) As a consequence they represent an excellent opportunity also for fundamental technological academic research. In this sense, it is obvious that academic teaching in computer science and engineering needs also significant research experience even in fields considered especially difficult like networks, operating systems, compilers and computer architecture. Especially the advanced computer architecture and compilers are domains in which research can be and must be done all over the world, not only in technologically advanced countries. At this moment, in our opinion, the main research in these elitist fields is done in countries belonging to West-Europe, Asia-Pacific, India and North America. The research tools (computer science methods and algorithms, cross-compilers, schedulers, execution-driven and trace-driven simulators, debuggers, benchmarks, etc.) must be mainly developed by the research teams in order to improve the digital processing paradigm.
I dared to write this book in English despite I know that my English is quite poor. The main reason for doing this is that an English written book might be useful for many people comparing with a book written in my Romanian native language. This is extremely important, especially today when we are living in a global society based on Information Technology and Communications. Anyway, I kindly ask the readers to excuse especially my language mistakes but also some possible technical errors. I would be faithful to all the persons that will inform me about these errors through my e-mail address: Lucian.Vintan@ulbsibiu.ro

Acknowledgments

I would like to express my gratitude to Professor Theo Ungerer from Augsburg University, Germany, for giving me the chance to work in 2003, for a two months period, in his research group, for his useful advises and for writing the book’s Foreword. All the researches presented in Chapters 8 and 9 were developed together with him and with Dr. Jan Petzold. I would like to thank Dr. Gordon Steven and Dr. Colin Egan from University of Hertfordshire, UK, for their continuous support during a scientific collaboration of over 10 years. I am also grateful to Dr. Mercè Griera-i-Fisa from Information Society Directorate-General of European Commission, for giving me the opportunity to be a reviewer of the FP6 NoE HiPEAC Research Program (2004-2008, Coordinator: Professor Mateo Valero from University Politecnica de Catalunya), developed by top-level researchers. It was a good chance for me to learn a lot about the newest European researches in Advanced Computer Architecture and Compilers from many prestigious worldwide professors. My full recognition to my nearest collaborators, Dr. Adrian Florea and my PhD student Arpad Gellert, from “L. Blaga” University of Sibiu, for the generous support that they provided me in developing this book and more than this. Both of them were very actively involved in doing a lot of researches presented during the book’s chapters. Another of my PhD students, Marius Oancea from the same university, also contributed in the unbiased branch research, presented during the third book’s chapter. Last but not least, my faithful and love to my wife, Dr. Maria-“Daniela” Vintan, and to my son Radu, for their understanding, encouragement and patience during the writing of the book.
2. ADVANCED BRANCH PREDICTION METHODS

2.1. DYNAMIC NEURAL BRANCH PREDICTION. GENESIS

As the average instruction issue rate and depth of the pipeline in multiple-instruction-issue (MII) processors increase, accurate dynamic branch prediction becomes more and more essential. Taking into account that present-day microprocessors have a branch delay slot of one to several (depending on the pipeline’s depth) cycles and the instruction issue rate is about 6 instructions/cycle, while a basic block contains at average 5-7 instructions, the branch prediction necessity becomes obvious. We can’t imagine a MII processor working without branch prediction. Very high prediction accuracy is required because an increasing number of instructions are lost before a branch misprediction can be corrected. As a result even a misprediction rate of a few percent involves a substantial performance loss. In the figure 2.1, based on an analytical model developed in [Vin2000], we present the relationship between average issue rate (IR) and branch prediction accuracy (Ap) in a processor having the superscalar factor equal to 4, considering some other realistic parameters.

If branch prediction is 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, all of the above must be completed in time to fetch instructions from the branch target address without interrupting the flow of new instructions to the processor pipeline.
The first efficient approach in hardware (dynamic) branch prediction consists in Branch Target Buffer (BTB) structures [Vin2000]. BTB is a small associative memory, integrated on chip, that retains the addresses of recently executed branches, their targets and optionally other information (e.g. target opcode). Due to some intrinsic limitations, BTB's accuracies are limited on some benchmarks having unpropitious characteristics (e.g. correlated branches). A classic Branch Target Cache (BTC) [Hen03] achieves these objectives by holding the following information for previously executed branches: the address of the branch instruction (its Program Counter – PC), the branch target address and information on the previous outcomes of the branch (local history). Branches are then predicted by using the PC address to access the BTC in parallel with the normal instruction fetch process. As a result each branch is predicted while the branch instruction itself is being fetched from the instruction cache. Whenever a branch is detected and predicted as taken, the appropriate branch target is then available at the end of the instruction fetch cycle, and instructions can be fetched from the branch target in the cycle immediately after the branch itself is fetched. Straightforward markovian prediction mechanisms based on the previous history of each branch give a prediction accuracy of around 80 to 95% [Hen03]. This success rate proved adequate for scalar processors, but is generally regarded as inadequate for MII architectures.

The requirement for higher branch prediction accuracy in MII systems and the availability of additional silicon area led to a dramatic breakthrough in the early 90s with branch prediction success rates as high as 97% [Yeh92].
being reported on some SPEC benchmarks (details on simulation methodology and benchmarking could be founded in [Flo03]). These high success rates were obtained using a new set of prediction techniques known collectively as Two-Level Adaptive Branch Prediction that were developed independently by Professor Yale Patt’s group at the University of Michigan [Yeh91] and respectively by Pan, So and Rahmeh from IBM and the University of Texas [Pan92]. Two-Level Adaptive Branch Prediction uses two levels of branch history information to make a branch prediction. The first level consists of a History Register (HR) that records the outcome of the last $k$ branches encountered. The HR may be a single global register, HRg, that records the outcome of last $k$ branches executed in the dynamic instruction stream or one of multiple local history registers, HRI, that record the last $k$ outcomes of each branch. The second level of the predictor, known as the Pattern History Table (PHT) records the behaviour of a branch during previous occurrences of the first level predictor. It consists of an array of two-bit saturating counters, one for each possible entry in the HR. $2^k$ entries are therefore required if a global PHT is provided, or many times this number if a separate HR and therefore PHT is provided for each branch. The prediction of the branch ($P$) is a function ($f$) of the actual prediction automaton state $S_t$.

$$P = f(S_t)$$

After the branch is resolved, HR is shifted left and the prediction automaton state becomes $S_{t+1}$.

$$S_{t+1} = g(S_t, B_t)$$

Where $g$ represents the automaton's transition function and $B_t$ represents the behaviour of the last branch encountered (taken/ not taken). A lot of interesting implementations of these correlated branch prediction schemes are known.

Although a single term is usually applied to the new predictors, this is misleading. Since the first level predictor can record either global or local branch history information, two distinct prediction techniques have in fact been developed. The global method exploits correlation between the outcome of a branch and the outcome of neighbouring branches that are executed immediately prior to the branch. In contrast, the local method depends on the observation that the outcome of a specific instance of a
branch is determined not simply by the past history of the branch, but also by the previous outcomes of the branch when a particular branch history was observed. Tutorials focused on dynamic branch prediction methods can be founded in [Hen03, Vin2000].

In complete contrast to earlier work, in 1999 we first explore the possibility of using neural networks to dynamically predict branch outcomes [Vin99a, Vin99c, Vin2000, Ega03]. The following paragraphs are presenting the dynamic neural branch prediction genesis, based on those cited original works. Conventional two-level branch predictors rely completely on one of only two correlation mechanisms. One of our main research objectives was therefore to use neural networks in order to identify new correlations that can be exploited by branch predictors. An important advantage is that, in contrast with conventional branch prediction, neural branch prediction can exploit deeper correlations at linear complexities rather than exponential complexities. We also wished to determine whether more accurate branch prediction is possible and to gain a greater understanding of the underlying prediction mechanisms. Finally, we hoped to design and evaluate hardware implementations of simplified neural branch predictors. Alternatively, the results of our initial research were really useful in designing more sophisticated two-level branch predictors but also new efficient dynamic neural predictors. Dynamic branch prediction with neural methods, as an alternative to the Two Level Adaptive Branch Prediction, was first introduced by L. Vintan [Vin99a], and further developed by D. Jiménez [Jim02] and other researchers [Des06, Mon05, Tar04, AKK04]. Despite the neural branch predictor’s ability to achieve very high prediction rates and to exploit deep correlations at linear costs, the associated complexity due to latency, large quantity of adder circuits, area and power are some disadvantages. Anyway, the neural methods seem to be successfully for future microprocessors taking into account that they are already implemented in a recent version of Intel’s IA-64 simulator.

Now we’ll explore the suitability of two neural networks, a Learning Vector Quantisation Network (LVQ, T. Kohonen) and a Multi-Layer Perceptron (MLP) with back-propagation learning algorithm, for branch prediction. Through trace driven simulation we demonstrate that neural predictors can achieve success rates that are better than those obtained by conventional two-level adaptive predictors. We therefore suggested that neural networks are indeed a suitable vehicle for further branch prediction research.
2.1.1. PREVIOUS WORK IN THE FIELD

Since most previous research on branch prediction has concentrated on two-level adaptive techniques [Cha95, Eve95, Lee97, McF93, Pan92; Sec95, Spr97, Yeh91, 92a, 92b, 93], it is useful to explore some of the drawbacks of two-level predictors. The main disadvantages can be found in the following areas:

- Size of PHT, having an exponential increase
- Branch interference
- Slow Initialisation
- “Hard-to-Predict” Branches

With a global predictor using a single Pattern History table (PHT), a total of $2^n$ entries are required where $n$ is the number of bits in HRg. This is the GA predictor in Patt’s classification system [Yeh92a]. To achieve a 3% misprediction rate with the SPEC benchmarks an 18-bit HRg is required [Yeh92a] giving a PHT with $2^{18}$ or 256K entries. Moreover, if a separate PHT is used for each branch, as in Patt’s GAp configuration, a far greater area must be devoted to the PHT. For example if HRl is 12 bits long and 16 bits are required to distinguish each branch, a total of $2^{28}$ entries is required. Similar high storage requirements apply to local branch predictors - PAg and PAp in Patt’s system - even though fewer local history register bits are generally required to achieve a given success rate.

The high cost of the PHT suggests that alternative configurations should be considered. One possibility is to replace the complete PHT with a conventional cache structure [Chen96, Vin99b]. Although a tag field must be added to each PHT entry, the very large size of conventional PHTs suggests that the total number of entries and therefore the total cost of the PHT could be significantly reduced. Furthermore, the impact of PHT cache misses can be minimised by adding a default prediction counter field to the conventional BTC which must still be included in all two-level predictors to furnish a target address for each branch [Ste99]. Alternatively, an entirely different approach, such as the neural branch predictors presented here, can be further investigated.

The second problem, branch interference, is a direct result of excessive PHT sizes. Because of the high cost of PHTs, researchers are reluctant to provide a separate PHT for each branch. Instead each PHT is usually shared between groups or sets of branches, giving rise to the GAs and PAs configurations in Patt’s classification. Unfortunately, prediction information is now shared
between multiple branches, leading to interference between different branches. We note that this problem can also be avoided by caching the PHT information.

The third problem is PHT initialisation. In the worst case, $2^n$ prediction counters, where $n$ is the length of the HR, must be initialised for each branch before the predictor is fully effective. Even allowing for the fact that a PHT is effectively a sparse matrix with many unused entries, this situation contrasts sharply with a classic BTC, which is fully initialised after one execution of each branch. This problem can be avoided by combining explicit PHT counter initialisation with the simple default predictor in the BTC mentioned earlier [Ste99].

Finally, some branches remain stubbornly hard-to-predict [Mud96, Vin99b]. There are two cases. The outcome of some data dependent branches is effectively random and these branches will never be accurately predicted. However, it should be possible to predict certain branches that are currently hard-to-predict more accurately by identifying new correlation mechanisms and adding them to the prediction process. We believe that neural predictors are an excellent vehicle for investigating potential new correlation mechanisms.

We have emphasised earlier that most branch prediction research is based on two closely related correlation mechanisms. Yet branch prediction is a specific example of a far more general time series prediction problem that occurs in many diverse fields of science. It is therefore surprising that there has not been more cross-fertilisation of ideas between different application areas. A notable exception is a paper by Mudge et al [Mud96], which demonstrates that all two-level adaptive predictors implement special cases of the Prediction by Partial Matching [PPM] algorithm that is widely used in data compression. Mudge uses the PPM algorithm to compute a theoretical upper bound on the accuracy of branch prediction, while Steven et al [Ste99] demonstrate how a two-level predictor can be extended to implement the PPM algorithm with a resultant reduction in the misprediction rate. Time series prediction is also an important research topic in neural networks. It therefore appears natural to look to neural networks for a further cross-fertilisation of ideas.

As far as we are aware, before our seminal papers focused on neural branch prediction, only one paper by Calder et al. [Cal95] has discussed the application of neural networks to the problem of branch prediction. Calder
is concerned entirely with static branch prediction at compile time. His predictions are therefore based entirely on information about a program’s structure that can be readily determined by a compiler. For example, a branch successor path that leads out of a loop or function is less likely to be followed than a path that remains within the loop or function. Using a neural network, Calder achieves a misprediction rate of only 20%, remarkably low for static branch prediction. Clearly, Calder was unable to feed the dynamic branch histories used by two-level predictors into his neural networks. Therefore, possibly the most useful contribution of his paper is to suggest a wide range of alternative inputs that might correlate with branch outcomes and which might therefore be usefully added to dynamic predictors.

In contrast, starting with 1999, we applied neural networks to dynamic branch prediction. Since our objective was to demonstrate that neural networks could achieve better prediction accuracy than a conventional two-level adaptive predictor, we restrict our neural network inputs to using the same dynamic global HR information as a conventional two-level predictor.

2.1.2 FIRST NEURAL DYNAMIC BRANCH PREDICTION MODELS

In this paragraph we compare the performance of a GAp two-level adaptive predictor with a Multi-Layer Perceptron (MLP). We also briefly consider the performance of a simple LVQ (Learning Vector Quantisation) neural predictor.

A Conventional GAp Predictor

The GAp predictor (Fig. 2.2) uses a k bit shift register, HRg, to record the outcome of the last k branches executed. The per-address PHT is accessed by concatenating the PC address (considered on l bits) with HRg, the global history register. Each PHT entry consists of a tag field, the branch target address and only one prediction bit. To avoid placing unnecessary restrictions on the performance of our GAp predictor, the PHT is of unlimited size.

An LVQ Neural Predictor

The first neural network we examined was an LVQ (Learning Vector Quantisation) clustering model (Fig. 2.3). Since LVQ [Gal93], like the simple perceptron network, is one of the simplest neural networks it is likely
Advanced Branch Prediction Methods

to be easier to implement in hardware than more complex neural nets. Our first developed neural branch predictor uses three input parameters: PC, HRg and HRl. Unusually, the local history register, HRl, is a function of the global history register, HRg (Fig. 2.3). A distinct local history pattern is therefore accumulated for every global history path leading to a branch.

The LVQ predictor contains two “codebook” vectors: the first vector, Vt, is associated with the branch taken event and the second, Vnt, with the not taken event. Vt is initialised to all ones and Vnt to all zeros. During the prediction process, the three input parameters are concatenated to form a single input vector, X. *Hamming* distances (HD) are then computed between X and the two codebook vectors.

\[
HD = \sum (Xi - Vi)^2
\]  

(1)

The vector with the smallest *Hamming* distance is defined as the winning vector, Vw, and it is used to predict the branch. A win for Vt therefore indicates predict taken, while a win for Vnt indicates predict not taken.

When the branch outcome is determined, the codebook vector Vw that was used to make the prediction is adjusted as follows:

\[
Vw(t + 1) = Vw(t) +/- a(t)[ X(t) - Vw(t)]
\]  

(2)

To reinforce correct predictions, the vector is incremented whenever a prediction was correct and decremented otherwise. The factor a(t) represents the learning factor (step) and is usually set to a small constant less than 0.1. In contrast, the losing vector is unchanged. The neural predictor will therefore be trained continuously as each branch is encountered. It will also be adaptive since the codebook vectors always tend to reflect the outcome of the branches most recently encountered.

**An MLP Neural Predictor**

For our main neural predictor we used a global MLP (Multilayer Perceptron) [Gal93] with a single intermediate layer (Fig. 2.4). Thus, only one MLP for all branches instead one predictor for each branch, just to rapidly evaluating the neural prediction idea. To allow for a direct comparison with a conventional GAp predictor, the PC address and global history register, HRg, were concatenated to form the input vector. The MLP then produces a true output for predict taken and a false output for predict...
not taken. The training process used the well-known backpropagation algorithm. As it is usually known, backpropagation algorithm is dedicated for learning in feedforward networks using mean squared error (MSE) and gradient descent method. It mainly consists in two steps: forward propagation step and respectively backward propagation step. The first step makes a bottom-up pass through the network to computed weighted sums and activations. The second step, starting with the outputs, makes a top-down pass through the output and intermediate cells computing gradients. Once we know the gradient we can take a small step to update the weights using a learning step. This process continues until the MSE becomes sufficiently small. After considerable experimentation we standardised on an intermediate layer, which always has four more cells than the number of bits in the input vector.

### 2.2. TRACE DRIVEN SIMULATION RESULTS

**Simulation Environment**

Our first simulation work used the Stanford integer benchmark suite, a collection of eight C programs designed to be representative of non-numeric code, while at the same time being compact. The benchmarks are computationally intensive with an average dynamic instruction count of 273,000. About 18% of the instructions are branches of which around 76% are taken. Some of the branches in these benchmarks are known to be particularly difficult to predict, harder than branches belonging to well-known SPEC benchmarks; see for example Mudges’ detailed analysis [Mud96] of the branches in quicksort benchmark.

The C Stanford benchmarks were compiled using a C compiler (GNU CC code generator) developed at the University of Hertfordshire, UK, for the HSA (Hatfield Superscalar Architecture) [Ste97]. Instruction traces were then obtained using the HSA instruction-level simulator, with each trace entry providing information on the branch address, branch type and target address. These traces were used to drive a series of trace-driven branch predictors developed at the “L. Blaga” University of Sibiu, Romania. The trace-driven simulators are highly configurable, the most important parameters being the number of HRg bits and the size of the PHT. As
output the simulators generate the overall prediction accuracy, the number of incorrect target addresses and other useful statistics.

**An LVQ Branch Predictor**

We were initially attracted to LVQ networks by their simplicity. We wished to determine whether respectable branch prediction success rates could be delivered by a simple global LVQ network that was dynamically trained after each branch prediction and used a minimal amount of branch history information [Vin99a, c].

The input vector for the neural network was constructed by concatenating the least significant bits of the PC with HRg and HRl (Fig. 2.3). Initially the values of the learning step $a(t)$ were varied between 0.1 and 0.001. Eventually the learning step’s value $a(t) = 0.01$ was standardised after it had been demonstrated that the predictor was largely insensitive to slight variations in $a(t)$.

The LVQ predictor achieved success rates ranging from 88.3% to 89.5% (Fig. 2.5). These results are highly encouraging; particularly since the LVQ predictor is dynamically trained after each prediction and the predictions are based on only 5 to 13 bits of HR information. We also simulated an equivalent two-level adaptive predictor for comparative purposes (Fig. 2.5). The conventional predictor achieved marginally better results. Nonetheless the neural predictor outperformed the conventional predictor on some individual benchmarks and, on average, less than 1.5% separated the two predictors in all configurations. Since the input vector was not tailored in any way for the neural network, we again found these initial results very encouraging. On the other hand, we compared a global LVQ predictor vs. per branch two-level adaptive predictors, therefore clearly disadvantaging LVQ!

**A Statically Trained MLP Predictor**

Our first objective was to determine whether respectable prediction results could also be obtained using an MLP branch predictor [Vin99a]. We therefore initially trained the MLP predictor off line to gain a feel for its potential. Initially the MLP’s weights were chosen in a randomly manner but all these initial weights are belonging to $[-2/l+k, 2/l+k]$, according to [Gal93]. For calculating the output’s value we used the classical sigmoid activation function (squashing function) given by the well-known expression $1/(1+\exp(-x))$. 
Static training was achieved by generating a training vector set, consisting of PC-HRg pairs, for each benchmark. First the benchmark’s instruction trace was pre-processed to determine how frequently individual branches were taken for specific values of HRg. The following fragment generated from the benchmark sort illustrates the output generated as a result of this analysis:

<table>
<thead>
<tr>
<th>Branch PC</th>
<th>HRg</th>
<th>Taken</th>
<th>Not Taken</th>
<th>Taken(%)</th>
<th>Not Taken(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>4055</td>
<td>121</td>
<td>41</td>
<td>74.69</td>
<td>25.31</td>
</tr>
<tr>
<td>68</td>
<td>3935</td>
<td>127</td>
<td>30</td>
<td>80.89</td>
<td>19.11</td>
</tr>
<tr>
<td>68</td>
<td>3453</td>
<td>17</td>
<td>138</td>
<td>10.97</td>
<td>89.03</td>
</tr>
<tr>
<td>68</td>
<td>1525</td>
<td>124</td>
<td>107</td>
<td>53.68</td>
<td>46.32</td>
</tr>
<tr>
<td>68</td>
<td>3925</td>
<td>109</td>
<td>143</td>
<td>43.25</td>
<td>56.75</td>
</tr>
<tr>
<td>68</td>
<td>1367</td>
<td>124</td>
<td>360</td>
<td>25.62</td>
<td>74.38</td>
</tr>
<tr>
<td>68</td>
<td>1373</td>
<td>210</td>
<td>234</td>
<td>47.30</td>
<td>52.70</td>
</tr>
<tr>
<td>68</td>
<td>765</td>
<td>4</td>
<td>3</td>
<td>57.14</td>
<td>42.86</td>
</tr>
<tr>
<td>68</td>
<td>3061</td>
<td>3</td>
<td>0</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>68</td>
<td>1399</td>
<td>72</td>
<td>200</td>
<td>26.47</td>
<td>73.53</td>
</tr>
<tr>
<td>68</td>
<td>1501</td>
<td>142</td>
<td>181</td>
<td>43.96</td>
<td>56.04</td>
</tr>
<tr>
<td>68</td>
<td>1909</td>
<td>126</td>
<td>196</td>
<td>39.13</td>
<td>60.87</td>
</tr>
<tr>
<td>68</td>
<td>3541</td>
<td>44</td>
<td>174</td>
<td>20.18</td>
<td>79.82</td>
</tr>
<tr>
<td>68</td>
<td>1213</td>
<td>4</td>
<td>1</td>
<td>80.00</td>
<td>20.00</td>
</tr>
</tbody>
</table>

As can be seen some of the branch outcomes are highly biased with branches being taken most of the time. In contrast, other branches are very hard to predict on the basis of the information given. For example, in the fourth line of the example, the branch is only taken 54% of the time. This input pair therefore provides virtually no useful training information. Only highly biased PC-HRg pairs, where the branch is either taken 70% of the time or not taken 70% of the time, were therefore included in the training set. The remaining PC-HRg pairs were discarded. For the above fragment, the following pairs were therefore retained for training:

<table>
<thead>
<tr>
<th>Branch PC</th>
<th>HRg</th>
<th>Taken/Not Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>4055</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>3935</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>3453</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>1367</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>3061</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>1399</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>3541</td>
<td>0</td>
</tr>
<tr>
<td>68</td>
<td>1213</td>
<td>1</td>
</tr>
</tbody>
</table>
Repeated training passes were then made using the training vectors until the mean squared error became smaller than 0.01. This training was performed separately for each benchmark. The statically trained MLP predictor was then used in a final trace-driven prediction run. To improve the success rate, the predictor was also trained dynamically, by running traces. This training consisted of a single backpropagation-learning step after each individual branch prediction.

The prediction results are summarised in Fig. 2.6 for values of HRg ranging from four to ten. For individual benchmarks the prediction accuracy ranges from 74% to 97% with an average accuracy of around 90%. Surprisingly, however, the average success rate rises only slowly from 89.0% to 89.5% as the number of bits in HRg is increased from four to ten. Table 2.1 presents the influence of the branches “polarisation degree” (PD) on the obtained prediction accuracies for three different HRg’s lengths. As it can be observed it is optimal recommended to include in the static training set PC-HRg pairs that are taken / not taken (“polarised”) about 60%-70% of time.

<table>
<thead>
<tr>
<th>PD</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>K=4</td>
<td>89.44</td>
<td>89.55</td>
<td>89.46</td>
<td>89.41</td>
</tr>
<tr>
<td>K=6</td>
<td>89.91</td>
<td>89.80</td>
<td>89.81</td>
<td>89.73</td>
</tr>
<tr>
<td>K=8</td>
<td>89.83</td>
<td>89.91</td>
<td>89.83</td>
<td>89.82</td>
</tr>
</tbody>
</table>

Table 2.1. The influence of the branches “polarisation degree” on the prediction accuracy

A Dynamically Trained MLP Predictor

While the results obtained with a statically trained MLP predictor give a very encouraging indication of the potential of an MLP predictor, we were also interested in dynamically trained branch prediction, thus, in other words, without the benefit of a static learning process. The trace-driven simulations of the previous section were therefore repeated without any prior static training of the neural network. The entire training therefore consisted of the single backpropagation pass after each individual branch prediction. Detailed results for individual benchmarks are presented in Fig. 2.7 for sizes of HRg between four and ten. Average results ranged from 88.76% with HR = 4 to 89.1% with HR = 10. The prediction accuracy therefore grows slowly as the size of HRg is increased.
The benefit of our static training developed technique can be appreciated by comparing the performance of the statically and dynamically trained MLPs (Fig. 2.8). While dynamic training performs well for small sizes of HRg, performance growths slowly as the size of HR is increased beyond four, and the dynamically trained MLP is increasingly unable to match the success rate achieved by the statically trained predictor. At average, for an HRg on 10 bits, the statically trained MLP obtains a prediction accuracy of about 90% compared with the 89% prediction accuracy obtained by the (only) dynamically trained MLP.

**MLP versus conventional GAP**

Further, we compared our MLP predictor with an equivalent classical GAP predictor. To avoid compromising the GAP predictor in any way the PHT table was of unlimited size having only one bit predictors (considering two bit predictors involves an unfair comparison between the GAP and MLP branch predictors due to the local history information used by the classical scheme). The prediction results for individual benchmarks are given Fig. 2.9. On average as the size of HRg was increased from four to ten the prediction accuracy also increased from 84.8% to 86%.

These results are compared with a statically trained MLP predictor in Fig. 2.10. Surprisingly, the MLP predictor is consistently more accurate than the GAP predictor by a margin of around 4%. Clearly a neural network is able to extract additional prediction information from a limited number of HRg bits. The situation is practically the same for a dynamically trained neural predictor, it also outperforms a GAP predictor as it can be seen comparing performances pointed out by figures 2.7 and 2.9.

**A New MLP Predictor**

In Figure 2.11 we investigated HRg (on k bits, varied from 2 bits to 10 bits) and HRl (on i bits, varied from 1 to 4) length’s influence on the prediction accuracy. The prediction accuracy grows as HRg and respectively HRl length grows. So, the minimum prediction accuracy is 88.31% (i=1, k=2) and the maximum accuracy is about 90% (obtained for i=4, k=10).

As we pointed out in [Vin99b], a common criticism for all the present Two Level Adaptive Branch Prediction schemes consists in the fact that they used an insufficient global correlation information (HRg). So, as our statistics clearly point out, for the same static branch and in the same HRg
Advanced Branch Prediction Methods

(containing the last $k$ branches encountered as taken or not taken) and HRl (containing the last $i$ occurrences of the same branch) context pattern, it's possible to find different ("alternative") branch's behaviours (for example about 50% taken and respectively 50% not taken), making that branch difficult to predict even through adaptive schemes. Otherwise, as it can be seen in [Sec95], "the role of adaptivity at the second level of two level branch prediction schemes is more limited than has been thought". In other words, it's difficult correctly to predict a branch that has randomly or unbiased behaviour in the same dynamic prediction context (HRg, HRl). If each bit belonging to HRg (on $k$ bits) will be associated during the prediction process with its corresponding PC, the correlation information will be more complete and therefore the prediction accuracy would be better. In this way it will be not only known if the previous $k$ encountered branches were taken or not (through HRg content), but it will be exactly known which branches they were, through their labels ($PC_1, PC_2, ..., PC_k$).

This information is the so-called path information, very useful in developing better branch predictors as we shown in this textbook. Therefore, instead of using only HRg, it could be used a more complex and complete prediction context, consisting of HRg together with its corresponding labels of branches with better performances. For each different pattern of this context, we'll have a corresponding prediction automaton. This path concept was independently discovered by Ravi Nair [Nai95] and us [Vin99b]. Based on this principle, we proposed a new efficient prediction scheme, first presented in [Vin99b]. The new idea presented now, is to hash all the $PC_1, PC_2, ..., PC_k$ through a logical XOR function, resulting a so named $PC_{\text{XOR}}$ output. Thus, the MLP input in this case will be $PC \& HRg \& HRl \& PC_{\text{XOR}}$ (\& operator means "concatenation"). Figure 2.12 presents some simulation results obtained for this MLP prediction scheme, considering different values of the local history (on $i$ bits) respectively global history length (on $k$ bits). As it can be observed comparing figures 2.10 and 2.11, this last scheme behaves slowly better than a corresponding MLP that doesn't use the additional $PC_{\text{XOR}}$ prediction information. The best prediction accuracy in this case is about 90.1%, obtained for $i=4$ and $k=10$.

**Neural Branch Predictor Training Methods**

As a research exercise, we have additionally developed other three distinct static training methods dedicated to our neural branch predictor, considering an HRg on 8 bits. The first one was presented in [Vin2000] and the training
vector set is based on some laborious statistics derived from the traces, as we briefly explained. The learning algorithm involves a prediction accuracy grow from 89.16% to 89.80%. The second static training method consists in learning all the traces, in a random order, until the error became smaller than a certain limit. Using this method, we presented in Figure 2.13 the learning behaviour after running 200 training iterations. As it can be observed, the prediction accuracy growths at average in this case from 89.16% to 90.11%. Finally, the third training method uses Genetic Algorithms in order to statically train our neural branch predictor. We started considering an initial set of 50 chromosomes, each of them containing the corresponding NN weights (genes). We have applied the well-known Genetic Algorithm [Koz92] using two genetic operators: crossover and mutation. Also we implemented in order to obtain next weights generations, some elitist genetic strategies. After training through this method, the prediction accuracy growth at average from 89.16% to 89.40%, therefore lower than the previous presented methods. The best training method seems to be the second one.

2.3. A STATIC BRANCH NEURAL PREDICTOR

One of the first and most simple methods for static branch prediction is called “backward-taken/forward-not-taken” (BTFNT). This technique relies on the statistical facts that backward branches are usually loop branches, and as such are likely to be taken. Using this technique we obtained in our experiments a static prediction accuracy rate of 57.83%. Applying other simple program-based heuristics information (branch’s opcode, branch’s direction, operands, characteristics of the branch successor blocks, knowledge about common programming idioms) can significantly improve the branch prediction accuracy over the simple BTFNT technique [Cal95].

The work presented in this paragraph does not use any of the heuristics stated above. Instead, a body of programs is used to extract common branch behavior that can be used to predict other branches. Every branch of those programs is processed and some static information is automatically extracted. The programs are then executed and the corresponding dynamic behavior is associated with each static element. Now we have accumulated a body of knowledge about the relation between the static program elements.
and its dynamic behavior. This body of knowledge can then be used at a later time to predict the behavior of instructions with similar static features from programs not yet “seen” (i.e. processed). However the prediction process is yet not complete. The static prediction is not linked with real time execution unless there is no connection with dynamic prediction (likely bits). Of course, compilers or integrated schedulers may still use static prediction alone in order to schedule the static program (e.g. trace scheduling). So we further generate such bits and use them as inputs into a dynamic predictor.

The only program’s elements that we are interested in are the conditional branches. Other kinds of branches are quite trivial from the direction prediction point of view. However, there are some branches that remain very interesting from the target prediction challenge (e.g. indirect jumps and calls generated by indirect calls through function pointers, statically or dynamically library function calls, polymorphism, etc.). For each conditional branch in the program a set of static useful features are recorded (Table 2.2). Some of these features are properties of the branch instruction itself (the branch’s opcode, branch’s direction, etc.) others are properties of the previous branch, while others are properties of the basic blocks that follow on the two program paths after the current branch. We mention that characteristics no. 5, 11 and 12 belonging to Table 2.2, are new useful characteristics first proposed in our work [Sbe01], completely different by those stated in [Cal95]. Of course, finding new relevant prediction features might be a very important open problem in our opinion.

<table>
<thead>
<tr>
<th>Index</th>
<th>Feature Name</th>
<th>Feature Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Branch opcode</td>
<td>The opcode of the branch instruction</td>
</tr>
<tr>
<td>2</td>
<td>Branch direction</td>
<td>The branch direction (forward or backward)</td>
</tr>
<tr>
<td>3</td>
<td>Loop header</td>
<td>The basic block is a loop header</td>
</tr>
<tr>
<td>4</td>
<td>RB type</td>
<td>RB is a register or an immediate constant</td>
</tr>
<tr>
<td>5</td>
<td>RB register index or constant</td>
<td>The RB register index or the LSB bits from the immediate constant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Feature Name</th>
<th>Feature Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Succ. Type</td>
<td>The branch type from the successor basic block</td>
</tr>
<tr>
<td>7</td>
<td>Succ. Loopheader</td>
<td>The successor basic block is a loop header</td>
</tr>
<tr>
<td>8</td>
<td>Succ. Backedge</td>
<td>The edge getting to the successor is a back edge</td>
</tr>
<tr>
<td>9</td>
<td>Succ. Exitedge</td>
<td>The edge getting to the successor is a loop exit edge</td>
</tr>
<tr>
<td>10</td>
<td>Succ. Call</td>
<td>The successor basic block contains a procedure call</td>
</tr>
<tr>
<td>11</td>
<td>Succ. Store</td>
<td>The successor basic block contains at least one store instruction</td>
</tr>
<tr>
<td>12</td>
<td>Succ. Load</td>
<td>The successor basic block contains at least one load instruction</td>
</tr>
</tbody>
</table>
Table 2.2. The Static Features Set

| Features of the not taken successor of the branch | As features 6 to 12 |

Similarly, the same type of feed-forward neural network, but another input vector, is used to predict with the dynamic predictor. This time the input vector consists of binary representation of the branch address concatenated with the transformed prediction obtained from the static predictor (the static prediction obtained may be a binary value taken/not taken or a “percentage value”), and, possible other useful dynamic information. The output is represented by one bit value (‘1’ for taken and ‘0’ for not taken). Our goal into this work was to predict the branch probability for a particular branch from its associated static features. The prediction rate obtained is then used as input into a dynamic branch predictor besides other inputs. Also the static prediction will be used as important information in the static scheduler (as an example, for the well-known trace scheduling optimization technique). The static features of a branch are mapped to a probability that the branch will be taken. A simple way to do this is using a feed-forward neural network. The neural network uses as input a numerical vector and maps it to a probability. Here, the numerical input vector is binary coded and consists of the feature values belonging to the static set (Table 2.2) and the output is a scalar indicating the branch’s probability.

Before this ensemble of predictors (static and dynamic) starting to work we have to record a body of knowledge into the static predictor’s neural network. So, a set of programs (benchmarks) is used to train the static neural network. The static features – automatically extracted for every conditional branch belonging to these programs - are mapped using the neural network to the probability that the branch will be taken. The output probability itself is obtained after running those programs and recording statistical information about each branch’s dynamical behavior. The process is reiterated until the difference between the output of the network and the statistical information gathered drops under a certain value. After the process ends we may say that we have recorded into the neural network weights a body of knowledge about mapping static features of branches to the probability that the branch will be taken. The static predictor may now be used to predict branches belonging to new unprocessed programs and the outcome is used to influence the dynamic predictor (there are several levels of influence, see further). The outcome of the static predictor is transformed according to the following rule: scalars greater than 0.5 are mapped to ‘1’
and less than 0.5 to ‘0’. This partial result is then concatenated with the branch’s address and set as input into the dynamic predictor. To evaluate the influence of this “likely bit” upon the dynamic prediction it is applied in various levels (multiplied horizontally as number of occurrence, obtaining more that one “likely bit”, all identical) as stated by the “Static Influence” parameter in our developed software dedicated program. The neural networks used in this work in order to map static features to a branch taken probability and to dynamically predict a branch outcome, were feed-forward kinds of networks.

Figure 2.14 represents two neural predictors working together (a static one and a dynamic one). Before this ensemble of predictors (static and dynamic) starts to work, we have to record a body of knowledge into the static predictor’s neural network. To quantify the performance of this static prediction scheme, during our experiments, we have used a set of 8 Stanford HSA (Hatfield Superscalar Architecture) benchmarks suite programs [Stev97]. In order to collect a body of knowledge, only 7 (of the total of 8) Stanford programs are used. Inspecting the sources of these 7 programs, the static feature sets are automatically extracted and a statistical evaluation is performed on the corresponding traces. Then, the neural net belonging to the static predictor is used to map these static feature sets to the probability that the branch will be taken or not taken. This probability is extracted from the statistical information earlier gathered. Now, taking into account that the body of needed knowledge is available, we may proceed in predicting programs yet “unseen” by our neural static predictor. In order to do this, we used the eighth program from the HSA benchmarks suite, left out until now, which we further call it the “test benchmark”. Because every program from this suite covers just a particular domain (puzzle games, recursion programs, matrix problems, etc.), each experiment is executed 8 times and every time is left out another program (as a test program). Finally an average of the eight results is computed. The neural net’s configuration presented in the results charts are represented as: [number of input units] x [number of hidden units] x [number of output units]. We have performed two distinct kinds of experiments. In the first one we predicted using just the first (static) neural network. The static prediction extracted from this first neural network output is compared with the perfect static prediction. By “perfect static prediction” we actually mean the asymptotic limit to which we could target out the maximum static prediction accuracy, considering an ideal “Oracle” processing model. In order to achieve this we considered the dynamic trace from which we extracted statistical information and for each benchmark we
calculated its associated perfect static prediction (P.S.P.) using the following formula:

\[
P.S.P. = \frac{ \sum_{k=1}^{L} MBR_k^2 }{ \sum_{k=1}^{L} NBR_k^2 }
\]  

(3), where:

NBR<sub>k</sub> – total number of dynamic instances of a branch belonging to a certain benchmark
NBR<sub>k</sub> = NT<sub>k</sub> (no. of taken instances for the k-th branch) + NNT<sub>k</sub> (no. of not taken instances for the k-th branch)
MBR<sub>k</sub> = Max( NT<sub>k</sub>, NNT<sub>k</sub> )
L = total number of static branches belonging to a certain benchmark

Now we will define the (imperfect) static prediction (S.P.) using the following formula:

\[
S.P. = \frac{ \sum_{k=1}^{L} \left( \frac{ NT_k^2 }{ NBR_k } * R_k + \frac{ NNT_k^2 }{ NBR_k } * (1 - R_k) \right) }{ \sum_{k=1}^{L} NBR_k }
\]  

(4), where:

R<sub>k</sub> – the output of the static neural network predictor for the branch k (1 – taken; 0-not taken). It’s obviously that:

\[
P.S.P. \geq S.P.
\]  

(5)

Prediction results obtained from the hybrid predictor (static + dynamic) are computed using the same formulas but while in formula (3) the input indices (MBR<sub>k</sub>) are “coming” from trace statistics, in this case these indices are gathered from the output of the first static neural network (R<sub>k</sub>). Also, the branch’s PC constitutes an input parameter for the neural dynamic branch predictor.

For the first type of experiments we have considered two ways of representing the output: as binary output (1 = taken, 0 = not taken) and as percentages of branch taken behavior (measured probability). Figures 2.14 and 2.15 present the static prediction accuracies (P.S.P. and S.P.) according
to these previous described two distinct conditions. Our simulations focused on the neural static prediction suggest that predicting binary results (taken / not taken) are less efficient than generating percentage results (prediction accuracies). The best average result performed for the binary output static prediction was at average 66.21% while with percentage output it reached 68.32%. Both these static prediction results must be compared with the perfect static prediction that was about 82.16%. Obviously, the neural net’s percentage outputs, even if are harder to be assimilated in the learning phase, contains more information than a simple binary prediction.

Another type of experiments consists in measuring the so called Static Influence (SI) of the static neural predictor regarded to the dynamic neural predictor’s behavior. More precisely, we added supplementary outputs belonging to the static network as entries for the dynamic neural predictor. The number of these supplementary inputs – having each of them the same associated value (probability) - is noted here with SI. The influence of SI parameter is presented in Figure 2.17 (SI=2 and 4). Related to Figure 2.17, it means that the dynamic predictor’s totally number of input cells is (8+SI) respectively (10+SI). A bigger number of inputs units derived from the static predictor means, theoretically, a bigger influence related to the neural dynamic predictor’s accuracy. The neural networks configurations are represented as:

- the configuration for the static net as: [number of input units] x [number of hidden units] x [number of output units]
- the configuration for the dynamic net as: [number of input units] x [number of hidden units] x [1 output unit]

Based on our simulations the influence (through the simulated likely bits) of static predictor upon a neural dynamic predictor is minimal. The best results obtained were 80.34% for a SI=2 and respectively 81.1% for SI=4, both using the same hybrid configuration (29x33x1, 8x10x1). Figure 2.17 exposes that as static influence grows for smaller neural nets configurations the result is also positively influenced, but on bigger neural nets the static influence becomes negative by expanding too much the net’s input layer and being harder to assimilate by the neural net.
2.4. SOME COMMENTS TO NEURAL BRANCH PREDICTION IDEA

In this paragraph we sought to determine whether a neural network could mimic a two-level adaptive branch predictor and achieve respectable branch prediction success rates. Two types of global neural predictors were developed and presented, an LVQ predictor and an MLP predictor. While the LVQ predictor achieved results comparable to an equivalent conventional predictor, the statically trained MLP predictor and also the (only) dynamically trained MLP, outperformed its conventional counterpart. The dynamic MLP predictor was a bit less successful but still managed to outperform a conventional predictor. These results suggested that not only can neural networks generate respectable prediction results, but also in some circumstances a neural predictor may be able to exploit correlation information more effectively than a conventional predictor and at a linear cost. The obtained results are extremely optimistic taking into account that we compared global neural predictors (one predictor to all branches!) with classical branch predictors (one predictor to each branch!), therefore totally unfair from neural point of view.

Clearly, therefore, neural networks are an useful vehicle for future branch prediction research. As we stated in our initial results “challenge is to construct composite input vectors for neural network predictors that will enable them to outperform conventional predictors”. This task involves both identifying new correlation mechanisms that can be exploited by neural prediction and tailoring the input information to fully exploit the capabilities of the neural predictor. Describing a neural network predictor that can be integrated into a high-performance superscalar processor is an even greater challenge (recent information on some efficient NN hardware implementations can be found at http://msia02.msi.se/~lindsey/nnwAtm.html or http://amesp02.tamu.edu/~sanchez/pubs/nnet.html). In order to do this, we also used some very simple neural methods, like a single cell perceptron, as an alternative to the commonly used two bits counters. Therefore, a two bits predictor will be replaced by a simple neural predictor (perceptron), instead of using a global neural predictor like we did in this section. In this sense in 2000 we already wrote: „We also suspect that the cost would be less than one of Two Level Adaptive Predictors and in the future it may even be possible to implement multiple cut-down neural predictors, associated
which each static cached branch” [Vin00]. However, the creditable performance achieved by the simple LVQ neural predictor suggests that this approach should not be discounted. Anyway, and most important at this point of our initial “integrated” research presentation, neural predictors could be a useful approach in establishing, estimating and understanding better, the processes of branch predictability and also their “limits”. Also this concept could be used, as a performance measurement of the predictability of branches and it is useful to compare its performance with the performance obtained through other prediction schemes. Thus a complex structure like a MLP together with its learning algorithm can serve as a diagnostic tool to measure some upper values of predictability, very important vehicle for computer architects. In the next paragraph we’ll present some very important further researches showing that neural branch prediction is feasible to be implemented in advanced microprocessors.

Figure 2.2. A Fully Associative GAp Scheme
Figure 2.3. An LVQ Neural Branch Predictor

Figure 2.4. An MLP Neural Branch Predictor

<table>
<thead>
<tr>
<th>PC</th>
<th>HRg</th>
<th>HRI</th>
<th>LVQ Predictor</th>
<th>Conventional Predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>87.30</td>
<td>88.60</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>88.36</td>
<td>89.76</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>88.76</td>
<td>90.03</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>88.77</td>
<td>90.06</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>88.76</td>
<td>89.89</td>
</tr>
</tbody>
</table>
3  6  4  89.29  89.95
3  3  9  89.19  90.57
3  9  4  89.51  89.70

Figure 2.5. LVQ Predictor Accuracy

Figure 2.6. Prediction Accuracies for a Statically Trained MLP Predictor

Figure 2.7. Prediction Accuracy for a Dynamically Trained MLP Predictor
Figure 2.8. A Statically Trained MLP vs. a Dynamically Trained MLP

Figure 2.9. Prediction Accuracy for an Unlimited GAp Scheme
Figure 2.10. The Statically Trained MLP Predictor vs. the GAp Predictor

Figure 2.11. Prediction Accuracies for the MLP Neural Branch Predictor
Figure 2.12. Prediction Accuracies for the Modified MLP Neural Branch Predictor

Figure 2.13. The learning process after running 200 trace iterations

Figure 2.14. Two neural predictors (static & dynamic) working together
Figure 2.15. Static prediction results with binary output

Figure 2.16. Static prediction results with percentage output

Figure 2.17. Hybrid (static + dynamic) prediction results influenced by a binary static prediction resolution
2.5 MAKING IT FEASIBLE: PERCEPTRON BRANCH PREDICTORS

One of the simplest neural networks is the so-called perceptron (single cell) illustrated in the next figure:

\[
X_0 = 1, \quad X_1, \ldots, X_n = x_i, \quad w_0, \ldots, w_n
\]

\[
\sum_{k=0}^{n} w_k x_k \rightarrow O(\vec{x}) \rightarrow Y
\]

**Figure 2.18.** Calculating perceptron’s output

\[\vec{X} = \{x_0, x_1, \ldots, x_n\}, \text{represents the input vector},\]
\[\vec{W} = \{w_0, w_1, \ldots, w_n\}, \text{represents the weight vector}\]
\[O(\vec{X}) = \vec{W} \cdot \vec{X} = w_0 + \sum_{k=1}^{n} w_k x_k, \text{represents the output}\]
\[Y = \begin{cases} +1 & \text{if } O(\vec{X}) > 0 \\ -1 & \text{if } O(\vec{X}) \leq 0 \end{cases}, \text{represents the output’s sign.}\]

The perceptron can be viewed as representing a hyperplane decision surface in the \(n\)-dimensional space of the inputs. The equation of this decision hyperplane is \(\vec{W} \cdot \vec{X} = 0\). Therefore, the perceptron might be used as a binary classifier or predictor (branch predictor in our case, predicting Taken = +1 or Not_Taken = -1). Of course, this sample perceptron can correct classify only linearly separable sets of examples (\(\vec{X}\)). It’s obvious that, for example, a logical XOR function can’t be represented by a simple perceptron.

The main problem is how to develop a learning rule for a simple perceptron, in order to correctly learn a set of training input vectors (examples) noted with \(D\). If we consider a supervised learning rule, for each example (training
vector) \( d \in D \) it is necessary to know the corresponding correct outputs called \( t_d \).

If \( O_d = w_0 + \sum_{k=1}^{n} w_k x_{dk} \) is the real output, one common convenient measure of the error \( E \) is:

\[
E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - O_d)^2
\]

Given this formula for \( E(\vec{w}) \), its surface must always be a parabola with a single global minimum. Of course, the particular \( \vec{w} \) that gives this minimum are classifying in the best manner a certain example \( X_{dk}, k=0,1,...,n \). The \( E(\vec{w}) \) gradient is written:

\[
\nabla E(\vec{w}) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n} \right] = \sum_{k=0}^{n} \frac{\partial E}{\partial w_k} \vec{i}_k,
\]

where \( \vec{i}_k \) are representing the orthogonal unit vectors in the \( n \)-dimensional space. It is well known that the gradient specifies the direction that produces the steepest increase in \( E \). Therefore, \( -\nabla E(\vec{w}) \) produces the direction of steepest decreases in \( E \). In this case a rational learning rule might be:

\[
\vec{W} \leftarrow \vec{W} + \Delta \vec{W}, \text{ where } \Delta \vec{W} = -\alpha \nabla E(\vec{W}), \alpha = \text{ the learning rate (a small positive real number)}. \]

This is equivalent with the following formula:

\[
w_k \leftarrow w_k - \alpha \frac{\partial E}{\partial w_k}, (\forall)k = 0,1,...,n
\]

But:

\[
\frac{\partial E}{\partial w_k} = \frac{\partial}{\partial w_k} \left( \frac{1}{2} \sum_{d \in D} (t_d - O_d)^2 \right) = \sum_{d \in D} (t_d - O_d) \frac{\partial (t_d - \vec{W} \cdot \vec{X})}{\partial w_k} = -\sum_{d \in D} x_{dk} \cdot (t_d - O_d)
\]

Finally the supervised learning rule is:

\[
w_k \leftarrow w_k + \alpha \sum_{d \in D} (t_d - O_d) x_{dk}, (\forall)k = 0,1,...,n
\]

This learning rule is called \textit{gradient descent or delta rule}. An algorithmically version might be the following:

\textbf{Initialize} each \( W_k \) to random values \( \in \left[-\frac{2}{n}, \frac{2}{n}\right] \)

\textbf{Until} \( E(\vec{w}) < T(\text{threshold}) \), \textbf{DO}:
Initialize each $\Delta W_k = 0$

For each pair $(x_d, t_d)$, from training examples, DO:

Compute $O_d$

For each $W_k$, DO:

$$\Delta w_k \leftarrow \Delta w_k + \alpha(t_d - O_d)x_{dk}$$

For each $w_k$, DO:

$$w_k = w_k + \Delta w_k$$

An alternative idea is to approximate this gradient descendent search by updating weights incrementally, following the error’s calculation for each training example. On way to implement this stochastic gradient descendent is to consider a distinct error $E_d(\tilde{w}) = \frac{1}{2}(t_d - O_d)^2$. Randomly giving the examples $X_d$, provides a reasonable approximation to descending the gradient respect to the original global error $E(\tilde{w})$. The stochastic gradient descent rule is:

Initialize each $w_k$ randomly to $[-\frac{2}{n}, +\frac{2}{n}]$

Until the termination condition is met ($E_d(\tilde{w}) < T$ or $|O_d| > T$, etc.), DO:

For each $(x_d, t_d)$, DO:

Compute $O_d$

For each $w_k$, DO:

$$w_k \leftarrow w_k + \alpha(t_d - O_d)x_{dk}$$

Summing over multiple examples the standard gradient descent rule is more time consuming but it is often used with a larger step-size per weight update than stochastic (incremental) gradient descendent. If $E(W)$ has multiple
local minima, stochastic gradient descent can sometimes avoid falling into these minima because it uses various $\nabla E_d(\bar{W})$ to guide its search.

If we consider perceptron’s output $O(\bar{X}) = sgn(\bar{W} \cdot \bar{X})$ instead of $O(\bar{X}) = \bar{W} \cdot \bar{X}$, this last simple rule is named the perceptron training rule:

$$w_k \leftarrow w_k + \alpha(t - o)x_k, \quad (\forall)k = 0, 1, ..., n$$

If the training example is correctly classified ($t=\circ$), no weights are updated. Suppose now $o=-1$ and $t = +1$. In this case all $w_k$ having $x_k$ positive are incremented and the others $w_k$ are decremented. Similarly, if $o = +1$ and $t = -1$, all $w_k$ having $x_k$ negative are incremented and the others $w_k$ are decremented. As a general intuitive rule: if $sgn(t) = sgn(x_k)$ then $w_k$ is incremented; otherwise, $w_k$ is decremented. Based on this last qualitative learning rule D. Jimenez [Jim02] proposed a pragmatically algorithm for training perceptron branch predictors:

If $sign(o) \neq t$ or $|o|<T$

For $k=0$ to $n$ do

$$w_k = w_k + tx_k$$

endfor

dendif

In the case of perceptron branch predictor the input vector $X$ represents the usual input vector in a classical branch predictor (local/global histories, branch address, path information, etc.). It is considered that if $t = -1$ the branch was not taken and if $t = 1$ the branch was taken. Since $t$ and $x_k$ are always either -1 or 1, these training algorithms increment the $k^{th}$ weight when the branch outcome agrees with $x_k$ (positive correlation), and decrements the weight when it disagrees (negative correlation).

The processor might contain a cache table of $N$ perceptrons, similar with the table of two bits saturating counters in the classical branch predictors. These perceptrons are indexed in the instruction fetch phase, using PC and other prediction information (contexts). Obviously, the numbers of perceptron’s weights is determined by the amount of branch history (context) we keep. When the CPU encounters a branch during the instruction fetch stage, the following 6 steps are taken [Jim02]:
1. The PC is hashed to produce an index into the predictors’ table, containing perceptrons’ weights.
2. The \(i^{th}\) perceptron is fetched from the table into a vector register containing weights (P vector).
3. Output \(O\) is computed as the dot product of \(P\) and the history register (global, local, hybrid, etc.).
4. Branch prediction process (Taken if \(O > 0\), Not Taken if \(O \leq 0\)).
5. After the real branch’s outcome is known, the training algorithm uses this outcome and the predicted value in order to update the weights in \(P\).
6. \(P\) is written back into the predictors’ table (PT).

Jimenez proposed some simplifications in order to adapt the dynamic perceptron branch prediction algorithm to hardware implementation restrictions and to reduce the prediction’s timing. In this sense, the weights for the perceptron predictor are signed integers (8-bit weights). Since the history information (entry in the perceptron predictors) is codified using only 1 and -1, multiplication is not need to compute the dot product. Instead
of multiplication, there are only adds (when the input bit is 1) respectively adds with two’s-complement (when the input bit is -1). In practice, adding the one’s complement approximate very well the two’s complement, being simpler to implement. Additionally, this avoids the delay of a small carry-propagate adder that finally might add +1 for calculating the weight two’s complement. More concretely, weights are bitwise eXclusive-ORed with the corresponding bits of the history register content. If the \( k \)th history bit is -1, the resulting weight \((w)\) is one’s complemented (XOR-ed with \(11111111_2\)); otherwise the resulting weight is unchanged. After processing weights, the sum is obtained using a Wallace-tree of 3 to 2 carry-save adders, which reduce adding \(N\) numbers to the problem of adding two numbers [Jim02]. The final two numbers are added with a carry-look-ahead adder. The computation is relatively quick \((\log n)\) because only the sign bit of the result is needed to make a prediction.

The perceptron branch predictor is the best one for linearly separable branches (comparing with all classical predictors). In order to define this concept, let \(H_n\) be the most recent \(n\) bits of branch’s history (global, local, etc.). For a static branch \(B\), there is a Boolean function \(f_B(H_n)\) that best predicts (classifies) behavior for branch \(B\). If \(f_B(H_n)\) is linearly separable the branch \(B\) is linearly separable, too. Theoretically, classical Two Level Adaptive Branch predictors can learn also linearly inseparable branch behaviors because each \(H_n\) context pattern indexes its own predictor. Jimenez shows that “most miss-predicted branches are linearly inseparable, so linear inseparability correlates highly with unpredictability in general” [Jim02]. Very interesting, he shows that “the longer the history required the better is the performance of the perceptron predictor, even on the linearly inseparable branches.”

In conclusion, the main advantage of the perceptron predictor consists in its ability to exploit long histories requiring linear resources growth (classical predictors are requiring exponential resources growth). Additional performance gains can be found for branch history lengths of up to 66 bits. Jimenez reports a global improvement of 5.7% over McFarling-style hybrid predictor (he used also a gshare/perceptron overriding hybrid predictors). The main disadvantage of the perceptron predictor consists in its high latency. Even if are using some high-speed arithmetic tricks like we already mentioned, the computation latency is relatively high comparing with the clock period of a deeply pipeline microarchitecture (Intel Pentium IV has over 20 stages in its integer pipeline and researchers conclude that
performance of aggressively clocked microarchitectures continues to improve until 52 stages!).

In order to reduce the prediction latency, in [Jim03] is presented a perceptron predictor choosing its weights for generating a prediction according to the current branch’s path, rather than according to the branch’s PC and a simple binary history register. Considering a path of the last $h$ branches, the predictor keeps a matrix $W$ of weight vectors, having $N$ lines and $(h+1)$ weights per line $w[i,0], w[i,1], \ldots, w[i,h]$. Each time a branch is fetched, its $0^{th}$ weight $w[i,0]$ is added to a running total that has been kept for the last $h$ branches, with each summand added during the processing of the previous branches. The key idea is that each branch is predicted correlated with the previous $h$ branches’ predictions rather than with the previous $h$ branches outcomes. Therefore, even the prediction information is speculative in this case.

Let $SR[0…h]$ be a vector of $(h+1)$ integer partial sums. The first column of $W$ forms the bias weights. $SR[h-j]$ stores the running total computing that will be used to predict the $j^{th}$ branch after the current one. The path-based neural prediction algorithm that predicts the branch at address PC is [Jim03]:

\[
\begin{align*}
\text{begin Prediction} \\
i &= \text{PC mod } N \\
O &= SR[h] + w[i,0] \\
\text{if } O \geq 0 \text{ then} \\
\quad \text{Prediction} = \text{Taken} \\
\text{else} \\
\quad \text{Prediction} = \text{Not\_Taken} \\
\text{endif} \\
\text{for } j = 0 \text{ to } h \text{ in parallel do} \\
\quad \text{if Prediction } == \text{ Taken} \\
\quad \quad SR'[h-j+1] = SR[h-j]+w[i,j] // \text{See next figure} \\
\quad \text{else}
\end{align*}
\]
SR'[h-j+1] = SR[h-j] - w[i,j]

endif; //Here it is prepared the partial sum for the jth next branch speculating that the current prediction is correct

endfor

SR = SR';

SR[0] = 0;

SG = (SG<<1) or Prediction //Shift prediction into the Speculative Global history (SG)

In order to better understand the path-based neural prediction process we try to develop an original synthetic model in a rigorous comprehensive intuitive manner. First, the partial sums calculation process might be represented like in the next figure:

**Figure 2.20.** The partial sums for the last $h$ branches

Considering the current predicted branch $i_h$ and the previous dynamic $h$ branches: $i_{h-1}$, $i_{h-2}$, ..., $i_1$, $i_0$. The path-prediction process can be resumed by the following temporal formula:

\[
P(i_h) = P(i_0) \cdot w[i_0, h] + P(i_1) \cdot w[i_1, h-1] + \ldots + P(i_{h-1}) \cdot w[i_{h-1}, 1] + w[i_h, 0],
\]
or:

\[
P(i_h) = w[i_h, 0] + \sum_{k=0}^{h-1} P(i_k) \cdot w[i_k, h-k],
\]
where:

$$(\forall)k = 0, h-1, P(i_k) = \begin{cases} 
+1, & \text{if branch } i_k \text{ was predicted as Taken} \\
-1, & \text{if branch } i_k \text{ was predicted as Not Taken} 
\end{cases}$$

If $P(i_h) \geq 0$, prediction for branch $i_h$ is Taken
If $P(i_h) < 0$, prediction for branch $i_h$ is Not Taken

It can be observed that $k$ is a temporal index representing a speculative history of $h$ branches previous instances. At this moment it becomes clear that the predictor really used path information and not only global history information. This is obvious taking into account that $i_k = PC_k \mod N$, therefore, representing the address of branch $i_k$ in the matrix $W$.

Updating the path-neural predictor is theoretically similar to the original perceptron’s update. It must be considered that each weights vector is associated with the last $h$ branches, rather than one branch. After a branch is completed, most of the weights vector associated with this branch cannot be updated because they are used to predict future branches, as we already explained. In order to solve this problem, matrix $W$ is designed as $(h+1)$ independently addressable high-speed memories, each representing the $N$ weights of single column. Therefore, when the predictor is updated the corresponding weights can be independently accessed. The bias weights are kept closest to the logic that computes the final output. Formally, we resumed the path-based predictor update algorithm in the following intuitive simplified manner:

Begin Update

If Prediction $\neq$ outcome or $|O| < T$ then:

$$w[i_h, 0] = w[i_h, 0] + \begin{cases} 
1, & \text{if } O = \text{Taken} \\
-1, & \text{if } O = \text{Not Taken} 
\end{cases}$$

For $j=1$ to $h$ in parallel do

$$k = v[j]$$

$$w[k, j] = w[k, j] + \begin{cases} 
1, & \text{if } O = H[j] \\
-1, & \text{if } O \neq H[j] 
\end{cases}$$

endfor

dendif
Vector \( H[1...h] \) represents the value of the global history shift register \( SR \) when the branch was predicted, therefore before the updating process. The array \( v[1...h] \) of integers represents the addresses (PCs) of the last \( h \) predicted branches \textit{modulo} \( N \). That is, \( v[j] \) is the index of row in \( W \) matrix used for predicting the \( j \)-th most recent branch. Here again clearly appears the role of path information (the PCs corresponding to the last \( h \) branches).

The path-based neural prediction method has two main advantages:

1) The prediction latency is almost completely hidden because computation of \( O \) can begin in advance of the effective prediction, with each step proceeding as soon as a new element of the path is executed. The most critical-timing operation is the sum of the bias weight and the current partial sum \( SR[h] \).

2) Prediction accuracy is improved because the predictor incorporates path information, too. In [Jim03] it is showed that it outperforms classical predictors at the same hardware budget. At 64KB hardware budget it delivers an IPC 16% higher than that of the simple perceptron predictor, due to its lower latency.

In [Tar04] the authors developed the path-based neural predictor by introducing the hashed perceptron predictor concept, which merges the concepts behind the \textit{gshare} and the perceptron predictor. The perceptron weights are fetched from a table using an index obtained by eXclusive-ORing (XOR) the branch’s PC and branch’s speculative global history (context). The authors showed that combining \textit{gshare} indexing with path-based indexing increased accuracy. The indexing formula is:

\[
\text{index} = (\text{PC} \text{xor} \text{SG}) \text{ modulo } \text{nr_weights}
\]

Combining \textit{gshare} with path-based perceptron, by using different parts of the history, weights are fetched by indexing into multiple tables. It is performed a series of small \textit{gshare}-like predictors instead of a single large predictor. This design can avoid an excessive training time. It is shown that this improved scheme reduces the number of adders by a factor of four and shortens the predictor’s pipeline by the same factor compared to a path-based perceptron. The predictor can be ahead pipelined to yield one cycle effective latency. The hashed ahead-pipelined path-based perceptron predictor improves accuracy by 20% over a path-based perceptron and improves IPC by 5.8%.
In [Mon05] it is proposed a neural predictor based on two perceptrons and called the combined perceptron branch predictor. This combined predictor consists of two concurrent perceptron–subpredictors, one using as inputs branch history information and the other one using some PC bits. Both these sub-predictors (history-based and address-based) are exploiting branch path information in the selection of the weight vector. The history-based predictor has the same structure as the path-based predictor, previously described. The address-based predictor used a perceptron selected by the branch path but the least significant PC bits give the input. Using PC bits as input, the combined predictor separates branches otherwise collapsing in the same perceptron. Therefore, the basic idea is to add to the final prediction a contribution to take into account branch address related information, de-aliasing some branches, which are collapsing in a single entry.

The outputs of the two perceptrons are calculated and summed together. If the calculated output is greater then zero the prediction is Taken, otherwise it is Not_Taken. In our opinion, some other more efficient metapredictors, based on some adaptive algorithms, might be developed. These metapredictors would decide based on the confidence information, attached to each of the two predictors. The proposed combined perceptron predictor archived 34% lower missprediction rate than baseline GShare predictor and 12% lower missprediction than a perceptron predictor.

As pipelines get deeper and superscalar factor is growing up, wasted miss-speculative execution due to branch misspredictions is increasing. As Akkary et al. [AKK04] are shown the percentage increase in total instructions executed is 24% on a 4-wide 20-cycle pipeline and respectively 50% on an 8 wide 20-cycle or 4-wide 40-cycle pipelines. As a consequence, the authors tried to solve this problem by developing a perceptron-based branch confidence estimator. They used pipeline gating for reducing wasted speculative execution due to a branch misprediction. Pipeline gating refers to halting the fetch of new instructions, until a likely mispredicted branch is resolved. Each branch has a branch confidence estimator inside the fetch unit. If the confidence estimator is high, the branch is likely to be correctly predicted, otherwise no and the speculative fetch process is stopped. After the branch is resolved, its confidence estimator is incremented if the prediction is correct and decremented else. Therefore, when a mispredicted branch was correctly identified, wastage due to the speculative execution is reduced. However, when a correctly predicted branch wasn’t correctly identified, the pipeline is stalled unnecessarily involving a performance loss. The authors reported that their perceptron-based branch confidence
estimator “is twice as accurate as the current best-known method and achieve reasonable mispredicted branch coverage”.

2.6 TOWARDS CROSS-FERTILIZATION BETWEEN COMPUTER ARCHITECTURE AND OTHER COMPUTER SCIENCE FIELDS [Vin01]

During the last 25 to 30 years, computer architecture researches have been followed an evolutionary path, based mainly on some quite old concepts and ideas. The performance growth over these years was really amazing but it's difficult to believe that it will be possible to follow the same path of development in the future. Thus, there are necessary some innovative ideas in this domain, revolutionary and evolutionary at the same time. Especially in Instruction Level Parallel Processors (ILPP) and Thread Level Parallel Processors research fields, the key seems to be an integrated approach based on cross-fertilization between hardware and software scheduling and respectively between technology and architecture. Also, the performance centrepiece criteria needs to be accomplished by other very important interdependent quality criteria like availability, maintainability and scalability of services.

In our opinion the most promising present-day approaches in ILPP domain consists in value prediction, dynamic instruction reuse (DIR), multiprocessors on a chip and simultaneous multi-threading. The value prediction super-speculative technique is based on the value locality concept, a third facet of locality that is frequently present in programs. Value locality describes the likelihood of the recurrence of a previously - seen value within a storage location. Researchers show that some micro-architectural enhancements that enable value prediction can effectively exploit value locality to collapse true dependencies, reduce memory latency and bandwidth requirements. On the other hand, empirical statistics suggest that many instructions and groups of instructions, having the same inputs, are executed dynamically. Such instructions do not have to be executed repeatedly because their results are obtained from a reuse buffer where they were saved previously. In the case of reusing groups of true dependent instructions, DIR can outperform the programs intrinsic sequentially ceiling (is Amdahl's low contradicted here?).
At this moment the main (unique?) way we can understand Computer Architecture is by doing laborious simulations. But, the question is: this means a real profound understanding? From our point of view certainly not, because sometime is very difficult or even impossible an in depth understanding of the hardware-software interface in a qualitative manner, based only on benchmarking. Simulation is, of course, absolutely necessary but, unfortunately, it isn't sufficient for an in depth, qualitative understanding. Sometime, the obtained quantitative results represent only effects and not the real causes of some processing phenomena.

Treated as separate entities, hardware and software seem to be really "sciences". As an example, automata theory, circuit complexity, digital design, hardware description languages, etc. could be considered, at this moment, being mature theoretic "hardware" fields. On the other hand, pure "software" domains like computer algorithms and complexity theory, formal languages or compiling theory, could be also considered "sciences". But could we consider Computer Architecture, therefore a hardware-software approach, being really a science? The answer is obviously no, because in this field, situated at the hardware-software interface, the main approach is based on benchmarking and simulation, empirical statistics, etc. That means a lot of empirics, heuristics, "strange" optimization techniques, statistic principles and rules (ex. "90/10 rule") but not a logic formalized mature theory. Why this? Perhaps because the "hardware-software" interface represents a too complex synergic concept and - up to this moment - we are unable to formalize and understand it in a real qualitative manner.

In our opinion, taking into account all these briefly presented aspects, Computer Architecture's pure evolutionary era must be finished. The alternative seems to be a more integrated approach that mainly means a synergism between the following aspects:
- Technology respectively architecture, concepts and algorithms
- Hardware, software and applications - not treated as separate entities
- Different (Computer) Science fields and techniques working together for a certain Computer Architecture challenge

We'll exemplify this new idea during the next sections, by analyzing some innovative proposals in branch prediction research field (neural branch predictors, markovian predictors, genetic predictors, decision tree predictors, etc.)
2.6.1. SOME OTHER NEURAL APPROACHES IN COMPUTER ARCHITECTURE

An interesting alternative approach in branch prediction research is to look to other computer science application areas for novel solutions to this important problem. As we already presented, we first proposed the application of neural networks to dynamic branch prediction. This is now an important topic in branch prediction research.

As we already presented, also an interesting neural nets approach in branch prediction was proposed by Calder et al. [Cal95]. The authors discussed the application of neural nets to the problem of static branch prediction at compile time. The prediction methodology is therefore entirely based on information about program’s structure that can be readily determined by a compiler. As an example, a branch successor path that leads out of a loop or function is less likely to be followed than a path that remains within the loop or function. In our opinion, the main contribution of this work consists in determining a static feature set of information related to the important static elements of the corpus of programs. So, in this set, there are included some branch features and also some features of the taken/not taken successor of the branch. All these static features that are correlated with branch outcomes are coded as binary inputs in the static neural predictor. Therefore a neural network is used to map static features associated with each branch to the probability that the branch will be taken. The main idea of this static neural branch predictor is that the behaviour of a corpus of programs (training set) can be used to infer the behaviour of new programs. Static branch prediction is important especially for compilers because it contributes to implement optimizations like trace-scheduling and other profile-directed optimizations [Cal95]. Using this approach, the authors reported a misprediction rate of only 20%, remarkably low for static branch prediction.

Another interesting neural approach applied in Computer Architecture was presented in [Kha95]. This paper presents an original neural algorithm (KORA) that uses a back propagation neural net in order to guide the block replacement decisions made by the cvasi-optimal algorithm (Belady, optimal only for read-only instruction flows). The key consists in identifying and discarding the dead blocks in cache memories. More precisely, the neural net is used here to identify two distinct types of cache blocks: transient blocks (new arrivals to the cache) respectively shadow blocks (recently replaced from the cache) stored in a so named shadow directory. As it’s known, there is a significant statistical probability that the
shadow blocks are going to be referenced soon. Therefore, the neural net identify these shadow lines and provide them with a preferential treatment. If the transient blocks do not become active during a certain time period, they are removed from the cache. The corresponding trace driven simulation results indicate that this neural replacement algorithm achieves approximately 8% performance improvement over a LRU scheme, measured for the SPEC benchmarks. This work seems to open new research directions when applied to the page replacement algorithms in virtual memory systems and disk caches.

Related to the problem of new original replacement algorithms, we found at least an other innovative paper. This is Stiliadis & Varma's paper [Sti94, Vin2000], that introduced a new heuristics named "Selective Victim Cache" in order to minimise the number of replacements between the main cache and the victim cache. In our opinion there is a similarity between the Selective Victim Cache and respectively the shadow directory proposed by Pomerene [Pom84] and improved by Khalid through neural nets algorithms.

2.6.2. A MARKOVIAN BRANCH PREDICTOR

During this section we analyzed a markovian branch prediction scheme, first introduced by Trevor Mudge et al. [Mud96]. The prediction is based here on the PPM (Prediction by Partial Matching) algorithm that represents an universal compression/prediction algorithm. PPM has been theoretically proven optimal in data compression and pre-fetching and also in some speech recognition problems. The bases of the PPM algorithm of order m are a set of (m+1) Markov predictors. A Markov predictor of order j predicts the next bit based upon the j immediately preceding bits pattern (a simple Markov chain). More precisely, the prediction process counts every time when that pattern on j bits was found, if it was followed by a ‘1’ or respectively by a ‘0’. The prediction is according to the most frequent bit that follows the searched and founded pattern. PPM uses the m immediately preceding bits to search a pattern in the highest order Markov model, in this case m. If the search succeeds, which means the pattern appears in the input sequence seen so far, PPM predicts the next bit using this m-th order Markov predictor. However, if the pattern is not found, PPM uses the (m-1) immediately preceding bits to search the next lower order (m-1)-th order Markov predictor. Whenever a search misses, PPM reduces the pattern by one bit and uses it to search in the next lower order Markov predictor. This
process continues until a match is found and the corresponding prediction can be made.

If we consider the search pattern for every Markov predictor as a HRg (Global History Register) pattern, it is proved [Mud96] that the complete PPM predictor can be viewed as a set of Two Level Predictors, having not one size of HRg but a set that spans m down to 0. It’s obviously that this PPM branch predictor generalizes the Two Level Adaptive Predictors. To implement this Markov predictor in hardware would require a doubling of the hardware, straining the limits of practicality. Trying to avoid this circuit complexity, we investigated a simplified PPM branch predictor consisted by only one Markov predictor of order m, quite feasible to be implemented in hardware [Vin2000]. Surprisingly, in both cases the obtained prediction accuracies are quite similar, showing clearly that a simplified PPM predictor outperforms a classical scheme (GAg in our experiments) and also it would be feasible to be implemented in hardware, results placed in a perfect concordance with those published by other researchers [Mud96]. In our opinion the main merits of this markovian approach is that it investigates the branch prediction problem as a mature scientific problem, involving also corresponding research tools.

2.6.3. GENETIC BRANCH PREDICTORS

In this section we discussed an interesting approach in order to search for new branch predictors using a genetic programming method. Emer and Gloy [Emer97] first developed this interesting idea. The authors proposed a language named BP that is an algebraic-style branch predictor language, based on the observation that every predictor can be divided in several elementary blocks such memories, functions and numbers, etc. The actual branch predictors can be divided into two distinct classes: non-adaptive and adaptive. In the case of a non-adaptive predictor the prediction is always the same logical function. On the other hand, dynamic adaptive predictors learn to make better predictions using information that is only available after the prediction is made. Dynamic adaptive predictors thus use feedback to learn from past behavior and hence make better predictions in the future.

In order for such a feedback control system to learn, it needs some sort of memory. To provide this memory was defined a primitive. This primitive noted \( P[w, d] (I; U) \), is basically a memory that is w bits wide and d entries deep. This memory has two types of operations: predict and update. The
predict phase consist in accessing memory at address index I, and the value read is used as the prediction P. Some time later, when the branch is resolved, an update value U is delivered to the predictor and written into the same location indexed by I. The static parameters \((w,d)\) allow describing a class of predictors of various sizes. The dynamic parameters \((I,U)\) are partitioned between the input arguments, listed first, and the update arguments, listed after the semicolons (:). Using of these predictors can be thought of as inputting a series of index \((I)\), generating a series of predictions \((P)\) and, when the prediction resolves, updating the predictor's state \((U)\). By using specific values or expression as the input to the predictor, we can generate a variety of predictors. The classical 1 bit predictor can be represented as:

\[
\text{Onelbit}[d](PC;T) = P[1,d](PC;T)
\]

where,

\[
\begin{align*}
PC &= \text{current Program Counter (index value)} \\
T &= \text{branch resolution (update value)} \\
(0 \rightarrow \text{not taken}, 1 \rightarrow \text{taken})
\end{align*}
\]

Also it can be defined the predictor “Onebit” with parameter \(d\), input \(PC\) and update value \(T\). This predictor can be parameterised by its depth noted \(d\). For example if we choose \(d = 2048\ (2K)\) we can write:

\[
\text{Onelbit}[2048](PC;T) = P[1,2048](PC;T)
\]

In the same formal manner we can build more complex structures. So we can define an array of \(n\)-bit saturating counters each of which counts up or down based on their update value.

\[
\text{Counter}[n,d](I; T) = P[n,d](I; \text{if } T \text{ then } P+1 \text{ else } P-1)
\]

The adding and subtraction operations are in this case saturating operations. If we combined this counter with function MSB – which return the most significant bit of a value - we can describe well known predictors like:

\[
\begin{align*}
\text{Twobit}(d)[PC; T) &= \text{MSB}(\text{Counter}[2,d](PC; T)) \\
\text{Twobit}(d)[PC; T) &= \text{MSB}(P[2,d](I; \text{if } T \text{ then } P+1 \text{ else } P-1))
\end{align*}
\]

This designing approach is based on automatic search for predictors using some Genetic Programming concepts. Genetic Programming is derived from Genetic Algorithms that represent an evolutionist method for efficiently searching extremely large problem spaces. A genetic algorithm encodes potential solutions to a given problem as fixed - length bits. In the BP representation each bit represents a language atom (function, memory, terminal, etc.). To generate the initial population is necessary to create the individuals. Each individual is created using a random algorithm.
Individuals are represented by a tree structure, which is easily translated into a corresponding expression in the BP language and vice-versa. First, it's necessary to evaluate the fitness – prediction accuracy in this case - of each individual through branch predictor simulation. The next step consists in creating new individuals from old ones by applying genetic operators that recombine the components of the old individuals in different ways. In this manner will be developed a new generation of genetic branch predictors. The individuals that serve as inputs to the genetic operations are chosen with a probability based on their fitness value. Individuals with a higher fitness value have a higher probability of being chosen, so that they may appear many times than individuals of lower fitness value. This means that the next generation will contain many individuals having one or more components from successful individuals, which makes it likely that the average fitness of this new generation will be better than that of the previous generation. By repeating this process many times, are produced successive generations. According to the authors, somewhere between 15 and 30 generations the experiments converge to a few distinct predictors. From the prediction accuracy point of view they are comparable with the most advanced human designed branch predictors. Unfortunately, they are logically much more complex and probably not feasible to be implemented [Emer97]. Also the computing process is very time consuming, therefore there are necessary to be used powerful computer systems. However, in our opinion, the paper opens a new era in branch prediction research and, more generally, in finding new efficient digital structures, based on evolutionary computing methods, useful in different Computer Engineering fields.

Another interesting machine-learning approach in branch prediction research is presented in [Fer06] where the authors proposed the idea of dynamic feature selection in order to avoid exponential growths of prediction tables and to permit the usage of a huge amount of predictive information. They designed a dynamic decision tree predictor, which exhibits only linear size growth in the number of additional features. Again an interesting approach using important ideas from artificial intelligence field. Also in [Cul05], starting from our neural branch prediction work, it is presented a new dynamic branch predictor based on Support Vector Machines (SVM), a classifying/prediction technique which was proved as being efficient for non-linear separable input data. SVM is a relatively new learning method based on kernels. The main idea of the kernel is to compute the norm of the difference between two vectors in a higher dimensional space without explicitly representing those vectors in the new space. The SVM branch predictor can return a greater accuracy than current state-of-
the-art predictors by exploiting its ability to learn linearly inseparable Boolean functions. Perhaps developing a SVM branch predictor feasible to be implemented in hardware might be an important further challenge.

As a conclusion, the present-day research paradigm in Computer Architecture field is too specialised and therefore, too limited. The corresponding research tools are quite old. It’s necessary that computer architects to be more opened at other Computer Science fields and research methods, in order to enlarge their view and to improve their results. During this paragraph we presented the necessity of a more integrated approach in Computer Architecture field where – as we previously claimed – the present day paradigm must be enlarged and improved. More precisely, we consider that using some more mature theoretical research tools, belonging to different Computer Science fields, could be an excellent alternative approach in the present-day Computer Architecture research.

This new research idea is concretely exemplified by considering as an example, the branch prediction challenge, through some very original papers, unfortunately not very well-known and cited. So, firstly it’s presented the neural dynamic branch predictor that uses simple neural networks in order to make predictions. Subsequently, we presented another interesting paper that also uses a neural net related to the block replacement decision in a cache memory. After this, we analyzed a markovian branch predictor first proposed by T. Mudge et al. [Mud96]. This branch predictor generalised the well-known Two Level Adaptive Branch Predictors. Finally, we briefly analyzed, a very original paper that proposes an innovative technique for developing branch prediction schemes using some genetic programming concepts.

From our point of view, an important common merit of all these researches consists in integrating a pure certain Computer Architecture problem with other interesting theoretical Computer Science fields and methods. Therefore, all these presented papers are exemplifying our initial idea: progresses in Computer Architecture research are possible in the nearest future, using a synergism between different mature science fields and techniques. It's essential trying to explore the potential benefits that could be realised through a cross-fertilisation of ideas between Computer Architecture and other Computer Science fields. We think that other similar original approaches, like those presented here, are very useful in this new Computer Architecture era.
References


[Cul05] Culpepper B., Gondre M., SVMs for Improved Branch Prediction, University of California, UCDavis, USA, ECS201A Technical Report, 2005 (Supervisor: Prof. Matthew Farrens)


[Emer97] Emer J., Gloy N., A Language for Describing Predictors and it’s Application to Automatic Synthesis, ISCA97, Denver, USA, 1997


Advanced Branch Prediction Methods


3. UNDERSTANDING SOME PRESENT-DAY BRANCH PREDICTION LIMITS

3.1. UNBIASED BRANCHES PROBLEM AND RELATED WORKS

Two trends – technological and architectural (conceptual) – are further increasing the importance of branch prediction. From technological point of view, modern high-end processors use an array of tables for branch direction and target prediction [20]. These tables are quite large in size (352K bits for the direction predictor in Alpha EV8) and they are accessed every cycle resulting in significant energy consumption - sometimes more than 10% of the total chip power [4]. For example, AMD’s Opteron microprocessor takes 7 clock cycles to fetch and decode operations and typically a further 12 clock cycles to execute instructions and a further 17 clock cycles to execute floating point operations. With such a deep pipeline, early and accurate branch prediction is essential, especially as the branch penalty for a mispredicted branch is 11 clock cycles. The Opteron uses a global branch predictor which stores 16KB of prediction counters, four times as many as its predecessor the Athlon. A further 2KB is used to store target addresses and a 12 entry return address stack for call/returns is also provided.

From architectural point of view, superscalar factors are getting wider and pipelines are getting deeper, allowing more aggressive clock rates in order to improve overall performance. A very high frequency will determine a very short clock cycle and the prediction cannot be delivered in a single clock cycle or maximum two cycles which is the prediction latency used in the actual commercial processors (see Alpha 21264 branch predictor) [12]. Also a very wide superscalar processor can suffer from performance’s point of view in the misprediction case, when the CPU context must be recovered and the correct paths have to be (re)issued. As an example, the performance of a 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 [28], it is very important to further increase the accuracy achieved by 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 approaches in branch prediction rely on usage of a greater number of input features (such as branch address, its global or local history, etc.) without taking into account of the real cause (unbiased branches) that produce a lower accuracy and implicit lower performance.

In our opinion, an important way to obtain considerably better overall performances consists in finding and solving difficult predictable branches – dynamic branches that are not predictable with the correlation information used in actual prediction methods (local branch history, global history, and path). More precisely, we initially considered that a branch in a certain unbiased dynamic context would be unpredictable and further we showed through simulation that this fact is true. Focalising on these branches with high entropy – in order to further design some efficient specific predictors for them by using more suitable context information – the overall prediction accuracy should increase with some percents, that would be quite remarkable.

Chang et al., introduced in [2] a mechanism called branch classification in order to enhance branch prediction accuracy by classifying branches into groups of highly biased (mostly-one-direction branches) respectively unbiased branches, and used this information to reduce the conflict between branches with different classifications. In other words, they proposed a method that classifies branches according to their dynamic taken rate and assigns branches from each class to different predictors. The class of branches is determined by their overall dynamic taken rate collected during program profiling. With their branch classification model they showed that using a short history for the biased branches and a long history for the unbiased branches improves the performance of the global history Two-Level Adaptive Branch predictors. In contrast to our work, presented in this chapter based on [27, 29], the authors are classifying branches irrespective of their attached context (local and global histories, etc.) involving thus an inefficient approach. Due to this rough classification the corresponding predictors are not optimally chosen, simply because it is impossible to find an optimal predictor for some classes.
Ravi Nair has first introduced dynamic branch prediction based on path correlation [18]. The basic observation behind both pattern-based and path-based correlation is that some branches can be more accurately predicted if the path leading to these branches is known. Information stored in a global history register does not uniquely identify the program path leading to a branch. Suppose, for example the final bit in the global history register is set to 1. The only information that can be truly extracted from this is that the previous branch was taken and that therefore the fall-through path from the preceding basic-block was not followed. However, the targeted basic-block may be the target of more than one branch, and therefore the path leading to the current branch is indeterminate. It can therefore be concluded that multiple program paths can map into a single global history register bit pattern.

Consider the Control Flow Graph shown in Figure 3.1. When branch D is encountered the history register will contain the same pattern, ‘XXX11’ irrespective of the path followed, ACD or BCD. Consequently, the same prediction will be obtained for both paths. However, if the probability is that the branch is not taken for the path ACD and that the branch is taken for the path BCD, then the prediction accuracy can be improved by recording the full path information.

![Figure 3.1. Full path prediction Control Flow Graph](image-url)
The path leading to a branch can be uniquely identified by recording the branch addresses as well as the branch outcomes. However, recording branch addresses is insufficient to uniquely identify a path. Consider the following example:

\[ \text{Bcc Label} \]
\[ \cdot \]
\[ \cdot \]
\[ \cdot \]
\[ \text{Label:Bcc Loop} \]

If there are no intervening branches, then the outcome of the first branch must also be recorded if the path leading to the second branch is to be correctly identified. Alternatively, the address of the instruction executed after each branch instruction can be recorded. This additional information, in combination with the history information, can then be used as an index to the PHT. Unfortunately a large amount of information is now required to record each branch outcome instead of two used by two-bit up/down saturating counters. This adds considerably to the already overburdened hardware budget.

Path-based correlation attempts to overcome the performance limitations of pattern-based correlation arising from pattern aliasing situations, where knowledge of the path leading to a branch results in higher predictability than knowledge of the pattern of branch outcomes along the path. Dr. Ravi Nair proposed a hardware scheme which records the path leading to a conditional branch in order to predict the outcome of the branch instruction more accurately. He adapted a pattern-based correlation scheme, replacing the pattern history register with a \( g \)-bit path history register, which encodes the target addresses of the immediately preceding \( p \) conditional branches. Ideally, all bits of the target address should be used to ensure that each sequence of \( p \) addresses has a unique representation in the register. Since such schemes are too expansive to be implemented in hardware, Nair used a simplified scheme, which uses a subset of \( q \) bits from each of the target addresses. Limiting the number of bits from the branch address could involve path aliasing – the inability of the predictor to distinguish two distinct paths leading to a branch. Unfortunately, this path correlation scheme does not show any significant improvement over pattern-based correlation [18]. Nair’s explanation for this is that for a fixed amount of hardware in the prediction tables, path-based correlation uses a smaller history than pattern-based correlation because the same number of bits
represents fewer basic blocks in the path history register than branch outcomes in the pattern history register. Despite this, path based correlation is better than pattern-based correlation on some benchmarks – especially when history information is periodically destroyed due to context switches in multithreading or multitasking environments – indicating that with a better hashing scheme the pattern correlation schemes could be outperformed.

Another adjacent approach is proposed by L. Vintan and C. Egan in [24] – their paper represents the genesis of the further work presented in [27]. The authors illustrated, based on examples, how a longer history could influence the behavior of a branch (changing it from unbiased to biased). They also showed that path information could also reduce branch’s entropy. The main contribution of this paper is related to the prediction accuracy gain obtained by extending the correlation information available in the instruction fetch stage. Based on trace-driven simulation the authors proved for relatively short global branch history patterns, that a path-based predictor overcomes a pattern-based predictor at the same hardware budget. The main difference, comparing with Nair’s approach, is that here the authors are using both the path and respectively the history information in order to obtain better predictions. They show that a scheme based on this principle performs better than a classical GAp scheme, at the same level of complexity. Particularly useful information has been gleaned regarding the interaction between path length and the number of replacements required in the PHT.

As we already showed in a previous dedicated chapter, dynamic branch prediction with neural methods, was first introduced by L. Vintan [25, 6], and further developed by D. Jiménez [11] and other researchers. Despite the neural branch predictor’s ability to achieve very high prediction rates and to exploit deep correlations at linear costs, the associated complexity due to latency, large quantity of adder circuits, area and power are some disadvantages. Anyway, the neural methods seem to be successfully for future microprocessors taking into account that they are already implemented in Intel’s IA-64 simulators. The advanced path-based neural predictors [8, 9] improve the instructions-per-cycle (IPC) rate of an aggressively clocked microarchitecture by 16% over the original perceptron predictor [11]. A branch may be linearly inseparable as a whole, but it may be piecewise linearly separable with respect to the distinct associated program paths. More precisely, the path-based neural predictor combines path history with pattern history, resulting superior learning skills to those of a neural predictor that relies only on pattern history. The prediction
latency of path-based neural predictors is lower comparing with the perceptron predictor, because the prediction’s computation is reduced to a simple add operation. This is due to the pipelining of prediction’s computation, each step (summing two integer weights) being processed as soon as a new element of the path is executed. Thus, the vector of weights used to generate prediction, is selected according to the path leading up to a branch – based on all branch addresses from that path – rather than according to the current branch’s address alone, as the original perceptron does. This selection mechanism improves significantly the prediction accuracy too, because due to the path information used in the prediction process, the predictor is able to exploit the correlation between the output of the branch being predicted and the path leading up to that branch. To generate a prediction, the correlations of each component of the path are aggregated. This aggregation is a linear function of the correlations for that path. Since there are many paths leading to a branch, there are many different linear functions for that branch, and they form a piecewise-linear surface separating paths that lead to predicted taken branches from paths that lead to predicted not-taken branches. The piecewise linear branch prediction [10], is a generalization of neural branch prediction [25,11], which uses a single linear function for a given branch, and respectively path-based neural branch prediction [8], which uses a single global piecewise-linear function to predict all branches. The piecewise linear branch predictors use a piecewise-linear function for a given branch, exploiting in this way different paths that lead to the same branch in order to predict otherwise linearly inseparable branches. The piecewise linear branch predictors exploit better the correlation between branch outcomes and paths, yielding an IPC improvement of 4% over the path-based neural predictor [10], already presented by us in Chapter 2.

A conventional path-based neural predictor achieves high prediction accuracy, but its very deeply pipelined implementation makes it both a complex and power-intensive component, since for a history length of \( p \) it uses – to store the weights – \( p \) separately indexed SRAM arrays organized in a \( p \)-stage predictor pipeline. Each pipeline stage requires a separate row-decoder for the corresponding SRAM array, inter-stage latches, control logic and checkpointing support, all of this adding power and complexity to the predictor. Loh and Jiménez proposed in [15] two techniques to solve this problem. The first decouples the branch outcome history length from the path history length using shorter path history and a traditional long branch outcome history. In the original path-based neural predictor, the path history was always equal to the branch history length. The shorter path history
allows the reduction of the pipeline length, resulting in decreased power consumption and implementation complexity. The second technique uses the bias-weights to filter out highly biased branches (almost always taken or almost always not taken branches), and avoids consuming update power for these easy-to-predict branches. For these branches the prediction is determined only by the bias weight, and if it turns out to be correct, the predictor skips the update phase, which saves the associated power. The proposed techniques improve the prediction accuracy with about 1%, and more important, reduce power and complexity by decreasing the number of SRAM arrays, and reducing predictor update activity by 4-5%. Decreasing the pipeline depth to only 4-6 stages it is reduced the implementation complexity of the path-based neural predictor.

Tarjan and Skadron introduced in [23] the hashed perceptron predictor, which merges the concepts behind the gshare [17] and path-based perceptron predictors [8]. The previous perceptron predictors assign one weight per local, global or path branch history bit. This means that the amount of storage and the number of adders increases linearly with the number of history bits used to make a prediction. One of the key insights of Tarjan’s work is that one-to-one ratio between weights and number of history bits is not necessary. By assigning a weight not to a single history bit but to a sequence of bits (hashed indexing), a perceptron can work on multiple partial patterns making up the overall history. The hashed indexing consists in XOR-ing a segment of the global branch history with a branch address from the path history. Decoupling the number of weights from the number of history bits used to generate a prediction allows the reduction of adders and tables. Using hashed indexing, linearly inseparable branches, which are mapped to the same weight, can be accurately predicted, because each table acts like a small gshare predictor [17]. The hashed perceptron predictor improves accuracy by up to 27.2% over a path-based neural predictor.

G. Loh and D. Jiménez introduced in [14] a new branch predictor that takes the advantage of deep-history branch correlations. To maintain simplicity, they limited the predictor to use conventional tables of saturating counters. Thus, the proposed predictor achieves neural-class prediction rates and IPC performance using only simple PHT (pattern history table) structures. The disadvantage of PHTs is that their resource requirements increase exponentially with branch history length (a history length of $p$ requires $2^p$ entries in a conventional PHT), in contrast with neural predictors, whose size requirements increase only linearly with the history length. To deal
with very long history lengths, they proposed a Divide-and-Conquer approach where the long global branch history register is partitioned into smaller segments, each of them providing a short branch history input to a small PHT. A final table-based predictor combines all of these per-segment predictions to generate the overall decision. Their predictor achieves higher global performance (IPC) than the original global history perceptron predictor, outperforms the path-based neural predictors, and even achieves an IPC rate equal to the piecewise-linear neural branch predictor. Using only simple tables of saturating counters, it is avoided the need for large number of adders, and, in this way, the predictor is feasible to be implemented in hardware.

V. Desmet et al. [5] proposed a different approach for branch classification. They evaluated the predictive power of different branch prediction features using Gini-index metric, which is used as selection measure in the construction of decision trees. Actually, as an alternative to the well-known informational entropy, Gini-index is a metric of informational energy and in this case is used to identify the branches with high energy (entropy). In contrast to our further presented work Desmet used as input features both dynamic information (global and local branch history) and static information (branch type, target direction, ending type of taken-successor-basic-block).

In [7] the authors identified some program corpus and data structures that create “hard to predict” branches. In order to accurately predict the difficult branches the authors find additional correlation information beyond local and global branch history. In their approach the prediction table is addressed by a combination between structural information, value information and history of values that are tested in the condition of respective branch. More precisely it is presented a technique called speculative branch execution that uses a conventional data-value predictor to predict the input values of the branch instruction and, after that, executes the branch instruction using the predicted values. The main disadvantage of this method consists in the relatively high prediction latency, because the data-value prediction is followed by the speculative execution of the branch instruction. Therefore, it is proposed a Branch Difference Predictor (BDP) that maintains a history of differences between branch source register operands and uses it in the prediction process. Consequently, the value history information is used directly for branch prediction, reducing thus the latency. Since branch’s outcomes are determined by subtracting the two inputs, the branch’s source differences correlate very well with the branch’ outcomes. The branch’s
difference history is maintained per static branch in a Value History Table (VHT) and it is retrieved using the branch’s PC. In their prediction mechanism, only the difficult branches are predicted based on the branch source differences using the Rare Event Predictor (REP), while most branches are predicted using a conventional predictor. The VHT and the REP are simultaneously accessed using the PC and global branch history. If the value difference history matches a REP tag, then the REP provides the prediction. If the REP does not contain that certain pattern, the conventional branch predictor generates the prediction. Unlike our work presented further based on references [24, 27, 29, 19], Heil et al. didn’t use the path history information in order to do better predictions. Using the proposed prediction method based on data values significantly improves prediction accuracy for some certain difficult branches but overall improvements are quite modest. However there are some unsolved problems: they tested only particular cases of difficult branches, and also, they didn’t approach branch conditions with two input values. Their final conclusion suggests that researchers must be focused on the strong correlation between instructions producing a value and, respectively, the branch’s condition that would be triggered by that certain value. Another interesting approach focused on the same problem is presented in González J., González A., *Control-Flow Speculation through Value Prediction*, IEEE Transactions on Computers, Vol. 50, No. 12, December 2001.

In [31] the authors proposed a new approach, called ARVI (Available Register Value Information), in order to predict branches based on partial register values along the data dependence chain leading up to the branch. The authors show that for some branches the correlation between such register value information and the branch’s outcome can be stronger than either history or path information (however, this is not proved in a rigorous manner). Thus, the main idea behind the ARVI predictor is the following: if the essential values in the data dependence chain that determine the final branch’s behavior can be identified, then the branch’s outcome should be known. If the values involved in the branch’s condition are the same as in a prior occurrence then the outcome of the branch will be the same, too. If the branch’s register values are available then a table look up can provide the last outcome of that branch occurred with the same values. Unfortunately, the branch’s register values are rarely available at the prediction’s time. However, if values are available for registers along the dependence chain that leads up to the branch, then the predictor can use these values to index into a table and reuse the last branch’s behavior. Therefore, instead of relying only on branch history or path, the ARVI predictor includes the data
dependent registers as part of the prediction information. The ARVI predictor uses Data Dependence Table (DDT) to extract the registers corresponding to instructions along the data dependence chain leading up to the branch. The branch’s PC and the identifiers of the data dependent registers are hashed together and used to index the prediction table. The values of the data dependent registers are hashed together and used as a tag to distinguish the occurrences of the same path having different values in the registers. Thus, the ARVI predictor uses both path and value-based information to classify branch instances. A two-level predictor using ARVI at the second level achieves a 12.6% overall IPC improvement over the state-of-the-art two level predictors, for the SPEC’95 integer benchmarks. (The authors selected SPEC’95 integer benchmarks because their branch behavior was extensively studied permitting comparisons between different works.)

In [3] the authors are focusing on some difficult predictable branches into a Simultaneous Subordinate Micro-Threaded (SSMT) architecture. They defined a difficult path being a path that has a terminating branch that is poorly predicted when it executes from that path. A path represents a particular sequence of control-flow changes. It is shown that from 70% to 93.5% from branch miss-predictions are covered by these difficult paths, involving thus a significant challenge into the branch prediction paradigm. The proposed solution in dealing with these difficult predictable branches consists in dynamically construct micro-threads that can speculatively and accurately pre-compute branch outcomes, only along frequently miss-predicted paths. Obviously, micro-thread predictions must arrive in time to be useful. Ideally, every micro-thread would complete before corresponding difficult branch’s fetch. Observing that the data-flow within the set of instructions guaranteed to execute each time the path is encountered, it can be extracted a subset of instructions that will pre-compute the branch. The proposed micro-architecture contains structures to dynamically identify difficult paths (Path Cache), construct micro-threads (Micro-Thread Builder) and communicate predictions to the main thread. The proposed technique involves realistic average speed-ups of up to 10% but the average potential speed-up from perfect predicting these difficult branches is about 100%, suggesting the idea’s fertility. Unfortunately the authors didn’t investigate why these paths, respectively their associated final branches, are difficult predictable. In other words, a very important question is: why these “difficult paths” frequently lead to miss-predictions? We could suspect that we already gave the answer in our work [27] because these “difficult branches” might be, at least partially, exactly the unbiased branches in the
sense defined by us, and, therefore, difficult predictable. They could be more predictable even in a single threaded environment, by sufficiently growing history pattern length or extending prediction information, as we’ll show further in this work. Thus, our hypothesis is that SSMT environment represents a sufficient solution in order to solve these difficult branches, as the authors shown, but not a strictly necessary one.

3.2 FINDING UNBIASED BRANCHES. RESEARCH METHODOLOGY

Through this paragraph, based on [27, 19, 29], we are presenting the methodology of finding difficult predictable branches, as they are defined in our approach. The first goal is to find the difficult predictable branches in the SPEC2000 benchmarks [22]. As we already pointed out, we consider that a branch in a certain context is difficult predictable if it is unbiased – meaning that the branch’s behavior (Taken/Not Taken) is not sufficiently polarized for that certain context (local branch history, global history, etc.) – and the taken and not taken outcomes are non-deterministically shuffled. More precisely, it is difficult predictable if its behavior can’t be approached as a Markov process of a certain order and therefore the branch’s direction is quite unpredictable through any simple Markov predictor. 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. A feature is the binary context of prediction information, represented on \( p \) bits, such as local history, global history or path. Each static branch finally has associated \( k \) dynamic contexts in which it can appear (\( k \leq 2^p \)). A context instance is a dynamic branch executed in the respective context frame. We introduce the polarization index (\( P \)) of a certain branch context as follows:

\[
P(S_i) = \max(f_0, f_1) = \begin{cases} f_0, & f_0 \geq 0.5 \\ f_1, & f_0 < 0.5 \end{cases}
\]  

(1)

where:
• $S = \{s_1, s_2, ..., s_k\}$ = set of distinct contexts that appear during all branch’s instances;
• $k$ = number of distinct contexts, $k \leq 2^p$, where $p$ is the length of the binary context;
• $f_0 = \frac{T}{T + NT}$, $f_1 = \frac{NT}{T + NT}$, $NT$ = number of “not taken” branch instances corresponding to context $S_i$, $T$ = number of “taken” branch instances corresponding to context $S_i$, $(\forall)i = 1, 2, ..., k$, and obviously $f_0 + f_1 = 1$;
• If $P(S_i) = 1$, $(\forall)i = 1, 2, ..., k$, then the context $S_i$ is completely biased (100%), and thus, the 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 is not predictable if the taken and not taken outcomes are non-deterministically shuffled.

As a real example taken from our work [24], the next table presents only a branch (labeled 35) belonging to “Permuate” Stanford HSA (Hatfield Superscalar Architecture) benchmark, having a strange behavior for $HRg=xxxxx101$ and $HRI=10$, indifferent if $HRg$ is on 3 bits or on 8 bits length.

<table>
<thead>
<tr>
<th>HRg</th>
<th>HRI</th>
<th>Taken (%)</th>
<th>Not taken (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>10</td>
<td>1680 (67%)</td>
<td>839 (33%)</td>
</tr>
<tr>
<td>11101</td>
<td>10</td>
<td>1680 (67%)</td>
<td>839 (33%)</td>
</tr>
<tr>
<td>11111101</td>
<td>10</td>
<td>840 (59%)</td>
<td>579 (41%)</td>
</tr>
<tr>
<td>01011101</td>
<td>10</td>
<td>840 (76%)</td>
<td>260 (24%)</td>
</tr>
</tbody>
</table>

Table 3.1. The behaviour of branch 35 belonging to “perm” benchmark

If the taken and respectively 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 appear as randomly shuffled, because in this case, the predictors cannot learn their behavior. Consider the following trivial example, a branch in a certain dynamic context shows the following behaviour: TTTTTT… or NNNNNN… in which case the transitions are always taken or always not-taken, and would be biased and easy-to-predict. However, a branch in a certain context that is stochastic (non-deterministic) will show a highly
shuffled behaviour, which would result in the branch being unbiased and
difficult-to-predict with its transitions toggling between T and N. We
therefore consider that the rate of transition between branch outcomes is an
important feature that can be applied to branch prediction. We introduce the
distribution index which is the based on the rate of transitions as shown by
equation (2):

\[
D(S_i) = \begin{cases} 
0, & n_r = 0 \\
\frac{n_r}{2 \cdot \min(NT, T)}, & n_r > 0 
\end{cases} 
\quad (2)
\]

where:

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

As it can be observed in Figure 3.2, 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 index \(P\)). We approached an iterative
methodology: a certain feature set is evaluated only on the unbiased
branches determined with the previous feature sets, because the rest were
solved with the previously considered feature sets. Gradually increasing the
lengths of feature sets and reapplying the algorithm shorten this list. For the
final list of unbiased branches we will try to find new relevant feature sets in
order to further improve their polarization index. More exactly, a certain
branch context is evaluated only if that branch was unbiased for its all
previously analyzed contexts. Thus, the final list of unbiased branches
contains only the branches that were unbiased for all their contexts. The
contexts’ lengths were varied from 16 bits to 28 bits.
This approach is more efficient than one that repeats each time the algorithm on all branches. Besides producing some unpleasant aspects related to simulation time (some days / benchmark), memory (gigabytes of memory needed), the second method would prove even not very accurate. This is because a shorter one can solve some of the branches that are not solved by a longer context. Following, we give a short example to explain the last statement. Suppose we have a dynamic context like below:

As it is shown in Figure 3.3, the branches in this context are highly predictable and should not be included in unbiased context list. Considering
now we have a longer context (10 bits instead of 9 bits), this solvable context would be split in two sub contexts:

![Diagram showing longer context and its split into two sub contexts]

Figure 3.4. Longer context might involve some false problems – *an unbiased context*

One of the longer contexts will have polarization rate $P' = 0.993525$ (*biased context*) and the other will have $P'' = 0.508167$ (*unbiased context*). The one with polarization index $P''$ will remain in the unbiased list even if it is nearly perfectly solvable by a shorter context (see $P$). Through our iterative approach we avoided the occurrence of false problems extending the context. In our experiments we concentrated only on benchmarks with a percentage of unbiased branch context instances (obtained with relation (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 were 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 difficult predictable branches during the current stage.

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

(3)
Where $\text{NUB}_i$ is the total number of unbiased branch context instances on benchmark $i$, and $\text{NB}_i$ is the number of dynamic branches on benchmark $i$ (therefore, the total number of branch context instances).

### 3.3 SOME EXPERIMENTAL RESULTS [27, 19]

All further SPEC2000 simulation results are reported on 1 billion dynamic instructions, skipping the first 300 million instructions. We note with LH(p)-GH(q)-GHPC(r) branches unbiased on local history (LH) of $p$ bits, global history (GH) of $q$ bits, and global history XOR-ed by branch address (GHPC) on $r$ bits. In the same manner, for all feature set extensions simulated in this work, LH(p)-GH(q)-GHPC(r)$\rightarrow$F(s) denotes that we measure the polarization rate using feature F on $s$ bits (if the feature is the local history, global history or global history XOR-ed by branch address) and/or on $s$ PCs (in the case of path), evaluating only the branches unbiased for local history of $p$ bits, global history of $q$ bits, and global history XOR-ed by branch address on $r$ bits.

**Pattern-based Correlation**

We started our study evaluating the branch contexts from SPEC2000 benchmarks on local branch history of 16 bits: LH(0)-GH(0)-GHPC(0)$\rightarrow$LH(16). The column *Dynamic Branches* from Table 3.2 contains the number of all dynamic conditional branches for each benchmark. The column *Static Branches* contains the number of static branches for each benchmark. For each benchmark we generated using relation (1) a list of unbiased branch contexts, having polarization less than 0.95. We considered that the branch contexts with polarization greater than 0.95 will obtain relatively high prediction accuracies (around 0.95); therefore, in these cases we considered that the potential improvement of the prediction accuracy is quite low. Table 3.2 compares the prediction accuracies obtained using the well-known PAg predictor stimulated by all contexts, and, respectively, only by unbiased contexts. We choose this particular predictor due to the fact that it is stimulated with branch’s local history. The predictors have the same configuration: 1024 entries in the first
level (L1size = 1024), 16 bit LHR lengths (W = 16), and $2^{16}$ entries in the second level (L2size = 65536). For the PAg predictor we used the sim-bpred simulator from SimpleScalar Simplesim-3.0 [21, 30], with the following options: -bpred 2lev -bpred:2lev 1024 65536 16 0.

<table>
<thead>
<tr>
<th>SPEC 2000 Benches</th>
<th>Dynamic Br.</th>
<th>Static Br.</th>
<th>Pred. Accuracy</th>
<th>PAg</th>
<th>PAg / Unbiased Contexts</th>
<th>Unbiased Context Instances (P&lt;0.95)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>118321124</td>
<td>370</td>
<td>Address</td>
<td>0.9838 0.8267</td>
<td>6812313</td>
<td>5.76%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.9838 0.8267</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td>85382841</td>
<td>1777</td>
<td>Address</td>
<td>0.9367 0.7748</td>
<td>17589658</td>
<td>20.60%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.9367 0.7748</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip</td>
<td>42591123</td>
<td>211</td>
<td>Address</td>
<td>0.9010 0.6913</td>
<td>11252986</td>
<td>26.42%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.9010 0.6913</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>71504537</td>
<td>136</td>
<td>Address</td>
<td>0.8905 0.7423</td>
<td>27692102</td>
<td>38.73%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.8905 0.7423</td>
<td></td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td>70616018</td>
<td>239</td>
<td>Address</td>
<td>0.8490 0.7078</td>
<td>31763071</td>
<td>44.98%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.8490 0.7078</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>90868660</td>
<td>17248</td>
<td>Address</td>
<td>0.9284 0.8049</td>
<td>9809360</td>
<td>10.80%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.9299 0.8059</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>79880717</td>
<td>3330</td>
<td>Address</td>
<td>0.9149 0.7579</td>
<td>17486582</td>
<td>24.55%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direction</td>
<td>0.9151 0.7581</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2. Prediction accuracy on the unbiased branch contexts for local history of 16 bits.

The column Unbiased Context Instances contains – for each benchmark – the number of unbiased context instances and respectively the percentage of unbiased context instances reported to all context instances (dynamic branches). As it can be observed in Table 3.2, the bzip, gzip and twolf benchmarks are difficult predictable with the original PAg predictor (prediction accuracies less than 0.9 – on all branches). The low prediction accuracies obtained with PAg predicting only the unbiased contexts (0.75 at average), confirm our initial hypothesis (unbiased branches are not predictable). Also, the relatively high percentages of unbiased branches (at average 24.55%) show high improvement potential from the predictability point of view.
We continue our work analyzing a global branch history of 16 bits only on the local branch contexts that we already found unbiased for local branch history (see Table 3.2 – last column). In other words, we further used a dynamic branch in our evaluations only if its 16 bit local context is one of the unbiased local contexts: \( \text{LH}(16)\)-\( \text{GH}(0)\)-\( \text{GHPC}(0) \) → \( \text{GH}(16) \). The column Simulated Dynamic Branches from Table 3.3 contains the number of evaluated dynamic branches (\( \text{LH}(16)\)-\( \text{GH}(0)\)-\( \text{GHPC}(0) \)) and respectively their percentages reported to all dynamic branches. The column Simulated Static Branches represents the number of static branches evaluated within each benchmark. For each benchmark we generated using relation (1) a list of unbiased branch contexts on local and global history of 16 bits (\( \text{LH}(16)\)-\( \text{GH}(16)\)-\( \text{GHPC}(0) \)), having polarization less than 0.95. Table 3.3 also compares the prediction accuracies obtained with the well-known GAg predictor predicting all branches, and respectively predicting only these unbiased branch contexts. We choose this particular predictor due to the fact that it is stimulated with global history. The last column contains the number of unbiased branch context instances and respectively their percentages reported to all dynamic branches. The predictors have the same configuration: one global history register of 16 bits, and \( 2^{16} \) entries in the second level (\( \text{L2size} = 65536 \)). For the GAg predictor we used the \texttt{sim-bpred} simulator from \textit{Simplesim-3.0} [21], with the following options: \texttt{-bpred 2lev -bpred:2lev 1 65536 16 0}. 

\[ \text{PC}_{\text{low}} \quad \text{PC}_{\text{high}} \]

\[ \log_2 \text{L1size} \]

\[ \text{LHR}_0 \]

\[ \text{LHR}_1 \]

\[ \text{LHR}_{\text{L1size}-1} \]

\[ \text{L2size} \]

\[ \text{W} \]

\[ \text{W} \text{ bits} \]

\[ \text{Predicted PC} \]

\[ \text{Prediction bits} \]

\textbf{Figure 3.5.} The PAg branch predictor scheme
Analyzing comparatively Tables 3.1 and 3.2, we observe that the global branch history reduced the average percentage of unbiased branch context instances from 24.55% to 17.48%, and it also increased the average prediction accuracy from 91% with the PAg to 92% with the GAg predictor. However, the remaining unbiased branch contexts (for local and global history of 16 bits) are more difficult predictable: on these branch contexts, with the GAg predictor, we measured an average prediction accuracy of only 72%, confirming again the fact that unbiased branches are highly unpredictable.

The next feature set we analyzed is the XOR between a global branch history of 16 bits and the lower part of branch address (PC bits 18÷3): LH(16)-GH(16)-GHPC(0)→GHPC(16). We used again only the branch contexts we found unbiased for the previous feature sets (local and global
branch history of 16 bits). In other words, we used a dynamic branch in our evaluations only if its 16 bit local context is one of the unbiased local contexts (Table 3.1), and its 16 bit global context is one of the unbiased global contexts (Table 3.3). For each benchmark we generated again using relation (1), a list of unbiased branch contexts with polarization less than 0.95 (LH(16)-GH(16)-GHPC(16)).

Table 3.4 compares the prediction accuracies obtained using the Gshare (McFarling) predictor stimulated by all contexts, and respectively by the determined unbiased branch contexts. The last column contains for each benchmark the number of unbiased branch context instances and respectively their percentages reported to all dynamic branches. The predictors have the same configuration: one global history register of 16 bits, and $2^{16}$ entries in the second level (L2size = 65536). For the Gshare predictor we used the sim-bpred simulator from Simplesim-3.0 [21], with the following options: -bpred 2lev -bpred:2lev 165536 16 1.

<table>
<thead>
<tr>
<th>SPEC 2000 Benchs</th>
<th>Simulated Dynamic Branches</th>
<th>Static Branches</th>
<th>Prediction Accuracy</th>
<th>Gshare</th>
<th>Gshare / Unbiased Contexts</th>
<th>Unbiased Context Instances (P&lt;0.95)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>3887052</td>
<td>3.28%</td>
<td>19</td>
<td>Address 0.9849 0.8302</td>
<td>3887050</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.9849 0.8302</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td>11064817</td>
<td>12.95%</td>
<td>504</td>
<td>Address 0.9510 0.7031</td>
<td>11063791</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.9510 0.7032</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip</td>
<td>9969701</td>
<td>23.40%</td>
<td>76</td>
<td>Address 0.9110 0.6563</td>
<td>9969678</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.9110 0.6563</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>20659305</td>
<td>28.89%</td>
<td>51</td>
<td>Address 0.9231 0.7352</td>
<td>20659290</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.9231 0.7352</td>
<td></td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td>22893014</td>
<td>32.41%</td>
<td>112</td>
<td>Address 0.8837 0.6676</td>
<td>22892985</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.8837 0.6676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>3563776</td>
<td>3.92%</td>
<td>2642</td>
<td>Address 0.9603 0.7388</td>
<td>3561998</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.9623 0.7398</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>12006278</td>
<td>17.48%</td>
<td>567.33</td>
<td>Address 0.9356 0.7218</td>
<td>12005798</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction 0.9360 0.7220</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4. Prediction accuracy on the unbiased branch contexts after XOR-ing between global history and branch address on 16 bits.
As it can be observed, XOR-ing global history and branch address by using the \textit{Gshare} predictor, the prediction accuracy increased with almost 1%, but it didn’t reduced the percentage of unbiased context instances. The high percentages of unbiased branch context instances in the case of \textit{bzip}, \textit{gzip} and \textit{twolf} benchmarks represent a potential improvement of prediction accuracy.

For the determined unbiased branch contexts we are analyzing now if the taken and respectively not taken outcomes are or aren’t 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 (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 (LH(16)-GH(16)-GHPC(16)). Table 3.5 shows for each benchmark the percentages of branch contexts with distribution indexes belonging to five different intervals in the case of local branch history. In the same way, Tables 3.5 and 3.6 present the distribution indexes in the case of global history and respectively XOR-ing between global history and branch address.

Tables 3.4, 3.5 and 3.6 show that in the case of unbiased branch contexts, the taken and respectively not taken outcomes are not grouped separately, more than this, they are highly shuffled: the percentage of unbiased branch contexts having highly shuffled outcomes is 76.30\% in the case of local history of 16 bits (see Table 3.5, D>0.4), 89.37\% in the case of global
history of 16 bits (see Table 3.6), and 89.37% in the case of global history XOR-ed by branch address on 16 bits (see Table 3.7). It can be observed that we obtained the same distribution indexes for both the global history and respectively XOR-ing between global history and branch address (Tables 3.5 and 3.6, D>0.4). These results are strongly correlated with the low prediction accuracies obtained on these unbiased branches.

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 behavior of the branch can be learned.

<table>
<thead>
<tr>
<th>SPEC 2000 Benchs</th>
<th>Simulated Dynamic Branches</th>
<th>Simulated Static Branches</th>
<th>Distribution Rate (D) [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>[0, 0.2)</td>
</tr>
<tr>
<td>mcf</td>
<td>3887050</td>
<td>3.28%</td>
<td>19</td>
</tr>
<tr>
<td>parser</td>
<td>11063791</td>
<td>12.95%</td>
<td>483</td>
</tr>
<tr>
<td>hzip</td>
<td>9969678</td>
<td>23.40%</td>
<td>75</td>
</tr>
<tr>
<td>gzip</td>
<td>20659290</td>
<td>28.89%</td>
<td>51</td>
</tr>
<tr>
<td>twolf</td>
<td>22892985</td>
<td>32.41%</td>
<td>110</td>
</tr>
<tr>
<td>gcc</td>
<td>3561998</td>
<td>3.91%</td>
<td>2553</td>
</tr>
<tr>
<td>Average</td>
<td>12005798</td>
<td>17.47%</td>
<td>548.5</td>
</tr>
</tbody>
</table>

Table 3.5. Distribution rates on local history of 16 bits evaluating only the branches that were unbiased on all their 16 bit contexts

<table>
<thead>
<tr>
<th>SPEC 2000 Benchs</th>
<th>Simulated Dynamic Branches</th>
<th>Simulated Static Branches</th>
<th>Distribution Rate (D) [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>[0, 0.2)</td>
</tr>
<tr>
<td>mcf</td>
<td>3887050</td>
<td>3.28%</td>
<td>19</td>
</tr>
<tr>
<td>parser</td>
<td>11063791</td>
<td>12.95%</td>
<td>483</td>
</tr>
<tr>
<td>hzip</td>
<td>9969678</td>
<td>23.40%</td>
<td>75</td>
</tr>
<tr>
<td>gzip</td>
<td>20659290</td>
<td>28.89%</td>
<td>51</td>
</tr>
<tr>
<td>twolf</td>
<td>22892985</td>
<td>32.41%</td>
<td>110</td>
</tr>
<tr>
<td>gcc</td>
<td>3561998</td>
<td>3.91%</td>
<td>2553</td>
</tr>
<tr>
<td>Average</td>
<td>12005798</td>
<td>17.47%</td>
<td>548.5</td>
</tr>
</tbody>
</table>

Table 3.6. Distribution rates on global history of 16 bits evaluating only the branches that have all their 16 bit contexts unbiased
Table 3.7. Distribution rates on the XOR between global history and branch address on 16 bits evaluating only the branches that have all their 16 bit contexts unbiased

We continued our evaluations extending the lengths of feature sets from 16 bits to 20, 24 and respectively 28 bits, our hypothesis being that the longer feature sets will increase the polarization index and, therefore, the potential prediction accuracy. We used the same simulation methodology, and in Table 3.8 we reported for each context length \( p \) (\( p=16, 20, 24, 28 \) bits) only the results obtained after we have eliminated the branch context instances that were biased on all considered feature sets with length \( p \) (local history, global history and respectively XOR of global history and branch address). Thus, column UBCI (Unbiased Branch Context Instances) shows for each benchmark the percentage of unbiased branch context instances after each context length extension (LH(16)-GH(16)-GHPC(16), LH(20)-GH(20)-GHPC(20), LH(24)-GH(24)-GHPC(24), LH(28)-GH(28)-GHPC(28)).

Column SDB (Simulated Dynamic Branches) shows the number of branch context instances that remain still unbiased after the previous context extension. Column SSB (Simulated Static Branches) shows the number of simulated static branches for each benchmark.

Table 3.8. Percentages of unbiased branch context instances (P<0.95) after each feature set extension
As it can be observed, in the case of the gcc benchmark, extending the feature set length to 28 bits, the percentage of the unbiased context instances is less than the considered threshold $T = 1\%$ (see relation (3)), and thus we eliminated this benchmark from our next evaluations. Therefore we consider that the conditional branches from the gcc benchmark are not difficult predictable using feature lengths of 28 bits. As a consequence the results obtained with the gcc benchmark on 28 bits are not included in the average results. Extending the feature set length from 16 bits to 28 bits, the percentages of unbiased dynamic branches decreased at average from 17.47\% to 6.19\% (see Table 3.8). Despite of the feature set extension, the number of unbiased dynamic branches remains still high (6.19\%), and thus, it is obvious that using longer feature sets is not sufficient. On the other hand, longer features are not feasible to be exploited through classic branch prediction.

![Figure 3.8](image_url)

**Figure 3.8.** Reduction of unbiased context instances (P<0.95) by extending the lengths of feature sets

The global history solves at average 2.56\% of the unbiased dynamic branches not solved with local history (see Figure 3.8). The hashing between global history and branch’s address (through a XOR operation) behaves just similar with the global history, and it does not further improve the polarization rate. In Figure 3.8 it can be also observed that increasing the branch history, the percentage of unbiased dynamic branches decreases, suggesting a correlation between branches situated at a large distance in the dynamic instruction stream. The results also show that the “ultimative
predictability limit” of history context-based prediction is approximatively 94%, considering biased branches as perfectly predictable and respectively unbiased branches as completely unpredictable. A conclusion based on our simulation methodology is that 94% of dynamic branches can be solved with prediction information of up to 28 bits (some of them are solved with 16 bits, others with 20, 24 or even 28 bits).

We have also studied the polarization of branches, using a little different simulation methodology. We evaluated local history concatenated with global history. The evaluation results presented in Table 3.9 show that these longer contexts, due to their better precision, have higher polarization index. Comparing the results, it is obvious that a certain feature set LH(p)-GH(p) from Table 3.9 is approximatively equivalent in terms of polarization rate with feature set GH(p+4) or GHPC(p+4) from Table 3.8. In other words, it is obtained the same percentage of unbiased context instances for LH(p)-GH(p), GH(p+4) and GHPC(p+4) feature sets, but the number of bits in the correlation information is different: (p+p) bits of local and global history, and respectively (p+4) bits of global history.

<table>
<thead>
<tr>
<th>SPEC2000Benchs</th>
<th>LH(0)-GH(0)</th>
<th>LH(16)-GH(0)</th>
<th>LH(16)-GH(16)</th>
<th>LH(20)-GH(20)</th>
<th>LH(24)-GH(24)</th>
<th>LH(28)-GH(28)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LH(0)-</td>
<td>LH(16)-</td>
<td>LH(16)-</td>
<td>LH(20)-</td>
<td>LH(24)-</td>
<td>LH(28)-</td>
</tr>
<tr>
<td></td>
<td>GH(0)</td>
<td>GH(16)</td>
<td>GH(16)</td>
<td>GH(20)</td>
<td>GH(24)</td>
<td>GH(28)</td>
</tr>
<tr>
<td>bzip</td>
<td>26.42%</td>
<td>12.83%</td>
<td>7.53%</td>
<td>4.70%</td>
<td>3.08%</td>
<td>2.10%</td>
</tr>
<tr>
<td>gzip</td>
<td>38.73%</td>
<td>24.58%</td>
<td>17.84%</td>
<td>12.67%</td>
<td>9.12%</td>
<td>6.16%</td>
</tr>
<tr>
<td>mcf</td>
<td>5.76%</td>
<td>3.09%</td>
<td>2.44%</td>
<td>2.09%</td>
<td>1.78%</td>
<td>1.49%</td>
</tr>
<tr>
<td>parser</td>
<td>20.61%</td>
<td>7.42%</td>
<td>4.77%</td>
<td>3.01%</td>
<td>1.98%</td>
<td>1.40%</td>
</tr>
<tr>
<td>twolf</td>
<td>44.98%</td>
<td>23.94%</td>
<td>12.79%</td>
<td>8.28%</td>
<td>5.70%</td>
<td>3.90%</td>
</tr>
<tr>
<td>gcc</td>
<td>10.85%</td>
<td>2.50%</td>
<td>1.41%</td>
<td>0.88%</td>
<td>0.58%</td>
<td>0.30%</td>
</tr>
<tr>
<td>Average</td>
<td>24.56%</td>
<td>12.39%</td>
<td>7.80%</td>
<td>6.15%</td>
<td>4.33%</td>
<td>3.01%</td>
</tr>
</tbody>
</table>

Table 3.9. The percentages of unbiased context instances using only the local history concatenated with the global history.

Taking into account that increasing the prediction accuracy with 1%, the IPC (instructions-per-cycle) is improved with more than 1% (it grows non-linearly) [28], there are great chances to obtain considerably better overall performances even if not all of the 6.19% difficult predictable branches 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 rate. Focalising on these unbiased branches – in order to design some efficient
path-based predictors for them [18, 24] – 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. 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 feasible for hardware implementation. The question is: what feature set length is really feasible 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 quite feasible solution in this case might be given by path-predictors.

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 [24]. 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 precisely, and therefore its prediction accuracy might be better. Our next goal is to extend the correlation information with the path, according to the above idea [24]. Extending the correlation information in this way, suggests that at different occurrences of a certain branch with the same global history context, the path contexts can be different.

We started our evaluations regarding the path, studying the gain obtained by introducing paths of different lengths. The analyzed feature consists of a global branch history of 16 bits and the last $p$ PCs. We applied this feature only to dynamic branches that we already found unbiased ($P<0.95$) for local and global history of 16 bits and respectively global history XOR-ed by branch address on 16 bits.
The column \( \text{LH}(16)\text{-GH}(16)\text{-GHPC}(16) \) from Table 3.10, presents the percentage of unbiased context instances for each benchmark. Columns codified with \( \text{LH}(16)\text{-GH}(16)\text{-GHPC}(16) \rightarrow \text{PATH}(1) \), \( \text{LH}(16)\text{-GH}(16)\text{-GHPC}(16) \rightarrow \text{PATH}(16) \) and \( \text{LH}(16)\text{-GH}(16)\text{-GHPC}(16) \rightarrow \text{PATH}(20) \) present the percentages of unbiased context instances obtained using a global history of 16 bits and a path of one, 16 and respectively 20 PCs. The last column presents the percentages of unbiased context instances extending the local history to 20 bits (without path). For each feature is presented the gain opposite to the first column average. It can be observed that a path of one introduces an insignificant gain of 0.2%. Even a path of 20 introduces a gain of only 2.11% related to the more significant gain of 4.30% introduced by an extended local branch history of 20 bits. The results show (Table 3.10) that the path is useful only in the case of short contexts. Thus, a branch history of 16 bits compresses and approximates well the path information. In other words, a branch history of 16 bits spreads well the different paths that lead to a certain dynamic branch.

We continue our presentation evaluating – on all branches (non-iterative simulation) – paths of different lengths \( p \) PCs used together with global histories of the same length \( p \) bits. The results are presented in Figure 3.9 where they are compared with the results obtained using only global history. The results obtained with long global history patterns (contexts) are closer to those obtained with path patterns of the same lengths, meaning that long global history \( p \) bits approximates very well the longer path information \( p \) PCs.
As it can be observed in Figure 3.9, an important gain is obtained through path in the case of short contexts (p<12). A branch history of more than 12 bits, compresses well the path information, and therefore, in these cases, the gain introduced by the path is not significant. Further, in the end of this section there are presented some results obtained applying the same methodology on Branch Prediction World Championship benchmarks – proposed and supported by Intel Co. [1, 13]; see and http://www.microarch.org/micro39/. We continue to evaluate – on all branches using the non-iterative simulation – paths of different lengths (p PCs) used together with global histories of the same lengths (p bits). The results are presented in Figure 3.10 and they are compared with the results obtained using only global history. As it can be observed, the results produced (unbiased context instances ratio) processing Intel’s benchmarks have the same profile like that obtained on SPEC2000 benchmarks. Actually, rich contexts (long pattern) reduce almost to zero the advantage introduced by using the path information (!). The main difference observed, analyzing Figures 3.8 and 3.9, consists in the different value of these ratios (much bigger on SPEC benchmarks) – due to their different characteristics and functions [13]. However, it must be mentioned again that while SPEC benchmarks were simulated on 1 billion dynamic instructions the Intel’s trace-benchmarks were entirely simulated, but the total number of dynamic instruction is smaller (under 30 millions).
Figure 3.10. The gain introduced by the path for different context lengths – Intel’s benchmarks

The same problem but a bit different approach [19]

The main idea is the same; to start with a shorter context and make a list of unbiased non-deterministic shuffled branches. We evaluated local history concatenated with global history. Gradually increasing the context length and reapplying the algorithm shorten this list. Figure 3.11 presents this approach as it was applied. Again we considered a branch context as unbiased if its corresponding $P < 0.95$. We started to analyze to polarization using GHR Len=0 and LHR Len=16. We continued by counting how many dynamic branches are found in unbiased contexts ($\text{sumIk}$) and we divide this number by the total number of dynamic context instances founded in the current simulation session ($\text{NDyn}$). For $\text{sumIk}/\text{NDyn} > T$, the algorithm is continued until remaining number of unbiased branches becomes insignificant. Summarizing the algorithm in pseudo-code we obtained the following description:

```python
countsConfigurations=16_0, 16_16, 20_20, 24_24, 28_28, 32_32
calculate NDyn = count (all conditional branches)
branchesToConsider = all conditional branches
foreach ctxCfg in countsConfigurations
    set1 = find contexts having polarization rate in [0.5-0.95)
    set2 = find contexts having polarization rate in [0.95-1.0)
    sumIk = branchCount( set1)
```

We applied the algorithm for each benchmark in the test set.

![Diagram of the algorithm](image)

**Figure 3.11.** Finding difficult branches: an iterative approach

Now, there are given the results of our analysis on the used SPEC 2000 benchmarks.

<table>
<thead>
<tr>
<th>Algorithm step</th>
<th>Step formalization</th>
<th>Unbiased branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td>A* = Local16</td>
<td>24.56%</td>
</tr>
<tr>
<td>B*</td>
<td>B* = Local16Global16(A*)</td>
<td>12.39%</td>
</tr>
<tr>
<td>C*</td>
<td>C* = Local20Global20(B*)</td>
<td>7.80%</td>
</tr>
<tr>
<td>D*</td>
<td>D* = Local24Global24(C*)</td>
<td>5.27%</td>
</tr>
<tr>
<td>E*</td>
<td>E* = Local28Global28(D*)</td>
<td>3.71%</td>
</tr>
<tr>
<td>F*</td>
<td>F* = Local32Global32(E*)</td>
<td>2.58%</td>
</tr>
</tbody>
</table>

**Table 3.11.** Centralized results created by averaging the data

As we can see from the above table, for a long history (32 bits GHR and 32 bits LHR) the maximum obtainable prediction accuracy is 97.42%. By using
a different chain of steps and a more sophisticated chaining method, smaller number of unbiased context will be obtained (e.g. Instead of having $B^* = \text{Local16Global16}(A^*)$ we can have $B^* = \text{Local16Global4}(A^*)$ and $C^* = \text{Local16Global16}(B^*)$. Therefore the chain $A^* B^* C^* D^* E^* F^*$ is definitely sub optimal.

Figure 3.12. Evolution of averaged unbiased list size

Most of the present-day predictors cannot use very long contexts and also cannot use dynamic reconfigurable contexts length to get the full advantages of the iterative approach. At this moment, our scientific hypothesis is that path information might further reduce the number of unbiased branches. As a consequence, we start our analysis by adding the last branch PC to the $B^*$ context. Without adding this path information the percentage of biased branches is 87.6066%. After adding path of size 1 the percentage of biased contexts increased to 87.6277%. The percentage of biased contexts is not significant higher that the one without any path. By increasing path length to 3, the percentage increases to 87.6644%. Even increasing the path length to 20 will not help very much.

A simple analytical model

High prediction accuracy is vital especially in the case of multiple instruction issue processors. In this section we assume the analytical model
proposed in [2] related to a superscalar processor that ignores stalls like cache misses and bus conflicts focalizing only about the penalty introduced by branch miss-prediction. Considering Branch Penalty (BP) as the average number of wasted cycles due to a branch miss-prediction for each dynamic instruction, it can be written the following relation:

$$BP = C \cdot (1 - Ap) \cdot b \cdot IR$$  \hspace{1cm} [wasted clock / instruction]  \hspace{1cm} (4)

Where we denoted:

- \(C\) = number of penalty cycles wasted due to a branch miss-prediction;
- \(Ap\) = prediction accuracy;
- \(b\) = the ratio of the number of branches reported to the number of total instructions;
- \(IR\) = the average number of instructions that are executed per cycle.

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

$$CPI_{\text{real}} = CPI_{\text{ideal}} + BP$$  \hspace{1cm} [clock cycle / instruction]  \hspace{1cm} (5)

Where:

- \(CPI_{\text{ideal}}\) = represents the average number of cycles per instruction considering a perfect branch prediction (\(Ap=100\% \Rightarrow BP=0\)). It is obviously that \(CPI_{\text{ideal}} < 1\).
- \(CPI_{\text{real}}\) = represents the average number of cycles per instruction considering a real branch prediction (\(Ap<100\% \Rightarrow BP>0 \Rightarrow CPI_{\text{real}} > CPI_{\text{ideal}}\)).

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

$$IR_{\text{real}} = \frac{1}{CPI_{\text{real}}} = \frac{1}{\frac{1}{CPI_{\text{ideal}}} + BP}$$  \hspace{1cm} [instruction / clock cycle]  \hspace{1cm} (6)

The relation (6) proves the non-linearly correlation between processing rate (IR) and prediction accuracy (Ap). With these metrics, we adapted the
model to our results presented during this paragraph. Further, we use the following notations:

\[ x = \text{the ratio of biased context instances} \]
\[ 1 - x = \text{the ratio of unbiased context instances} \]

In our simulations (see Table 3.4) we obtained using the \textit{Gshare} predictor the global prediction accuracy \( A_{\text{global}} = 93.60\% \) (prediction applied to all branches) and respectively the unbiased prediction accuracy \( A_{\text{unbiased}} = 72.2\% \) (only unbiased branches were predicted). Since \( A_{\text{global}} \) represents a weighted mean among predictions accuracies applied both to bias and unbiased branches, it can be determined the biased prediction accuracy \( A_{\text{biased}} \).

\[
A_{\text{global}} = x \cdot A_{\text{biased}} + (1-x) \cdot A_{\text{unbiased}}
\] (7)

Considering \( A_{\text{global}} = 0.936 = 0.8253 \cdot A_{\text{biased}} + 0.1747 \cdot 0.722 \), it results that \( A_{\text{biased}} = 0.9813 \). Obviously, if we would predict the unbiased branches with a more powerful branch predictor having, for example, 95\% prediction accuracy, involves an accuracy gain: \( \text{Accuracy gain} = (0.95 - 0.722) \cdot (1-x) \). More than that, this accuracy gain involves a processing rate speed-up according to (4) and (6) formulas. This gain justifies the importance and the necessity of finding and solving the difficult predictable branches. However, finding of predictor that obtains so high prediction accuracy was beyond the scope of our work.

Therefore, further we determined how much is influenced the branch penalty (BP) by the increasing of context length and what is the speed-up in these conditions. For this, we softly modified the initial model proposed by Chang [2] by substituting \( A_{\text{p}} \) with our \( A_{\text{global}} \), according to formula (7). Thus, it is considered a penalty introduced for mis-prediction of biased branches – the term \( (1-A_{\text{biased}}) \cdot x \), respectively for considered wrong prediction of all unbiased branches \( (A_{\text{unbiased}}=0) \) – the term \( (1-x) \).

<table>
<thead>
<tr>
<th>Model proposed by Chang</th>
<th>Our “modified” model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( BP = C \cdot (1-A_{\text{p}}) \cdot b \cdot IR )</td>
<td>( BP = C \cdot b \cdot IR \cdot [1-x \cdot A_{\text{biased}}] ) (8)</td>
</tr>
</tbody>
</table>

Figure 3.8 proves a decrease of unbiased branches \((1-x)\) by extending the context length that leads to a reduction of branch penalty (BP) according to (8), and implicitly to a greater IR according to (6). It can be written the followings relations (↗ means a growth):
Context (Features Set) Length $\mathcal{F} \Rightarrow x \mathcal{F} \Rightarrow BP \mathcal{V} \Rightarrow IR \mathcal{F} \Rightarrow \exists$ Relative Speedup > 0

Next, we computed the IR relative speedup when varying the context length. Starting from the well-known metric $Speedup = \frac{IR(L)}{IR(16)} \geq 1$, where $L$ is the feature’s length, $L \in \{20, 24, \text{and } 28\}$, we obtained the relative speedup:

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

Figure 3.13 illustrates the IR relative speedup obtained when varying the context length. The baseline processor model has an $IR_{\text{ideal}}$ of 4 [instruction / clock cycle] and incorporates a branch predictor with 98.13% prediction accuracy for biased branches. The number of penalty cycles wasted due to a branch miss-prediction considered in our model is 7. The ratio of the number of simulated branches over the number of total simulated instructions is $b=8\%$ (generated by Table 3.1).

**Figure 3.13.** The IR relative speedup obtained growing the context length

Figure 3.13 illustrates not only the necessity of a greater number of prediction features to improve the processor’s performance, but also the necessity of new performing branch predictors that could consider a larger amount of information in generating predictions (but without scaling exponentially with input features set length).
3.4 SOME CONCLUSIONS AND FURTHER WORK

During this chapter we showed that for designing better branch predictors it is necessary first to identify difficult predictable branches. We proved that a branch in a certain dynamic context is difficult predictable if it is unbiased and its outcomes are shuffled. Based on laborious simulations we showed that the percentages of difficult branches are quite significant (at average between 6% and 24%, depending on the different used contexts and their lengths), giving a new research challenge and a useful niche for further research. State of the art predictors, including neural, obtained low accuracies on these unbiased branches. Even after cascading branches through a series of prediction contexts it remains a significant number of difficult-to-predict branches and the frequency of these difficult-to-predict branches varies between different programs and between different prediction contexts. Computer Architects cannot therefore continue to expect a prediction accuracy improvement with conventional predictors and alternative approaches are necessary. They need some specific efficient predictors that are using some new more relevant prediction information [27].

Predicting the unbiased branches with a more powerful branch predictor leads to a prediction accuracy gain proportional with ratio of unbiased context instances. More than that, this accuracy gain involves a processing rate speed-up, justifying finally the necessity of finding and solving these difficult predictable branches. Also, based on analytical model and simulations results, we determined how much is influenced the branch penalty (number of wasted cycles due to a branch miss-prediction for each dynamic instruction) by increasing the context length and what is the relative speedup in these conditions. The obtained results illustrate the necessity of a greater number of relevant prediction features to improve the processor performance and also the necessity of a new performing branch predictor that can consider a larger amount of information in doing predictions. The IR relative speedup varies between 19% and 57% as the context length growth from 20 to 28 bits.

Also the simulations show that the path is relevant for better polarization rate and prediction accuracy only in the case of short contexts. As Figures 3.8 and 3.9 suggest, our conclusion is that despite some branches are path-
correlated, long history contexts (local and global) approximate well the path information. The results presented here lead us to the conclusion that the path is only useful in case of short contexts. Long contexts seem to disperse and compress very well the path information. In our further work, we’ll try to reduce the path information extracting and using only the most important bits. Thus, the path information could be built using only a part of the branch address instead of all the 32 bits of the complete PC.

Present-day branch predictors are using limited information: local and global correlation and path information. As we already shown in this paper, for some branches this information is not always sufficiently relevant, at least for short feasible contexts. Therefore we think it is important to find other relevant information that is determining branches’ behavior in order to use it for designing better predictors. For example, each branch has some specific information that better involves its behavior. This information might be a particularly relevant “piece” of the dynamic CPU context, for example branch’s condition [26]. In order to efficiently use this (new) more relevant information we think there is necessary the compiler’s help. The compiler could transmit to the predictor, through ISA, this specific relevant information. Anyway, finding a new relevant context to aggressively reduce the number of unbiased shuffled branches remains an open problem. In this sense we believe that a solution might be “branch prediction through value prediction” In our used SimpleScalar (MIPS) architecture the branch’s condition is based on the difference between two source operands. Obviously, this can be positive, negative or zero. Predicting this difference (because branch’s sources aren’t usually available at the prediction time) might involve predicting the branch itself. Therefore branch’s condition prediction represents a value prediction problem and this could be approached based on the branch’s condition histories (a string of values). The prediction problem is what is the next condition’s sign (positive, negative or zero) rather than its concrete value.

Also a next stage of the work will consist in exploiting the information regarding the branch polarization in pre-training neural branch predictors. For example, we can pre-train a perceptron branch predictor [25, 8], only with dynamic contexts having polarization index greater than 0.95, avoiding in this way the contradictory behavior of the unbiased branches that is difficult to be learned. Pre-training the perceptron with the biased branch contexts, we expect to obtain higher prediction accuracy and superior overall performances (IPC) comparing with those of the original perceptron predictors.
References


4. PRE-COMPUTING BRANCHES

4.1. INTRODUCTION

In this chapter we present a technique that is trying to avoid branch prediction through a new concept called “pre-computed branches”. The following presentation is based on our previous work firstly described in [19]. As we already explained in the previous chapters, excellent branch handling techniques are essential for current and future advanced microprocessors. These modern processors are characterized by the fact that many instructions are in different stages in the pipeline. Instruction issue also works best with a large instruction window, leading to even more instructions that are “in flight” in the pipeline. However, approximately every seventh instruction in an instruction stream is a branch instruction, which potentially interrupts the instruction flow through the pipeline [4,5,6,7]. As the pipeline depths and the issue rates increase, the amount of speculative work that must be thrown away in the event of a branch misprediction also increases. Thus, tomorrow’s processors will require even more accurate dynamic branch prediction to deliver their potential performance [3,14,15,16,17]. Dynamic branch prediction forecast the outcome of branch instructions at run-time. This forecast, or prediction, may change for each occurrence of the branch even the dynamic context is the same. Dynamic branch predictors are composed of a single level, such as a classical Branch Target Cache (BTC), or even two levels, such as the Two-Level Adaptive Branch Predictors [8,9,10].

A BTC predicts (Taken/Not Taken and the corresponding Target Address) on the overall past behavior of the branch. In contrast, a Two-Level Adaptive predictor bases its prediction on either global history information or local history information. The first level history records the outcomes of the most recently executed branches (correlation information) and the second level history keeps track of the more likely direction of a branch
when a particular pattern is encountered in the first level history. Global schemes exploit correlation between the outcome of the current branch and neighboring branches that were executed leading to the branch. Local schemes exploit the outcome of the current branch and its past behavior. During the last years there has been interest in hybrid branch predictors where the fundamental idea is to combine different dynamic predictor schemes having different advantages, in a run-time adaptive manner [13].

In [21] call targets are correlated with the instructions that produce them rather than with the call’s global history or the previous branches’ targets. The proposed approach pre-computes virtual function call’s (v-call) targets. V-calls’ targets are hard predictable even through path-history based schemes that exploit the correlation between multiple v-calls to the same object reference. Object oriented programming increases v-calls’ importance. The proposed technique dynamically identifies the sequence of instructions that computes a v-call target. Based on this instructions’ sequence it is possible to pre-calculate the target before the actual v-call is encountered. This pre-calculation can be used to supply a prediction. The approach reduces v-call target miss-predictions with 24% over a path-based two level predictor.

### 4.2 THE PRE-COMPUTED BRANCH ALGORITHM

We suggest through this chapter an alternative approach to the present – day dynamic branch prediction schemes. Instead of predicting based on history information, we propose to pre-calculate the branch outcome. A pre-calculated branch (PCB) determines the outcome of a branch as soon as all of the branch’s operands (used for calculating its condition) are known. This occurs when the last operand of the branch’s instruction is produced at execution. The instruction that produced the last operand may trigger supplementary branch condition estimation and, after this operation, it correspondingly computes the branch’s outcome (Taken/Not Taken). Similarly to branch history prediction schemes, branch information is cached into a “prediction table”, as it will be further presented. Through this method, excepting the first one, every instance of a branch can be computed and therefore correctly “predicted”, before its issue.
In our PCB study we used MIPS-I microprocessor’s Instruction Set Architecture (ISA) since a branch instruction has addressing modes with two register operands and no immediate operands. Considering for example the following MIPS –I code sequence:

```
ADD R9, R5, R7; //R9<- (R5) + (R7)
BNE R9, R8, offset; //if (R9!= R8) PC<-(PC) + offset
```

The first instruction (ADD) modifies the R9’s content and therefore it directly influences the branch condition. This means that the ADD instruction will correspondingly modify R9’s content in the PCB structure. After this operation the PCB structure estimates the condition and, at the moment when the branch instruction itself is encountered, its behavior will be perfectly known. Figure 4.1 depicts our new proposed PCB scheme. It uses two tables: the PT (we still called it “Prediction Table” even if it doesn’t really predict anything) table and an extension of the register file called RU (Register Unit). As the reader can see further, PC doesn’t indexes the RU table. It is used for some associative searches in PT table and also, in some certain cases, it will be updating into the LDPC field. We mention that the letters associated with the arrows in figures 4.1, 4.2 and 4.3 (a, b, c, d and e) represents sequential operations.

Each entry in the PT table consists the following fields:

- a TAG field (the branch’s PC high order bits)
- PC1 and PC2 – which are pointers to the last branch’s operands producers (the PCs of the instructions that produced the branch’s operands values)
- OPC – the branch’s opcode
- nOP1 and nOP2 – the register names of the branch’s operands
- PRED – the branch’s outcome (Taken or Not-Taken) and a LRU field (*Least Recently Used*)

RU table maintains the register file meanings but additionally, each entry, has two new fields named LDPC and respectively RC.
• The “value” field contains the register data value.
• LDPC – represents the most recent instruction label (PC) that wrote in that register.
• The RC field – is a reference counter that is incremented by one by each instruction writing in the attached register and linked by every branch instruction stored in PT table (therefore the instruction’s label is necessary to be found in PC1 or PC2 field). The RC field is decremented by one when the corresponding branch instruction is evicted from the PT table. Therefore, if the RC field attached to a certain register is 0 it involves that in the PT table there isn’t any branch having that register as a source operand.

**Figure 4.1.** The new proposed PCB scheme. A) When a non-branch instruction is encountered; B) when a branch instruction is encountered

In the newly proposed PCB algorithm, the PC of every non-branch instruction, that modifies at least one register, is recorded into the LDPC
field belonging to its destination register. The first issue of a particular branch in the program is predicted with a default value (Not Taken). After branch’s execution, if it was taken, an entry in the PT table is inserted and the LRU field is correspondingly updated. The newly added PT entry fields are filled with the updated information from the branch itself (PC into TAG, OPC, nOP1, nOP2) and data from the RU table (LDPC into PC1 or PC2). Every time after a non-branch instruction - having the corresponding RC field greater than 0 - is executed, the PT table is searched upon its PC, in order to find a hit with the PC1 or PC2 fields (if RC=0, obviously it isn’t any reason for searching the PT table). When a hit occurs, the branch’s opcode stored in that PT line is executed and the corresponding result (taken/not-taken) is stored into the PRED bit. Next time when the program reaches again the same branch, the outcome of the branch is founded in the PT table, as it was previously calculated, and thus its direction it is surely known before branch’s execution. In this way the processor knows for sure which of the program’s path should be further processed. The only miss-predictions that may arise are coming from the initial learning (so named compulsory or cold miss-predictions) or from the fact that PT table has a limited size and therefore capacity miss-predictions may also occur and from exit loops branches. However, recording every first instance of a branch in the PT table (not only those with taken result) will eliminate the exit loop miss-prediction source.

However, the designer must be very careful about the pipeline timing. There are needed at least one and up to several cycles, depending on the pipeline’s length and structure, between the last instruction that alter a register used in the branch’s condition and respectively the corresponding branch instructions. This is because the branch may follow the instruction that produces its operands too closely in the program flow and thus the former instruction cannot finish its execution properly. The branch instruction cannot start its execution right away because it would trigger a Read after Write (RAW) data dependence and it cannot be used the result from the PCB structure because it hasn’t been yet calculated. So, we should postpone the branch processing few cycles and allow the previous instruction to finish and, after this, trigger the supplementary comparison. The minimum number of cycles that should separate the instruction that alter the registers from the corresponding branch instruction we named, analogously with the well-known Branch Delay Slot term, PIDS (Producer Instruction Delay Slot). In order to fill this PIDS we proposed some program scheduling techniques, that will fill this PIDS when necessary, with control independent instructions (statically or dynamically). This was proven as being a feasible
Prediction Techniques in Advanced Computing Architectures

exciting solution, but we’ll not focus on it during this presentation. Anyway, the PCB structure will therefore only help in those cases where the comparison (or equivalent) can be successfully moved several instruction slots ahead of the branch without increasing the length of the schedule. It must therefore be combined with other more conventional techniques. The proposed PCB technique is then used if the comparison is far enough ahead else conventional prediction is used. Some previous valuable work about filling the PIDS with control independent instructions with very optimistic results and details could be found in [18].

We illustrate the working of this scheme using the example shown in Figure 4.2 and Figure 4.3. Figure 4.2.A shows the sequential actions took after the execution of the instruction from “p1” address. The LDPC field

Figure 4.2. Example of the first instance of a particular branch; A), B) actions took when issuing non-branch instructions; C) actions took before b1 branch’s execution.
corresponding to the destination register (R1 here) is filled with the instruction’s PC (p). The number that follows the PC label says that is the first encounter of that instruction; next time it will be the second and so on. Because the RC field of the same register is 0 it means we have completed our actions related to instruction “p”. Similar actions are followed for the instruction having the PC noted “c”. After decoding the “b” branch, the PT table is searched for a hit upon TAG, PC1, PC2 fields (in the “b” set). Due to a miss (this being the first instance of “b” branch) a default prediction is used. If after the “p” instruction’s execution, its outcome is taken then a new line in the PT table is added; also the LRU field is correspondingly updated.

**Figure 4.3.** Example of the second instance of a particular branch; A), B) actions took when issuing non-branch instructions; C) actions took before b2 branch execution.
This time when the “p” instruction is issued again (Figure 4.3.A) the RC field attached to R1 register is greater than zero and the PT table is fully searched for a hit upon PC1 or PC2 field. A hit is obtained and it triggers a supplementary branch’s execution (after obtaining the operands values from RU) and the result (taken/not taken) is correspondingly updated into the PRED field. Similarly actions are presented in Figure 4.3.B for the second issue of the “c” instruction. When the branch about we are talking itself is issued (Figure 4.3.C) the PT is searched into the “b” set. This time a hit occurs and the “prediction” of the branch “b” is extracted from the PRED field. This outcome is 100% accurate, because it has been correctly calculated in the previous described steps. For a more complete in depth understanding of the proposed PCB algorithm, we have provided a pseudo-code description in Appendix.

4.3 COMPLEXITY AND COSTS EVALUATIONS

The global performance of a branch prediction scheme can be investigated from, at least, two points of view: prediction accuracy (local performance) and respectively architectural complexity (costs). The costs themselves can be split in two parts: the table’s sizes and the time spent to access them. In order to evaluate the time corresponding to one branch prediction process (e.g. tables searches, supplementary branch’s condition execution, etc.), we defined the next time quotas:

\[ T_{DM} \] – time needed for one direct mapped table access (RU)
\[ T_{FA} \] – time needed for one fully-associative table access (PT)
\[ T_{SA} \] – time needed for one set-associative table access (PT)
\[ T_{EX} \] – time spent for one supplementary branch execution

Also we have considered:

\[ N_B \] – the number of branch instructions
\[ N_{NB} \] – the number of non-branch instructions
\[ N_{NB} = k_N \cdot N_B \], where \( k_N \) is a statistical constant based on some program profilings \( \approx 7 \)
$N_{\text{NBL}}$ – the number of non-branch instructions “linked” (through the RC field) with a branch instruction

$N_{\text{NBL}} = k_L \cdot N_B$, where $k_L$ is a statistical constant based on some program profiling $≈ 5$

$N_{\text{NBEX}}$ – the number of non-branch instructions followed by a “supplementary branch execution”

$N_{\text{NBEX}} = k_{ex} \cdot N_B$, where $k_{ex}$ is a statistical constant $= 1.3$ (about 30% branches)

Now, the time spent in the branch prediction for one branch is formed by:

A) Search time spent by a non-branch instruction

$N_{\text{NB}} \cdot T_{DM}$ – needed to check the RC field from RU

Every non-branch instruction that writes into a register triggers a search into the PT table for a hit with PC1 or PC2. To reduce these full table searches we have used instead this direct mapped table access to check the RC field (if this instruction is “linked”) and proceed to the full table search only when the RC is not 0.

B) Search time and execution time needed by a “linked” non-branch instruction

$N_{\text{NBL}} \cdot T_{FA(\text{in PT})} + N_{\text{NBEX}} \cdot (T_{DM(\text{in RU})} + T_{EX})$, “linked” instructions search the PT table. When a hit arises, the operands values are taken from the RU table and an execution follows ($T_{EX}$)

C) Search time needed by a branch instruction

$N_B \cdot (T_{DM(\text{in RU})} + T_{SA(\text{in PT})})$, when a branch is encountered a search in the PT table is performed to extract the corresponding prediction computed before

The overall time needed for one branch prediction is: $T = N_{\text{NB}} \cdot T_{DM} + N_{\text{NBL}} \cdot T_{FA} + N_{\text{NBEX}} \cdot (T_{DM} + T_{EX}) + N_B \cdot (T_{DM} + T_{SA})$

$T_{PT} = N_B \cdot (k_N \cdot T_{DM} + k_L \cdot T_{FA} + k_{ex} \cdot (T_{DM} + T_{EX}) + T_{DM} + T_{SA})$ \hspace{1cm} (1.1)

The time costs presented above we think that it should be necessary to be compared with a classic BTB having the same number of rows and a fully associative organization. For the considered BTB, the time needed to predict a branch is reduced to search the BTB at every branch instruction. So, the overall time in this case is $T_{BTB} = N_B \cdot T_{FA}$ \hspace{1cm} (1.2)
Considering common sense values for constants involved as: \(k_N = 7\), \(k_L = 5\), \(k_{ex} = 1.3\) and \(T_{FA} = 4 \times T_{DM}\), \(T_{SA} = 1.2 \times T_{DM}\), the (1.1) and (1.2) equations become:

\[
T_{PT} = N_B \times T_{DM} \times 30.5 + N_B \times T_{EX} \times 1.3 \quad (1.1')
\]
\[
T_{BTB} = N_B \times T_{DM} \times 4 \quad (1.2')
\]

At first sight the time cost difference needed for one branch may seem overwhelming, but we think that other internal processes can hide some of the times from \(T_{PT}\). So, the times wrote with italic font in the \(T_{PT}\) expression \(N_B \times (k_N \times T_{DM} + k_L \times T_{FA} + k_{EX})\) may overlap with the next instruction processing or data reuse process and the \(T_{PT}\) expression becomes now:

\[
T_{PT} = N_B \times (k_{EX} \times (T_{DM} + T_{EX}) + T_{DM} + T_{SA})
\]

Using this new expression we have obtained:

\[
T_{PT} = N_B \times T_{DM} \times 3.5 + N_B \times T_{EX} \times 1.3 \quad (1.1'')
\]
\[
T_{BTB} = N_B \times T_{DM} \times 4 \quad (1.2'')
\]

Now the two expressions, in our opinion, are relatively comparable as processor time spent.

As we have stated above, the actions expressed by the times wrote with italic fonts in the \(T_{PT}\) expression may overlap with some other actions corresponding to the same non-branch instruction. While the instruction is executed (or even reused!) the RU table may be checked for the RC field and, on a hit, the PT table searched upon PC1 or PC2 fields. All these operations can be done in parallel because these actions do not depend on each other, thus they are hidden into the real processor time consumed. The part from the \(T_{PT}\) expression that cannot yet be hidden is that which express the times involved in the supplementary branch execution: accessing the RU table for branch’s operands values and the branch opcode execution. It’s quite obvious that we cannot offset these actions above the end of the current instruction’s execution, when the instruction’s result is produced. In place of trying to overlap these last actions with actions over the current instruction we could overlap them with the next instruction’s execution if they do not totally depend on each other. For this purpose we defined an average overlap probability (OP), which points out the overlapping degree with the next instruction’s execution. After this (1.1”’) and (1.2”’) expressions becomes:

\[
T_{PT} = N_B \times T_{DM} \times 3.5 + (1-OP) \times N_B \times T_{EX} \times 1.3 \quad (1.1'''')
\]
\[
T_{BTB} = N_B \times T_{DM} \times 4 \quad (1.2'''')
\]
The improvement in prediction accuracy brought by this scheme must be paid some way in costs. As we initially felt, if timing costs can be partially reduced by hiding them, the physical costs cannot. Considering a register file RU with 32 registers and a PT table with M (M = 2^j) entries, the total size, in bits, is:

\[ D_{PT} = M \cdot (32-j)/T + 2 \cdot 32/P + 5/OPC + 2 \cdot 5/nOP1 \]
\[ + nOP2 + 1/PR + 2/LRU + 32/(32/LDC + 2/RC) = M \cdot (114-j) + 1088 \]

For a corresponding BTB having the features discussed above:

\[ D_{BTB} = M \cdot (32-j)/T + 2/p + j/LRU = M \cdot 34 \]

Considering tables (PT and BTB) with 1024 entries, we have obtained:

\[ D_{PT} = 1024 \cdot (114-10) + 1088 = 105 \text{ KBits} \]
\[ D_{BTB} = 1024 \cdot 34 = 34 \text{ KBits} \]

### 4.4 PERFORMANCE EVALUATIONS THROUGH SIMULATION

The result for this second part of the chapter were gathered using a complex simulator built in our Advanced Computer Architecture & Processing Systems (ACAPS) Research Center belonging to “Lucian Blaga” University of Sibiu (http://acaps.ulbsibiu.ro/research.php), on the kernel of the SimpleScalar simulation tool-set [1], an execution-driven simulator based upon the MIPS-I processor’s ISA. The benchmarks used falls into two categories: the Stanford HSA (Hatfield Superscalar Architecture) benchmarks as described in [2, 4, 11, 16], recompiled to run on SimpleScalar architecture and the SPEC ‘95 benchmark programs [10] having as inputs, the files listed in Table 4.1. The benchmarks were run for a maximum 500 millions instructions (SPEC) or to completion if they were shorter (HSA).
We performed several experiments to evaluate the newly proposed scheme. For doing this we have used table sizes of 128, 256, 512, 1024, 2048 entries having an associativity degree of 4. The results obtained on our PCB scheme were then compared with a BTB prediction scheme having an equivalent number of entries and two kinds of associativity degree: full associative and respectively 4-way set associative. For PCB we performed two experiments in order to evaluate the two ways of adding new entries in the PT table. First way is to add an entry in the PT table only if the branch was taken. The politics adopted is to don’t fill the table with branches that have a not-taken behavior (ANT=0). This solution reduces capacity misses, but we will have supplementary misses when the branch will be taken (end loop misses). The other way (ANT=1) is to add taken and not taken branches preventing the end loop misses. Of course, this will have a big impact on capacity misses when using small size tables.
Inserting entries in the PT table only when it is really necessary performs better on smaller tables because it reduces the capacity misses. But on larger table sizes where the capacity misses are not so frequent, adding every entry in PT reduced the end loop misses. The next experiment was to compare the newly proposed scheme (PCB) with similar classical dynamic prediction schemes. Figure 4.6 shows the amount of accuracy brought by the PCB scheme compared with a corresponding set associative BTB scheme using SPEC ‘95 benchmarks, is about 11%. As depicted in Figure 4.6, even with a full associative BTB the PCB scheme performs better. The difference of accuracy between the PCB scheme and BTB schemes are even greater when using the HSA Stanford benchmarks, about 18%, because these programs are more difficult to predict than SPECs, using classical schemes.
Based on our work, an interesting further analysis of the pre-computed branch concept is presented in [20].

![Graph showing prediction accuracies](image)

**Figure 4.6.** Average “prediction” accuracies obtained on SPEC ‘95 benchmarks

### 4.5 SOME CONCLUSIONS AND FURTHER WORK

The new proposed PCB dynamic algorithm outperforms clearly all the branch prediction schemes because it pre-computes the branch outcome before the branch will be really processed. From the pure prediction accuracy point of view this algorithm seems to be almost perfect. Similarly to branch history prediction schemes, branch information is cached into a so called “prediction table”. Through this method, excepted the first one, every instance of a branch can be computed and therefore correctly (non-speculatively) “predicted”, before its issue. The improvement in “prediction accuracy” brought by this scheme must be paid some way in timing and costs. Unfortunately, if the PCB’s timing can be partially reduced by hiding it through some overlapping processes, the structural costs cannot be reduced so easy. So, a PCB prediction scheme is about 105 KBits complex comparing with a full associative BTB scheme having only 34 KBits complexity at the same number of PT entries (1024 in this case).

As a further work we intend to measure the average PIDS (in cycles) based on SPEC ‘2000 benchmarks, and, as a consequence, trying to develop a software scheduler in order to fill – where it will be necessary - with some branch condition independent instructions these PIDS. This research might
be a solution of the unbiased branch problem presented in this book (see the previous chapter). Also we’ll try to analyze in more depth other overlapping possibilities in order to reduce the PCB timing and also investigate the integration of the PCB scheme in some very powerful processor models, having some advanced architectural skills like value prediction and dynamic instruction reuse concepts.

4.6 APPENDIX. THE PCB ALGORITHM

We are using the following notations and abbreviations in this annex:
PC – current instruction address
PC.nOP1 – register name for the first operand corresponding to the current instruction
PC.nOP2 – register name for the second operand corresponding to the current instruction
PC.OPCODE – instruction opcode corresponding to the current instruction
dimSet – the number of entries in a set
dimPT – the total number of PT entries
PC_{m-1..0} – Least Significant m Bits of the PC
PT (Prediction Table) – set-associative organization after TAG and fully-associative after PC1 and PC2

To implement the PCB algorithm we have used the following helper functions:
- FOUND(j) - tests if a previous search in the PT table finished with success or not
- FIND_PT_ENTRY - it searches the PT table, in the PC's corresponding set, for a hit on the PC and PC1 and PC2 fields. When a hit occurs it returns the index of that entry in the PT table otherwise -1.
- ADD_PT_ENTRY - records a new entry in the PT table. The entry to be filled is selected using the FREE_PT_ENTRY function. If we had R0 as operand we will perform no decrementing because for the R0 register is useless to consider a RC field (there is no instruction to have R0 register as destination). Now we can update the entry with the new data (TAG, PC1, PC2, nOP1, nOP2, OPC). Finally we have to link this entry with the corresponding operands by incrementing the RC field of those registers.
- FREE_PT_ENTRY - Its aim is to find a suitable entry in the PT table to be, first, evicted and then in that position to add a new entry.
- **SCH_and_UPD_PT_TABLE** - searches the entire PT table for a hit in the PC1 or PC2 fields. When a hit occurs the data stored into that entry (OPC, nOP1, nOP2) is used to execute a supplementary conditional operation. The result is then stored back in the PRED field of the same entry.

START:
0. FETCH_INSTR
1. DECODE_INSTR
2. IF isBRANCH(PC) THEN //this is a branch
3. IF FOUND(FIND_PT_ENTRY(PC)) THEN
4. \[
    \text{PREDICTION} = \text{PT}[\text{FIND_PT_ENTRY(PC)}].PRED  \quad //100\%
\]
accuracy
5. ELSE
6. \[
    \text{PREDICTION} = \text{NotTaken} \quad //\text{default prediction}
\]
7. IF EXEC_BRANCH=TAKEN THEN
8. ADD_PT_ENTRY(PC)
9. ELSE //not a branch instruction
10. RD.REGVAL=EXEC_INSTR //RD-destination register for the current instruction
11. RD.LDPC=PC
12. if RD.RC >0 THEN
13. SCH_and_UPD_PT_TABLE(PC) //search the whole PT table for PC1=PC or PC2=PC
//on hit, update the prediction field of those entries
14. PC=PC+offset
15. \[\text{GOTO START}\]

Next we show the functions implementations:

FOUND(j)
\[
\text{IF } j<0 \text{ THEN RETURN FALSE}
\]
ELSE
\[
\text{RETURN TRUE}
\]
END //FOUND

//searches for an entry in the PC set with (TAG=PC) and (PC1=PC.nOP1) and (PC2=PC.nOP2)
FIND_PT_ENTRY(PC)
\[
\text{[stSet=dimSet*PC}_{m-1.0}]' \quad //\text{first entry in the PC set}
\]
\[
\text{[endSet=dimSet*([PC}_{m-1.0}+1)]'} \quad //\text{first entry in the PC+1 set}
\]
WHILE stSet < endSet DO //all this searches overlap
IF (PT[stSet].TAG=PC) AND (PT[stSet].PC1=RU[PC.nOP1].LDPC) THEN
    IF NOT PC.OP2 THEN     //there is no second operand
        RETURN stSet
    ELSE                     //there is a second operand
        IF (PT[stSet].PC2=RU[PC.nOP2].LDPC) THEN
            RETURN stSet
        stSet++
    //end while
END  //FIND_PT_ENTRY(PC)

//adds an entry in the PT table
ADD_PT_ENTRY(PC)
IF PT[FREE_PT_ENTRY(PC)].nOP1 > 0 AND //if this PT entry was taken and
    nOP1 is not R0
    RU[PT[FREE_PT_ENTRY(PC)].nOP1].RC > 0 THEN //don’t go below 0
        RU[PT[FREE_PT_ENTRY(PC)].nOP1].RC--
        IF PT[FREE_PT_ENTRY(PC)].nOP2 > 0 AND
            RU[PT[FREE_PT_ENTRY(PC)].nOP2].RC > 0 THEN
                RU[PT[FREE_PT_ENTRY(PC)].nOP2].RC--
                PT[FREE_PT_ENTRY(PC)].TAG=PC
                PT[FREE_PT_ENTRY(PC)].PC1=RU[PC.nOP1].LDPC
                PT[FREE_PT_ENTRY(PC)].nOP1=PC.nOP1
            IF PC.OP2 THEN
                PT[FREE_PT_ENTRY(PC)].PC2=RU[PC.nOP2].LDPC
                PT[FREE_PT_ENTRY(PC)].nOP2=PC.nOP2
            ELSE
                PT[FREE_PT_ENTRY(PC)].PC2=-1
                PT[FREE_PT_ENTRY(PC)].nOP2=-1
                PT[FREE_PT_ENTRY(PC)].OPC=PC.OPCODE
                RU[PT[FREE_PT_ENTRY(PC)].nOP1].RC++
                //increment the new refcount
            IF PC.OP2 THEN
                RU[PT[FREE_PT_ENTRY(PC)].nOP2].RC++
                //increment the new refcount
        END  //ADD_PT_ENTRY
//full PT table search for PC in PC1 or PC2 fields
SCH_and_UPD_PT_TABLE(PC)

\[ j=0 \]

//this long time searches may overlap with EXEC_INSTR or data reuse process
WHILE \( j \lt \text{dimPT} \) DO
  IF (PT[\( j \)].PC1=PC) OR (PT[\( j \)].PC2=PC) THEN
    PT[\( j \)].PRED=EXEC(PT[\( j \)].OPC,
    RU[PT[\( j \)].nOP1].REGVAL,
    RU[PT[\( j \)].nOP2].REGVAL)
    \( j++ \)
  END //SCH_and_UPD_PT_TABLE

References

[7] Egan C., Steven G., Vin\( \text{\char'\=a} \)an L., Quantifying the Benefits of Multiple Prediction Stages inCached Two Level Adaptive Branch Predictors, Proceedings of International Conference SBAC-PAD, Brasil, Brasilia, September, 2001 (accepted)


[12] SPEC - The SPEC benchmark programs (www.spec.org)


5. DYNAMIC INSTRUCTION REUSE

5.1. FETCH BOTTLENECK. DYNAMIC INSTRUCTION REUSE THROUGH TRACE-CACHE PROCESSORS

A common limitation of the multiple instruction issue processors is called fetch bottleneck. This means that the CPU can’t fetch more than a basic-block per cycle due to the final branch instruction. If this branch is a taken one, the subsequent instructions will not be processed. In the general-purpose programs the average basic block is about 6 instructions. It can be observed that this limitation is due to the programs’ intrinsic characteristics. As a consequence, a processor can’t fetch more than 6 instructions/cycle from the I-Cache and, therefore, this means also an issue rate less than 6 instructions/cycle. Solving this problem means answering to the following questions:

- How to fetch simultaneously multiple basic-blocks?
- How to learn their input addresses?
- How could we simultaneously access these multiple basic-blocks?
- However, how could the fetch process to be limited to only one cycle?

An interesting paradigm situated in the extension of the superscalarity concept, that can be a novel solution regarding the fetch bottleneck limit of superscalar architectures (or so called Flynn’s bottleneck), consists in the trace-processor, a processor with a trace-cache (TC) structure [Rot96]. In the same manner as instruction caches (IC), the TC is accessed in parallel with the IC using the address of a new instruction block (Program Counter - PC) that must be processed. In the case of miss in the TC, the instruction is fetched from the IC or – in case of miss in the IC – from the upper memory hierarchy. Unlike the IC, the TC stores continuously instructions in their sequential execution order, in successive memory locations. A TC entry stores a segment of dynamically and sequentially executed instructions.
called trace-segment. Obviously, a trace can contain more basic-blocks (sequential program units). Therefore, a TC entry can contain N instructions or M basic-blocks (N>M) written at the time they were executed [Vin02].

According to Figure 5.1 the TC is accessed with the address of basic-block A, in parallel with the multiple-branch predictor. Unlike a simple branch predictor, the multiple-branch predictor anticipates not only the address of the next basic-block which must be executed, but also the addresses of all the next (M-1) basic-blocks that follow basic-block A. The (M-1) bits generated by the multiple-branch predictor (taken / not taken) select from the corresponding TC entry only the basic-blocks that are predicted as being further executed (in this case only blocks A and B because the predictor selected blocks ABD, while the TC entry contained at the moment only blocks ABC).

A TC entry usually contains:

- N decoded instructions, each of them having specified the block it belongs to. There might be also information related to instruction dependences.

**Figure 5.1.** Trace-cache with a multiple-branch predictor
- M targets from the all-possible $2^{M-1}$ destination addresses of the M blocks stored in the TC entry.
- A field that codifies the number and “directions” (0/1) of the branches stored in the TC entry.

Before the instructions are stored in the TC, they can be pre-decoded in order to write into the TC some information regarding the data dependences corresponding to the instructions from the current TC entry. Such information will facilitate processes like data bypassing between execution units, dynamic renaming of registers to avoid WAR (Write After Read) or WAW (Write After Write) dependences, etc., useful during the out of order execution. A TC entry can have different associativity degrees because it can contain more block paths, all of them having certainly the same starting address (A in this case), as it is shown in Figure 5.2.

![Figure 5.2. Selection from an associative trace-cache entry](image)

Therefore, in this case, different segments that begin at the same address (A) are stored in the same associative entry from the TC. Like in the previous non-associative TC structure, the validity of the selected entry is verified by comparing (searching) tags. The essential difference consists in the necessity to select – according to the pattern generated by the multiple-branch predictor – the longest trace (path) from the respective associative entry. This complex selection may take longer than in the non-associative case, with a negative repercussion on the timing of the instruction fetch process. The great advantage, as it can be observed in Figure 5.2, is that
longer block sequences can be provided to the processor than in the case of simple direct mapped TC. As an example, if the real path of executed blocks is ABD, the TC structure will provide it entirely, while a direct mapped TC structure that contains only the ABC path, obviously will provide in this case only blocks A and B.

As a group of instructions is processed, it is loaded to a so-called fill unit (FU). The role of the FU is to assemble dynamic instructions, as they are executed, into a trace-segment. The segments obtained in this way are stored in the TC. As it was already emphasized, before the segment is written into the TC, the FU may analyze the instructions within the segment in order to mark explicitly the dependences between them. This will make easier the issue of these instructions, because they will be fetched from the TC and introduced directly into the reservation stations corresponding to the functional units. Thus, the FU collects instructions issued for execution, assembles them into a group of N instructions (or M basic blocks), and writes the group into a certain TC entry. There are certain cases when the FU can create multiple copies of some blocks in the TC. This informational redundancy might involve performance degradation, but on the other hand, without redundancy the fetch rate degrades resulting thus a lower global performance. The optimal trade-off must be found.

![Figure 5.3. Trace-segments assembled during the execution of a loop](image)

Thus, it can be stated that a TC exploits the efficient reuse of dynamic instruction sequences, frequently re-processed on the basis of two statistical principle reasons: the temporal localization of the traces and, respectively, the predictable behavior of the branches, anticipated in the virtue of their previous behavior. The TC stores traces for an efficient program execution –
because a trace segment contains only instructions that will be executed—and not only for an efficient instruction fetch process. As we already explained defining fetch bottleneck, in the case of a simple IC, if in a block there is a taken branch instruction, the next instructions are fetched uselessly because they would not be executed.

Because the TC is highly dependent on the branch predictor, the performances of these predictors must be improved. A future solution probably will consist in a multiple-branch predictor, whose important role is to predict simultaneously the next (M-1) branches associated to maximum M blocks that can be stored in a TC entry. As an example, in order to predict simultaneously three branches using a two-level prediction scheme, each entry from the PHT (Pattern History Table) prediction structure must be expanded from a single two-bit saturating counter, to seven such prediction automata as in Figure 5.4. Thus, the prediction generated by the first predictor (taken / not taken) will select through a selector circuit the results of the two predictors associated to the second possible branch from the current TC entry. Both predictions corresponding to the first two branches will select one of the four possible predictors for the third branch that can be resident in the TC-entry, predicting thus multiple branches simultaneously. As it can be observed there is a correlated prediction process. If the multiple-branch predictor provides simultaneously multiple PCs, the TC elegantly solves the multiple instruction fetch, without the multi-porting implied in the case of a conventional cache.

Figure 5.4. Correlated prediction of three successive branches
The three branch predictors are indexed with three PC’s belonging to the trace-cache, corresponding to the three predicted branches. Such multiple-branch predictors in conjunction with TC structures lead to a new program processing paradigm called multi-flow that might processes in parallel multiple basic-blocks from a program. It must be mentioned that the multi-flow architectures that process both sides of a branch require, however, the target address of its taken side. However, more than two paths are hard to be processed simultaneously at the level of the actual technologies (there are needed distinct CPU contexts, etc. [Sil99]). Even if only two paths are processed in parallel, correct speculative mechanisms must be assured that do not modify wrongly and definitively the state of the system (e.g. do not allow the alteration of a memory location by the “wrong” path of the branch). These architectures [Kla98] eliminate the penalties introduced by mispredictions within mono-flow architectures, which decrease performance approximately with 50% reported to the perfect prediction case. In [Pat97] a simulation based research is presented that focalizes on novel TC concepts and multiple-branch predictors, integrated into a highly aggressive superscalar architecture developed at University of Michigan, USA by Professor’s Y. Patt’s group. The investigation emphasizes the following aspects:

- The growth of the TC’s associativity degree from 0 (direct mapped) to 4 (associativity in blocks with 4 entries/block) can increase the average processing rate up to 15%.
- Equal TC and respectively instruction cache capacities (64 Kbytes, 128 Kbytes) lead to quasi-optimal performances.
- The global performance compared to that of an equivalent architecture, but without TC, grows approximately with 24%, and the fetch rate grows with 92% at average.
- Trace cache exploits the basic-blocks reuse based on trace’s temporal locality.
- Its efficiency is due to the predictable branches (multiple predictor’s efficiency)
- Trace cache might improve also the instructions issue due to some additional information related to data dependencies.
- Reducing redundant information is a further challenge.

The Intel Pentium IV processor represents the first commercial architecture that replaces the usual cache level L1 with an Execution Trace Cache. Other architectural characteristics of the processor constitutes: the integration of
42 million transistors, a pipeline that can work on over 20 stages, 4 issued instructions per cycle simultaneously, a 17 bit GAs outcome branch predictor (Prescott core), a 400 MHz system bus with a memory transfer rate of 3.2 Gb/s [Vin00b]. An extension of the dynamic instruction reuse, focalized on solving the execution limits of the superscalar processors (issue bottleneck), is presented in the next paragraph (5.2) respectively next chapter of this book, together with the connections between the reuse concept and respectively the instruction centric value prediction concept, the last one so useful in the implementation of speculative executions.

5.2 ISSUE BOTTLENECK. DYNAMIC INSTRUCTION REUSE. FUNCTION REUSE

Issue Bottleneck (or Data-Flow Bottleneck, Critical Path Bottleneck) means that conventional processing models are limited in their processing speed by the dynamic program’s critical paths. The critical path represents the longest dependence (Read after Write) chain belonging to the program’s dependence graph, measured in CPU’s clock cycles. In other words, if the critical path’s latency is N cycles the program will run, even on an Oracle processor model with infinite hardware resources [Vin00], in a time greater or at least equal than N CPU cycles. It is the price paid due to the intrinsic sequentially of the programs in respect to the well-known Amdahl’s low.

The genuine idea of dynamic instruction reuse was first introduced by the researchers A. Sodani and G. Sohi from University of Wisconsin – Madison, USA, at International Symposia on Computer Architecture (ISCA’97) conference, held in Denver, USA [Sod97]. The dynamic instruction reuse is a relatively new non-speculative micro-architectural technique that exploits the repetition of dynamic instructions, reducing thus the quantity of code necessary for execution with remarkable benefit on the processing speed. Here is a good example when a fundamental and fertile computer-engineering concept, the reuse, actually migrates from the software to the hardware (vertical migration). First, the authors show that the reuse of instructions or dependent instruction sequences is relatively frequent and the compact program writing and respectively the intrinsic generic characteristics of the data structures generate it. A dynamic instruction is reusable if it occurs with the same entries and produces the same outputs as an earlier instance of the same instruction. The main idea is that if an
Dynamic Instruction Reuse

In [Sod98] the authors show that the reuse degree of the dynamic instructions is significant. Less than 20% of static instructions generate over 90% of repeated dynamic instructions. Their analysis shows that at average (harmonic mean) 26% of dynamic instructions are reusable on the SPEC’95 benchmarks (Standard Performance Evaluation Corporation). There are two qualitative reasons explaining this fact: the programs are written generically while they operate a variety of input data represented in some data structures (lists, trees, stacks, hashing tables, etc.), and, on the other hand, these programs are written in a concise manner – meaning a compact static representation of a sequence of dynamic operations – to obtain the desired results (e.g. recursive programs, loops, inheritance, etc., are representative).

For a better understanding of the instruction repetition phenomenon, the programs’ dynamic execution is analyzed on three levels: global, function-level and respectively local (inside the function). In the global analysis, the data patterns used in programs are retained as integer entities and the instruction repetition sources are determined (external inputs, global data initializations or internal program values). Since the instruction repetition is due mostly to the last two repetition sources, the repetition phenomenon is moreover a property of the manner in which the computations are expressed through programs rather than an intrinsic property of the input data. The conclusions of the function-level analysis are that the functions are mostly
invoked repeatedly with the same values of the input parameters and relatively few function calls do not have repeated arguments. The percentage of reusable dynamic instructions can be significant, even in the case when a function is repeatedly called with different input parameters. At the level of local analysis, the instructions belonging to functions/procedures are classified based on the source of the used values (e.g., function arguments, global data, values returned by other functions, etc.) and based on the executed task (e.g., register save – restore, prolog – epilog, global address computation, etc.). The instruction repetitions are due mostly to global values or function’s arguments and also to the function’s prolog and epilog [Vin02].

A suggestive example taken from [Sod97] that illustrates the phenomenon of dynamic instruction repetition is further analyzed. The function `search`

```c
int search(x, list, size) {
    int i;
    for(i=0; i<size; i++) {
        if(x == list[i]) return i;
    }
    return -1
}
```

```c
main_search(a,b,c) {
    ...
    search(a, list, size);
    ...
    search(b, list, size);
}
```

---

**Figure 5.5.** Dynamic instruction reuse during function’s execution [Sod97]

A suggestive example taken from [Sod97] that illustrates the phenomenon of dynamic instruction repetition is further analyzed. The function `search`
Dynamic Instruction Reuse

(Figure 5.5a) searches a value $x$ in the array list of length size. The main function main_search (Figure 5.5c) calls function search several times to search for different elements in the same array. When function search is called, the value is searched in the array element by element until the tail of the array, the search being finished when the element is found. The expansion of the loop from function search corresponding to an iteration is presented in Figure 5.5b. The dynamic instances of instructions generated by the first call of search are described in Figure 5.5d. In each iteration of the loop, instruction 2 is dependent on parameter size, instructions 3 and 4 are dependent on parameter list, instruction 5 is dependent on parameter list and on the value $x$ searched in the array, and instruction 6 is dependent on counter $i$. If search is called again in the same array list (of the same length size) but with another searching parameter, then all dynamic instances of instructions $1\div 4$ and $6$ will produce the same results as those produced at the previous function call. Only dynamic instances of instruction 5 produce results that might differ from the previous calls of function search. The repetition of the results generated by dynamic instances of instructions $1\div 4$ and 6 is directly attributed to the fact that function search was written as an array-searching generic function, but in this particular case, only one of the parameters modifies between different calls. As a consequence of using loops to express the desired computations in a concise manner, the different dynamic instances of instruction 6 ($i=0, i=1, i=2,...$) will produce the same values as those generated at the first call of the function, even if search would be called with all parameters different between calls. However, if parameter size will be different at a second call of function search, then only minimum (size1, size2) dynamic instances of instruction 6 will produce the same results. Consequently, this suggestive example shows that the repeatability of dynamic instructions is considerable and, therefore, the reuse is possible. The dynamic instances marked with “*” in Figure 5.5 will produce the same results for both calls of function search.

Beyond this simple example, increased efforts are imposed for a better understanding of instructions reuse, more exactly, of the answers gave to some questions like: which characteristics of the programs’ structures written in high level languages lead to high reusability degrees? Currently it is also relatively unknown, for example, what relationships are between the object-based programming and dynamic reuse or how do the local and global program optimization techniques contribute to instruction reusability? Such high interest problems remain currently opened.
Dynamic Instruction Reuse – where the reusability comes from?

- Generic programs – operates on data structures;
- Concise programs – have a compact static representation of the
dynamic program (loops, recursive structures, inheritance, etc.);
- The repetition phenomenon is moreover a property of the manner in
which the computations are expressed through programs rather than a
property of the input data;
- The functions are often invoked with the same parameters.

Dynamic Instruction Reuse – some opened important problems (at least
for the author…)

- Which characteristics of the algorithms and HLL program structures
lead to high reusability degrees? E.g. Polling algorithms vs. Event-
Driven algorithms;
- Which are the relations between the characteristics of procedural
programming (call, recursive call, etc.) / object-based programming
(inheritance, polymorphism, etc.) and the dynamic instruction reuse? I
believe that object-oriented programming increase instruction/function
reusability.
- Which characteristics of data structures and data representation (e.g.
data redundancy) increase reusability?
- How do the local and global code optimization techniques (loop
unrolling, software pipelining, trace scheduling, code percolation, etc.)
contribute to instruction reusability? (I think they are decreasing the
reusability…).
- How could we exploit a hybrid dynamic & static reuse? In other
words, how could we unify dynamic instruction reuse with basic-
block, trace and HLL function reuse (dynamic programming)?

Based on the above-presented premises, the American researchers Dr.
Avinash Sodani and Prof. Guri Sohi developed three dynamic instruction
reuse schemes, the first two at instruction level and the last, at the level of
RAW (Read After Write) dependent instructions’ chain. The instructions
that have been already executed are stored into a small cache structure
called the Reuse Buffer (RB). The RB can be addressed with the Program
Counter (PC) during the Instruction Fetch (IF) stage and it also has a
mechanism for selective invalidation of the entries, based on some certain
events. Certainly, the RB must allow a mechanism to test the reusability of the selected instructions. The reusability test verifies if the information accessed from the RB represents a reusable result. The test’s implementation details depend on the reuse scheme. Two RB management aspects must be also treated: establishing which instruction will be written into the buffer and maintaining the consistency of the reuse buffer. The decision regarding how the instructions are inserted into the RB can vary between an unrestricted one (“no policy”) that inserts every instruction into the buffer if it is not already there, and a more selective one that filters the instructions based on the statistical reuse probability. The consistency problem takes into consideration the guarantee that the result of the instruction reused from the RB is correct. Maintaining the consistency of the information from the RB depends on each reuse scheme, as it would be showed further. For compatibility with the superscalar model that issues multiple independent instructions simultaneously, the RB is usually multi-ported in order to allow many currently issued instructions to be reused. Obviously, the multi-porting degree of the RB is less than the maximum execution window.

![Reuse Buffer's principle](image)

**Figure 5.6.** The Reuse Buffer’s principle

As it can be observed in Figure 5.6, the RB is indexed with the instruction’s PC. It presents a selective entry invalidation mechanism based on some certain events. In the case of instruction-level reuse, an RB entry could have one of the following formats [Sod97]:

1. **Entries**
2. **PC**
3. **Reuse test**
4. **Reused instruction**
### Figure 5.7. RB Entry’s formats

<table>
<thead>
<tr>
<th>TAG</th>
<th>OP1</th>
<th>OP2</th>
<th>ADDRESS</th>
<th>RESULT</th>
<th>MEM_VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG</th>
<th>OP1</th>
<th>OP2</th>
<th>ADDRESS</th>
<th>RESULT</th>
<th>RES_VALID</th>
<th>MEM_VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG</th>
<th>OP1</th>
<th>OP2</th>
<th>ADDRESS</th>
<th>RESULT</th>
<th>RES_VALID</th>
<th>MEM_VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SRC-INDEX</td>
<td>SRC-INDEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NAME</td>
<td>NAME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TAG** – might be represented by the instruction’s PC;

**OP1, OP2** – represent the value or name of source registers used by the instruction;

**RESULT** – represents the actual result of the instruction, which will be reused in the case of a “hit” in the RB;

**RES_VALID** – indicates, in the case of arithmetic/logic instructions, if the result “RESULT” is valid or not. In the case of Load and Store instructions, if it is set, shows that the address of the instruction is valid in the RB and thus it can be reused. It is set when the instruction is introduced into the RB. It is reset automatically by any instruction that writes one of the source registers (OP1, OP2);

**ADDRESS** – is the (reusable) memory address in the case of a Load/Store instruction;

**MEM_VALID** – indicates if the value from the “RESULT” field is reusable in the case of a Load instruction. The bit is set when the Load instruction is written into the RB. The bit is reset by any Store instruction that has the same access address.

Thus, the reuse of arithmetic/logic instructions is assured if RES_VALID=1. If RES_VALID=1, also guarantees the correct address for any Load/Store instruction and exempts the processor from its computation (indexed addressing \( \equiv [\text{Register} + \text{Offset}] \)). On the other hand, the result of a Load instruction can be reused only if MEM_VALID=1 AND RES_VALID=1. The performance gain associated to the dynamic instruction reuse technique is introduced by short-circuiting some pipeline stages and also by reducing the
Dynamic Instruction Reuse

structural hazards and the pressure over several hardware resources. Therefore, by reusing instructions the stalls in the reservation stations (Instruction Window) are avoided, reducing thus the execution latencies, the results of the reused instructions being written earlier into the Reorder Buffer. Also it involves an earlier release of the functional units, which do not have to process the reused instructions and, respectively, a faster unblocking of the RAW dependent instructions. The evacuation from the RB must take into account that the invalidated instructions have priority in this process. A reuse scheme that contains even the values of the source operands is presented in the following figure.

Figure 5.8. An Sv Reuse Scheme

In the case of a scheme that reuses dependent instruction chains (so called Sn+d scheme) the structure of an RB entry is the same with the previous one, but there are two new subfields associated to the source operands, \( \text{SrcIndex1} \) and \( \text{SrcIndex2} \), indicating the addresses of the instructions from the RB that produced the source operands corresponding to the current instruction stored in the RB. In this case, the instructions are classified into three categories: source – that produce results for other dependent instructions from the chain and respectively independent – whose operands
are not produced within the considered instruction chain. Another essential difference in contrast to the previous scheme consists in the fact that for the scheme of reusing (dependent) instructions chains, if a source operand is modified, only the independent instructions that contain this operand are invalidated in RB. In order to understand better the benefits of this selective invalidation rule, consider the following RAW dependent instruction sequence:

Figure 5.9. A conservator chain instruction-reusing scheme
In this case, the execution of instruction R will not invalidate instructions J and K because they are dependent on instruction I. Instruction R will not invalidate even independent instruction I, because register R1 is not source in this instruction. Thus, if a new instance of the chain <I, J, K> occurs, its previous results (R1=0, R2=R3=5) are simply reused, the effective processing of the corresponding instructions being not necessary. Unfortunately in the case of the previous scheme (instruction-level reuse) instruction R would uselessly invalidate instructions J and K, not allowing their reuse, despite they are obviously reusable. Anyway, a difficult question is: how could we reuse instructions belonging to critical path? As an alternative to dynamic instruction reuse in [Yeh04] the authors start their
research observing that many sequences of dependent instructions can be interpreted as data flow graph functions and these functions can be implemented as hardware operators into very fast but exponentially costly two-level combinational schemes. They present a novel method that exploits this opportunity and speeds up programs due to circuit-level parallelism and redundancy, avoiding the exponential costs. The proposed implementation consists of a superscalar processor with a large specific functional units associated with specific back-end transformations. The reported average performance compared with an equivalent superscalar architecture without this facility, vary from 3.5% to 19% on optimized code sections.

Certainly, the integration of branch instructions into these dynamic reuse schemes must also be taken into consideration. Integrating these instructions lead to restrictions regarding the introduction of instructions into the RB. In order to analyze this problem, the following instruction sequence is considered:

\[
\begin{align*}
I_1: & \quad R_1 \leftarrow 1 \\
I_2: & \quad \text{BRANCH <Cond>, I4; If <Cond>=True, Jump to I4} \\
I_3: & \quad R_1 \leftarrow 0 \\
I_4: & \quad R_2 \leftarrow R_1+4 
\end{align*}
\]

In the case of speculative execution, the RB instruction allocation management as it was described above can be improper. The previously described instruction-level reuse scheme will be considered first. We suppose that I2 is initially predicted as Not Taken, resulting thus the allocation of instructions I3 and I4 into the RB. If later the prediction of I2 turns out to be wrong, the context recovery and the execution of Taken path are necessary. Consequently, the reuse of instruction I4 is possible, generating a wrong result, since, source operand \( R_1=0 \), as it was modified by I3. This anomaly can be resolved by allowing the introduction of a speculative instruction into the RB only if its source instructions are not speculative. Thus, in the considered example, since I3 is speculative, results that I4 will not be introduced into the RB and the error will not occur.

In the case of the reuse scheme at dependent instructions chain level, the approach is more complicated. In this case, after the introduction of instructions I3 and I4 into the RB, a pointer from I4 to I3 will indicate that I3 is a source. Thus, when - due to the misprediction - the alternative path will be processed, I4 will not be reused because its connection with I3 will disappear, its source instruction being in this case I1. However, the same
problem that appeared at the previous scheme will appear in this case too if I4 would be inserted into the RB as an independent instruction (e.g. if for some reasons, I3 wouldn’t be stored in the RB anymore). Consequently, in the case of this scheme, it is necessary to impose the insertion of a speculative instruction into the RB only if its source instructions are not speculative or if all its source instructions are in the RB.

Under the influence of the researches initiated by Sodani and Sohi, also in [Max98] the authors try to develop some schemes for dynamic reuse of RAW dependent instructions chains. During the processing of instructions, so named instruction sets are built dynamically. An instruction “i” is appended to a set S if “i” depends RAW on at least one of the instructions belonging to set S. Otherwise, instruction “i” will be the first that belongs to a new set. Practically, building these sets implies the generation of a data dependence graph attached to the program, as in the following sequence taken from [Max98].

![Data Dependence Graph](image)

**Figure 5.11. Building the code reuse sets**

After the instructions are processed, the resulting sets are written into a special buffer named by the authors TDIS (Table of Dependent Instruction Sequences), in order to be reused. A TDIS entry contains three important parts:

- **IN** – stores the values of the input operands not produced through that sequence, but taken from outside;
- **INSTRUCTION** – contains the addresses of the instructions inserted into sets;
- OUT – contains the names of destination registers afferent to a set together with their values.

For exemplification, the previous program sequence needs a TDIS buffer with two entries, as follows (Figure 5.12, taken from [Max98]).

<table>
<thead>
<tr>
<th>IN</th>
<th>INSTRUCTION</th>
<th>OUT</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td>11 13 14 16</td>
<td>R1=0</td>
<td>R2=1</td>
<td>R3=1</td>
<td>R6=4</td>
</tr>
<tr>
<td>9</td>
<td>12 15</td>
<td>R5=17</td>
<td>R7=11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.12.** The TDIS structure

Thus, as each new instruction is fetched, its PC is compared with the address of the first instruction from each TDIS entry. After that, the actual values of the processor’s registers are compared with the IN field belonging to the TDIS structure. In the case of a hit, the instruction sequence from the TDIS can be successfully reused, avoiding thus all RAW hazards between these instructions. The “execution” of these instructions implies only an update of the processor’s context according to the OUT values from TDIS. Consequently, in the same manner as in the previous cases, reusing instructions through this mechanism will have a benefic effect in terms of processing time. Considering a superscalar processor that can fetch, decode and execute maximum four instructions per cycle, the previous sequence is processed as in the following two figures.

**Figure 5.13.** Executing the program on a superscalar processor
It can be observed that the TDIS buffer determines the execution of the sequence through instruction reuse in only 4 cycles opposite to 7 cycles in the case of classical processing. If 6 instructions would be decoded instead of 4, the execution of the sequence would be reduced to only 3 cycles. The tests effectuated on the SPEC'95 benchmarks, presented in [Max98], and showed that between 17% and 26% of instructions corresponding to these programs were successfully reused by implementing such schemes. The TDIS concept is efficient because, like in the case of using combined instructions [Vin00], it eliminates the necessity to sequentially execute RAW dependent instructions. More than this, in the author’s opinion, from a certain point of view, the concept of dependent instructions sequence reuse ignores somehow the famous low of Gene Amdahl, since it surpasses the intrinsic sequential program and processes aggressively in parallel, even in this case, by updating the outputs corresponding to a sequence of dependent instructions. Therefore, the RAW hazard is not an immutable obstacle in this approach. It is possible a favorable combining of this concept with the previously presented trace-cache concept that operates specially over the limitations corresponding to the instructions’ fetch rate.

Dependence collapsing represents another technique, developed before DIR, proposed by Stamatis Vassiliadis from IBM (now at Delft University), that resolves execution data Read After Write hazards for instructions requiring ALU operation and therefore might reduce program’s critical path [Vas93]. Dependence collapsing can reduce the latency eliminating data dependencies by combining dependencies among multiple instructions into one more complex instruction. This technique improves the processor performance by “restructuring” the data dependence graph. A general
scheme capable of collapsing involves arithmetic and logical instructions. Instructions that will be collapsed can be even non-consecutive. There are at least two possible implementation strategies: the first run-time strategy, based on a combining hardware mechanism of instructions from pre-fetch buffer, and second, a static strategy, a software combining realized by an instruction scheduler. Into a previous work [Vin00], we presented run-time implementation of this technique: in instruction buffer there are detected possible data dependencies and if it is possible then the collapse is done under some certain rules (the dependence is generate by an arithmetic instruction). Three dependent instructions or two groups of two dependent instructions belonging to the “instructions window” may be collapsed. The distance separating the collapsed instructions is nearly always less than 8. In the same work we proved through trace driven simulation method that dependence collapsing and raising of instruction number that could be collapsed from 2 to 3 improves the processor performance with about 22%. Although, a growth of the collapsed instructions number from 3 to 6 doesn’t involve significant performance growth (under 3%). Considering a typical superscalar processor with two-port data cache, dependence collapsing improves processor’s performance with 8.51% at average.

In [Con99] the authors proposed to exploit dynamic instruction repetition by integrating some hardware-software techniques. More exactly, in this approach the compiler analyses the code in order to identify those program zones that could be reused during dynamic instruction execution. The compiler communicates to the object code through a simple interface, the reusable code zone, specifying the afferent inputs and outputs. During program execution, the results of these reusable zones are stored into hardware buffers for a potential reuse. For the exemplification of these novel ideas consider a macro-definition taken from SPEC benchmark 008.espresso. It computes the number of bits set to 1 within a word $v$ on 32 bits. The word $v$ is divided into 4 bytes and after they are preprocessed, each byte is used as an index in the $bit\_count$ table. This table contains words, each of them on 32 bits. Each $bit\_count$ function returns the number of bits set on logical one in the corresponding byte. After that, the results obtained for each byte of the word are summed, resulting thus the number of bits of 1 in word $v$.

```c
#define count_ones(v)
   (bit_count[v&255] + bit_count[(v>>8)&255])
   + bit_count[(v>>16)&255] + bit_count[(v>>24)&255])
```
Considering: A – arithmetic/logic instructions, L – Load, R – shift right, S – shift left, the graph of data dependences afferent to the previous sequence written in language C, is presented in Figure 5.15.

Through instructions A1, A2, A4 and A6 there are calculated the logical AND for each bit_count function call. Instructions S1 to S4 are doing logical shift to left with 2 bits, because bit_count table contains words on 32 bits. Instructions A3, A5 and A7 are adding the partial sums returned by bit_count function. Instructions R1, R2 and R3 are shifting to right the word v with 8, 16 and 24 bits respectively. It can be observed that the entire graph contains a single input (r3 – having value v) and respectively generates a single output in register r26 – the number of bits set to 1. The compiler can determine through anti-alias (disambiguation) analysis that the table bit_count is static and, consequently, it does not change during execution. Thus, it is obvious that the entire instruction graph would be reusable in the
hypothesis that the value introduced in r3 is the same. The reuse schemes based only on hardware excited by object code will be unable to determine the structure of this sequence and respectively the fact that its single output is r26. More than this, storing all these instructions implies high resource consumption. On the other hand, at compiler level, the graph is visible since the compiler built it. Consequently, the compiler can build an alternative graph to that from Figure 5.15, using for example a special instruction named REUSE. The REUSE instruction communicates with the hardware buffers in order to decide if r26 can be reused. In the case it can be reused, the REUSE instruction will update only r26 with that value (see Figure 5.15). That is, a hardware-software hybrid reuse scheme has the advantage, opposite to the previously presented purely hardware schemes, that provides a superior view over the program’s structure and thus implies better performances. More, this approach exploits a superior semantic of the programs written in high-level languages that is usually invisible in object programs. Obviously, the instruction set of the architecture must be extended with an appropriate interface that will intermediate the communication between the software and the hardware reuse logic. The performance gains reported in [Con99] for such hybrid architecture opposite to an equivalent superscalar one are approximately 30%, showing that the proposed hybrid approach is remarkable and efficient.

Figure 5.16 illustrates a superscalar microarchitecture with instruction reuse. The single difference opposite to the superscalar model consists in the apparition of the reuse buffer. During the fetch stage, instructions are extracted from instruction cache or main memory and are introduced into the instruction queue. The fetch stage is followed by the dispatch stage and the register renaming in order to eliminate name conflicts (WAR dependences – Write After Read and WAW dependences – Write After Write). In the stage of reading operands, the operand values afferent to the instructions are read from the register set or from the reorder buffer (the “rough copy” of the superscalar processor’s state where operations are finishing out of order), depending on the structure that contains the last version of registers. The access to the reuse buffer can be pipelined and overlapped with the fetch stage. Immediately after the instruction is decoded, during the stage of reading operands, a reuse test is performed on the inputs read from the RB in order to determine if the instruction results are reusable. If a reusable result is found, the corresponding instruction must not be processed further, and the result is sent directly to the reorder buffer. Load instructions avoid the instruction window only if the results of both micro-operations (address computation and memory access) are reusable.
Taking into account that the reuse test is a sequential process that depends on the number of dependent instructions in the chain, it can take one or even more cycles.

In the case of branch misprediction, the context recovery mechanism must be sufficient selective to not invalidate in the RB the instructions situated immediately after the possible ramification convergence point that could be reused. Thus, the reuse is exploited at maximum in this case too, with obvious benefits on performance. It would be interesting to study a hybrid architecture, obtained by combining an architecture with dynamic instruction reuse and an architecture with trace-cache, previously presented in this chapter. Both structures are in connection with dynamic instruction reuse. While the reuse buffer pre-eminently breaks the execution limits of the actual superscalar architectures, the trace-cache eliminates those limits connected with the instruction fetch process (although, as it was shown, it has favorable influence on the execution rate too). Summarizing, several advantages have been introduced by the dynamic instruction reuse technique as follows:

- Short-circuiting some levels of the pipeline structure through reused instructions, reducing the pressure over the resources (reservation stations, functional units, data-cache ports, etc.) that are needed by other waiting instructions.
- When an instruction is reused, its result becomes available earlier than in the normal processing situation (up to 40% gain), allowing other dependent instructions to be executed faster.
- Reduces branch misprediction penalty, by reusing, even partially, the code that is successor of the convergence point (squash reuse).
- The data dependence compression improves the execution time increasing the parallelism degree of the architecture.
- The percentage of reused dynamic instructions, computed on the SPEC’95 benchmarks, is significant: up to 76%.
- The speed-up obtained opposite to the superscalar model on the same benchmarks, is not as high as the reuse percentage: 7-15% at average, with a maximum of 43%.

<table>
<thead>
<tr>
<th></th>
<th>Sv</th>
<th>Sn</th>
<th>Sn+d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse_LD</td>
<td>21%</td>
<td>7%</td>
<td>12%</td>
</tr>
<tr>
<td>Reuse_ALU</td>
<td>37%</td>
<td>22%</td>
<td>21%</td>
</tr>
<tr>
<td>Speedup</td>
<td>15%</td>
<td>7.5%</td>
<td>10.5%</td>
</tr>
</tbody>
</table>

Table 5.1. DIR Quantitative Aspects (RB 1024 entries)
In [Kav03, Che06] the authors show that for integer benchmarks, due to the fact that some functions are re-executed with the same arguments it is possible to eliminate some function invocations. They observed that recursive programs frequently replicate their computation. As a simple example, considering a recursive Fibonacci function implementation (called $fib$), it creates multiple copies of $fib(1)$, $fib(2)$, ..., since $fib(n)$ and $fib(n-1)$ both spawn $fib(n-2)$. For calculating $fib(40)$ the function is called 127,278,861 times and it is called more than once with the same inputs 86,167,788 times. The authors showed that from 10% to 68% of function invocations might be skipped by buffering function results. The function reuse concept is integrated in the Reuse Buffer (RB) microarchitecture. Considering a MIPS based system, function’s arguments are available in registers R4-R7. RB is accessed concurrently with instruction’s fetch and it is indexed by the program counter of the function call ($JumpToSubRoutine$ JSR or $JumpAndLink$ JAL instructions). When the function is encountered again with the same inputs, its previous results are read from RB (Return Value field) and directly saved in R2 and R3 registers, designated for function results in MIPS convention.

<table>
<thead>
<tr>
<th>PC</th>
<th>Input1</th>
<th>Input2</th>
<th>Input3</th>
<th>Input4</th>
<th>Return Value</th>
</tr>
</thead>
</table>

**Figure 5.16.** Generic superscalar microarchitecture with reuse buffer

**Figure 5.17.** RB’s Entry format for function reuse
In [Sha06] the author is focused also on the concept of dynamic reuse of procedures’ results. The described method uses a hardware mechanism that caches the address of the called procedure along with its arguments and returned value. When the same procedure is called again using the same arguments, the returned value can be provided without procedure’s re-execution. Although this approach can be highly effective in some cases, it is limited to procedures that do not use side effects, and use only by-value parameter passing. The proposed method might be appropriate for specific computing-intensive applications.

References

[Con99] Connors D., Hwu W.M. – Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results, IEEE 1072-4451, 1999
[Sil99] Silc J., Robic B., Ungerer T. - Processor Architecture, from Dataflow to Superscalar and beyond, Springer-Verlag, 1999
6. DYNAMIC INSTRUCTION VALUE PREDICTION AND SPECULATION

6.1. INTRODUCTION TO VALUE PREDICTION

Prediction methods are more and more used in Computer Science and Computer Engineering research, applications and commercial hardware/software products [Vin02]. For example, almost all present-day multiple instruction issue microprocessors are using advanced branch prediction techniques in order to increase Instruction Level Parallelism (ILP). Through dynamic branch prediction microprocessors are speculatively processing multiple basic-blocks in parallel and therefore their ability to increase ILP is stronger. Dynamic predictive methods are implemented also in cache blocks or virtual memory pages replacement techniques, during some cache miss processes or page fault exceptions. These dynamic adaptive methods are more efficient than non-adaptive methods like LRU et al., that cannot run-time adapt to program’s behaviour.

Dynamic Instruction Value Prediction (VP) is a relatively recent developed processing technique (M. Lipasti et al. [Lip96a]) independently developed by four research groups (AMD NexGen – Widigen & Sowatski, they have a patent in March 1996, Haifa University – F. Gabbay & A. Mendelson, they also have an USA patent in 1997, Carnegie Melon University – M. Lipasti et al., Wisconsin-Madison University – Y. Sazeides & J. Smith). In 2004 there were published about 70 papers at the most prestigious conferences focused on VP. Its main aim is to early predict the instructions results, during their fetch or decode pipeline stages, and, therefore, to speculatively execute the instruction flow. The predicted value can then be used as an input to some subsequent dependent instructions so that they can execute earlier. If the prediction is incorrect, recovery mechanism must be employed to squash speculative results and re-execute all instructions that already used the incorrectly predicted value. An important challenge for VP technique is to
compress the program’s dynamic critical path and therefore to solve the so-called Issue Bottleneck (or Data-Flow Bottleneck, Critical Path Bottleneck). Otherwise, if the predicted instructions do not belong to the critical path, the technique is obviously not efficient in reducing critical path’s limitation. Therefore, the VP technique tries to avoid a fundamental limitation in the present day computing paradigm, the Read After Write (RAW) data hazards, thus the intrinsic program sequential execution.

**Dynamic Value Locality**

According to [Lip96a], Value Locality represents a third facet of the statistical locality concepts used in computer engineering (close by the well known, and used, temporal and spatial locality concepts). Value Locality means the statistical probability (frequency) to produce a value belonging to the previous \( k \) value instruction’s instances. The concept is strongly related with the redundant computing concepts (like the memoization technique) including here the relatively recently introduced Dynamic Instruction Reuse (DIR) technique [Sod97]. As the authors showed, 90% of a program’s dynamic instructions are produced by only 20% of the program’s static instructions. That means the degree of use of some instruction is high and might involve a significant value locality.

In my opinion, the main difference between temporal and spatial locality and, respectively, value locality consists in the fact that the first two metrics are focused on item’s (data’s or instruction’s) address reuse rather than on the instruction’s result reuse. More precisely, temporal locality means a significant statistic probability that the current accessed memory address to be accessed again in the nearest future. Value locality means a bit more: the re-fetched instruction will produce a result belonging to its previous instances, with a significant probability, too. Therefore, not only that the processor will fetch again the same instruction, but this instruction will produce – with an important statistical probability – the same result as one of its previous instance. Temporal locality is a consequence of value locality. Temporal locality is necessary but not sufficient for value locality to exist. Obviously, value locality might be exploited through value prediction [Vin04].

Measurements using SPEC benchmarks shows that value locality on Load instructions is about 50% using a history of one (producing the same value like the previous one) respectively 80%, using a history of 16 previous instances (see figures 6.1, 6.2). Therefore, it is natural to introduce the
metric of \textit{k-value locality}, meaning the new produced result belongs to the previous \(k\) produced results. It is necessary to observe that, for example, a \textit{k-value locality} = 95\% theoretically means that the best context predictor might predict the new value with a probability less or equal then 95\%. In other words, the \textit{k-value locality metric} theoretically represents the ultimate accuracy limit of a contextual predictor (Markov predictor). More generally, value locality consists in a dynamic correlation between a resource name (register, memory location, I/O port, etc.) and the value attached momentary to that resource. However, value locality and value predictability is not the same concept. You can have 100\% locality and be very unpredictable (as a simple example, a random sequence of 0s and 1s has 100\% value locality with history of two values but can be very unpredictable). A more general example: a difficult predictable sequence using any Markov predictor but having a high value locality degree.

Value locality is justified by some statistical empirical observations like the followings [She03]:

- Data redundancy – the input data sets for general-purpose programs are redundant (sparse matrices, file texts with many blanks and many repetitive characters, free cells in table calculus, etc.)
- Exploiting compiler error tables when there are generated repetitive errors.
- Program constants, meaning that is more efficient to load program constants from memory than constructing them as immediate operands.
- In \textit{case} – \textit{switch} constructions, it is needed the repetitive load of a constant (branch’s base address)
- Virtual function calls – loaded a function pointer that is a constant during the run-time. Similar in object oriented programming is polymorphism’s implementation.
- Computed branches – for calculating a branch address it is necessary to load a register with the base address for the branch jump table, which might be a run-time constant.
- Register spill code – when all the CPU registers are busy, variables that may remain constant are spilled to data-memory and loaded repeatedly.
- Polling algorithms – the most likely outcome is that the event being interrogated for has not yet occurred, involving in redundant computation to repeatedly check for the event, etc.

Of course, value locality especially for Load instructions might be influenced by some compiler optimisation techniques like \textit{loop unrolling},
software pipelining, tail replication, etc. [Hen02]. These optimisations generate multiple Load instances. The value locality can be exploited to bypass the redundant operation through value prediction or dynamic instruction reuse. This is especially beneficial when the operations that are on the critical path can be skipped.

In order to better understand the value locality degrees, I present now some simulation results obtained using a dedicated simulator developed at University “Lucian Blaga” of Sibiu, in our Advanced Computer Architecture & Processing Systems (ACAPS) Research Centre [Flo05] - http://acaps.ulbsibiu.ro/research.php. All the simulations used SimpleScalar Tool ver. 3.0 and SPEC benchmarks [Bur97], [SPEC]. In this particular investigation were simulated 5,000,000 dynamic instructions for each SPEC benchmark. SPEC 95 and 2000 were compiled for SimpleScalar using GNU Compiler Collection (GCC) that represents the leading tool suite for portable developments on open platforms. At this moment (November 2006) GCC supports at least 6 languages and about 30 target processor architectures. It offers inter-procedural analysis and optimisations (including profile-directed optimisations), loop optimisations, automatic vectorizations, etc., being de facto standard for program optimisation. As it can be observed from the first two figures, from the value locality point of view, the optimal history length is 8 instances for both instruction centric and, respectively, data memory address centric approaches. This might involve an optimal context predictor having an 8 values context. Unfortunately, despite of the fact that memory centric locality is higher (see Figure 6.3) there is an important disadvantage. Load Value Prediction using data address cannot be done during the Instruction Fetch stage. It can be done only later, in ID or ALU pipeline stages, depending on the Load addressing mode (MIPS CPU). Figure 6.4 compare ALU instructions value localities with Load instructions value localities, showing better average degrees with about 2.5% for Load instructions.
Figure 6.1. Value Locality for Load Instructions (Instruction Centric, PC)

Figure 6.2. Value Locality for Load Instructions (Memory Centric, Load Data Address)
Figure 6.3. Instruction Centric vs. Memory Centric Load Value Locality Approaches (PC vs. Data Address, 8 history values)

Figure 6.4. Load Value Locality vs. ALU Value Locality using a history depth of 32
6.2 PREDICTING LOAD INSTRUCTIONS

Considering an integer processor having 32 bits general registers, at a first look someone might conclude that the probability of a correct value prediction is $1/2^{32}$, therefore it is practically zero. Fortunately, due to the previous value locality explained concept, the value prediction probability is quite optimistic.

Value Prediction is a speculative technique that is based on instructions value prediction, followed by speculative execution of the successive dependent instructions. This speculative execution is later validated, after the instruction’s result is produced. In the case of a correct prediction, the (critical path’s) execution time is reduced. Otherwise, the wrong predicted instructions must be re-executed again (recovery process), with corresponding time penalties. Prediction and speculation need dedicated mechanisms for:

- Detecting mispredicted values and checking the prediction’s accuracy
- Processor’s context recovery after a miss-prediction (example: recycling the branch misprediction recovery mechanism that flushes out speculative instructions and re-fetches all instructions after the mispredicted one)
- Issuing dependent instructions speculatively (involving the standard out-of-order logic with some minor modifications)
- Storing and bypassing predicted values for the next dependent processed instructions. This speculative mechanism is the main VP’s advantage.
As it can be seen in the figure 6.5, the Value Prediction technique predicts next value based on the last values belonging to that instruction; it executes the operations using the predicted value and the speculative execution will be validated when the correct value is known, (after execution stage). If the value was correctly predicted the critical path might be reduced. Otherwise, the instructions already executed with wrong entries must be executed again (recovery process). Above it can be seen a figure that intuitively presents the differences between VP and DIR techniques. The VP key consists in the speculative prediction process in DIR case the key consists in a Reuse Buffer structure (RB) as we detailed in the previous chapter. Anyway, VP technique is far more debated by the research community comparing with DIR.

During the instruction’s fetch stage, the VPT table is accessed with the Program Counter (PC). In the case of a valid prediction, the VPT will generate the predicted value to the subsequent corresponding RAW dependent instructions (waiting in the Reservation Stations). After execution, when the real value is known, it is compared with the predicted value. In the case of a miss-prediction the speculatively executed dependent instructions are re-issued again for a correct execution.

The recovery mechanism has two approaches: re-fetch and selective reissue [She03]. As we already pointed out, in re-fetch based recovery, value
mispredictions are treated exactly like branch mispredictions. This mechanism is attractive because it requires few changes in branch misprediction recovery mechanism. The obvious disadvantage is a severe misprediction penalty. The goal of selective reissue is to reduce this penalty. With this approach only those instructions that are data dependent on a mispredicted value are required to reissue. This requires a mechanism for propagating misprediction information through the data flow graph to all dependent instructions. Intel Pentium 4 implements selective reissue to recover from cache hit speculation; data are forwarded to dependent instructions speculatively, thus before tags’ checking. In the case of a tag miss only the dependent instructions are reissued.

Due to their big potential latencies (misses in caches hierarchy – memory wall limitation, Page Fault exceptions, a Load instruction blocking the Reorder Buffer, etc.), predicting Loads is a very important challenge in Computer Architecture. The first developed value prediction scheme was the so-called Load Value Prediction Unit (LVPU) [Lip96a] and it was integrated in an equivalent PowerPC 620 simulated microarchitecture. It reduces the memory latency and, on the other hand, it increases memory bandwidth (there is no need for so many memory ports).

The LVPU components are:

- **LVPT (Load Value Prediction Table)** – It is a direct mapped cache memory indexed by Load’s Program Counter. It contains the predicted values, each value having attached a confidence counter.

- **LCT (Load Classification Table)** – It is a mechanism that dynamically classifies Load instructions in 3 distinct categories: unpredictable, (weak) predictable and strong predictable (so called “constant Loads”). In the case of an unpredictable Load, there is no generating any prediction (the instruction must access the memory hierarchy – see figures 6.6 and 6.7). Through this classification, there can be avoided costs involved by mispredictions. On the other hand, when the Load is classified as predictable, it can be avoided the memory hierarchy access. LCT is a direct mapped cache, having the same number of entries as LVPT, addressed with PC\textsubscript{LOW}. Researchers showed that over 80% of Load instructions are correctly classified, using a confidence counter like that presented in figure 6.7.

- **CVU (Constant Verification Unit)** – It is a checking unit utilised only for strong predictable Loads (constant Loads classified by the corresponding LCT automata, see figure 6.7). A Load classified as strong predictable (constant) will have a 100% correct value in the LVPT structure.
CVU assures the necessary coherence mechanisms - between the LVPT value, and, respectively the real value (stored in the memory system) - for the strong predictable Loads. If a LVPT entry is classified as being “strongly predictable”, its LVPT index (program counter’s low part) and memory address are stored in an associative manner into the CVU. Thus, a CVU entry has 3 fields: Load’s PC, Load’s Data Address and a Valid Bit. If a subsequent Store instruction writes to a valid memory address belonging to CVU, the corresponding CVU entry is invalidated and therefore, the corresponding LCT automata – pointed by CVU Load’s PC field - will transit from the constant state to the weak predictable state. If at one moment is executed a constant Load, during its EX1 stage there will be an associative search in CVU based on the LVPT index (PC) concatenated with the Load memory address. If there is obtained a hit search (meaning that the CVU entry was not invalidated by a Store or it was not already evicted), the corresponding LVPT value is correct for sure. In this case, there is no need for a further comparison between the predicted value and the value fetched from the memory system. Unfortunately, this correct predicted value validation seems to be done a bit too late in the pipeline, meaning that the subsequent dependent instructions might stall for one cycle. For a weak predictable Load, this comparison is obviously necessary (checking). Figure 6.6 presents the prediction mechanism and the structure interactions (LVPT, LCT and CVU).
LVPT and LCT structures are simultaneously addressed during the Load instruction’s fetch stage using PC\textsubscript{LOW}. If the corresponding confidence automaton is in its weak predictable state, the predicted value will be immediately forwarded to the dependent instructions waiting on it in the pipeline structures. During EX1/EX2 stage (data address calculus/cache access, see figure 6.6) Data Cache and CVU structure are simultaneously accessed. After the Data Cache (or the other memory hierarchy components) will generate the requested data, it will be compared with the weak-state predicted data value (Comp stage in figure 6.6). Obviously, in a miss-prediction case there are necessary efficient recovery mechanisms in order to re-execute the program from the first miss-predicted instruction based on a correct CPU context. If the corresponding confidence automata is in its constant state, the predicted value will be forwarded to the dependent instructions, too. In this case the LVPT predicted value is necessary to be
later validated by the CVU unit, assuring its correctness. We discriminate here two sub cases. First, in a **miss**-Data Cache case, therefore being necessary to access the superior memory hierarchy, the correct prediction advantages are very important from the general processing timing point of view. In this case the correct predicted value is given in significant timing advance. On the other hand, in a **hit** first-level Data Cache case, having the access time one CPU cycle, the correct predicted value might be too late generated in EX1/EX2 stage (as it was already shown, Data Cache is simultaneously addressed with the CVU data structure). This means that the correct predicted value is a bit too late generated because, at the same moment, the Data Cache might generate the same correct value. Considering an infinite LVPT structure, the authors reported a speedup of about 30% compared with an equivalent superscalar without value prediction. The speedup depends on the superscalar factor (issue rate). A great issue rate will obviously improve the potential critical path compression.

![LCT Classification Automata](image)

**Figure 6.7. LCT Classification Automata**

In [Sam04] are described several approaches to develop more efficient load-value predictors by exploiting the type of load instructions (byte, word, double-word, etc.). These techniques are implementable in architectures that support these multiple load types. The type of a load is used as a selector in a hybrid predictor, too. Furthermore, it is shown that it is possible to split predictor tables into smaller tables with faster access times without degrading the performance.

Relatively recent studies [Lep00] introduced the Store locality and Store prediction methods, with good results especially for multiprocessors systems (interconnection network traffic reducing). As for Load
instructions, the Store value locality was measured using its PC (instruction centric) or its accessed data address (memory centric). In both cases the value locality degree is between 30% and 70%. It is introduced the “silent Store” concept, meaning that a Store writes the same value like its previous instance (34% - 68% of the dynamic Store instructions are silent Stores). These silent Store instructions could be cancelled, with an important performance benefit. So, removing these Store instructions at some points in the program’s execution (either statically at compile time, or dynamically at run time) some potential benefit can be gained in execution time and/or code size. Also, there are reduced: the pressure on cache write ports, the pressure on store queues and the data bus traffic outside the processor chip. The free silent store squashing concept is based on idle read port stealing to perform store verifies and aggressive load/store queue to exploit temporal and spatial locality for store squashing.

In [Lep01] the authors describe how to enhance the performance of mono-processor programs by squashing silent stores. Frequently occurring stores are highly likely to be silent. The fact that many stores are silent creates the opportunity for removing a significant fraction of all the instructions in the dynamic instruction stream, since many of those instructions are merely computing the redundant results being stored by the silent stores (for example, related instructions that are used in the store’s data or address computation). Of special interest are critical stores that represent specific dynamic silent stores that, if not squashed, will cause a cache-line to be marked as dirty and hence require a write-back (it is sufficient to only squash the critical silent stores to obtain maximal write-back reduction; in contrast, squashing non-critical silent stores has no benefit in terms of write-back reduction). Also in these works it is shown how to reduce address and data bus traffic on shared memory multiprocessors for reducing bus traffic.

The work from some researchers extends the definition of silent store to the so called “temporarily silent stores”. These stores change the value stored at a memory location, but only temporarily, and subsequently return a previous value of interest to the memory location. Multiprocessor sharing is redefined to account for temporal silence stores. It is shown that up to 45% of communication misses in scientific and commercial applications can be eliminated by exploiting values that are changing only temporarily. Also, it is developed a practical mechanism that detects temporally silent stores and removes the coherence traffic they cause in conventional multi-processors. It is shown that up to 42% of communication misses can be eliminated with a simple extension to the MESI cache coherence protocol.
6.3 GENERALISED VALUE PREDICTION

In [Saz97] the authors developed an empirical classification related to the dynamic generated instruction values. According to them, there are 3 distinct types of instruction values: constant, incremental and non-incremental. Constant sequences are the simplest (8, 8, 8, ...), and are surprising frequently generated. Incremental sequences (1, 3, 5, 7, 9, ...) are characterised by the fact that the difference between two successive values is a constant, named stride (if the stride = 0, we have constant sequences). They can be generated by loops index, by accessing a table, a matrix, some data structures, etc. Non-incremental sequences represent any other value sequences (ex. 28, 13, 99, 107, 23, 456, ...). For example, crossing a simple linked list can frequently generated addresses belonging to this last class. Of course, we can now imagine also some hybrid classes, based on these 3 main classes. Also, from another point of view, there can be considered repetitive and, respectively, non-repetitive sequence values (incremental – 1, 3, 5, 1, 3, 5, ... respectively non-incremental – 1, 13, 88, 7, 1, 13, 99, 7, ...).

These classes involve at least two distinct main categories of predictors: computational and contextual. Computational predictors are predicting the next value based on some previous values, in an algorithmic (computational) manner, therefore according to a deterministic recurrence formula. An incremental predictor belongs to the computational class. As it can be observed, a computational predictor doesn’t directly derive from the value locality concept. A contextual predictor predicts the next value based on a particular stored pattern (context) that is repetitively generated in the value sequence, in a markovian stochastic manner. Theoretically they can predict any repetitive value sequences. A context predictor is of order \( k \) if its context information includes the last \( k \) values, and, therefore, the search is done using this pattern of \( k \) values length. As we already pointed out, a contextual predictor of order \( k \) derives from the \( k \)-value locality metric that represents an idealised \( k \)-context predictor.

6.3.1 COMPUTATIONAL PREDICTORS

A. “Last Value Predictors”
They predict – as being the new value - the last value produced by that instruction.

![Figure 6.8. “Last Value” Prediction Scheme](image)

However, there are some variants that change the value prediction strategy based on hysteresis. This could be implemented through an up/down-saturated counter attached to each value-predictor (confidence counter, incremented when a correct prediction is done, decremented otherwise – see figure 6.7). The stored value is evicted only if its confidence is under a certain threshold. Another similar hysteresis mechanism doesn’t evacuate a value until the new value was generated successively over a certain threshold.

The field “Num” implements the hysteresis mechanism (confidence counter). In order to maintain a small prediction table – very important for low power consumption and integration area, an important request especially in embedded systems - there can be used some hashing mechanisms for addressing VHT (figure 6.8). The average prediction accuracy - measured on SPEC benchmarks - obtained for Load instructions using this scheme was 49%. Considering that VHT stores the last 4 generated values and that the predictor will chose the right one (if this would be possible), the prediction accuracy growth to 61% [Lip96].

B. Incremental Predictors
Considering that $v_{n-1}$ and $v_{n-2}$ are the last generated 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 $(v_{n-1} - v_{n-2})$ represents the stride. Of course, the stride might be variable in time (constant only for some certain real-time periods). The stored stride is modified based on a confidence mechanism (“Num” counter, see figure 6.9). As in the previous case the counter is incremented / decremented according to the prediction result, respectively correct/incorrect. Another hysteresis strategy is the so-called “$2$ – delta” method. In this case there are stored the last encountered two strides ($s_1$, being the most recent one, and $s_2$), $s_1 = v_n - v_{n-1}$. Stride $s_2$ is currently used in the prediction process. Only when the same stride value ($s_1$) is generated two times consecutively, it is done the transfer $s_2 \leftarrow s_1$. In the case of repetitive incremental sequences, both strategies reduced the number of wrong predictions from two, to one. Figure 6.9 presents a typical incremental predictor.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure6.9.png}
\caption{Incremental Predictor Structure}
\end{figure}
An important action for an incremental predictor consists in stride’s detection. First time when an instruction produces a result it will be stored in field “Val” from VHT and its corresponding automata will be in the “Init” State (see figure 6.10). While the next instruction instances generate the same result (stride = 0), the automata will be in its initial state. If a next instruction instance will produce a different result ($R_1$) then it will be calculated the stride. Simultaneously the automata will transit in Transition State (see fig. 5). A new instruction dynamic instance will involve $S_2 = R_2 - Val \rightarrow VHT$ and $R_2 \rightarrow VHT$. If $S_1 = S_2$ then the automata will transit in its Stable State, otherwise it will stay in the Transition State. Only in the Stable State it will generate a prediction value given by formula $R_2 + S_2$. There will be possible other similar solutions.

### 6.3.2 CONTEXTUAL PREDICTORS

In this case the prediction will be done based on the most frequent value that follows a pattern context in the string of history values. Thus, Value Prediction is in this case a classic time series problem. Predictability is an intrinsic property of the string values. An important contextual predictor class implements, with some simplifications, the general “Prediction by Partial Matching” (PPM) algorithm used also in speech recognition, data compression, etc. PPM works on a set of $(k+1)$ Markov predictors of order $k, k-1, k-2,..,2, 1,0$ respectively [Mud96] (see figure 6.11).
Value sequence:  aaabcaabcaaa?

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
9 & 2 & 2 \\
\end{array}
\quad \quad
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
6 & 2 & 0 \\
\end{array}
\]

0\textsuperscript{th} order Markov Prediction: a

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
3 & 2 & 0 \\
\end{array}
\quad \quad
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
0 & 2 & 0 \\
\end{array}
\]

2\textsuperscript{nd} order Markov Prediction: a

3\textsuperscript{rd} order Markov Prediction: b

\textbf{Figure 6.11.} Prediction by Partial Matching Algorithm

A Markov predictor might be represented by a graph of states and transitions between states. Each transition from state X to state Y in the corresponding graph has assigned a weight representing the fraction of all X references that are followed by a reference Y. Obviously, the sum of weights belonging to a certain state must be equal to 1 (normalization or equilibrium equation). If the predictor of order \( k \) cannot predict anything, it will be used the predictor of order \((k-1)\) and so on, up to the 0\textsuperscript{th} order predictor. Details on Markov predictors are given in chapter 9.

Considering figure 6.11, it can be observed that only a Markov predictor of order 3 will predict the correct value (in this case is b!). Of course, a greater order Markov predictor will also correctly predict. A long pattern context might be useful but, on the other hand, in real implementations it could act like a noise (because a lot of its information is not correlated with predictor’s behaviour). It’s prohibiting to hardware implement a complete PPM predictor, but it will be useful to implement some simplified versions. Anyway, the PPM predictor is a good general model for context predictors and it might offer an ultimate limit of context predictability, as Trevor Mudge claimed in his work focused on branch prediction [Mud96] (even if he hasn’t prove it; perhaps other prediction techniques like Time Series,
Neural Networks, Bayesian Predictors, Hidden Markov Models, Support Vector Machine, etc. might be better, like some researchers proved, at least in branch prediction area [Vin00], [Ste01]).

In [Saz97] the authors defined two important orthogonal characteristics, in order to understand better a predictor’s behaviour. The first characteristic is called *Learning Time* – LT, and it represents the minimum number of values belonging to the string values, generated before the first correct prediction. The second one is called *Learning Degree* – LD, and it represents the prediction accuracy after the initial learning period (LT).

We made the following notations:

- SC = a constant value sequence
- SI = an incremental value sequence
- NSI = a non-incremental value sequence
- SIR = a periodical (repetitive) SI
- NSIR= a periodical (repetitive) NSI

- \( T \) = the period of a repetitive value sequence
- \( O \) = PPM’s order

Taking into account these notations, the following table describes some predictors’ behaviour, considering the above described value sequences.
Table 6.1. Some predictors’ behaviour for different value sequences

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Last Value</th>
<th>Incremental</th>
<th>PPM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LT</td>
<td>LD(%)</td>
<td>LT</td>
</tr>
<tr>
<td>SC</td>
<td>1</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>SI</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>NSI</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SIR</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>NSIR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

As it can be observed, there is not a best particular predictor and no one predictor can correctly predict NSI sequences. On the other hand, it’s obvious that for any NSI values that can be generated through an algorithm (function), we ideally can develop a corresponding predictor that will correctly predict the next values. Considering, for example, a loop that calculates Fibonacci’s recurrent string \( v_n = v_{n-1} + v_{n-2} \), for a perfect prediction it must activate a corresponding predictor in hardware, implemented by the previous formula. Unfortunately, generally speaking, it is not possible to predict NSI sequences generated by some algorithms under the present-day instruction-processing paradigm (hardware prediction structures accessed by information derived from the object code). In order to do this, perhaps we need a new hardware-software interface model (a new Instruction Set Architecture – ISA philosophy), knowing better HLL code semantics. In other words, the compiler must transmit to the predictor, mainly through the ISA, some High Level Language (HLL) relevant semantic information, related to the HLL data-structures, program-corpus, processing information semantics, etc. in order to be efficient exploited by the machine. Unfortunately, in the present day approach, after compilation, the resulted object code, lose almost all of the HLL program’s semantics. Thus, in my opinion, I believe we need a more semantic approach in Computer Architecture based on a new code generation paradigm, on a new hardware-software interface model that will include more HLL application semantics.
Returning to Table 6.1 it can be observed that periodical value sequences are 100% correctly predicted by context predictors after a learning period that is greater than that for incremental predictors. This fact is pointed out also in the next figure (fig. 6.12) where there are compared a PPM predictor of order 2 with an incremental predictor. The value sequence is a SIR one with the period $T = 4$. After a greater learning period (LT=6 > 2), PPM predictor obtains 100% accuracy instead of the incremental one that obtains only 75%.

<table>
<thead>
<tr>
<th>Sequence Predictor</th>
<th>1 2 3 4 1 2 3 4 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental Predictor</td>
<td>$\begin{array}{c} X \ X \ 3 \ 4 \ 5 \ 2 \ 3 \ 4 \ 5 \ 2 \ 3 \ 4 \ LT=2 \end{array}$</td>
</tr>
<tr>
<td>PPM Predictor</td>
<td>$\begin{array}{c} X \ X \ X \ X \ X \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \ LT=6 \end{array}$</td>
</tr>
</tbody>
</table>

**Figure 6.12. PPM Predictor vs. Incremental Predictor**

Figure 6.13 presents a generic scheme for a contextual predictor. During the instruction fetch stage the context belonging to VHT (Value History Table) is addressed using PC. Generally speaking, this context is obtained from hashing in some way the last $k$ instruction’s values and/or even other relevant information. This context will address the VPT (Value Prediction Table). A VPT word contains two fields: $Val$ – the last instruction value or values, and, respectively, $Confid$ – confidence degrees attached to each value belonging to $Val$ field (saturated counters, incremented for correct predictions and decremented otherwise). The predicted value will be that value with the highest attached confidence, if this confidence is greater than a certain threshold. Practically the scheme implements a simplified Markov predictor of order $k$ (because it counts the frequencies for each value following a certain context).
Two Level Adaptive Value Predictors

Some authors proposed – through an analogy with the well-known Two Level Adaptive Branch Predictors [Yeh92] – some Two Level Adaptive Value Predictors [Wang97]. The scheme presented in figure 6.14 stores in the first level (VHT) the last 4 values per each dynamic instruction (Data Values field). A selection mechanism might choose one of these 4 values as the predicted value. These 4 data values are binary codified {00, 01, 10, 11}. While an instruction produces one of the 4 stored values, eventually one of these values will be predicted. Otherwise - if it generates a new value - this new value will be introduced in VHT (Data Values field) and another value belonging to VHT will be evicted (ex. through a LRU algorithm or based on the attached data values confidences C0 – C3).

Figure 6.13. A generic contextual predictor scheme
The VHP (Value History Pattern) field belonging to the VHT stores a binary pattern on $2p$ bits. This pattern codes the last $p$ results generated by the corresponding instruction (each result is coded using two bits as we already shown). VHP will update like a shift logic left register, according to the corresponded predicted value. The VHP field address the PHT table (Pattern History Table). A PHT line contains 4 independent saturated counters ($C_0$, $C_1$, $C_2$, $C_3$). If the current instruction produces the value associated with the counter $C_k$, ($k=0,1,2,3$) belonging to the corresponding PHT line, then $C_k$ is incremented (+3), and, the other 3 counters are decremented (-1). The MAX circuit works as following: if $\text{MAX} (C_0, C_1, C_2, C_3) = C_k$ and $C_k \geq A$, where A is a certain threshold, then MAX circuit generates the binary value of $K$, codified on 2 bits. Otherwise, if $C_k < A$, it will be not generated any prediction ($C_k$ value associated with the VHP pattern hasn’t a sufficient high frequency). It can be observed that – through the proposed codification mechanism - VHP index represents a compressed CPU context (hashing). This involves reasonable PHT capacities.

The scheme presented in figure 6.14 represents a simplified feasible implementation of the PPM generic predictor (see figure 6.11). Each instruction has associated its own context stored in VHT, representing the last $p$ binary codified values produced by this instruction (VHP pattern).
Using this pattern, the Two Level Predictor calculates – using $C_0$ to $C_3$ counters – the most frequently generated value in the last $p$ instances of the instruction. Now it’s clear why this predictor belongs to the most general PPM contextual predictor. On the other hand, someone might answer: why the VHT structure stores only the last 4 instruction’s values? There is a statistical explanation: 15% - 45% from the instructions produce only one value in their last 16 dynamic instances and 28% - 67% produce maximum 4 distinct values considering their last 16 instances.

Obviously, there might be some interference in the PHT structure. An interesting solution in this sense is given in [Des02] where the authors proposed to use a second hashing function ($\text{hash}_2$), independent of the first one. The next two figures (6.15 and 6.16) suggest how these structures are working and updated after instruction’s execution. The interferences are strongly reduced with great benefits on prediction’s accuracy.

![Figure 6.15. Using two independent hashing functions](image)
6.3.3 HYBRID PREDICTORS

Some instructions values are better predicted by a particular type of predictor while other instructions values by other type of predictor. Therefore, the hybrid prediction idea seems to be very natural in order to obtain the optimal performance-cost trade-off. That means using two or many predictors working together. Figure 6.17 presents a hybrid predictor composed by a Two Level Adaptive Predictor and an Incremental Predictor [Wang97]. The main idea is quite simple: if the contextual predictor is generating a value, it will be the predicted value, and otherwise, the incremental predictor might generate the predicted value. In my opinion this fixed prioritisation is not optimal but it is quite simple to be implemented; a dynamic prioritisation based on some confidences should be better, but in this case it should be necessary a dynamic (adaptive) meta-predictor in order to select the best predictor at a certain moment. According to [Wang97], by simulating Eqntott SPEC benchmark, an incremental predictor obtained 63% prediction accuracy, a Two Level Predictor having the threshold value A=6 obtained 67% accuracy and the hybrid prediction obtained 81% accuracy, involving thus a kind of synergism.
In [Tho04] the authors implemented the stride value predictor (Figure 6.9), the Two Level Adaptive Value Predictor (Figure 6.14) and the hybrid predictor (Figure 6.17) using perceptrons rather than state automata or confidence-saturated counters. The perceptron having the highest output value (if it is greater then a certain threshold) would select the predicted data value. This value will be forwarded to the dependent instructions only if a selection is done. The Value History Pattern (VHP) represents the input of the perceptrons. Training is required only if the perceptron’s output is incorrect or below the threshold. It is shown that value prediction with perceptrons outperform classic value predictors. Unfortunately, perceptrons involve more time for prediction and update.

In [Cal99] the authors are developing some interesting techniques in order to reduce the prediction table pressure, by filtering the instructions that accessed these tables. The ideal case is to select those dynamic instructions belonging to the critical path. Unfortunately the critical path is not known.
during run-time. Practically, for detecting the time consuming instructions we need the compiler’s help together with some profiling information. For simplicity, the authors proposed a technique that selects the dynamical instructions belonging to the current longest RAW chain from the instruction window. For prediction tables of 1024 entries they report an average performance growth of about 11%, comparing with a classical superscalar structure. Also in [Fie01] the work is focused on critical path run-time detection and value prediction. So, egalitarian policies could be replaced with cost-sensitive strategies that will grow increasingly effective as processor become more parallel. Their so-called focus value prediction technique implements an optimization that uses the critical path for reducing the frequency of miss-speculations while maintaining the benefits of useful predictions. The key approach consists in using a dependence graph model of the microarchitectural critical path that identifies run-time bottlenecks by incorporating data and machine-specific dependences. There are at least two reasons why critical path is difficult to be exploited in dynamical approaches. First reason is related to its the global nature. While adaptive compilers can frequently guess the critical path, processors are seeing only a small instruction-window at a certain moment. The second reason is that the compiler’s view of the critical path, consisting of data dependences, doesn’t represent exactly the critical path on a particular CPU implementation because a real CPU involves resource constraints (a finite reorder buffer, finite fetch bandwidth, branch mispredictions, etc.) that introduce additional dependences. The authors divided the design tasks into two problems: development of a dependence-graph model of the critical path (1) and a predictor that follows this model when learning the critical path run-time (2). Each node, representing a dynamic instruction, corresponds to instruction’s dispatch, executed or committed phases. Edges represent various data and resource dependences between these three events. Data dependences connect execute nodes. A resource dependence due to a mispredicted branch induces an edge from the execute node of the branch to the dispatch node of the correct target. It is developed a token-passing algorithm that computes the critical path. The validation model indicates that it closely represents the real dynamic critical path (accuracy about 88%). In essence, the algorithm would build the dependence graph for the entire execution, label the edges with observed latencies, and then find the longest path through the graph. By focusing on critical path instructions the predictor can give priority to critical resource consuming instructions and avoid speculation on non-critical instructions. Finally the authors shown that critical path based dynamic instruction scheduling improves performance at average with 10% and focusing value prediction only on critical instructions
improves performance by as much as 5% [Fie01]. However, in our opinion, until now the value prediction’s performance gain is not quite well understood.

In [But02] the authors developed a new metric, in my opinion related to the well-known register life time metric (e.g. register life), called degree of use, being the number of times a particular value is used by all subsequent instructions in the program. Of course, this proposed metric has sense only for instructions that modify a register. As a consequence, the final degree of use of a value (initially produce by an instruction) is only certain when the value is overwritten by an unrelated value (produced by another instruction). Any instruction that reads the value between its generation and its destruction affects the actual degree of use and must be tracked. Values with a predicted degree of use of one need not even use a CPU register (using a bypassing network). Many instructions generate a single unique degree of use. The majority of static instructions (84% on SPEC 2000) generate the same degree of use every execution. Degree of use two values account for only 14% of all values. The number of values with a higher degree of use is very small. The authors proposed a degree of use predictor, because knowing the number of consumers of a dynamic value before that value is even generated enables several potential optimizations to the value communication structures (instruction window, bypass network, register file). Unfortunately the authors didn’t focus on this important aspect, just informing “the exploitation of degree of use prediction to optimize value communication is part of our ongoing work”’. They only claim that a degree of use predictor will be a key component of many communication optimizations. A common relationship between value locality and value prediction and, respectively, degree of use and degree of use prediction is the following: when instances of a particular static instruction generate values with many different degree of use, predicting the correct degree of use becomes harder. The main difference between value locality vs. degree of use consists in the fact that the first metric is focused on the past (the actual value belongs to the previously $k$ values) instead the second one is focused on the future (how many subsequent instructions will effectively use this produced value).

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

The majority of value prediction techniques focus on instruction level prediction. However, in [Mar99] the authors proposed a trace-based value prediction method that uses value prediction to predict the outputs for a trace based on its inputs, representing an attempt to obtain better speculative thread level parallelism. The trace is limited here to a loop trace. The speculative multithreading paradigm is based on the concurrent execution of control-speculative threads. These threads might be for example different iterations of an innermost loop. Data speculation is required to eliminate the serialization imposed by inter-thread dependences. The basic idea is to have a two level speculation approach: the conventional superscalar control speculation and respectively a prediction mechanism in order to predict some certain points in the control flow that are likely to be visited in the future. The speculation mechanism should consider the data dependences among concurrent threads (input/output data values, inter-thread memory dependences). Threads are likely to be dependent, thus the microarchitecture should provide mechanisms to ensure that all dependences among threads are detected. A simple approach is to force serialization between the producer and the consumer of every dependence. However, data value speculation can significantly improve the performance by predicting those values that flow through such dependences. If values are correctly predicted, dependent threads are executed as if they were independent. Speculative and non-speculative threads are executed concurrently since the CPU provides support for multiple hardware contexts. Also, the value reuse technique presented in [Hua98] exploits the natural boundary in programs and treats a basic block as a unit for reuse. In [Wu01] the authors proposed an integrated value prediction and computation reuse scheme that exploits value locality of programs in the region level. The compiler statically identifies general regions of code that are suitable for reuse or value prediction. At run-time, the hardware reuses or predicts the outputs of a region as a whole. Their proposed integrated scheme resembles the compiler – directed computation reuse scheme in the way it exploits computation reuse opportunity. But if the computation region cannot be reused, the scheme attempts to predict the results of the region and use the predicted values to speculatively advance the instruction flow. The obtained speedup, compared with a reuse or value
prediction approach is from 1.25 to 1.40 (on SPECint ‘95). Therefore, the integrated approach significantly outperforms either computational reuse or value prediction alone. In particular, the proposed integrated approach outperforms a computation reuse that has twice as much reuse buffer entries.

Rychlik et al. [Ryc98] combined a last, a stride, and a two-level value predictor to an overall hybrid value predictor. In order to efficiently utilize the hardware resources, they provided the dynamic classification scheme to dynamically distribute instructions into proper component predictors during run-time. Although this dynamic classification scheme uses the hardware resources more efficiently, it cannot provide higher prediction accuracy than the hybrid value predictor. Other researchers modified the dynamic classification scheme by reclassifying instructions after they cannot be predicted well by their previously assigned component predictor. Their modification improved this kind of hybrid value predictor.

As some researchers showed [Gab98], the value prediction potential is more significant if the instruction’s fetch rate is big. As an example, for an 8 instructions fetch rate the speedup is about 8% and for a 40 instructions fetch rate the speedup growths at 80%. As we already explained in the previous chapter, Trace Cache could be a feasible solution in order to assure a better fetch rate. Also simulations show that the number of simultaneously predicted branches also strongly influenced the value prediction’s performance. Anyway, until this moment (October 2006) no real microprocessors incorporate value prediction. Even if, together with dynamic instruction reuse, we believe they are powerful feasible techniques, only time will tell if their performance potential will compensate the additional complexity and power consumption required for effective implementation. Other important problems to be solved to have commercial implementations are: subtle memory ordering issues and misprediction recovery.

A particularly difficult challenge consists in target prediction for indirect jumps and calls. Polymorphism, indirect function calls and some case/switch structures are the main sources responsible for generating these indirect jumps and calls. Because the target of an indirect jump (call) can change with every dynamic instance of that jump, predicting the target of such an instruction is really difficult. Using conventional prediction mechanisms (BTB), the misprediction rates for indirect branches measured on some SPEC benchmarks were about 50% [Cha97]. The authors proposed a new prediction mechanism, named the Target Cache, for predicting the targets of indirect jumps. As the program executes, the target cache records
the target address for each indirect jump target encountered. Therefore, the prediction mechanism consists in selecting its prediction from all the targets of the indirect jump that have already been encountered. Just as an example, for Perl and Gcc belonging to SPECint95 benchmarks, the authors reported quite high misprediction rates varying from 30% to 48%. There are developed some prediction schemes for indirect jumps by constructing the index into the prediction table using the last $N$ target addresses, and using profiling information to select the proper value of $N$ for each branch. In this manner, good accurate jump target prediction is achieved. In fact there are developed some context predictors that are not derived from the general complete PPM prediction method, because in their schemes there are not encountered frequencies associated with the searched pattern. The indirect jumps prediction challenge is particularly gained through object-oriented programming techniques that exercise different aspects of computer architecture to support the object-oriented programming style. These languages tend to use indirect function calls, where the address of the call target is loaded from memory. Therefore, the necessity of understanding and accurate predicting these indirect jumps grow more and more.

6.4 THE VALUE PREDICTION SPEEDUP. A SIMPLE ANALYTICAL MODEL

In [Gab98] the authors proposed an analytical model dedicated for determining the speedup involved by a value prediction architecture comparing with a superscalar one. During the following presentation we determine the speedup based on this model. We have eliminated some omissions that the original model contains, hopefully making it more explicit and correct. Thus we consider an abstract machine having an unlimited number of general registers and execution units. Also we consider that the average prediction accuracy – and therefore speculative execution - for an instruction is $p$. For simplicity we consider that a correct predicted instruction is instantaneously executed, otherwise it is executed in time $T$. The model takes into account the critical path, and, it considers that the whole program belongs to processor’s instruction window. Taking into account that the processor has unlimited resources involves that there is possible a perfect renaming in order to avoid WAR and WAW dependences (anti- and output-dependences). As a consequence, we suppose that all the
instructions that aren’t belonging to the critical path will be processed simultaneously with the instructions belonging to that path. A graphical presentation of the proposed model is given in figure 6.18.

Figure 6.18. A Speculative Model of Instruction Processing Process

The figure’s 6.18 meaning is: if I_k instruction is correctly predicted (probability p) then its execution is practically instantaneous, otherwise (probability 1-p) it will be executed in time T. Therefore, each transition has assigned two components: a speculatively execution probability and, respectively, an execution time (0=speculative, T=non-speculative).

The total program’s execution time is:

$$ET(n) = T + T_{n-1},$$ where $T_{n-1}$ represents the execution time for the next (n-1) instructions following instruction I_1. Thus it represents the execution time for instructions I_1, I_2, …, I_n.

We can write:

$$\Pr(T_{n-1} = kT) = C_k^n (1-p)^k p^{n-1-k}, \forall k \in \{0,1,2,\ldots,n-1\}$$

Obviously there is fulfilled the normalisation equation:

$$\sum_{k=0}^{n-1} \Pr(T_{n-1} = kT) = 1$$

Therefore, the total execution time is:

$$ET(n) = T + T \sum_{k=1}^{n-1} k \Pr(T_{n-1} = kT) = T + T \sum_{k=1}^{n-1} k C_k^n (1-p)^k p^{n-1-k}$$

Considering the well-known equality:

$$C_n^k = \frac{n!}{(n-k)!k!}$$
It can be immediately be proved the identity:

\[ kC^k_{n-1} = (n-1)C^k_{n-2} \]

Taking into account this last identity and considering the last \( ET(n) \) expression, we can successively write:

\[ T_{n-1} = T \sum_{k=1}^{n-1} (n-1)C^k_{n-2}(1-p)^k p^{n-1-k} = T(n-1)(1-p) \sum_{k=1}^{n-1} C^k_{n-2}(1-p)^{k-1} p^{n-1-k} = \]

\[ = T(n-1)(1-p)(p + (1-p))^{n-2} = T(n-1)(1-p) \]

Therefore, the total execution time is:

\[ ET(n) = T + T(n-1)(1-p) \]

As a consequence the involved speedup \( S(n) \) vs. an equivalent processor without value prediction is:

\[ S(n) = \frac{nT}{ET(n)} = \frac{n}{1 + (n-1)(1-p)} \]

Taking into account that practically the number of dynamic executed instructions is infinite, the speedup is:

\[ S(\infty) = \lim_{n \to \infty} S(n) = \frac{1}{1 - p} \]

Of course, taking into account our multiple simplifications, in reality the speedup is less then the obtained \( S(\infty) \), but even so, the obtained result is quite suggestive.

We should develop now a speedup model, even in the more realistic case with a finite instruction window. Considering that \( w \) means instruction window’s capacity, the probability to correctly predict \( k \) successive instructions is:
Prediction Techniques in Advanced Computing Architectures

\[ \text{Prob}(L = k) = \begin{cases} (1 - p) p^{k-1}, & 1 \leq k \leq w - 1 \\ p^{w-1}, & k = w \end{cases} \]

\( L \) is a random variable that represents the average evicted instructions / cycle from the instruction window. This number is given by the RAW dependent instructions up to the first wrong predicted instruction. \( E(L) \) will be calculated like a weighted mean of the instructions evicted in each model’s state. The obtained value will be the speedup.

Taking into account that: \( \sum_{k=1}^{w} p(L = k) = 1 \) \( \Rightarrow \)

\[ E(L) = 1 \cdot P(L=1) + 2 \cdot P(L=2) + ... + (w-1) \cdot P(L=w-1) + w \cdot P(L=w) \]

\[ E(L) = \sum_{k=1}^{w} k \cdot p(L = k) \] \( \quad (6.0') \tag{6.0'} \]

\[ E(L) = \sum_{k=1}^{w-1} k \cdot p(L = k) + w \cdot p(L = w) = \sum_{k=1}^{w-1} k \cdot (1 - p) p^{k-1} + w \cdot p^{w-1} = (1 - p) \left( \sum_{k=1}^{w-1} k \cdot p^{k-1} \right) + w \cdot p^{w-1} \]

Because:

\[ \sum_{k=1}^{w-1} k \cdot p^{k-1} = \sum_{k=1}^{w-1} \left( \sum_{i=1}^{k} p^{i} \right) = \left( \sum_{i=1}^{w-1} p^{i} \right) = \left( p \cdot \frac{1 - p^{w-1}}{1 - p} \right) = \left( \frac{p - p^{w}}{1 - p} \right) = \frac{(1 - w \cdot p^{w-1})(1 - p) + (p - p^{w})}{(1 - p)^{2}} \]

\[ = \frac{1 - w \cdot p^{w-1} - p + w \cdot p^{w} + p - p^{w}}{(1 - p)^{2}} \]

It involves that:

\[ E(L) = \frac{1 - w \cdot p^{w-1} + p^{w}(w-1)}{(1 - p)} + w \cdot p^{w-1} = \frac{1 - w \cdot p^{w-1} + p^{w}(w-1) + w \cdot p^{w-1} - w \cdot p^{w}}{1 - p} \]

\[ = 1 - \frac{p^{w}}{1 - p} - \frac{p^{w}}{1 - p} = \frac{1}{1 - p} - \frac{p^{w}}{1 - p} \]

\[ \quad (6.2) \tag{6.2} \]

The obtained value is the real speedup. Obviously, for a finite instruction window as we considered, \( E(L) < S(\infty) \).

References


[Sam04] Sam N., Burtscher M. - Exploiting Type Information in Load-Value Predictors, The Second Value Prediction and Value-Based Optimization Workshop (in conjunction with ASPLOS 11 Conference), Boston, USA, October 2004


[Sil99] Silc J., Robic B., Ungerer T. - Processor Architecture, from Dataflow to Superscalar and Beyond, Springer-Verlag, 1999


Dynamic Instruction Value Prediction and Speculation


7. FOCALIZING DYNAMIC VALUE PREDICTION TO CPU’S REGISTERS

7.1. THE REGISTER VALUE PREDICTOR CONCEPT

The main aim of this chapter consists in focalizing dynamic value prediction to CPU’s context rather than on the current fetched/processed instruction, as we already proceed in the previous chapter. The idea of attaching a value predictor for each CPU register – we called it register centric prediction - instead of instruction or memory centric prediction was developed by us in [26, 28] and could involve new architectural techniques for improving performance and reducing the hardware complexity/cost of speculative microarchitectures. In some earlier works [25], [3], we 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 on these registers) leading to the conclusion that value prediction might be successful applied at least on these favorable registers.

Whether until our cited published works, the prediction process was only instruction or memory centered with great complexity and timing costs. By implementing the well-known value prediction schemes [11, 15] centered on CPU’s registers will be mainly decreased the hardware complexity, power consumptions (both static and dynamic) and costs. However, there are some disadvantages. Addressing the prediction tables with instructions’ destination register name (number) during the decode stage instead with the Program Counter (PC), will cause some interferences. However, as we’ll show further, with a sufficiently large history, a hybrid predictor could eliminate this problem and achieve very high prediction accuracy (85.44% at average on 8 MIPS registers using SPEC’95 benchmarks and respectively 73.52% on 16 MIPS registers using SPEC2000 benchmarks). The main benefit of the
proposed VP technique, from the pure performance point of view, consists in unlocking the subsequent dependent instructions, too.

As we already shown in the previous chapter, Lipasti and Shen [11] first introduced the concept of value locality as the third facet of the principle of locality (temporal and spatial). They defined the value locality as "the likelihood of the recurrence of a previously-seen value within a storage location" [11] in a processor. When the "storage location" is a single condition bit into a processor, it serves as the basis of branch prediction techniques. In an older work Tullsen and Seng [19] proposed a bit similar technique, however different by that presented in [26, 05], entitled “register-value prediction”, which identifies instructions that produce values that are already in the register file (especially in the instruction’s destination register). Therefore, the corresponding results are predicted using the current values belonging to the register file. About 75% of time the values loaded from the memory are already stored in CPU registers or they were stored recently. Mainly, this technique uses the previous value in the instruction’s destination register as a prediction for the new result, in a statically or dynamically manner. This technique produced speedups of up to 11% for the SPECint95 benchmarks and respectively up to 13% for SPECfp95 benchmarks. This method was further developed in [23] where the authors proposed to exploit the global correlation between the current instruction and, respectively, the previous N instructions, in order to predict its result using a simple perceptron. The perceptron’s input represents the global history of most recent committed instructions. A bit of 1 in the input vector indicates that the instruction produced the same value as that already stored in the destination (so called redundant instruction) and a –1 indicates otherwise. Each time an instruction is executed, the perceptron predicts if it is redundant or not. This prediction is compared with the real outcome and based on this feedback the perceptron is further trained. The authors proved that for a given size predictor, a perceptron performs better than a saturating counter based register value predictor. In contrast to our proposal, further summarized in this chapter, this approach is instruction-centric, like almost all the developed papers in this research area, instead of register-centric, as our proposals are. According to this, the authors pointed out that in their value prediction approach “confidence counters are associated with instructions rather than registers”. Additionally, they defined register value locality as the probability that the next value produced by an instruction to belong to the values already stored in the register file. In contrast, we define it as the probability that the next value associated with destination register \( R_{\text{dest}} \) to belong to the previous \( k \) values stored in that register. Therefore, our original register value prediction
technique consists in predicting register’s next value based on the previously seen values. In order to implement this strategy we attached a value predictor for all the CPU’s favorable registers (having a high value locality degree). After instruction’s decode the corresponding register value predictor is activated. Based on this approach, we developed in a systematical manner some context/incremental predictors. Gabbay and Mendelson [4] proposed a so-called register-file predictor that is the closest predecessor to our register value prediction technique. They predict the destination value of a given instruction according to the last previously seen value and the stride of its destination register, but the authors did not pursue further this particular idea by systematically developing new register-centric predictors and evaluating them through simulations.

7.2 REGISTER VALUE PREDICTORS

The main goal of this paragraph is to adapt the previously presented instruction-centric value predictors for registers’ values prediction. Whereas in the producer-centric prediction case the structures are addressed during the instructions’ fetch stage using PC, in the register-centric prediction case the tables are indexed only in the second part of instructions’ decode stage, using destination register’s number. Each instruction-value predictor presented in the previous chapter has its own corresponding predictor, attached to each CPU logical register. The “last value predictors” predict the new value as the same with the last value stored in the corresponding register. Exploiting the correlation between registers’ names and the values stored in those registers, will decrease the instructions’ latencies. Perfectly similar with the classical instruction value prediction method, each entry in the prediction table has its own automata, which is incremented when the prediction is correct and it is decremented otherwise. Based on the dynamic behavior of the register content there is developed the following classification: unpredictable and predictable registers contents. By treating separately each group of registers it can be avoided the miss-prediction’s costs. Obviously, it is necessary the verification of the value generated by the VHT. The automata’s state will be changed accordingly with the comparison between the generated value and the predicted value.
The VHT generates the predicted values and the “State” field could implement the histerezis mechanism using an automata. 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 with the number of logical registers.
When a register is used as destination for the first time, there is a miss in the prediction table and no prediction is done. When an instruction stores a value in the destination register that value is stored also in the VHT’s “Val” field and the automata will be in the initial unpredictable state. If that register is used again as a destination register, no prediction is made but it is calculated the stride \( \text{Str}_1 = V_1 - \text{Val} \), and \( V_1 \) and \( \text{Str}_1 \) are introduced in the “Val” and respectively “\( \text{Str}_1 \)” fields. If it’s used again as a destination register no prediction is done, but it is again calculated the stride \( \text{Str}_2 = V_2 - \text{Val} \). The value from the field “\( \text{Str}_1 \)” is introduced in the field “\( \text{Str}_2 \)”. Also \( V_2 \) and \( \text{Str}_2 \) are introduced in the fields “Val” and respectively “\( \text{Str}_1 \)”.

If the register is used again as an instruction’s destination and \( \text{Str}_1 = \text{Str}_2 \), it is calculated the predicted value adding the stride \( \text{Str}_2 \) to the value stored in the VHT (Val). If the automata are in the predictable state, the prediction is generated. The automata are incremented if the prediction is correct otherwise it is decremented.

In the case of context register predictors, the predicted value is the value that followed the current used context with the highest frequency. Therefore, the predicted value depends on the current context. A longer context frequently drives to higher prediction accuracy but sometimes it can behave as noise. As we already explained, a complete PPM predictor contains N simple Markov predictors, from \( 0^{\text{th}} \) order to \( (N-1)^{\text{th}} \) order. If the \( (N-1)^{\text{th}} \) Markov predictor produces a prediction (the context is matched in the sequence) the process is finished, otherwise the \( (N-2)^{\text{th}} \) order Markov predictor will be activated, and so on until order 0.

![Figure 7.3. The structure of a context based predictor](image-url)
In figure 7.3 is presented the main structure of the context-based predictor. Each entry from the VHT has an associated automaton that is incremented when the prediction is correct and 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.

Again, it naturally appears the idea of hybrid prediction; two or more value predictors dynamically working together during the prediction process. In the next figure (figure 7.4) it is presented a hybrid predictor composed by a context-based predictor and respectively a stride predictor. The context predictor has always the highest priority, predicting the first if its confidence state is higher than a certain threshold. The scheme is similar with the corresponding one presented in the previous chapter; the single difference is that here, VHT structure is indexed with destination register’s number instead of PC.

![Figure 7.4. The hybrid predictor (contextual & stride)](image-url)
7.3 SIMULATION METHODOLOGY AND EXPERIMENTAL RESULTS [26, 05]

In order to quantitatively investigate the previous proposed register centric schemes we developed a cycle-accurate execution driven simulator derived from the Sim-Outorder simulator in the SimpleScalar tool set [15]. The baseline superscalar processor supports out-of-order instruction issue and execution. We modified it to incorporate the registers’ values predictors previously described. Table 7.1 shows the configuration of the baseline processor used to obtain the results. To perform our evaluation, we collected results from different versions of SPEC benchmarks: 5 integer (li, go, perl, jpeg, compress) and 3 floating point (swim, hydro, wave5) SPEC’95 benchmarks. From the CINT SPEC2000 set, there were simulated 7 benchmarks (gzip, b2zip, parser, crafty, gcc, twolf and mcf). We simulated some SPEC’95 benchmarks too, in order to compare their behavior with that one involved by newer SPEC 2000. In other words, we intend to discover how these different benchmarks influence the value predictors’ micro-architectural features.

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

<table>
<thead>
<tr>
<th>Processor Core</th>
<th>Fetch / Decode / Issue Width</th>
<th>8 instruction / cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder Buffer Size</td>
<td>128 entries</td>
<td></td>
</tr>
<tr>
<td>Load-Store queue</td>
<td>64 entries</td>
<td></td>
</tr>
<tr>
<td>Integer ALUs</td>
<td>8 units, 1-cycle latency</td>
<td></td>
</tr>
<tr>
<td>Integer Multiply / Divide</td>
<td>4 units, 3 / 12-cycle latency</td>
<td></td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Hybrid branch predictor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gshare with 16K entries, 14 bit history, Bimodal with 16K entries.</td>
<td></td>
</tr>
<tr>
<td>Branch and Value miss-prediction</td>
<td>7-cycle latency</td>
<td></td>
</tr>
<tr>
<td>Memory Access</td>
<td>60-cycles latency</td>
<td></td>
</tr>
<tr>
<td>Memory Width</td>
<td>32 bytes</td>
<td></td>
</tr>
</tbody>
</table>
Caches

<table>
<thead>
<tr>
<th>Level-one data cache</th>
<th>4-way set associative, 64 KB, 1-cycle hit latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-one instruction cache</td>
<td>direct mapped, 128 KB, 1-cycle hit latency</td>
</tr>
<tr>
<td>Level-two cache (unified I&amp;D)</td>
<td>4-way set associative, 1024 KB, 10-cycle hit latency</td>
</tr>
</tbody>
</table>

Table 7.1. Machine configuration for the baseline architecture

Statistical results based on simulation proved that common-used programs are characterized by strong value repetitions [11, 17]. As we previously presented, 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 exhibits the number of time 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 destination field. In the next two figures we calculated the Value Locality metric using the following developed formula:

\[
VL_j(R_k) = \frac{\sum_{i=1}^{n} VL_j^k(i)}{\sum_{i=1}^{n} V Ref^k(i)}
\]  

(1)

where:

- \( n \) = number of benchmarks (8 for SPEC’95 respectively 7 for SPEC2000)
- \( j \) = history length (4, 8, 16 respectively 32)
- \( k \) = register’s number
- \( VL_j^k(i) \) = Number of time when register \( R_k \) is written with a value that was previously seen in last \( j \) values of the same register (on benchmark \( i \)).
- \( V Ref^k(i) \) = The total numbers of dynamic instructions that have register \( R_k \) as destination field (on benchmark \( i \)).

The next two figures give emphasis to the concept of value locality on registers. As it can be observed (see figures 7.5a, 7.5b), the value locality on some registers is remarkable high (90%), and this predictability naturally leads us to the idea of value prediction implemented at least on these favorable registers. Taking into account the simulated MIPS CPU architecture, some possible qualitative explanations are [26, 27]:

202  Prediction Techniques in Advanced Computing Architectures
• $at (1) register is frequently used in data memory address in order to access some data structures and rarely generate more than one unique value. $v0 keeps few OS system call codes.
• $sp (29) register is the MIPS stack pointer and, thus, it has little variations.
• $fp (30) is the frame pointer register.
• $ra (31) stores the return address for a jal (jump and link) instruction. There are few calling points in SPEC benchmarks. The greater locality exhibited by this register is due to fewer procedure calls within a program.

Figure 7.5a. Value locality on registers (SPEC'95 simulation results)

Figure 7.5b. Value locality on registers (SPEC2000 simulation results)
The 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, like it is shown in figure 7.6. The Value Prediction technique predicts next register’s value based on the last values belonging to that register; it executes the operations using the predicted value and the speculative execution will be validated when the correct value is known, after execution stages. If the value was correctly predicted the critical path might be reduced. Otherwise, as we explained in the previous chapter, the instructions executed with wrong entries must be executed again (recovery process).

In [28], [5] were developed and simulated several different basic value predictors, such as the last value predictor, the stride value predictor, the context-based predictor and a hybrid value predictor in order to capture certain type of value predictabilities from SPEC benchmarks and to obtain higher prediction accuracy. All these predictors were easily adapted to our specific proposed prediction model. After the instruction’s decode stage, the VPT table is accessed with the name of the destination register. In the case of a valid prediction, the VPT will generate 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. In the case of a misprediction, the speculatively executed dependent instructions are re-issued again for execution. Starting with a minimal superscalar architecture, we studied how will be affected the simulator’s performance by the variation of its parameters. In this paragraph we present the results obtained with a register value predictor. Each register has associated a 4-state confidence automata. A prediction is done only if the automata is in one of the two predictable states. In figures 7.7a respectively 7.7b, each bar represents the average of register value prediction accuracy obtained for 8 of SPEC’95 benchmarks respectively for 7 integer SPEC2000 benchmarks.
In the previous two figures it was calculated the Prediction Accuracy (PA) metric using the following the formula:

\[
PA(R_k) = \frac{\sum_{i=1}^{n} CPV^k(i)}{\sum_{i=1}^{n} \text{Ref}^k(i)}
\]  

(2)

n = number of benchmarks (8 for SPEC’95 respectively 7 for SPEC2000)

k = register number

\( CPV^k(i) \) = Number of correctly predicted values of register \( R_k \) (on benchmark \( i \)).
V_{Ref}^k (i) = The total numbers of dynamic instructions that have register R_k as destination (on benchmark i).

In the next investigations, we are focusing only on the predictable registers, having prediction accuracy higher than a certain threshold (60% respectively 80%), measured using the hybrid predictor on the SPEC benchmarks. As it can be seen in Figures 7.7a and b, the registers having a prediction accuracy higher than 60% are: 1, 5, 7÷13, 15, 18÷20, 22, 29÷31 for SPEC’95, and respectively, 1, 6÷8, 10÷16, 18÷25, 29÷31 for SPEC2000. The statistic results on SPEC’95 benchmarks exhibit a using degree of 19.36% concerned to these 17 registers. This means that 19.36% of instructions use as a destination one of these registers. The equivalent average result on SPEC2000 is 13.24% using 22 general purposes registers. In Figures 7.8a and 7.8b are compared the previously presented value prediction techniques: last value prediction, stride prediction, context based prediction and hybrid prediction. In the prediction process are used only the 17 favorable registers on SPEC’95 benchmarks and respectively 22 favorable registers on SPEC2000 benchmarks. The context based and the hybrid predictors use a history of 256 values and a search pattern of 4 values.

![Figure 7.8a. Prediction accuracy using 17 favorable registers (SPEC’95 simulation results)](image-url)
These results represent the global prediction accuracies of the favorable registers for each benchmark. It can be observed the hybrid predictor synergism, involving an average prediction accuracy of 78.25% on SPEC’95 benchmarks and respectively 72.93% on SPEC2000 benchmarks. Now, there are considered only the registers with prediction accuracy higher than 80% (see Figures 7.9a and 7.9b). The selection is based again on Figures 7.7a and 7.7b. We can observe that there are 8 registers that fulfill this condition: 1, 10÷12, 18, 29÷31 on SPEC’95 benchmarks and respectively 16 registers: 1, 8, 11÷15, 20÷25, 29÷31 on SPEC2000 (registers 1, 29÷31 are included even if they don’t fulfill this condition because they exhibit high degree of value locality; also they have special functions). The global using rate of these registers is 10.58% on SPEC’95 benchmarks, and respectively 9.01% on SPEC2000.
Figures 7.9 emphasizes for each benchmark the global prediction accuracy obtained with the implemented predictors using 8 respectively 16 selected registers (threshold over 80%, according to the previous explanations). In these figures, each bar represents the register value prediction for a certain benchmark, measured by counting the number of times when prediction is accurate for any of the favorable registers and dividing to the total number...
when these registers are written. The simulation results offered by last value predictor are relatively close to the stride predictor’s results. The best average prediction accuracy of 85.44% was obtained with the hybrid predictor, quite remarkable (on some benchmarks with values over 96%). In the next two figures (10a and b) we show the speedup obtained compared to the baseline processor when using each of the registers’ values predictors described earlier.

**Figure 7.10a.** Speedup over baseline machine using 8 favorable registers (SPEC ‘95)

**Figure 7.10b.** Speedup over baseline machine using 16 favorable registers (SPEC 2000)
7.4 REGISTER VALUE PREDICTION USING METAPREDICTORS [26]

As we already mentioned, a hybrid instruction/register value predictor combines two or more component predictors that each predicts the value of the current instruction destination. The hybrid predictor therefore needs a selection mechanism to predict which of the predictors is likely to be the most accurate at a certain moment. This “prediction of the best prediction” structure is called in literature metaprediction (dynamic predictor’s selection).

In this paragraph we used the following component predictors, all of them centered on CPU’s registers instead on program’s instructions: a last value predictor, a stride predictor and a context-based predictor. Every component predictor provides two values: the predicted value and its confidence. A confidence mechanism performs speculation control by limiting the prediction to those that are likely to be correct. A high confidence represents a lot of correct predictions in a given history of that register. The metapredictor represents a second prediction level and it dynamically selects, based on the predictors last behaviors (confidence), one of the component predictors (last value, stride or contextual), in order to predict the next value for a certain destination register. The architecture used for metaprediction process is presented in Figure 7.11.

![Figure 7.11. The hybrid-predictor’s architecture](image-url)
**Heuristic Non-Adaptive Metapredictors**

The first non-adaptive metapredictor (we called it “heuristic” even if it is not driven by a true heuristic algorithm) selects one of the component predictors based on their last $k$ behaviors (correct/incorrect prediction). A logical one (‘1’) means a correct prediction and a logical zero (‘0’) means a miss-prediction. These behaviors are stored in a logic left shift register, associated with each predictor, like in the next figure.

![Logic left shift register](image)

Figure 7.12. Logic left shift register associated with each predictor

The predictor with the highest value of its corresponding shift register has priority. As it can be observed, the prioritization rule is pure static and, thus, non-adaptive. Therefore, in this case the metapredictor doesn’t learn from its past behavior because it doesn’t update its state after the real result is known, toward a better further prediction. The predicted value is used only if the selected predictor encountered a number of favorable behaviors higher than a certain threshold. In other words, a prediction is generated only in the case of an acceptable confidence degree.

**Adaptive Metapredictors With Automata**

The second developed metapredictor is an adaptive one, using three 4-states confidence automata saturating counters: one for the context-based predictor, one for the stride predictor and another one for the last value predictor. The automata initially are in the unpredictable state and they are adjusted after each prediction, when the real values are known. Therefore the automata are learning from the real behavior. The predictor with the highest predictable state has priority and a prediction is generated only in the corresponding two predictable states. The structure of the 4-states automata is presented in the next figure. We explained the difference between the first metapredictor and respectively the second one, considering the following example: if the component predictor 1 has the attached behavior $H_1=111110000$ (1=correct prediction) and the component predictor 2 has the history $H_2=000011111$, the “heuristic” metapredictor will chose predictor 1 (because $H_1>H_2$) while the
automata predictor will select predictor 2 (it is in the strongest predictable state, while predictor 1 is its strongest unpredictable state).

Figure 7.13. The structure of the 4-state automata

Neural Adaptive Metapredictors

Another idea consists in implementing a dynamic metapredictor represented by a simple neural network (NN) in order to select the best predictor at a certain moment [24]. It is well known that a great advantage of the artificial neural networks consists in their capacity to learn based on examples (supervised learning). The network extracts the information from the training samples. In this way it is able to synthesize implicitly a certain model of the problem. In other words, the neural network builds up alone an adaptive algorithm to solve the problem. We firstly implemented as a metapredictor a multi-layer perceptron with one hidden layer and back-propagation learning algorithm, just for understanding the idea’s potential performance (at this moment we don’t care if the idea is feasible to be hardware implemented or not). The input vector consists of one \( k \)-bit confidence sequence for each predictor (\( N=3k \)), representing the last \( k \) binary codified behaviors: 1 for a correct prediction and 0 (or better, -1) for a misprediction. The neural network returns through its output layer the selected predictor, using one neuron for each predictor (\( P=3 \)). It is selected the predictor corresponding to the neuron with the highest output value, but a prediction is generated only if this value is greater than a certain threshold. We proved based on experiments that the optimal number of hidden layer neurons is \( N+1 \), \( N \) being the number of input layer neurons. In Figure 7.14 is presented the structure of the adaptive neural metapredictor with one hidden layer.
For the training/learning process we used the well-known back-propagation algorithm [22], adapted as below:

1. Create a feed-forward network with $N$ inputs, $M = N + 1$ hidden units and $P$ output units.

2. Initialize all network weights $W_{i,j}^{1}$; $i = 1, N$; $j = 1, M$ and $W_{i,j}^{2}$; $i = 1, M$; $j = 1, P$, to small random numbers belonging to the $[0.3, 0.7]$ interval.

In the following steps $t_k$ represents the value of $k$’s neuron from the output layer and $O_k$ is the desired value of the same neuron.

3. Until $E(W) = \frac{1}{2} \sum_{k \in \text{Outputs (P)}} (t_k - O_k)^2 \leq T$ (threshold), do:

   3.1. Input the instance $\vec{x}$ to the network and compute the output $\vec{O}$ (matrix product).
   
   $$\vec{O} = \vec{x} \cdot W^1 \cdot W^2$$  \hspace{1cm} (1)

   3.2. For each network output unit $k$, $k = 1, P$, calculate its error term $\delta_k$.
   
   $$\delta_k = O_k (1 - O_k) (t_k - O_k)$$  \hspace{1cm} (2)

   3.3. For each hidden unit $h$, $h = 1, M$, calculate its error term $\delta_h$.
   
   $$\delta_h = O_h (1 - O_h) \sum_{k \in \text{Outputs (P)}} W_{k,h}^2 \cdot \delta_k$$  \hspace{1cm} (3)

   3.4. Update each network weight $W_{i,j}$.
\[ W_{i,j} = W_{i,j} + \Delta W_{i,j} \]  \hspace{1cm} (4) \\
\[ \Delta W_{i,j} = \alpha \cdot \delta_i \cdot X_{i,j} \]  \hspace{1cm} (5)

Where \( \alpha \) is the learning step.

We used the well-known sigmoid activation function:

\[ F(x) = \frac{1}{1+e^{-x}} \]  \hspace{1cm} (6)

### 7.5 Metapredictors’ Simulations and Quantitative Evaluations

For researching these metaprediction ideas we developed an execution driven simulator derived from Simple Scalar tool set. It was simulated the execution of 5 million dynamic instructions from different SPEC’95 benchmarks. We developed and evaluated several different already presented metapredictors in order to capture certain type of value predictabilities from SPEC’95 benchmarks and to obtain higher prediction accuracy. As we previously mentioned, the prediction accuracy represents the number of correctly predicted registers divided to the total number of dynamic instructions having these registers as destinations. The confidence metric further introduced represents the number of correctly predicted values related to a certain register \( R_k \) when the attached predictor was into a predictable state, divided to the total number of predictable states associated to that register \( R_k \).

Starting with a minimal superscalar architecture, we studied how will be affected the simulator’s performance by the variation of its parameters. We began evaluating the “heuristic” non-adaptive metapredictor, which selects one of the predictors based on their last \( k \) behaviors stored in a logic left shift register associated with each predictor. The predictor with the highest value has priority. The predicted value is used only if the selected predictor encountered a number of favorable behaviors at least equal with a certain threshold. The next two figures show how the threshold’s value affects the prediction accuracy and respectively the confidence, using a “heuristic” metapredictor with \( k=3 \).
We continued our study evaluating the adaptive metapredictor with confidence automata. Each predictor has associated a 4-state automata and the predictor with the highest confidence has priority. A prediction is generated only in the predictable strong states. Figures 7.17 and 7.18 show how the threshold’s value affects the prediction accuracy and respectively the confidence of these predictors.
We continued our evaluations with the adaptive neural metapredictor, with an input vector consisting of one $k$-bit sequence for each predictor ($N=3k$), representing the last $k$ binary codified behaviors: 1 for a correct prediction and 0 (in fact –1) for a misprediction. The neural network returns through its output layer the selected predictor, using one neuron for each predictor ($P=3$). It is selected the predictor corresponding to the neuron with the highest output value, but a prediction is generated only if this value is greater than a certain
Focalizing Dynamic Value Prediction to CPU’s Registers

threshold. Figures 7.19 and 7.20 show how the threshold’s value affects the prediction accuracy and respectively the confidence of the neural network.

**Figure 7.19.** The neural metapredictor’s prediction accuracy for different thresholds (k=3)

**Figure 7.20.** The neural metapredictor’s confidence for different thresholds (k=3)
It can be observed that each metapredictor, for a high threshold value, has less global prediction accuracy but a higher confidence. Practically, the probability that local prediction generated by high confidence states to be correct, significantly increases through reducing the cases when the structure makes a prediction. The disadvantage is that the percentage of cases in which is made a prediction dramatically decreases. The next parameter we varied is the history length \((k)\). Fig. 7.21 shows how is affected the prediction accuracy and respectively the confidence, both generated by a neural network predictor, by the behavior history length, using a threshold of 0.1. It can be observed that the optimal value of \(k\) is 3 (prediction accuracy of 84.78%).

![Figure 7.21. Study of prediction accuracy and confidence varying the behavior history length for a threshold of 0.1](image)

Taking into account all the previous experimental results, the next table exhibits an interesting conclusion: in spite of pretty small global prediction accuracy (72.45% in average on all favorable registers) there are some registers (R1, R22, R29 and R31) with prediction accuracies greater than 90%. The accuracy gain obtained on these registers using a “heuristic” metapredictor versus our previous described hybrid predictor – where the context predictor has always the maximum priority - is 2.27%, that is quite good.
Focalizing Dynamic Value Prediction to CPU’s Registers

Register Value Prediction Accuracy

<table>
<thead>
<tr>
<th>Register number</th>
<th>Metapredictor with automata (threshold=1 state)</th>
<th>The heuristic metaprediction: (threshold = 1)</th>
<th>The neural metapredictor (threshold=0)</th>
<th>Hybrid predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>89.034</td>
<td>89.028</td>
<td>88.999</td>
<td>86.81</td>
</tr>
<tr>
<td>22</td>
<td>85.477</td>
<td>85.647</td>
<td>84.146</td>
<td>81.43</td>
</tr>
<tr>
<td>29</td>
<td>95.289</td>
<td>95.287</td>
<td>95.284</td>
<td>94.74</td>
</tr>
<tr>
<td>31</td>
<td>95.642</td>
<td>95.643</td>
<td>95.599</td>
<td>94.48</td>
</tr>
<tr>
<td>Average</td>
<td>91.3605</td>
<td>91.40125</td>
<td>91.007</td>
<td>89.36</td>
</tr>
</tbody>
</table>

Accuracy gain obtained vis a vis of hybrid predictor [%] 2.23 2.27 1.83

Table 7.2. The prediction accuracy obtained on the best four favorable registers

7.6 CONCLUSIONS AND SOME FURTHER WORK IDEAS

In this chapter, we have introduced and studied the register value prediction concept, first introduced by our research group in [25, 26, 27, 28]. As we discussed, the intention of the register value prediction concept is to reduce the unfavorable effect of the RAW dependencies, reducing the wait times of the subsequent dependent instructions and the complexity/costs. Therefore, the prediction focused on CPU registers instead on program’s instructions, is advantageous because there are needed much fewer predictors, saving complexity, power consumption and costs. We demonstrated that there is a dynamic correlation between the names of the destination registers and the values stored in these registers. The simulations show that the hybrid predictor exploits with the best results this correlation and the value locality concept. We proposed to exploit the value locality on registers, using different proposed prediction techniques. For the dynamic selection of the favorable registers, we used a hybrid predictor. We continued after that with the evaluation of the predictors, using the registers with prediction accuracy higher than 60%. The best results were obtained with the hybrid predictor: an average prediction accuracy of 78.25% and a using rate of 19.36%. We tried after that a more elitist selection of the registers and we continued the evaluation of the predictors using only the registers with prediction accuracy...
higher than 80%. The best results were obtained again with the hybrid predictor: an average prediction accuracy of 85.44% (on some benchmarks with values over 96%) and a using rate of 10.58%. Also, considering an 8-issue out-of-order superscalar processor, simulations show that register centric value prediction produce average speedups of 17.30% for the SPECint95 benchmarks, respectively of 13.58% for the SPECint2000 benchmarks.

Further we proposed to improve the register value prediction’s accuracy by metaprediction. We introduced several different metapredictors (dynamic selectors), in order to properly select the current best predictor. One of the developed adaptive metapredictors uses three 4-state confidence automata: one for the context-based predictor, one for the stride predictor and another one for the last value predictor. The predictor with the highest history value has priority and a prediction is generated only in the predictable strong states. Using this hybrid predictor we obtained an average global prediction accuracy of 69.81% with a confidence of 94.7%. Predicting only in the strongest state of the automata we obtained an average global prediction accuracy of 68.1% with a confidence of almost 100%. The second metapredictor selects one of the predictors based on their last k behaviors (correct/incorrect prediction), in a static, non-adaptive, manner. These behaviors are stored in a logic left shift register associated with each predictor. The predictor with the highest confidence state has priority. As an example, with a selection based on the last three behaviors of each predictor (k=3), and generating predictions only if the number of favorable predictions is at least 2 (threshold=2), we obtained a global prediction accuracy of 68.93% with a confidence of 94.57%. For a threshold of 1 (predict only in the strongest state) the global prediction accuracy is 65.09% with a confidence of 100%.

Another developed idea consisted in implementing a dynamic adaptive metaprediction structure represented by a simple neural network (NN), in order to select the best predictor at a certain moment. Concretely, we implemented as metapredictor a multi-layer perceptron with one hidden layer and back-propagation learning algorithm. The input vector consists of one k-bit sequence for each predictor, representing the last k binary codified behaviors: 1 for a correct prediction and 0 for misprediction. The neural network returns through its output layer the selected predictor, using one neuron for each predictor. It is selected the predictor corresponding to the neuron with the highest output value, but a prediction is generated only if this value is greater than a certain threshold. With a neural selection, based on the
last three behaviors \((k=3)\) and a threshold value of 0.1, we obtained an average global prediction accuracy of \(71.58\%\) with a confidence of \(84.77\%\). For a threshold of 0.9 the global prediction accuracy is 59.01\% with a confidence of 95.85\%. Judging from the confidence point of view it can be observed that the adaptive metapredictor with automata and prediction in the strongest state is the optimal. Using this metapredictor we obtained a global prediction accuracy of 68.11\% with a confidence of almost 100\%. A further challenge will be to implement in hardware an efficient neural dynamic metapredictor (might be a simple perceptron predictor?).

Generally speaking, a further challenge in our Value Prediction Research would be how to integrate all these developed structures and algorithms into an aggressive out-of-order superscalar architecture, feasible to be implemented in hardware by the next generation microprocessors, like SimpleScalar Out-Order simulator or Decoupled Kilo-Instruction-Processor (DKIP) simulator developed at University Politecnica Catalunya of Barcelona and offered us (ACAPS Research Center) for doing research. At this moment (2006) our research group understood both these complex simulation frames. Also it is essential to find new solutions to the well-known Reorder Buffer (ROB) bottleneck, as a centralized structure into a superscalar environment. At this moment ROB do not scale to large instruction windows because of timing and power constraints. Also ROB enlarges the memory-wall limitation, consisting in the gap between the processor and main memory (memory latency have improved at a much slower pace – 7\% per year than microprocessor speed - 60\% per year). In this sense we’ll develop some new aggressive instruction issue algorithms based on some new works like [29, etc.] (Increasing the number of in-flight instructions, CPU state Checkpointing, early release of resources, better locality exploitation, more intelligent management of the instructions queues), etc. Another idea is to exploit cross-fertilisation between speculative processing (Value Prediction) and Dynamic Instruction Reuse. As benchmarks we’ll use Standard Performance Evaluation Corporation SPEC2000 suite benchmarks (we already have a license) and also MediaBench suite. Hopefully we’ll use also the newest SPEC 2006 benchmarks. It is also possible for us to use Branch Prediction World Championship benchmarks proposed by Intel http://www.microarch.org/micro39/. An important goal will be to use the new Embedded Microprocessor Benchmark Consortium (EEMBC) for understanding processor behavior in embedded applications in industry like multimedia, real-time automotive, network processors, telecom, signal processing, consumer devices, office automation, VoIP applications, multicore platforms, etc. Using EEMBC complex benchmarks we’ll better
understand embedded applications characteristics and we should synchronize with top computer architecture researches like those from FP6 HiPEAC (High Performance Embedded Architectures and Compilers – see www.hipeac.net) or FP6 ARTEMIS platform, etc. Finally another important goal will be to evaluate all these new developed microarchitectures from the hardware complexity and energy consumption point of view using dedicated CAD tools. As far as we know until this moment, there aren’t researches in neural branch prediction power consumption. Also there are few approaches in estimating power consumption for speculative microarchitectures like those briefly presented above. Obviously it is essential to develop some low power algorithms and structures in order to be implementable in an efficient embedded frame.

References

5. 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. Vințan, PhD).
17. SPEC *The SPEC benchmark programs*, http://www.spec.org


29. Cristal A. et al., *A Decoupled KILO-Instruction Processor*, High Performance Embedded Architectures and Compilers Conference (HiPEAC), Barcelona, November 2005
8. NEURAL NETWORKS MODELS WITH APPLICATIONS IN UBIQUITOUS COMPUTING.
NEXT LOCATION PREDICTION

8.1. INTRODUCTION TO UBICOM

An important measure of progress in computing involves rendering it as an inseparable part of our everyday life while simultaneously making it disappear. Ubiquitous Computing (abbreviated UbiCom – first proposed by Mark Weiser in his seminal paper from 1991) represents an emerging technology, in which computers will be embedded in our natural interactions with the environment. Its growing evolution has recently been accelerated by improved wireless communications capabilities, open networks, continued increases in computing power, development of embedded systems, improved battery technology and low power consumption, and the emergence of flexible open software architectures.

The movement into the UbiCom area integrates the advances from both mobile and pervasive computing. Mobile computing is fundamentally about increasing our capability to physically move computing services with us, benefiting from important wireless communication progresses. As a consequence, the computer becomes an ever-present device that expands our capabilities to inscribe, remember, communicate, and reason independently of the device’s location. On the other hand, the pervasive computing concept (context-aware computing) implies the computer has the capability to obtain relevant information from the environment in which it is embedded and utilize it to dynamically develop models. A (mobile) computer is aware of its user’s state and surroundings (spatial and temporal awareness), and modifies its behavior based on this information. The process is reciprocal: the environment can and should also become “intelligent” in that it also has a capability to detect other computing devices entering it or modifying their states. This mutual dependency and interaction results in a new capacity of computers to act “intelligently” upon and within the environments in which
the people move. Pervasive Computing involves an environment populated with sensors, pads, badges, and virtual or physical models of the physical and social/cognitive environments. Context-aware agents can deliver relevant information to the user when it is needed. Three examples of first-generation context-aware agents might include:

- The Notification Agent alerts users if they are passing within a certain spatial distance to a task on their to-do list. For example, if a user is walking near his/her computer, the agent alerts the user if he/she have an (important) e-mail message to retrieve/send.
- The Meeting Reminder Agent alerts the user if they are moving away from the site of a meeting whose location has been changed.
- The Activity Recommendation Agent recommends possible activities/meetings a user might like to attend based on their interests.

Currently, the main challenge of pervasive computing is the limited scope and large effort involved to teach a computer about its environment. UbiComs are characterized by a high level of embeddness (any place), context awareness and mobility (any time). Their main goal should be to provide “the right information to the right person at the right place at the right time”. A paradoxical outcome of UbiCom is that it is simultaneously very personal and, on the other hand, extremely global. In its ultimate form, UbiCom means any computing device, while moving with us, can build incrementally dynamic models of its various environments and self-configure its services accordingly. Furthermore, the devices will be able to either “remember” past environments they operated in, thus helping us to work when we reenter, or proactively build up services in new environments whenever we enter them, based on some sophisticated prediction methods. UbiCom applications, by virtue of being available everywhere at all times, will have to adapt to the dynamism of users’ environments and the resulting uncertainties. In these environments, users may change their goals or adapt their actions to a changing environment and therefore UbiComs have to adapt to these facts, too.

8.2 A RELATED CONCEPT: AUTONOMIC COMPUTING

Autonomic Computing (AC) or Self Governing Computers, concept introduced by IBM in 2001, has little to do with the usual obstacles that
preoccupy computer engineers (Moore’s Law, Machine Intelligence, hardware-software interface, etc.) but is closely related with UbiCom paradigm. It is mainly focused on the complexity obstacle. Dealing with it is one of the most important challenges facing present day IT industry. Through AC we might free our minds and resources to concentrate on previously unattainable tasks [Hor01].

Hardware Computer Complexity: microprocessor power up by a factor of about 10000, storage capacity by a factor of 45000, communication speeds by a factor of 1000000 – but at a price: complexity growth. As an example, the last Intel Pentium Dual Core microprocessor contained about 1.3 billion transistors, an enormous number comparing with about 40,000 transistors an eight-bit microprocessor like Z80, in the end of ’70s.

Software Computer Complexity: There are increasingly sophisticated architectures governed by software whose complexity now routinely demands tens of millions lines of code. Some operating environments contain over 30 million lines of code created by over 4000 programmers. Similar in complexity are the actual compiler platforms. Actual microprocessors incorporate about 1 billion of transistors (an instruction would be equivalent with a transistor?). The Internet adds yet another layer of complexity by allowing us to connect this world of computing systems with telecommunication networks.

According to IBM’s Vice-President, Paul Horn (2001), at current rates of expansion, there will not be enough skilled IT people to keep the world’s computing systems running. Even in uncertain economic times, demand for skilled IT workers is expected to increase by over 100% in the next years. Even if we could somehow come up with enough skilled people, the complexity is for sure growing beyond human ability to manage it! Paradoxically, to solve the problem – make things simpler for developers, administrators and users of IT – we need to create more complex systems. How will this possibly help? By embedding and hiding the complexity in the system architecture itself then automating its management, answered Horn. For this approach IBM found inspiration in the massively complex systems of the human body. One of the key systems is the autonomic nervous system that does a lot of actions like heart beat control, checking blood sugar and oxygen level, control pupils, etc., without any conscious recognition or effort. That is probably what we need to build new generation of computing systems. Thus, according to IBM, it’s time to design and build computing systems
capable of running themselves, adjusting to varying contexts, and preparing their resources to handle most efficiently the workload we put upon them. These AC systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there. In IBM’s opinion the term AC is emblematic of a vast hierarchy of natural self-governing systems, many of which consists of billions of interacting, self-governing components that in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down.

But focusing on automating the piece parts of computing systems will not be enough. It is the self-governing operation of the entire system, and not just parts of it, that delivers the ultimate benefit (for example, an increase in heart rate without a corresponding adjustment to blood pressure would be dangerous). It is needed therefore a global optimisation method. AC represents thus a holistic vision that will enable the whole of computing to deliver much more automation than the sum of its individually self-managed parts, involving thus synergism.

Autonomy, pro-activity, and goal-directed interactivity with their environment are essential characteristics of software agents. Viewing autonomic elements as agents and autonomic systems as multi-agent systems built on a web services infrastructure makes it clear that agent-oriented architectural concepts and technology will be critically important. According to IBM, such a high – level systems could have at least eight key elements named “self-x” elements [Hor01]:

1. To know itself (context-awareness) and comprise components that also posses a system identity. It needs detailed knowledge of its components, current status, ultimate capacity, and all connections with other systems in order to manage itself. To build this ability into computing systems, clearly defined policies embodied in adaptable software agents will have to govern a system’s definition of itself and its interaction with IT systems around it. Context information is distributed with redundancy over the different units of the system. The information of all units together builds up the complete context. If one unit fails and there is another unit with the same information (tolerance failure), the system will continue working without any notice to the end-user [Tru03].
2. Configure and reconfigure itself (self-configuration) under varying and unpredictable conditions, following powerful adaptive algorithms. System configuration or set-up must occur automatically, as must dynamic
adjustments to that configuration to best handle changing environments. A system may need for example to create multiple images of critical software (a kind of software cloning), and reallocate its resources as needed (memory, communication bandwidth, processing elements). In other words, when a component is introduced in an AC system, it will incorporate itself and the system will adapt to its presence.

3. Always looking for ways to self-optimise its processing. Components and systems continuously seek opportunities to improve their own performance and efficiency. To be able to optimise itself, a system will need advanced feedback control mechanisms to monitor its performance and take appropriate action. We’ll need to answer questions such as how often a system takes control actions, how much delay it can accept between an action and its effect (real-time constraints), and how all these affect overall system’s stability. Another requirement is that the application must be able to start with predefined parameters or even without any initial information. This kind of self-configuration is based on the assumption that the configuration can be completed based on information extracted from application messages or from specialized configuration messages.

4. Self-healing means calling into actions redundant or under-utilised elements to act as replacement parts. Thus, system automatically detects, diagnoses, and repairs localized software and hardware failures. It must be able to recover from routine and extraordinary (exceptions) events that might cause malfunctions. Initially healing rules will follow rules generated by a human expert (supervised learning) but further the system will begin to discover new rules on its own (unsupervised machine learning). Particularly, at the level of individual components, well-established techniques – many of which fall under the domain of fault tolerance – have led to the development of rarely fail, which is one important aspect of being autonomic. An AC system should continue working by compensation of failing components similar to a human body that adapts to the new circumstances during the healing process. Unlike the human body an AC system needs also self – healing if the system structure changes – not only missing parts, but also newly added parts must be recognized and smoothly integrated into the system. According to [Tru03] in a deeper vision self – healing can be extended toward self-optimisation.

5. Self-protection is required in order to maintain overall system’s security and integrity. System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system wide failures. By simulating the human immune system, a digital immune
system can detect suspicious code (worms, viruses, etc.) without the user being aware such protection is in process.

6. **Adapting** meaning that it knows its environment and the context surrounding its activity, and acts accordingly. This is almost self-optimisation turned outward: an AC system will find and generate rules for how optimal to interact with neighbouring systems. This context sensitivity includes improving service based on knowledge about the context of a transaction. AC systems will need to be able to describe themselves and their available resources to other systems, and they will also need to be able to automatically discover other devices in the working environment.

7. An AC system must function in a **heterogeneous world** and implement open standards (an autonomic computing system cannot, by definition, be a proprietary solution). Open standard ways of system identification, communication and negotiation need to be further invented and agreed on.

8. AC system will **anticipate (predict)** the optimised resources needed while keeping its complexity hidden for the administrator or users. Anticipation is the capability to have a projection of user needs and actions into the future. To context anticipate the future behaviour of a person, a system must analyse the past. This might involve some proactively performed actions [Tru03, Tru03b]. We’ll need to make progress along the following tracks:

- Making individual system components autonomic, and achieving autonomic behaviour at the level of global enterprises IT systems.
- How to create this global system awareness and management? How do we globally optimise the entire stack of computing layers as a whole?
- How to design an architecture for autonomic systems that provides consistent interfaces and points of control while allowing for a heterogeneous environment?

Some Engineering Challenges for achieving AC requirements, according to [Kep03], are the followings:

- Design, test, and verification. Extending Web services with programming tools and techniques that aid in managing relationships with other autonomic elements. Programmers will need tools that help them acquire and represent policies and map them onto lower level actions. Testing is very difficult; it will be virtually impossible to build test systems that capture the size and complexity of realistic systems and workloads.
- Installation and configuration. Will most likely entail a bootstrapping process that begins when the element registers itself in a directory service
by publishing its capabilities and contact information. The element might also use the directory service to discover suppliers that may provide information or services it needs to complete its initial configuration.

- Monitoring and problem detection. Elements will permanently monitor themselves to ensure that they are meeting their own objectives, and they will log this information to serve as the basis for adaptation, self-optimisation, and reconfiguration. An autonomic element that detects poor performance or failure in a supplier may not attempt a diagnosis; it may simply work around the problem by finding a new supplier (through system’s redundancy).

- Upgrading. The AC elements might subscribe to a service that alerts them to the availability of relevant upgrades and decide for themselves when to apply the upgrade, possibly with guidance from another element or even a human.

According to [Kep03] the relationship between elements, are the followings:

a. Specification – an autonomic element must have associated with it a set of output service it can perform and a set of input services that it requires, expressed in a standard format so that other autonomic element can understand it. (Are here formal domain ontologies required?)

b. Location – to locate other elements dynamically, the element can look them up by name or function in a directory service; possibly using a heuristics search process that involves sophisticated reasoning about service ontologies (semantic web concept).

c. Negotiation – once an element finds potential providers of an input service, it must negotiate with them to obtain that service as soon as possible and in optimal conditions (time, costs, etc.).

d. Provision – once two elements reach an agreement, they must provision their internal resources.

e. Operation - if the agreement were violated, one or both elements would seek an appropriate remedy.

f. Termination

Some Scientific Challenges for implementing AC concepts, according to the same [Kep03], are the followings:

1. Behavioural abstractions and models – we need fundamental mathematical work aimed at understanding how the properties of self-configuration, self-optimisation, self-maintenance and robustness arise from or depend on the
behaviours, goals, and adaptivity of individual autonomic elements; the patterns and type of interactions among them; and the external influences or demands on the system. How can we derive a set of behavioural and interaction rules that, if embedded in individual autonomic elements, will induce a desired global behaviour?

2. Robustness theory (see http://discuss.santafe.edu/robustness)

3. Learning and optimisation theory – at this moment learning multi-agent theory is a challenging but relatively unexplored problem, actually based on biologic models (example: stigmergic approaches in multi-agent learning).

4. Negotiation theory

5. Automated statistical modelling – statistical models of large networked systems will let autonomic elements or systems detect or predict overall performance problems from a stream of sensor data from individual devices.

8.3 THE NEXT LOCATION PREDICTION PROBLEM IN AN UBICOM ENVIRONMENT

Ubiquitous systems in an AC context strive for adaptation to user needs by utilizing information about the current context in which a user’s appliance works. A new quality of ubiquitous systems may be reached if context awareness is enhanced by predictions of future contexts based on current and previous context information. Such a prediction enables the system to proactively initiate actions that enhance the convenience of the user or that lead to an improved overall system. Humans typically act in a certain habitual pattern, however, they sometimes interrupt their behavior pattern and they sometimes completely change the previous pattern. Our aim is to relieve people of actions that are done habitually without determining a person’s action. The system should learn habits automatically and adaptively reverse assumptions if a habit changes. The predictor information should therefore be based on previous behavior patterns and applied to speculate on the future behavior of a person. If the speculation fails, the failing must be recognized, and the predictor must be updated to improve future prediction accuracy [Pet03]. On the other hand, for example in mobile phone communications there is very useful to predict next cell movement of a certain mobile station in order to request in advance, from its corresponding Base Transceiver
Neural Networks Models with Applications in Ubiquitous Computing. Next Location Prediction

Station, a new communication channel (a new radio-communication carrier frequency). Similarly, in some GPS & Digital Maps applications, it is useful to predict the optimal route for example, based on some run-time traffic information, etc. Also it is very useful to implement a next web page predictor for a certain user, based on the accessed URLs history, on user’s profile and schedule, etc., in order to reduce the access latency through a implemente cache buffer. Generally speaking, in the Ubiquitous Computing paradigm, the pro-activity – as a consequence of the context awareness – is implemented using some advanced prediction methods (Neural, Markov, Bayes, Time Series, Support Vector Machine and other kernel methods, etc.).

It is very important in many applications to predict what the user will do, when and where he/she will do it and, if it would be possible, even the reason for her/his action. User modelling is requested for acquiring this understanding. Prediction techniques are also implemented in a lot of computer engineering applications like: image reconstruction, network adaptive routing, strategic computer games, speech recognition (basically here are frequently used Hidden Markov Models), handwritten character recognition, gesture understanding, artificial intelligence applications, pro-active web browsers, etc.

Further we’ll present an interesting research project developed by Professor’s Theo Ungerer’s group from Augsburg University entitled Smart Doorplates. For our application domain we chose next location prediction instead of general context prediction. The algorithms may also be applicable for other more general context domains; however, there already exist numerous scenarios within our presented applications domain. Some sample scenarios may be the following [Pet03]:

- Smart doorplates that are able to direct visitors to the current location of an office owner based on a location-tracking system and predict if the office owner is soon coming back.
- Elevator prediction could anticipate at which floor an elevator will be needed next.
- Routing prediction for cellular phone systems may predict the next radio cell a cellular phone owner will enter based on his previous movement behavior (intra-cells and inter-cells movements).

To predict or anticipate a future situation learning techniques as e.g. Markov Chains, Hidden Markov Models, Bayesian Networks, Time Series or Neural Networks are obvious candidates. The challenge is to transfer these
algorithms to work with context information. Petzold et al. in their work [Pet03], transformed some prediction algorithms used in branch prediction techniques of current high-performance microprocessors, already presented by us in Chapter 2, to handle context prediction in a certain ubiquitous system. They evaluated the one-level one-state, two-state, and multiple-state predictors, and the two-level two-state predictors with local and global first-level histories. The evaluation was performed by simulating the predictors with behavior patterns of people walking through a building as workload. Their simulation results show that the context predictors perform well but exhibit differences in training and retraining speed and in their ability to learn complex patterns.

The global two-level context predictors with 2-state predictors in the second level (see Figure 8.2) regard a sequence of the last rooms that a person entered to predict the next room. The visited rooms are stored in a kind of shift register that constitutes the first level of the predictor. If a new room is entered all entries of the register are shifted to the left and the new room is filled in from the right. The length of the shift register is called the predictor’s order, which denotes the number of last visited rooms that influence the prediction. The second level consists of a pattern history table that stores all possible patterns of room sequences in different entries. Each pattern has associated a 2-state predictor (see Figure 8.1). In fact, the scheme implements a simplified (saturated) Markov predictor for 3 rooms (B, E, S) and a common corridor. States B0, E0, S0 are weak predictions instead states B1, E1, S1 are strong predictions.

![Floor plan with four rooms and prediction graph of 2-state predictor for the corridor with three neighbor rooms (B – boss, E – employee, S – secretariat)](image)
Figure 8.2 shows the case when a third order Markov predictor is used, assuming that the sequence of visited rooms is the following: CBSBSECBSECBSBSEC. After the first occurrence of the pattern SEC the initial state B0 is set. The predictor changes to state B1 after the next occurrence of SEC pattern. Now the prediction is that the next room is boss’ office B.

![Figure 8.2. Global two-level predictor with 2-state predictors in the second level](image)

Petzold et al. also introduced the global two-level context predictors with frequency analysis in the second level. The second level stores for each pattern the frequencies of all previous accesses to all neighbors’ rooms from the current room. Now the room with the highest frequency is predicted. This predictor corresponds to the Markov Predictor. Figure 8.3 presents the structure of the global two-level context predictors with frequency analysis in the second level, considering the same example as that given in Figure 8.2.

![Figure 8.3. Global two-level predictor with frequency analysis in the second level](image)

In another work [Pet04], Petzold et al. introduced context prediction techniques based on previous behavior patterns, in order to anticipate a person’s next location movement. They analyzed the two-level predictors
with global first-level histories and the two-state predictors and they compared these predictors with the Prediction by Partial Matching (PPM) method. As we already explained in Chapter 6, PPM successively works on a set of \((k+1)\) Markov predictors of order \(k, k-1, k-2, \ldots, 2, 1, 0\) respectively. They evaluated the predictors by some movement sequences of real persons within an office building reaching up to 59% accuracy in next location prediction without pre-training and, respectively, up to 98% with pre-training.

Based on the previous briefly presented ideas and references related to Augsburg Smart Doorplate Project, below I’m trying to suggest some further ideas to be analysed for Augsburg’s “context prediction based on previous behaviour patterns project”, described in [Pet03, Pet04]. Some of these ideas were really further developed and presented in this chapter, too. So, for example a simple static predictor, having implemented a simple prediction rule (for example: the person will return in his own office) would be useful in this particular next location prediction problem, even as a comparative scheme. Some of mine ideas are open problems at this moment. I think that some certain human behaviour (person’s scheduler – fixed courses, fixed meetings, department scheduler, etc.) must be introduced in a flexible manner into the system. So, for example, if a professor has a course Monday from 10 to 12, the system will know this and it will correspondingly action. The points 1 - 4 are related to the branch prediction analogy, the points 5 – 13 are related to a new value prediction analogy and the last two points (14, 15) propose new approaches for your next location predictor methods.

1. Another solution could be a Local or Global Predictor having an additional confidence (Confid) field, implemented as an up / down counter (correctly predicted / incorrectly predicted, respectively). This new scheme should be a good trade-off between an advantage of the proposed predictor (rapidly trained 2-state predictor) and a disadvantage (oscillations between weak states). The prediction might be done based on a heuristics between the predictor’s state and the confidence level. See figures 8.4 to 8.6. This solution was efficiently investigated by Ausburg group.

2. A New Prediction Scheme based on a hybrid predictor (Global & Local). That means a Shift Register that will point to a table’s location (Context Table - CT). This CT location will contain two fields: a Tag (Global History Pattern) and a Local History Pattern (LHP). This LHP will address the Prediction Table’s location (second table). This location will contain two fields: a Tag (Local History Pattern) and a 2 state prediction automata (Pred). After we’ll know the real next movement, the Global History
Pattern will be updated (shift logic left), and, if it is necessary, only its corresponding LHP field belonging to CT, in the same manner. In order to avoid two sequential table-indexing processes, we can cache the \( \text{Pred} \) field from the second table into the first table (CT), after checking the prediction (thus, after the automata from the LHP will be correspondingly updated). Obviously, additionally might be used also in this case a \( \text{Confid} \) field like I suggested above. For implementation see figures 8.4 to 8.6.

3. In order to minimise the previous hybrid predictor costs and complexity, someone could try to hash the Global History Register with the Local History Register (what hashing function is better?). For doing this, you might binary encoded the rooms, and using, for example a XOR function for hashing. After this, the obtained index will address the Prediction Table (PT). PT’s location will contain a Tag (that will be checked if it is identical with the generated index), and, of course, the \( \text{Pred} \) field (eventually even a \( \text{Confid} \) field). See figure 8.7 for a better understanding.

4. Another hybrid predictor (Global & Local) might consist by a Local and a Global Predictor, having an additional \( \text{Confid} \) field (see above), working together. The prediction’s prioritisation might be done based on a heuristics between the 2-predictor states and the 2 attached confidence levels.

5. The next location context prediction research is clearly closer with the value prediction domain, already presented by us in Chapter 6, rather than to the branch prediction domain. Therefore, a cross-fertilisation with the contextual value prediction schemes might be a good idea. Additionally, in contrast with instruction value prediction, you apriori know that the next value belongs to a certain values set (you know all the possible rooms). How could someone exploit this little advantage?

6. A good artefact of a context predictor’s accuracy (based on previous behaviour movements patterns) might be a Person Centric Value Locality, measured on Augsburg benchmarks, using different history lengths. Considering a person’s movement locations string (benchmark), we’ll measure if the current produced value belongs to the previously \( k \) values, for different \( k \) (history). This metric might be important for establishing the optimal context predictor’s order and also for suggesting us the (ultimate?) obtainable predictability limit.
7. Somehow inspired from the degree of use metric, previously presented in Chapter 6, I propose a metric called “the associated room distance”. Again, considering a person’s movement string (benchmark), we’ll measure for each room, the average distance between two consecutive instances of that room. If the room distance is big (for example, over the pattern length), I believe that we can eliminate that room from the benchmark (filter), because it might act as a noise, and therefore is not efficient for predictor’s static training process.

8. An important challenge consists in developing a prediction scheme derived from that presented in Figure 6.14 from Chapter 6. The scheme could be local or global. Person Centric Value Locality measurements (see point 6), will suggest how many values to store in VHT structure. Some qualitative advantages of this scheme comparing with the already implemented schemes could be:

- It uses confidence counters attached to the predicted rooms (C0 – C3).
- The prediction is associated with a longer compressed context (VHP field, \( p > 4 \)) and this might increase prediction’s accuracy.

9. It should have sense to implement a computational predictor? It might be useful in some non-repetitive movements, generated by a possible computational rule? Anyway, if so, it’s possible a hybrid predictor in a new sense (contextual & computational, like in value prediction area).

10. I think it might be useful to implement also the complete PPM predictor, consisting in \( k \) Markov predictors of order \( k, k-1, \ldots, 1 \). After this, all the implemented schemes might be compared with this complete PPM predictor that could represent an "ultimate prediction limit" (a useful metric for understanding the obtained prediction accuracy, using more realistic schemes). As I pointed out at point 6, Person Centric Value Locality might be another useful “predictability limit”.

11. After this, it might be useful perhaps to implement also some simplified PPM predictors (for example, containing only a Markov predictor of order \( k \) and respectively a Markov predictor of order 1). In [Pet03], it is implemented an observable Markov model of order 3. If in this scheme, the associated pattern frequencies are 0, it might be useful to regress to a Markov predictor of order 1. How far is its performance related to a complete PPM predictor?
12. There are needed relatively simple prediction algorithms due to the PDAs memory and computing limitations (64 Mbytes DRAM, only 1 Mbytes space for Java programs, Windows CE, ARM CPU at 400 MHz etc). However, predictors could be pre–trained into a powerful desktop PC and, after this, putting them into PDAs.

13. The question is: how much time could be saved when we are searching N consecutive persons, and the probability of correctly predict a person is p? (If I predict it correct, I’ll save time T). I think it’s quite easy to develop an analytical model. After this, it might be interesting to develop a more realistic model, based on time distributions given by Augsburg’s benchmarks.

14. After researching based on branch and value prediction analogy, next major step consists in developing a back-propagation neural predictor with a single hidden layer. Inputs: Person Identifications & Global/Local History Pattern, both binary encoded. For example, considering 8 persons, 8 rooms and a (global/local) history pattern length of 3, we need 12 bits as an input pattern. Outputs: Rooms Identifications (A, B, C, D, E, F, G, H). It was recommended a hidden layer having (N+1) cells, where N is the input layer cells number. As it can be seen we model and simulate this neural predictor further in this chapter. Therefore, the prediction problem will be modelled as a classification problem. In other words: to which class belongs person X in context Y (Global/Local History Pattern? If the obtained results are encouraged, we can further simplify the neural network architecture, using for example simple perceptrons instead of the 2 – state predictors (that is useful only if our prediction problem is linear separable).

15. “How might be the predictor developed time-dependent?” it’s an open problem at this moment. However, at a first look a relatively simple Room Centric & Person Centric markovian approach could be possible. Considering – for a given person - the room sequence: A(10 minutes), B(6 minutes), B(3),C(2),D(6),A(5),C(3),C(1),B(11),D(12),C(3),A(4)… [For example A(10) means that the person visited room A for a time period of 10 minutes]. Supposing that the predictor will predict that the next room will be C, we can extract the sub-string: C(2),C(3),C(1),C(3), and, based on a Markov predictor of order 1 it will predict that the person will stay in room C about 1 minute. Of course, there will be necessary some time
approximations (example, 3 minutes = [2 minutes and 30 seconds, 3 minutes and 30 seconds], etc.). Other statistic models might be developed. As it is further shown, even a Hidden Markov Model might be useful for solving this problem.

**LOCAL PREDICTOR (Room Centric & Person Centric)**

![Figure 8.4. A Local Predictor Scheme with Confidences (focused on Room A)](image)

**GLOBAL PREDICTOR (Person Centric & Non-Room Centric)**

![Figure 8.5. A Global Predictor Scheme with Confidences](image)
Based on the previous suggestion from point 14 in [Vin04] the authors proposed neural prediction techniques to anticipate a person’s next location movement. The work focused on neural predictors (multi-layer perceptron
with back-propagation learning algorithm) with and without pre-training. The optimal configuration of the neural network was determined by evaluating movement sequences of real persons within an office building, at Augsburg University. The simulation results, obtained with one of the pre-trained neural predictors, show accuracy in next location prediction reaching up to 92%. Most of the further results presented in this chapter were based on this work.

Mozer in his work [Moz98], proposed an Adaptive Control of Home Environments (ACHE). ACHE monitors the environment, observes the actions taken by the inhabitants, and attempts to predict their next actions, by learning the anticipation needs. The predictors are implemented as feed-forward neural networks with back-propagation learning algorithm. Unfortunately, the author doesn’t present the predictor’s architecture, and, as a consequence, the paper doesn’t contain any quantitative results. In another paper [Moz04], Mozer proposed and implemented a smart home environment. The intelligence of the home arises from the home’s ability to predict the behavior and needs of the inhabitants by having observed them over a period of time. He focused on home comfort systems, specifically air temperature regulation, and lighting. Instead of being programmed to perform certain actions, the house essentially adapts dynamically itself by monitoring the environment and sensing actions performed by the inhabitants, observing the occupancy and behavior patterns of the inhabitants, and learning to predict future states of the house. The author uses as a predictor a feed-forward neural network with one hidden layer for anticipating the next action (as an example, the system will predict when an inhabitant returns home and therefore will start the heater). When the predictions are incorrect, the inhabitants can simply indicate their preferences via ordinary interfaces, e.g., light switches, thermostats, and simply turning on the hot water.

Aguilar et al., in their work [Agu03], implemented a system to reduce latency in virtual environment applications. In Virtual-Environment Applications, where virtual objects are presented in a head-mounted display, virtual images must be continuously stabilized in space against the user’s head motion. Latencies in head-motion compensation cause virtual objects to swim around instead of being stable in space. This results in an unnatural feel, disorientation, and simulation sickness in addition to errors in matching of virtual and real objects. Visual update delays are a critical technical obstacle for implementation of head-mounted displays in a wide variety of applications. To address this problem, Aguilar et al., used machine-learning techniques to define a forward model of head movement based on angular velocity information. They use a recurrent neural network to capture temporal
patterns of pitch and yaw motion. Their results demonstrate an ability of the neural network to predict head motion up to 40 ms ahead thus eliminating the main source of latencies. The results also demonstrate successful generalization by the learning system.

Further in this chapter neural prediction techniques are used in order to anticipate persons’ next movements into an intelligent ambient (institute). The application predicts the next room based on the history of rooms visited by a certain person who is moving within an office building. Three movement prediction types are used. The complete prediction mechanism predicts all destination rooms within a benchmark (history of movements for a certain person). It is also defined the restrictive prediction – a prediction mechanism that predicts the next room from all rooms except for the own office. The binary prediction is the simplest mechanism that predicts, from all rooms excepting the own office, if the person will return in his own room or not (if not, it does not predict the room). All the further research methodology and results are presented based on [Vin04, Gel06].

### Movement Sequences

The optimal configurations of the predictors are determined by evaluating them on some movement sequences of real persons developed at the University of Augsburg [Pet04]. The following table shows the number of rooms that appear in each benchmark:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of rooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>10</td>
</tr>
<tr>
<td>Employee 2</td>
<td>14</td>
</tr>
<tr>
<td>Employee 3</td>
<td>11</td>
</tr>
<tr>
<td>Boss</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 8.1. The number of visited rooms by each person

Each line from the original benchmarks [Pet04] represents a person movement (his/her entry in a room). It contains the movement’s date and hour, the room’s name, the person’s name and a timestamp (example: 2003.07.08 10:54:15; Room 406; Person_Name; 1057654455005). In the codification process the common corridor was eliminated from the benchmark, because it is common and it could behave as noise. The rooms belonging to each benchmark were codified separately (dynamic
codification), because in this way the room codes consist in only 4 bits instead of 5 bits. Thus, different encoding is used for the same room for different benchmarks. Obviously, a certain room has the same codification in a certain benchmark. Anyway, it does not matter that a room could have different codification in different benchmarks. For the predictor is important to find significant correlation in order to predict. Table 8.2 shows how a benchmark looks before and after the room codification process.

<table>
<thead>
<tr>
<th>Original benchmark</th>
<th>Benchmark after room codification</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003.07.07 10:13:45; 402; Employee2; 1057565625801</td>
<td>0</td>
</tr>
<tr>
<td>2003.07.07 10:21:41; corridor; Employee2; 1057566101067</td>
<td>-</td>
</tr>
<tr>
<td>2003.07.07 10:21:45; 411; Employee2; 1057566105152</td>
<td>1</td>
</tr>
<tr>
<td>2003.07.07 10:21:48; corridor; Employee2; 1057566108771</td>
<td>-</td>
</tr>
<tr>
<td>2003.07.07 10:21:54; 402; Employee2; 1057566114338</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.2. The first lines from a certain benchmark (with a movement sequence of Employee 2) before and after the room codification process

After the codification process the benchmarks contain only the room codes (0÷13 in Employee2’s case), because in this starting stage of work only this information is used in the prediction process. Two benchmark types are used: some short benchmarks containing about 300-400 movements and some long benchmarks containing about 1000 movements. The short benchmarks are used for static training and the long benchmarks for evaluations.

8.4 PERSON MOVEMENT PREDICTION USING NEURAL NETWORKS

The artificial neural networks (NN) are composed of a multitude of interconnected neurons representing simple processing elements that operate in parallel. A great advantage of the artificial neural networks is their capacity to learn based on examples (supervised learning). In order to solve a problem traditionally, we have to elaborate its model, and after that we have to indicate a succession of operations that represents the solving algorithm of the problem. However there are practical problems with a high level of
complexity, and for this kind of problems it is very hard or even impossible to establish such a deterministic algorithm.

In the connection models like neural networks we are not forced to give a solving algorithm dedicated to a certain problem; we have to offer to the NN only a multitude of consistent examples in order to learn and generalize them. The network extracts the information from the training samples. In this way it is able to synthesize implicitly a certain model of the problem. In other words, the neural network builds up alone an algorithm to solve a problem. The capacity of the neural network to solve complex practical problems using a multitude of samples gives them a highly large potential of applicability. Building intelligent systems that can model human behavior has captured the attention of the world for years. So, it is not surprising that a technology such as neural networks has generated great interest during the last years.

The neural network’s structure

In this work we used a multi-layer perceptron (MLP) with one hidden layer (see Figure 8.8) and the well-known back-propagation learning algorithm. The rooms are binary codified, thus we chose bit encoding with complexity $\log_2 N$ (entries in MLP), instead of one room - one neuron encoding with complexity $N$ (no. of entries in MLP). This codification might be useful taking into account further enlargements of the project, too ($N$ will probably grow).

![Figure 8.8. The multi-layer perceptron](image)
Another neural network studied in this work is the Elman’s recurrent neural network [Elm90]. The difference between the Elman network and the multi-layer perceptron with one hidden layer consists in the additional context units in the input layer of the Elman network. These units are also hidden in the sense that they interact exclusively with other nodes internal to the network, and not with the outside world. The context units are initially set to 0 (the middle of the [-1, 1] interval). Both the input units and context units activate the hidden units; and then the hidden units feed forward to activate the output units. The hidden units also feed back to activate the context units. This constitutes the forward activation. At the next time step \( t+1 \) the above sequence is repeated. This time the context units contain values, which are exactly the hidden unit values at time \( t \). These context units thus provide the network with memory. In Figure 8.9 is presented a simple recurrent Elman network in which activations are copied from hidden layer to context layer on a one-for-one basis, with fixed weight of 1.0. Dotted lines represent trainable connections.

![The structure of the Elman’s recurrent network](image)

**Figure 8.9.** The structure of the Elman’s recurrent network

For both the multi-layer perceptron and the Elman recurrent neural network, each person has his/her own predictor, and therefore, the input data consists only in the code of the last visited rooms. One of the parameters of the network is the number of rooms in the input vector. This parameter is varied between 1 and 8, in order to see how is affected the prediction accuracy by the length of room history (analogous with implementing Markov predictors of order 1 to 8). Since the maximum number of rooms in a benchmark is 14
(see Table 8.1) the rooms are codified on 4 bits. As an example, for a history of four rooms visited by the second employee – considering the current 4-rooms history: 5, 2, 1, 3 – the input vector obtained after the binary codification of the input data is the following: \( V_i = 0101 \ 0010 \ 0001 \ 0011 \). The neural network will return through its output layer the predicted room. In the output layer one neuron is used for each room. The neural network predicts the room corresponding to the neuron with the highest output value, if this value is greater than a certain threshold.

**Training the neural network**

For better results the input data is codified with -1 and 1 and thus, the following activation function is used:

\[
f(y) = \frac{1 - e^{-y}}{1 + e^{-y}}
\]

(1)

The back-propagation learning algorithm [Mit97] with the sigmoid activation function consists of the following steps:

1. Create a feed-forward network with \( N \) inputs, \( M \) hidden units and \( P \) output units.
2. Initialize all network weights \( w_{ij}^1; i = 1, N; j = 1, M \) and \( w_{jk}^2; j = 1, M; k = 1, P \), to small random numbers belonging to the \( \left[ -\frac{2}{N}, +\frac{2}{N} \right] \) interval.
3. Until \( E(W) = \frac{1}{2} \sum_{k=1}^{P} (t_k - o_k^2)^2 \leq T \) (threshold), do:
   3.1. Input the instance \( \overline{X}^1 \) to the network and compute the output \( \overline{O}^2 \).
      \[
      net_i^1 = \sum_{j=1}^{M} w_{ij}^1 \cdot x_i^1, \quad j = 1, \ldots, M
      \]
      \[
      x_j^2 = o_j^1 = f (net_j^1), \quad j = 1, \ldots, M
      \]
      \[
      net_k^2 = \sum_{j=1}^{M} w_{jk}^2 \cdot x_j^2, \quad k = 1, \ldots, P
      \]
      \[
      o_k^2 = f (net_k^2), \quad k = 1, \ldots, P
      \]
   3.2. For each network output unit \( k, k = 1, P \), calculate its error term \( \delta_k^2 \).
\[ \delta_k^2 = \left( o_k^2 - o_k^2 \right) \cdot f'(\text{net}_k^2) = \frac{1}{2} \left( o_k^2 - o_k^2 \right) \cdot \left( 1 - o_k^2 \cdot o_k^2 \right) \] (2)

3.4. For each hidden unit \( j \), \( j = 1, M \), calculate its error term \( \delta_j^i \)

\[ \delta_j^i = \sum_{k=1}^{P} \delta_k^2 \cdot w_{jk}^2 \cdot f'(\text{net}_j^1) = \frac{1}{2} \left( 1 - o_j^1 \cdot o_j^1 \right) \cdot \sum_{k=1}^{P} \delta_k^2 \cdot w_{jk}^2 \] (3)

3.5. Update each network weight

\[ w_{jk}^2 = w_{jk}^2 + \alpha \cdot \delta_k^2 \cdot x_j^2, \quad j = 1, M, \quad k = 1, P \] (4)

\[ w_{ij}^1 = w_{ij}^1 + \alpha \cdot \delta_j^1 \cdot x_i^1, \quad i = 1, N, \quad j = 1, M \] (5)

where \( \alpha \) is the learning step.

**Static Learning**

The static learning means that before the effective run-time prediction process, the predictor will be trained based on some room history patterns belonging to the previous presented benchmarks (person movement sequences). A very important parameter is the threshold’s value \( (T) \). As an example, for a threshold of 0.2, the output values are accepted only if they belong to the \([-1, -0.8]\) interval for \(-1\) (0) or in the \([0.8, 1]\) interval for \(+1\). If the output values are not in one of those intervals, the backward step is generated until this condition is fulfilled. In other words, this iterative training process will continue until all network’s outputs will belong to those intervals. Obviously, the training will be stopped if the error function will be less than a certain threshold. Another important parameter is the learning rate \( (\alpha) \).

**Dynamic Learning (Run-Time Prediction Process)**

One of the differences between the static and dynamic learning is that during dynamic learning we predict based on the result of the feed-forward step. If the output value is belonging to \([-1, 0)\) interval it will be considered \(-1\) (logical 0) and if it belongs to the \([0, 1]\) interval it will be considered +1. Another difference is that in the case of dynamic learning a single backward step is applied even if the prediction is not correct (this means that the prediction process is faster, and, thus, better adapted to real-time restrictions). More than that, the simulation results from prior work [Gel06, Vin04] show that the prediction accuracy is the highest when only one backward step is applied after each prediction during the dynamic learning process.
In the case of static & dynamic learning the network is statically trained before its effective use. In this case the dynamic learning process is started with the weights generated by the static training process. If only dynamic learning is used, the weights are initially randomly generated, and, after this, the network will effectively predict. An iteration step means to run one time all the benchmarks.

8.5 EXPERIMENTAL RESULTS. ANALYSIS AND COMMENTS

The developed simulators have the following parameters [Gel06]: the number of neurons in the input layer ($N$) practically determined by the room history length ($H$), the number of neurons in the hidden layer ($M$), the threshold’s value used in the static learning process ($T$), and the learning rate ($\alpha$). The simulator’s output represents the predicted room. Varying all these parameters, hopefully it will be obtained the optimal configuration of the studied neural networks. The predictors are evaluated by the movement sequences of real persons [Pet04] previously presented. We introduced the following definitions:

- NM = number of movements from all rooms excepting own office;
- NP = number of predictions;
- NCP = number of correctly predicted movements;
- PA = prediction accuracy (NCP/NP);
- Quantity = NP/NM;
- Gain = the factor which gives the improvement of the quality without confidence estimation vs. the quality with confidence estimation.

Prediction Using Multi-Layer Perceptrons

We introduced the notation MLP (H=h, N=n, M=m, $\alpha=\alpha$, T=t) in order to represent a certain configuration of the multi-layer perceptron predictor. The above notation indicates that predictions are generated based on a history of $h$ rooms using a neural network with $n$ input layer neurons, $m$ hidden layer neurons, a learning rate $\alpha$ and in the case of pre-training, a threshold $t$. The simulation results showed that the best configuration of a multi-layer
perceptron for this specific application implies $N+1$ neurons in the hidden layer (where $N$ is the number of neurons in the input layer). The number of output layer neurons ($P$) is always equal to the number of rooms used in the prediction process (see Table 8.1), thus, the neural network has in its output layer one neuron for each room and it predicts the room corresponding to the neuron with the highest output value.

The first varied parameter is the learning rate ($\alpha$). We used the complete prediction mechanism of a dynamically trained MLP ($H=2$, $N=8$, $M=N+1$) predictor on the long benchmarks. The results from Table 8.3 were generated after 10 iterations of each benchmark. Table 3 shows that the optimal learning rate is 0.1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$\alpha = 0.05$</th>
<th>$\alpha = 0.10$</th>
<th>$\alpha = 0.15$</th>
<th>$\alpha = 0.20$</th>
<th>$\alpha = 0.25$</th>
<th>$\alpha = 0.30$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>57.69</td>
<td>58.63</td>
<td>58.12</td>
<td>57.08</td>
<td>57.60</td>
<td>56.71</td>
</tr>
<tr>
<td>Employee 2</td>
<td>57.54</td>
<td>56.70</td>
<td>56.27</td>
<td>55.89</td>
<td>54.31</td>
<td>53.37</td>
</tr>
<tr>
<td>Employee 3</td>
<td>47.36</td>
<td>47.62</td>
<td>46.82</td>
<td>45.54</td>
<td>46.14</td>
<td>46.58</td>
</tr>
<tr>
<td>Boss</td>
<td>57.28</td>
<td>57.50</td>
<td>57.21</td>
<td>56.79</td>
<td>55.60</td>
<td>52.64</td>
</tr>
<tr>
<td>Average</td>
<td>54.96</td>
<td><strong>55.11</strong></td>
<td>54.60</td>
<td>53.82</td>
<td>53.41</td>
<td>52.32</td>
</tr>
</tbody>
</table>

**Table 8.3.** Prediction accuracies using different learning rates ($\alpha$), reported after 10 iterations

The room history length ($H$) is varied using a complete prediction mechanism with a dynamically trained MLP ($N=4*H$, $M=N+1$, $\alpha=0.1$) complete predictor on the long benchmarks and the results are reported after 10 iterations of each benchmark. Table 4 shows that the optimal room history length is 2.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$H=1$</th>
<th>$H=2$</th>
<th>$H=3$</th>
<th>$H=4$</th>
<th>$H=5$</th>
<th>$H=6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>58.55</td>
<td>58.68</td>
<td>57.54</td>
<td>56.77</td>
<td>56.14</td>
<td>55.69</td>
</tr>
<tr>
<td>Employee 2</td>
<td>57.35</td>
<td>56.70</td>
<td>54.88</td>
<td>55.37</td>
<td>54.40</td>
<td>53.73</td>
</tr>
<tr>
<td>Employee 3</td>
<td>47.36</td>
<td>48.02</td>
<td>46.86</td>
<td>46.54</td>
<td>45.69</td>
<td>44.96</td>
</tr>
<tr>
<td>Boss</td>
<td>54.10</td>
<td>57.33</td>
<td>55.33</td>
<td>54.57</td>
<td>53.22</td>
<td>53.04</td>
</tr>
<tr>
<td>Average</td>
<td>54.34</td>
<td><strong>55.18</strong></td>
<td>53.65</td>
<td>53.31</td>
<td>52.36</td>
<td>51.85</td>
</tr>
</tbody>
</table>

**Table 8.4.** Prediction accuracies using different room history lengths ($H$), reported after 10 iterations
Figure 8.10. Prediction accuracies using different room history lengths (H), reported after 10 iterations

The threshold’s value, used in the static training process, is varied using a statically pre-trained dynamic MLP (H=2, N=8, M=N+1, \(\alpha = 0.1\)) with complete prediction mechanism. The MLP was statically pre-trained before its effective use in the prediction process. This means that the dynamically trained predictor is initialized with the weights generated by the static learning process. The results are reported after 10 iterations of the benchmarks. For these simulations we used the short benchmarks in the static training process and respectively the long benchmarks in the dynamic prediction process. Table 8.5 and Figure 8.11 show how is affected the prediction accuracy by the threshold’s value used in the static training process.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>T=0.1</th>
<th>T=0.3</th>
<th>T=0.5</th>
<th>T=0.7</th>
<th>T=0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>61.64</td>
<td>60.70</td>
<td>59.76</td>
<td>58.82</td>
<td>58.45</td>
</tr>
<tr>
<td>Employee 2</td>
<td>57.25</td>
<td>56.85</td>
<td>57.12</td>
<td>57.52</td>
<td>56.45</td>
</tr>
<tr>
<td>Employee 3</td>
<td>49.84</td>
<td>49.31</td>
<td>49.04</td>
<td>49.17</td>
<td>49.91</td>
</tr>
<tr>
<td>Boss</td>
<td>59.18</td>
<td>59.06</td>
<td>58.34</td>
<td>57.78</td>
<td>58.13</td>
</tr>
<tr>
<td>Average</td>
<td>56.97</td>
<td>56.48</td>
<td>56.06</td>
<td>55.82</td>
<td>55.73</td>
</tr>
</tbody>
</table>

Table 8.5. Prediction accuracies, reported after 10 iterations, using different thresholds (T) in the static training process
The optimal threshold is 0.1. As it can be observed from Tables 8.4 and 8.5, for a history length of 2, without pre-training the prediction accuracy is at average 55.18% and with pre-training (T=0.1) the prediction accuracy grows to 56.97%. Further we extracted from the presented previous results the prediction accuracies obtained when the prediction process is simplified (binary prediction). That means a binary MLP (H=2, N=8, M=N+1, α =0.1, T=0.1) predictor (the neural network predicts if the person will return in his own room or not), using the short benchmarks in the static training process and respectively the long benchmarks in the run-time prediction process. Table 8.6 compares the prediction accuracies obtained after 10 iterations using a dynamically trained predictor and respectively a statically pre-trained dynamic predictor. Table 8.7 compares the prediction accuracies obtained with the same predictors, reported after a single iteration.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic training</th>
<th>Static &amp; Dynamic training</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>91.26</td>
<td>91.53</td>
</tr>
<tr>
<td>Employee 2</td>
<td>79.92</td>
<td>80.07</td>
</tr>
<tr>
<td>Employee 3</td>
<td>69.81</td>
<td>70.52</td>
</tr>
<tr>
<td>Boss</td>
<td>79.45</td>
<td>79.45</td>
</tr>
<tr>
<td>Average</td>
<td>80.11</td>
<td><strong>80.39</strong></td>
</tr>
</tbody>
</table>

Table 8.6. Comparing the accuracies generated through binary prediction using a dynamic predictor and a statically pre-trained dynamic predictor – results reported after 10 iterations

**Figure 8.11.** Prediction accuracies, reported after 10 iterations, using different thresholds (T) in the static training process
Table 8.7. Comparing the accuracies generated through binary prediction using a dynamic predictor and a statically pre-trained dynamic predictor – results reported after one iteration

In order to further improve the prediction accuracy, a 4-state local confidence automaton is associated to each pattern (sequence) of two rooms (because the optimal room history length is 2) like that presented in Figure 8.12. We used a prediction table, which is addressed with the sequence of the last two visited rooms’ codes, like that presented in [Pet03]. The difference is that each pattern of two rooms has associated a confidence automaton (the prediction process is done in this case through a global neural network instead of multiple finite automata), and in [Pet03] it has associated a prediction automaton. The next room is predicted only if the selected confidence automaton is in the strongest predictable state.

Figure 8.12. The confidence automaton associated with each distinct pattern of 2 rooms
The following tables and figures present the experimental results obtained using 4-state local confidence automata. We varied again the learning rate using a dynamically trained complete MLP (H=2, N=8, M=N+1) predictor on the long benchmarks. The next table shows the prediction accuracies obtained with different learning rates using the local confidence automata. The results were reported after 10 iterations of the benchmarks. It can be observed that this time the optimal learning rate is 0.2.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>α =0.05</th>
<th>α =0.1</th>
<th>α =0.15</th>
<th>α =0.2</th>
<th>α =0.25</th>
<th>α =0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>91.00</td>
<td>90.92</td>
<td>90.84</td>
<td>90.67</td>
<td>90.23</td>
<td>90.54</td>
</tr>
<tr>
<td>Employee 2</td>
<td>85.56</td>
<td>85.46</td>
<td>85.74</td>
<td>86.41</td>
<td>86.21</td>
<td>86.11</td>
</tr>
<tr>
<td>Employee 3</td>
<td>77.42</td>
<td>77.34</td>
<td>77.45</td>
<td>77.28</td>
<td>76.68</td>
<td>76.42</td>
</tr>
<tr>
<td>Boss</td>
<td>85.27</td>
<td>86.84</td>
<td>86.86</td>
<td>87.21</td>
<td>87.19</td>
<td>87.06</td>
</tr>
<tr>
<td>Average</td>
<td>84.81</td>
<td>85.14</td>
<td>85.22</td>
<td>85.39</td>
<td>85.07</td>
<td>85.03</td>
</tr>
</tbody>
</table>

Table 8.8. Prediction accuracies reported after 10 iterations for different learning rates (α) using confidence automata

The threshold’s value is varied using a statically pre-trained dynamic MLP (H=2, N=8, M=N+1, α=0.1) with confidence automata and complete prediction mechanism. The results are reported after 10 iterations of the benchmarks. For these simulations we used the short benchmarks in the static training process and respectively the long benchmarks in the dynamic prediction process. Table 8.9 shows how is affected the prediction accuracy by the threshold’s value used in the static training process, when we used the 4-state local confidence automata.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>T=0.1</th>
<th>T=0.3</th>
<th>T=0.5</th>
<th>T=0.7</th>
<th>T=0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>90.71</td>
<td>91.47</td>
<td>90.58</td>
<td>91.29</td>
<td>90.87</td>
</tr>
<tr>
<td>Employee 2</td>
<td>86.80</td>
<td>87.06</td>
<td>86.87</td>
<td>87.16</td>
<td>87.08</td>
</tr>
<tr>
<td>Employee 3</td>
<td>77.13</td>
<td>77.61</td>
<td>76.31</td>
<td>77.11</td>
<td>77.37</td>
</tr>
<tr>
<td>Boss</td>
<td>86.38</td>
<td>86.10</td>
<td>86.42</td>
<td>86.24</td>
<td>86.09</td>
</tr>
<tr>
<td>Average</td>
<td>85.25</td>
<td>85.56</td>
<td>85.04</td>
<td>85.45</td>
<td>85.35</td>
</tr>
</tbody>
</table>

Table 8.9. Prediction accuracies reported after 10 iterations for different thresholds (T) using 4-state confidence automata
Using the local confidence automata, the optimal threshold is 0.3. The following results (see Tables 8.10 and 8.11) were obtained after 10 iterations of each long benchmark, using a dynamic MLP (H=2, N=8, M=N+1, $\alpha =0.1$) without confidence automata and respectively a dynamic MLP (H=2, N=8, M=N+1, $\alpha =0.2$) with confidence automata, both with complete prediction mechanism.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NM</th>
<th>NP</th>
<th>NCP</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>2130</td>
<td>2130</td>
<td>1253</td>
<td>0.5882</td>
</tr>
<tr>
<td>Employee 2</td>
<td>4800</td>
<td>4800</td>
<td>2700</td>
<td>0.5625</td>
</tr>
<tr>
<td>Employee 3</td>
<td>4510</td>
<td>4510</td>
<td>2131</td>
<td>0.4725</td>
</tr>
<tr>
<td>Boss</td>
<td>4280</td>
<td>4280</td>
<td>2466</td>
<td>0.5761</td>
</tr>
</tbody>
</table>

Table 8.10. Person movement prediction without confidence automata – results reported after 10 iterations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NM</th>
<th>NP</th>
<th>NCP</th>
<th>PA</th>
<th>Quantity</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>2130</td>
<td>1030</td>
<td>922</td>
<td>0.8951</td>
<td>0.48</td>
<td>1.52</td>
</tr>
<tr>
<td>Employee 2</td>
<td>4800</td>
<td>1961</td>
<td>1688</td>
<td>0.8607</td>
<td>0.40</td>
<td>1.53</td>
</tr>
<tr>
<td>Employee 3</td>
<td>4510</td>
<td>1431</td>
<td>1094</td>
<td>0.7645</td>
<td>0.31</td>
<td>1.61</td>
</tr>
<tr>
<td>Boss</td>
<td>4280</td>
<td>1713</td>
<td>1461</td>
<td>0.8528</td>
<td>0.40</td>
<td>1.48</td>
</tr>
</tbody>
</table>

Table 8.11. Quality improvements introduced by predictors with automata, reported after 10 iterations

Further we extracted again from the presented previous results the prediction accuracies obtained using the confidence automata, when the prediction process is simplified (binary prediction). It was used a binary MLP (H=2, N=8, M=N+1, $\alpha =0.2$, $T=0.3$) predictor working on the short benchmarks in the static training process and respectively working on the long benchmarks in the run-time prediction process. Table 8.12 compares the prediction accuracies obtained after 10 iterations using a dynamically trained predictor and respectively a statically pre-trained dynamic predictor. Table 8.13 compares the prediction accuracies obtained with the same predictors, reported after a single iteration of the benchmarks.
Table 8.12. Prediction accuracies – reported after 10 iterations – obtained using a dynamic predictor and a statically pre-trained dynamic predictor, both with confidence automata.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic training</th>
<th>Static &amp; Dynamic training</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>92.54</td>
<td>92.61</td>
</tr>
<tr>
<td>Employee 2</td>
<td>88.75</td>
<td>88.78</td>
</tr>
<tr>
<td>Employee 3</td>
<td>78.09</td>
<td>77.53</td>
</tr>
<tr>
<td>Boss</td>
<td>89.42</td>
<td>89.71</td>
</tr>
<tr>
<td>Average</td>
<td><strong>87.20</strong></td>
<td><strong>87.15</strong></td>
</tr>
</tbody>
</table>

Table 8.13. Prediction accuracies – reported after one iteration – obtained using a dynamic predictor and a statically pre-trained dynamic predictor, both with confidence automata.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic training</th>
<th>Static &amp; Dynamic training</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>92.64</td>
<td>92.64</td>
</tr>
<tr>
<td>Employee 2</td>
<td>86.86</td>
<td>87.32</td>
</tr>
<tr>
<td>Employee 3</td>
<td>75.47</td>
<td>78.09</td>
</tr>
<tr>
<td>Boss</td>
<td>90.82</td>
<td>89.62</td>
</tr>
<tr>
<td>Average</td>
<td><strong>86.44</strong></td>
<td><strong>86.91</strong></td>
</tr>
</tbody>
</table>

Table 8.13 shows that the gain of prediction accuracy, obtained by pre-training the MLP predictors, is not significant. More than that, the results reported after 10 iterations (when the prediction accuracy is saturated), show that is better to use only dynamic training (see Table 8.12). It was also studied a restrictive multi-layer perceptron predictor (which predicts the next room from all rooms excepting the own office). It was used a MLP \((H=2, N=8, M=N+1, \alpha =0.1, T=0.1)\) predictor on the short benchmarks in the static training process and respectively on the long benchmarks in the run-time prediction process. Table 8.14 presents comparatively the prediction accuracies obtained using a dynamically trained MLP and respectively a statically pre-trained dynamic MLP, both without confidence automata. Table 8.15 shows the results (number of movements, number of predictions, number of correct predictions, prediction accuracy) obtained with the dynamically trained MLP. In the same manner, Table 8.16 presents the results obtained...
with the statically pre-trained dynamic MLP. The results from Tables 8.14, 8.15 and 8.16, were reported after a single iteration of the benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic training</th>
<th>Static &amp; Dynamic training</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>87.38</td>
<td>87.38</td>
</tr>
<tr>
<td>Employee 2</td>
<td>75.28</td>
<td>75.65</td>
</tr>
<tr>
<td>Employee 3</td>
<td>63.39</td>
<td>68.67</td>
</tr>
<tr>
<td>Boss</td>
<td>77.82</td>
<td>74.05</td>
</tr>
<tr>
<td>Average</td>
<td>75.97</td>
<td>76.44</td>
</tr>
</tbody>
</table>

Table 8.14. Prediction accuracies obtained with a dynamically trained MLP and a statically pre-trained dynamic MLP, both without confidence automata

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NM</th>
<th>NP</th>
<th>NCP</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>113</td>
<td>111</td>
<td>97</td>
<td>0.8738</td>
</tr>
<tr>
<td>Employee 2</td>
<td>269</td>
<td>267</td>
<td>201</td>
<td>0.7528</td>
</tr>
<tr>
<td>Employee 3</td>
<td>267</td>
<td>265</td>
<td>168</td>
<td>0.6339</td>
</tr>
<tr>
<td>Boss</td>
<td>241</td>
<td>239</td>
<td>186</td>
<td>0.7782</td>
</tr>
</tbody>
</table>

Table 8.15. Person movement prediction using a dynamically trained MLP without confidence automata

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NM</th>
<th>NP</th>
<th>NCP</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>113</td>
<td>111</td>
<td>97</td>
<td>0.8738</td>
</tr>
<tr>
<td>Employee 2</td>
<td>269</td>
<td>267</td>
<td>202</td>
<td>0.7565</td>
</tr>
<tr>
<td>Employee 3</td>
<td>267</td>
<td>265</td>
<td>182</td>
<td>0.6867</td>
</tr>
<tr>
<td>Boss</td>
<td>241</td>
<td>239</td>
<td>177</td>
<td>0.7405</td>
</tr>
</tbody>
</table>

Table 8.16. Person movement prediction using a statically pre-trained MLP without confidence automata

We repeated the above simulations using a restrictive multi-layer perceptron with 4-state confidence automata. The results were generated with a MLP (H=2, N=8, M=N+1, $\alpha=0.2$, $T=0.3$) using the short benchmarks for the static training process and respectively the long benchmarks for the run-time prediction process. Table 8.17 presents comparatively the prediction accuracies obtained using a dynamically trained MLP and respectively a statically pre-trained dynamic MLP, both with 4-state confidence automata. Table 8.18 shows the results (number of movements, number of predictions, number of correct predictions, prediction accuracy) obtained with the dynamically trained MLP. Table 8.19 presents the results obtained with the
statically pre-trained dynamic MLP. The results from Tables 8.17, 8.18 and 8.19, were reported after a single iteration of the benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic training</th>
<th>Static &amp; Dynamic training</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>91.35</td>
<td>92.50</td>
</tr>
<tr>
<td>Employee 2</td>
<td>84.79</td>
<td>85.96</td>
</tr>
<tr>
<td>Employee 3</td>
<td>74.63</td>
<td>74.17</td>
</tr>
<tr>
<td>Boss</td>
<td>86.82</td>
<td>85.93</td>
</tr>
<tr>
<td>Average</td>
<td>84.40</td>
<td><strong>84.64</strong></td>
</tr>
</tbody>
</table>

Table 8.17. Prediction accuracies obtained with a dynamically trained MLP and a statically pre-trained dynamic MLP, both with 4-state confidence automata

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NM</th>
<th>NP</th>
<th>NCP</th>
<th>PA</th>
<th>Quantity</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>113</td>
<td>81</td>
<td>74</td>
<td>0.9135</td>
<td>0.71</td>
<td>1.04</td>
</tr>
<tr>
<td>Employee 2</td>
<td>269</td>
<td>171</td>
<td>145</td>
<td>0.8479</td>
<td>0.63</td>
<td>1.12</td>
</tr>
<tr>
<td>Employee 3</td>
<td>267</td>
<td>138</td>
<td>103</td>
<td>0.7463</td>
<td>0.51</td>
<td>1.17</td>
</tr>
<tr>
<td>Boss</td>
<td>241</td>
<td>128</td>
<td>112</td>
<td>0.8682</td>
<td>0.53</td>
<td>1.11</td>
</tr>
</tbody>
</table>

Table 8.18. Person movement prediction using a dynamically trained MLP with confidence automata

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NM</th>
<th>NP</th>
<th>NCP</th>
<th>PA</th>
<th>Quantity</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>113</td>
<td>80</td>
<td>74</td>
<td>0.9250</td>
<td>0.70</td>
<td>1.05</td>
</tr>
<tr>
<td>Employee 2</td>
<td>269</td>
<td>171</td>
<td>147</td>
<td>0.8596</td>
<td>0.63</td>
<td>1.13</td>
</tr>
<tr>
<td>Employee 3</td>
<td>267</td>
<td>151</td>
<td>112</td>
<td>0.7417</td>
<td>0.56</td>
<td>1.07</td>
</tr>
<tr>
<td>Boss</td>
<td>241</td>
<td>128</td>
<td>110</td>
<td>0.8593</td>
<td>0.53</td>
<td>1.16</td>
</tr>
</tbody>
</table>

Table 8.19. Person movement prediction using a statically pre-trained dynamic MLP with confidence automata

The column Gain from Tables 8.18 and 8.19 was obtained dividing the prediction accuracies from these tables (obtained with confidence automata) to those from Tables 8.15 and respectively 8.16 (obtained without confidence automata). The results show that the accuracy is higher when the prediction is generated using confidence automata, but in the same time the column Quantity shows that the number of predictions is significantly lower.

**Prediction Using Elman Recurrent Neural Predictors**

Further is studied the possibility of person movement prediction using the Elman recurrent neural network [Elm90], presented in Figure 8.9. Each room has associated a 4-state local confidence automaton like that presented in
Figure 8.12. We used a prediction table, which is addressed with the code of the last visited room. The next room is predicted only if the selected confidence automaton is in the strongest predictable state. We introduced the notation ELMAN (N=n, M=m, $\alpha=\alpha$, T=t) in order to represent a certain configuration of the Elman predictor. The above notation indicates that predictions are generated using an Elman neural network with $n$ input layer neurons, $m$ context neurons, $m$ hidden layer neurons, a learning rate $\alpha$ and in the case of pre-training a threshold $t$. The number of output layer neurons ($P$) is equal to the number of rooms used in the prediction process (see Table 8.1), thus, the neural network has in its output layer one neuron for each room and it predicts the room corresponding to the neuron with the highest output value.

The first varied parameter is the number of neurons in the hidden layer ($M$). We used the complete prediction mechanism of a dynamically trained ELMAN (N=4, $\alpha=0.1$) predictor on the long benchmarks. Table 8.20 shows how is affected the prediction accuracy by the number of neurons in the hidden layer for one iteration in the dynamic training process. It can be observed that the optimal number of hidden layer (context) neurons is 4.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$M=1$</th>
<th>$M=2$</th>
<th>$M=3$</th>
<th>$M=4$</th>
<th>$M=5$</th>
<th>$M=6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>94.11</td>
<td>93.58</td>
<td>93.50</td>
<td>93.50</td>
<td>93.50</td>
<td>93.50</td>
</tr>
<tr>
<td>Employee 2</td>
<td>82.09</td>
<td>81.59</td>
<td>81.43</td>
<td>81.25</td>
<td>81.13</td>
<td>80.39</td>
</tr>
<tr>
<td>Employee 3</td>
<td>72.79</td>
<td>72.79</td>
<td>73.68</td>
<td>76.00</td>
<td>75.80</td>
<td>75.40</td>
</tr>
<tr>
<td>Boss</td>
<td>89.11</td>
<td>89.04</td>
<td>89.04</td>
<td>89.04</td>
<td>89.04</td>
<td>88.35</td>
</tr>
<tr>
<td>Average</td>
<td>84.52</td>
<td>84.25</td>
<td>84.41</td>
<td><strong>84.94</strong></td>
<td>84.86</td>
<td>84.41</td>
</tr>
</tbody>
</table>

Table 8.20. Prediction accuracies obtained for different numbers of hidden layer neurons ($M$) using 4-state confidence automata – reported after a single iteration

The learning rate is varied using a dynamically trained complete ELMAN (N=4, $M=4$) predictor with 4-state confidence automata on the long benchmarks. The results were reported after a single iteration. Table 8.21 and Figure 8.13 show how is affected the prediction accuracy by the learning rate. It can be observed that this time the optimal learning rate is 0.1.
Prediction accuracies obtained for different learning rates ($\alpha$) using 4-state confidence automata – reported after one iteration

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$\alpha$ =0.1</th>
<th>$\alpha$ =0.2</th>
<th>$\alpha$ =0.3</th>
<th>$\alpha$ =0.4</th>
<th>$\alpha$ =0.5</th>
<th>$\alpha$ =0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>93.50</td>
<td>93.15</td>
<td>92.20</td>
<td>90.76</td>
<td>88.13</td>
<td>85.00</td>
</tr>
<tr>
<td>Employee 2</td>
<td>81.25</td>
<td>81.34</td>
<td>79.19</td>
<td>79.22</td>
<td>79.60</td>
<td>80.71</td>
</tr>
<tr>
<td>Employee 3</td>
<td>76.00</td>
<td>75.20</td>
<td>73.68</td>
<td>71.65</td>
<td>73.33</td>
<td>72.46</td>
</tr>
<tr>
<td>Boss</td>
<td>89.04</td>
<td>88.96</td>
<td>90.22</td>
<td>88.61</td>
<td>90.38</td>
<td>87.50</td>
</tr>
<tr>
<td>Average</td>
<td><strong>84.94</strong></td>
<td>84.66</td>
<td>83.82</td>
<td>82.56</td>
<td>82.86</td>
<td>81.41</td>
</tr>
</tbody>
</table>

Table 8.21. Prediction accuracies obtained for different learning rates ($\alpha$) using 4-state confidence automata – reported after one iteration

Figure 8.13. Prediction accuracies obtained for different learning rates ($\alpha$) using 4-state confidence automata – reported after one iteration

The threshold’s value is varied using a statically pre-trained dynamic ELMAN (N=4, M=4, $\alpha$=0.1) predictor with confidence automata and complete prediction mechanism. The dynamically trained predictor is initialized with the weights generated by the static learning process. The results are reported after one iteration of the benchmarks. For these simulations we used the short benchmarks in the static training process and respectively the long benchmarks in the dynamic prediction process. Table 8.22 shows how is affected the prediction accuracy by the threshold’s value used in the static training process. It can be observed that better results were obtained without pre-training. The reason is that if the recurrent network is pre-trained, it learns with a certain threshold the last movements from the
Neural Networks Models with Applications in Ubiquitous Computing. Next Location Prediction

benchmark in the static learning process. In the dynamic training process the recurrent network has to adapt to the new movements, and it adapts easier if the weights are randomly initialized.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>T=0.1</th>
<th>T=0.3</th>
<th>T=0.5</th>
<th>T=0.7</th>
<th>T=0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>93.10</td>
<td>92.42</td>
<td>90.54</td>
<td>93.15</td>
<td>90.66</td>
</tr>
<tr>
<td>Employee 2</td>
<td>77.19</td>
<td>77.84</td>
<td>79.64</td>
<td>79.76</td>
<td>77.45</td>
</tr>
<tr>
<td>Employee 3</td>
<td>72.41</td>
<td>75.70</td>
<td>70.50</td>
<td>70.31</td>
<td>70.28</td>
</tr>
<tr>
<td>Boss</td>
<td>86.77</td>
<td>88.32</td>
<td>88.40</td>
<td>89.55</td>
<td>87.94</td>
</tr>
<tr>
<td>Average</td>
<td>82.36</td>
<td>83.57</td>
<td>82.27</td>
<td>83.19</td>
<td>81.58</td>
</tr>
</tbody>
</table>

Table 8.22. Prediction accuracies obtained for different thresholds (T) – reported after a single iteration

Further we extracted again from the presented previous results the prediction accuracies obtained when the prediction process is simplified (binary prediction). Table 8.23 and Figure 8.14 compare a dynamic ELMAN (N=4, M=4, $\alpha=0.1$) predictor with a statically pre-trained dynamic MLP (N=8, M=N+1, $\alpha=0.2$, T=0.3) predictor, both with binary prediction mechanism and 4-state confidence automata. We used again the short benchmarks in the static training process and respectively the long benchmarks in the run-time prediction process. The results were reported after a single iteration of each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Elman</th>
<th>MLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>93.50</td>
<td>92.64</td>
</tr>
<tr>
<td>Employee 2</td>
<td>82.11</td>
<td>87.32</td>
</tr>
<tr>
<td>Employee 3</td>
<td>73.68</td>
<td>78.09</td>
</tr>
<tr>
<td>Boss</td>
<td>89.04</td>
<td>89.62</td>
</tr>
<tr>
<td>Average</td>
<td>84.58</td>
<td><strong>86.91</strong></td>
</tr>
</tbody>
</table>

Table 8.23. Comparing the MLP with the Elman recurrent network
It can be observed that better results were obtained with the multi-layer perceptron in terms of prediction accuracy (see Figure 8.14). The costs of the approach (time and memory size) are the following [Gel06]:

**Time costs**
For static learning the neural network needs about 10 to 45 seconds to learn an entire short benchmark, using a Pentium III, f(CLK)=650 MHz, 128 MB DRAM. The dynamic learning is practically instantaneous because a single backward step is used.

**Memory costs**
For a local predictor with a room history length of 2 (H=2), codifying the room with 4 bits (B=4), we have N=B*H=8, M=B*H+1=9, P=B (N/M/P - the number of input/hidden/output layer neurons). For this optimal configuration of the neural network, the system needs 168 memory units (160 float value memory units, and 8 bits for the input vector). More generally, the memory costs (C) are given by the following formula:

\[
C = M(N+B) + P(M+B) + N - \text{the number of memory units}
\]
\[
C_F = M(N+B)+P(M+B) - \text{float value memory units}
\]
\[
C_B = N - \text{the number of memory units necessary to store 1 or -1 (1 bit)}
\]
The time and memory costs of the MLP approach presented in this work show again that this method is a realistic and attractive solution for PDAs.

8.6 SOME CONCLUSIONS AND FURTHER WORK IDEAS

In conclusion, after a short introduction in ubiquitous area and in autonomic computing concept, in this chapter we analyzed neural prediction techniques used in an ubiquitous computing application. In ubiquitous environments often relatively simple prediction algorithms are required e.g. due to the PDA’s memory, computing, and communication restrictions. We used in this work one of the simplest neural networks, a multi-layer perceptron with one hidden layer, trained with back-propagation algorithm. Also we used a recurrent Elman network. Two predictor types were analyzed: the local and respectively the global predictors. In the case of local predictors, each person has his/her own neural predictor and in this way each neural network will be trained with the movements of a single person. Alternatively, it is possible to use one global neural network for all persons, and in this second case the persons must be codified, too. The evaluations show that the local predictors have higher prediction accuracy than the global predictors, probably due to benchmarks interferences. Therefore, the global predictor was not able to exploit potential useful correlations between different persons’ movements. We found a formula for determining the optimal number of neurons in the hidden layer as a function of the neurons’ number in the input layer; we could consider that the optimal number of hidden layer neurons $M = N+1$, whereas $N$ is number of input layer neurons. The neural network is more efficient when only one backward step is applied in the run-time prediction process. The next varied parameter was the learning rate. The evaluations show that the learning rate of 0.1 is near to optimal. More important, after we fixed all these parameters, we studied how the prediction accuracy is affected by the room history length (and implicitly by the number of neurons in the input layer). The simulation results show that the optimal room history length is 2. We continued our study implementing a statically trained neural network. The last parameter we varied is the threshold used in the static training process. For this we used a dynamic neural predictor, which was statically learned before it’s effective use. The results show that the optimal threshold is 0.1.
We extracted from the presented previous results the prediction accuracy obtained using a simplified prediction process. We compared the dynamic predictor with the statically trained dynamic predictor. The experimental results show that the pre-trained dynamic predictors are more efficient than the dynamic predictors. The arithmetical mean of the prediction accuracies obtained with the pre-trained local predictors is 89.81%, but the prediction accuracy measured on some local predictors grew up to over than 92%. For an efficient evaluation, the static training process was made with some summer (short) benchmarks, and in the run-time prediction process were used fall (long) benchmarks. One of the further development directions is to compare the neural predictors presented in this work with other neural predictors (which might exploit some inter-benchmarks correlations) and with the state predictor techniques with exactly the same experimental setup.

References


9. HIDDEN MARKOV MODELS WITH APPLICATIONS IN UBIQUITOUS SYSTEMS.
NEXT LOCATION PREDICTION

9.1. APPLICATIONS BASED ON HIDDEN MARKOV MODELS

During this chapter we’ll focused on the same problem like in the previous one: predicting the next room based on the history of rooms, visited by a certain person moving within an office building. In fact, comparing with the previous chapter, we are developing now a complete different solution to the same problem problem. For doing this, we developed and evaluated some Hidden Markov Model (HMM) predictors by some movement sequences of real persons developed by the research group at the University of Augsburg [6]. Before presenting this new approach we’ll briefly presented below some HMM related works in context prediction in ubiquitous computing field.

Liu et al. in their work [3], describe a HMM based framework for hand gesture detection and recognition. The goal of gesture interpretation is to improve human-machine communication and to bring human-machine interaction closer to human-human interaction, making possible new applications such as sign language translation. They present an efficient method for extracting the observation sequence using the feature model and Vector Quantization, and demonstrate that, compared to the classic template-based methods, the HMM-based approach offers a more flexible framework for recognition.

Galata et al. [2], present a novel approach for automatically acquiring stochastic models of the high-level structure of a human activity without the assumption of any prior knowledge. The process involves temporal segmentation into plausible atomic behavior components and the use of
variable length Markov models for the efficient representation of behaviors. Their experimental results demonstrate that the use of variable length Markov models provides an efficient mechanism for learning complex behavioral dependencies and constraints.

Machine Learning techniques based on HMMs have been also applied to problems in computational biology and they can be used as mathematical models of molecular processes and biological sequences. The goal of computational biology is to elucidate additional information required for drug design, medical diagnosis and medical treatment. The majority of molecular data used in computational biology consists in sequences of nucleotides corresponding to the primary structure of DNA and RNA, or sequences of amino acids corresponding to the primary structure of proteins. Birney [1] reviews gene-prediction HMMs and protein family HMMs. The role of gene-prediction in DNA is to discover the location of genes on the genome. HMMs have also been used in protein profiling to discriminate between different protein families and predict a new protein-family or subfamily. Yoon et. al. in their work [10], proposed a new method based on context-sensitive HMMs, which can be used for predicting RNA secondary structure. The RNA secondary structure results from the base pairs formed by the nucleotides of RNA. The context-sensitive HMM can be viewed as an extension of the traditional HMM, where some of the states are equipped with auxiliary memory. Symbols that are emitted at certain states are stored in the memory, and they serve as the context that affects the emission and transition probabilities of the model. They demonstrated that the proposed model predicts the secondary structure very accurately, at a low computational cost.

L. Rabiner in his seminal paper [7] shows how HMMs can be applied to selected problems in speech recognition. His paper presents the theory of HMMs from the simplest concepts (discrete Markov chains, see below the next paragraph) to the most sophisticated models (variable duration, continuous density models, recognition of isolated/connected words, etc.). He also illustrated some applications of the theory of HMMs to simple problems in speech recognition, and pointed out how the HMM techniques have been applied to more advanced speech recognition problems. Other HMM research approaches are focused on web pages pre-fetching by user’s next URL prediction, in order to decrease the long time waiting due to the network’s bandwidth limitations and therefore to improve the quality of web servers.
The following paragraphs are focused on a Hidden Markov Model (HMM) prediction approach, introducing the HMM-based predictors. The presented application predicts the next room based on the history of rooms, visited by a certain person moving within an office building. The developed predictors are evaluated by some movement sequences of real persons, acquired from the Smart Doorplates project developed at Augsburg University [4, 5].

9.2 DISCRETE MARKOV PROCESSES

We consider a discrete process with N states $S_1, S_2, \ldots, S_N$. At a certain moment $t$, the process transits from the current state $S_i(t)$ into the next state $S_j(t+1)$, according with the transition matrix probabilities $a_{ij}$, $i, j \in \{1, 2, \ldots, N\}$. In a Markov process of order $k$, $k \in \{1, 2, \ldots, N\}$ the next state transition depends on the previous $k$ states. In other words, there are the following conditional probabilities:

$$P[S_j(t+1) | S_i(t), S_l(t-1), \ldots, S_m(t-k+1)]$$

where $i, j, l, m \in \{1, 2, \ldots, N\}$.

For a first order Markov process, the transition conditional probabilities are:

$$a_{ij} = P[S_j(t+1) | S_i(t)], \quad 1 \leq i, j \leq N;$$

Notations: $p(S_i) = p_i$, $|p| = [p_1, p_2, \ldots, p_N]$ and $|A| = \sum_{j=1}^{N} a_{ij}$.

We can write the following formulas:

$$|p| = |p| \cdot |A| \quad \text{si}$$

$$\sum_{j=1}^{N} a_{ij} = 1 \quad \text{(Equilibrium relation)}$$

A stochastic process like the previous one is called observable process because its states $S_i$ are corresponding to the observable events. As an example, we consider a three state $S_1, S_2, S_3$ Markov process with the associated transition matrix:

$$A = \begin{bmatrix}
0.4 & 0.3 & 0.3 \\
0.2 & 0.6 & 0.2 \\
0.1 & 0.1 & 0.8
\end{bmatrix}$$
Taking into account the proposed Markov model, the problem is to determine the probability of the observable states sequence \( O = \{S_3, S_3, S_3, S_1, S_1, S_3, S_2, S_3\} \) knowing that \( P[S_3(t = 1)] = 1 \). The probability of sequence \( O \) is:

\[
P(O|Model) = P[S_3, S_3, S_3, S_1, S_1, S_3, S_2, S_3|Model] = P[S_3|S_3] \cdot P[S_3|S_3] \cdot P[S_3|S_3].
\]

\[
P[S_3|S_3] \cdot P[S_3|S_1] \cdot P[S_3|S_2] \cdot P[S_3|S_2] = 1 \cdot a_{33} \cdot a_{33} \cdot a_{31} \cdot a_{13} \cdot a_{32} \cdot a_{23} =
\]

\[
= 1 \cdot (0.8) \cdot (0.8) \cdot (0.1) \cdot (0.4) \cdot (0.3) \cdot (0.1) \cdot (0.2) = 1.536 \cdot 10^{-4}
\]

Another problem is the following: knowing that the Markov process is in a certain state \( S_i \), calculate the probability of being in this state exactly \( d \) time instances. Therefore:

\[
O = \{S_i(t), S_i(t + 1), ..., S_i(t + d - 1), S_i(t + d) \neq S_i(t + d - 1)\}
\]

The requested probability is:

\[
P(O|Model, S_i(t)) = (a_{ii})^{d-1} \cdot (1-a_{ii}) = p_i(d)
\]

A natural further problem is the following: how many cycles is staying a Markov process in a certain state \( S_i \)? The answer is given by the following formula formula:

\[
\overline{d}_i = \sum_{d=1}^{\infty} d \cdot p_i(d) = \sum_{d=1}^{\infty} d (a_{ii})^{d-1} \cdot (1-a_{ii}) = \frac{1}{1-a_{ii}}
\]

Particularly for the considered process we have the following values:

\[
\overline{d}_1 = 1.67, \overline{d}_2 = 2.5 \text{ and } \overline{d}_3 = 5(\text{cycles})
\]
Prediction principle based on a Markov process

Considering an order one Markov process being in the current state $S_i(t)$, the next predicted state is $S_i(t + 1)$ if \( \text{Max}\{a_{i,j}\}_j = a_{i,l} \), where \( i, j, l \in \{1, 2, ..., N\} \).

Note: The probabilities \( a_{ij} \) are modified after each cycle according to the real transition. If the prediction was correct (\( S_i \)) then \( a_{i,l} \) is growing up and the other transition probabilities are decreasing; else (the next state is \( S_m, m \neq l \)) \( a_{i,m} \) is growing up and the others transition probabilities are correspondingly decreased. Therefore the model is dynamically trained.

Example:
Considering the following observable states sequence:

\[ S_1, S_1, S_1, S_2, S_3, S_1, S_1, S_2, S_3, S_1, S_1, S_1, ... \]

The problem is, what state is coming next?

In order to give the answer using an order one Markov model, it is necessary to calculate the probabilities \( a_{i1} \), \( a_{i2} \) and \( a_{i3} \). Taking into account the given sequence, the corresponding transition frequencies are:
\[ f_{11} = 6, \quad f_{12} = 2, \quad f_{13} = 0 \]
equivalent with statistical probabilities \( a_{11} = \frac{3}{4}, a_{12} = \frac{1}{4}, a_{13} = 0 \).

Therefore the next state prediction is \( S_1 \) (it seems to be a miss-prediction because according to the intrinsic sequence correlation the correct prediction might be \( S_2 \)). Considering an order two Markov process, it is predicted \( S_1 \), too (analogous, it is searched in the history string what states are following after the sequence \( S_1S_1 \)). Only an order 3 Markov model will correctly predict the next state (\( S_2 \)) because:

\[
\begin{align*}
P[S_1 | S_1, S_1, S_1] &= 0 \\
P[S_2 | S_1, S_1, S_1] &= 1 \text{ (frequency 2)} \\
P[S_3 | S_1, S_1, S_1] &= 0
\end{align*}
\]

Therefore, the model’s order is essential in predicting with a simple discrete Markov model. A greater order involves the possibility of detecting deeper correlations into the history observable states. It is very important to compute the transitions in a saturated manner in order to easily adapt to some quick radical behavior’s changes.

![Figure 9.2. A 3-primitive state Markov model of 3rd order](image-url)
A Markov model of order $k$ having $N$ primitive states involves $N^k$ possible combinations of length $k$ of these $N$ states. In this case, considering the current super-state ($i$) consisting by the last $k$ primitive states $i \in \{1,2,\ldots,N^k\}$, it is computed $\text{Max}\{a_{ij}\} = a_{ij}$. $(\forall) \ j \in \{1,2,\ldots,N\} \Rightarrow S_j$ is the next predicted primitive state. A Markov process is defined as being ergodic if from any current state it can transit to any other state in a finite number of steps (All $a_{ij} > 0$). If the requested finite transitions number is one step, the process is called strong-ergodic (full connected graph).

### 9.3 Hidden Markov Models of Order 1

Discrete Markov Prediction Models are characterized by the fact that each state corresponds to an observable event. There are many problems where this prediction model is not effective. In this sense, it can be considered the case where the observation is a probabilistic function of the state. This involve a more refined model called Hidden Markov Model, that is a doubly stochastic process with an underlying stochastic process that is hidden, but produce in a probabilistic manner the sequence of observations. In other words, an underground Markov process represents “the hidden semantic process” that generates the observable stochastic (also Markov!) process. Therefore, one key problem in HMMs consists in “uncovering” the underground semantic stochastic process that generates the observable events. As an intuitive example, we consider a (long) sequence of colored balls (there are $M$ distinct colors of the balls) very poor predictable with a simple Markov model. Why? Because in underground, the next observable ball is selected from one of the $N$ urns (each urn contains a big number of colored balls), according to a simple stochastic Markov process. Each of the $N$ states belonging to the hidden Markov process has attached a corresponding observable Markov process, representing a certain sub-string of the whole observable string events. As a consequence, it would be possible to be easier to correctly predict the next observable event knowing its corresponding sub-string rather than not (thus knowing only the whole observable string). It seems be obvious that, if we could discover the hidden Markov process that stochastically generates the observable events (balls), the prediction accuracy could be increased. Unfortunately, in real prediction problems we aren’t
know the hidden stochastic process, but – as it might be further shown – it will be sufficient to guess its number of states (N).

**Elements of a HMM of order 1**

1. **N** - the number of hidden states, with \( S = \{S_0, S_1, \ldots, S_{N-1} \} \) the set of hidden states, and \( q_t \) the hidden state at time \( t \). N will be varied in order to obtain the optimal value. Although the states are hidden, for some applications there is often some physical significance attached to the states. Generally, each state can be reached from any other static (an ergodic model)

2. **M** - the number of observable states, with \( V = \{V_0, V_1, \ldots, V_{M-1} \} \) the set of observable states (symbols), and \( O_t \) the observable state at time \( t \).

3. **A** = \( \{a_{ij}\} \) - the transition probabilities between the hidden states \( S_i \) and \( S_j \), where \( a_{ij} = P(q_{t+1} = S_j | q_t = S_i), \quad 0 \leq i, j \leq N - 1 \).

4. **B** = \( \{b_j(k)\} \) - the probabilities of the observable states \( V_k \) in hidden states \( S_j \), where \( b_j(k) = P(O_t = V_k | q_t = S_j), \quad 0 \leq j \leq N - 1, \quad 0 \leq k \leq M - 1 \).

5. **\( \pi \)** = \( \{\pi_i\} \) - the initial hidden state probabilities, where \( \pi_i = P(q_1 = S_i), \quad 0 \leq i \leq N - 1 \).

We also define the following variables:

- \( \alpha_t(i) = P(O_t, O_{t+1} \ldots O_T, q_t = S_i | \lambda) \) - the forward variable [7], representing the probability of the partial observation sequence until time \( t \), and hidden state \( S_i \) at time \( t \), given the model \( \lambda = (A, B, \pi) \).

- \( \beta_t(i) = P(O_{t+1}, O_{t+2} \ldots O_T | q_t = S_i, \lambda) \) - the backward variable [7], representing the probability of the partial observation sequence from \( t+1 \) to the end \( T \), given hidden state \( S_i \) at time \( t \) and the model \( \lambda = (A, B, \pi) \).

- \( \xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j | O_t, O_{t+1} \ldots O_T, \lambda) \) - the probability of being in hidden state \( S_i \) at time \( t \), and hidden state \( S_j \) at time \( t+1 \), given the model \( \lambda = (A, B, \pi) \) and the observation sequence.

- \( \gamma_t(i) = P(q_t = S_i | O_t, O_{t+1} \ldots O_T, \lambda) \) - the probability of being in hidden state \( S_i \) at time \( t \), given the model \( \lambda = (A, B, \pi) \) and the observation sequence.

- **H** - the history (the number of observations used in the prediction process). In [7] and [8] the entire observation sequence is used in the prediction process (H=T), but in some practical applications the observation sequence increases continuously, therefore it is necessary its
limitation. Thus, the last $H$ observations can be stored in a left shift register.

- $I$ - the maximum number of iterations in the adjustment process. Usually the adjustment process ends when the probability of the observation sequence doesn’t increase anymore, but for a faster adjustment, it is limited the number of iterations.

**Adjustment process of a HMM of order 1**

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

**Initialization of the model of order 1**

- The transition probabilities between the hidden states $A(NxN) = \{a_{ij}\}$, are randomly initialized to approximately $1/N$, each row summing to 1 (equilibrium condition).
- The probabilities of the observable states $B(NxM) = \{b_{j}(k)\}$, are randomly initialized to approximately $1/M$, each row summing to 1.
- The initial hidden state probabilities $\pi(1xN) = \{\pi_i\}$ are randomly set to approximately $1/N$, their sum being 1.

**9.4 PREDICTION ALGORITHM USING A HMM OF ORDER 1**

1.) $T=1$ (T is the length of the observation sequence);
2.) $T=T+1$;
   if $T<H$ go to 2.)
3.) $c=0$ (c is the number of current iteration, its maximum value is given by $I$);
4.) The model $\lambda = (A, B, \pi)$ is repeatedly adjusted based on the last $H$ observations $O_{t-H+1}, O_{t-H+2}, ..., O_{t}$ (the entire observation sequence if...
H=T), in order to increase the probability of the observation sequence 
$P(O_{T-H+1} O_{T-H+2} \ldots O_T | \lambda)$. Below, in 4.1, 4.2 and 4.3 further steps, the
denominators are used in order to obtain a probability measure, and to
avoid underflow. As Stamp showed in [8], underflow is inevitable
without scaling, since the probabilities tend to 0 exponentially as $T$
increases.

4.1. Compute the forward variable $\alpha$ in a recursive manner:

\[
\alpha_{T-H+1}(i) = \frac{\pi_i \cdot b_i(O_{T-H+1})}{\sum_{j=0}^{N-1} \pi_j \cdot b_j(O_{T-H+1})}, \quad i = 0, \ldots, N-1,
\]

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

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

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

\[
\alpha_t(j) = P(O_{T-H+1} O_{T-H+2} \ldots O_T, q_T = S_j | \lambda),
\]

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

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

4.2. Compute the backward variable $\beta$ in a recursive manner:

\[
\beta_T(i) = \frac{1}{\sum_{j=0}^{N-1} \sum_{i=0}^{N-1} \alpha_{T-j}(i) \cdot a_{ij} \cdot b_j(O_T)}, \quad i = 0, \ldots, N-1;
\]
\[ \beta_i(t) = \frac{\sum_{j=0}^{N-1} a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)}{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)}, \quad t = T-1, \ldots, T-H+1, \quad i = 0, \ldots, N-1 \]

where

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

4.3. Compute \( \xi \):

\[ \xi_i(i, j) = \frac{\alpha_i(i) \cdot a_{ij} \cdot b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \alpha_i(i) \cdot a_{ij} \cdot b_j(O_{t+1}) \beta_{t+1}(j)}, \quad t = T-H+1, \ldots, T-1, \quad i = 0, \ldots, N-1 \]

\[ j = 0, \ldots, N-1 \]

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

4.4. Compute \( \gamma \):

\[ \gamma_i(i) = \sum_{j=0}^{N-1} \xi_i(i, j), \quad t = T-H+1, \ldots, T-1, \quad i = 0, \ldots, N-1 \]

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

4.5. Adjust \( \pi \):

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

4.6. Adjust \( A \):

\[ a_{ij} = \frac{\sum_{t=H+1}^{T-1} \xi_i(i, j)}{\sum_{t=H+1}^{T-1} \gamma_i(i)} - \text{the probability of transition from hidden state } S_i \text{ to } S_j. \] The numerator represents the expected number of transitions
from state $S_i$ to $S_j$, while the denominator represents the expected number of transitions from state $S_i$ to any state.

4.7. Adjust $B$:

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

4.8. $c=c+1$;

if $\log[P(O_{T-H+1}...O_T|\tilde{\lambda})] > \log[P(O_{T-H+1}...O_T|\hat{\lambda})]$ and $c<I$ then go to 4.).

Since $P$ would be out of the dynamic range of the machine [7], we compute the log of $P$, using the following formula [8]:

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

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

- choose hidden state $S_i$ at time $T$, $i=0, ..., N-1$, maximizing $\alpha_T(i)$;
- choose next hidden state $S_j$ (at time $T+1$), $j=0, ..., N-1$, maximizing $\overline{a_{ij}}$;
- predict next symbol $V_k$ (at time $T+1$), $k=0, ..., M-1$, maximizing $\overline{b_j(k)}$.

If the process continues, then $T=T+1$ and go to 3.).

The model parameters $(A, B, \pi)$ are adjusted in order to maximize the probability of the observation sequence $P(O|\hat{\lambda})$. The model $\hat{\lambda} = (A, B, \pi)$ can
be chosen such that \( P(O|\lambda) \) is globally maximized using an iterative procedure, or using gradient techniques like the Lagrange multipliers method. In this document we use the Baum-Welch iterative method introduced by Baum et al. [Baum72]. The Baum-Welch algorithm – identical to the Expectation Maximization (EM) method for this particular problem – improves iteratively an initial model. If we define the current model as \( \lambda = (A, B, \pi) \) and use it to compute the re-estimated model \( \lambda = (\bar{A}, \bar{B}, \bar{\pi}) \) – through steps 4.5, 4.6 and 4.7 from the prediction algorithm presented above – then, as it has been proven by Baum, the model \( \lambda \) is more likely than model \( \lambda \) in the sense that \( P(O|\lambda) \geq P(O|\lambda) \). Thus, if \( \lambda \) is used iteratively in place of \( \lambda \) repeating the re-estimation calculation, the probability of the observation sequence can be improved until some limiting point is reached. Rabiner shows in [Rab89] that the same re-estimation formulas can be obtained using the techniques of Lagrange multipliers.

9.5 A POSSIBLE GENERALIZATION: HIDDEN MARKOV MODELS OF ORDER R>1

In this paragraph we present a Hidden Markov Model of order R, R≥1 based on [11,12]. 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 that the generic primitive hidden state becomes now a generic super-state. The elements of a HMM of order R are the followings:

1. R - the order of HMM (the number of primitive hidden states based on a transition is made).
2. N - the number of primitive hidden states, with \( S = \{ S_0, S_1, ..., S_{N^R-1} \} \) the set of hidden states (sequences of \( R \) primitive hidden states), and \( q_t \) the hidden state at time \( t \). \( N \) will be varied in order to obtain the optimal value.

3. M - the number of observable states, with \( V = \{ V_0, V_1, ..., V_{M-1} \} \) the set of observable states (symbols), and \( O_t \) the observable state at time \( t \).

4. \( A = \{ a_{ij} \} \) - the transition probabilities between the hidden states \( S_i \) and \( S_j \), where \( a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \quad 0 \leq i, j \leq N^R - 1. \)

5. \( B = \{ b_j(k) \} \) - the probabilities of the observable states \( V_k \) in hidden states \( S_j \), where \( b_j(k) = P(O_t = V_k | q_t = S_j], \quad 0 \leq j \leq N^R - 1, \quad 0 \leq k \leq M - 1. \)

6. \( \pi = \{ \pi_i \} \) - the initial hidden state probabilities, where \( \pi_i = P[q_0 = S_i], \quad 0 \leq i \leq N^R - 1. \)

We also define the following variables:

- \( \alpha_t(i) = P(O_1, O_2, ..., O_t, q_t = S_i | \lambda) \) - the forward variable [7], representing the probability of the partial observation sequence until time \( t \), and hidden state \( S_i \) at time \( t \), given the model \( \lambda = (A, B, \pi) \).
- \( \beta_t(i) = P(O_{t+1}, O_{t+2}, ..., O_T | q_T = S_i, \lambda) \) - the backward variable [7], representing the probability of the partial observation sequence from \( t+1 \) to the end \( T \), given hidden state \( S_i \) at time \( t \) and the model \( \lambda = (A, B, \pi) \).
- \( \xi_t(i, j) = P(q_t = S_i, q_{t+1} = S_j | O_1, O_2, ..., O_T, \lambda) \) - the probability of being in hidden state \( S_i \) at time \( t \), and hidden state \( S_j \) at time \( t+1 \), given the model \( \lambda = (A, B, \pi) \) and the observation sequence.
- \( \gamma_t(i) = P(q_t = S_i | O_1, O_2, ..., O_T, \lambda) \) - the probability of being in hidden state \( S_i \) at time \( t \), given the model \( \lambda = (A, B, \pi) \) and the observation sequence.
- \( H \) - the history (the number of observations used in the prediction process). In [7] and [8] the entire observation sequence is used in the prediction process \( (H=T) \), but in some practical applications the observation sequence increases continuously, therefore it is necessary its limitation. Thus, the last \( H \) observations can be stored in a left shift register.
- \( I \) - the maximum number of iterations in the adjustment process. Usually the adjustment process ends when the probability of the observation sequence doesn’t increase anymore, but for a faster adjustment, it is limited the number of iterations.
For a HMM of order $R$ with $N$ hidden states, the transition probabilities between the hidden states $A(N^R \times N^R) = \{a_{ij}\}$, are stored in a table with $N^R$ rows and $N^R$ columns but not all cells of the table are used; there are only $N$ consistent transitions from each state. The following table, for example, corresponds to a HMM of order 3 ($R=3$) with 2 hidden states ($N=2$):

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>AAA</td>
<td>AAB</td>
<td>ABA</td>
<td>ABB</td>
<td>BAA</td>
<td>BAB</td>
<td>BBA</td>
<td>BBB</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1. Consistent transitions for a HMM of order 3 ($R=3$), with 2 hidden states ($N=2$)

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

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

- For current hidden states (rows) $i = 0, ..., N^R - 1$, are consistent only the next hidden states (columns)
  
  $$j = (i \mod N^{R-1}) \cdot N, \ldots, (i \mod N^{R-1}) \cdot N + N - 1.$$

### Adjustment process of a HMM of order R

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

**Initialization of the model of order R**

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

**9.6 PREDICTION ALGORITHM USING A HMM OF ORDER R>1**

1.) $T=1$ (T is the length of the observation sequence);
2.) $T=T+1$;
   if $T<H$ goto 2.)
3.) $c=0$ (c is the number of current iteration, its maximum value is given by $I$);
4.) The model $\lambda = (A, B, \pi)$ is repeatedly adjusted on the last $H$ observations $O_{T-H+1}, O_{T-H+2}, \ldots, O_T$ (the entire observation sequence if $H=T$), in order to increase the probability of the observation sequence $P(O_{T-H+1}, O_{T-H+2} \ldots O_T|\lambda)$. In 4.1, 4.2 and 4.3 the denominators are used in order to obtain a probability measure, and to avoid underflow. As Stamp showed in [8], underflow is inevitable without scaling, since the probabilities tend to 0 exponentially as $T$ increases.
4.1. Compute the forward variable $\alpha$ in a recursive manner:
\[ \alpha_{T-H+1}(i) = \frac{\pi_i \cdot b_i(O_{T-H+1})}{\sum_{i=0}^{N^R-1} \alpha_{T-H+1}(i)} \quad i = 0, ..., N^R - 1 \]

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

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

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

\[ \alpha_r(j) = P(O_{T-H+1} O_{T-H+2} \ldots O_T | q_T = S_j | \lambda), \]

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

\[ P(O_{T-H+1} O_{T-H+2} \ldots O_T | \lambda) = \sum_{j=0}^{N^R-1} \alpha_r(j). \]

4.2. Compute the backward variable \(\beta\) in a recursive manner:

\[ \beta_T(i) = \frac{1}{\sum_{j=0}^{N^R-1} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(O_T)} \quad i = 0, ..., N^R - 1; \]

\[ \beta_T(i) = \frac{\sum_{j=(i \mod N^R)N+N+1}^{j=(i \mod N^R)N+N-1} a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)}{\sum_{j=0}^{N^R-1} \sum_{i=0}^{N^R-1} a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)} \quad t = T - 1, ..., T - H + 1, \quad i = 0, ..., N^R - 1, \]
where $\beta_t(i)$ is the probability of the partial observation sequence from $t+1$ to the end $T (O_{t+1}O_{t+2}...O_T)$, given hidden state $S_i$ at time $t$ and the model $\lambda = (A, B, \pi)$.

4.3. Compute $\xi$:

$$\xi_t(i, j) = \frac{\alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{j=(i \mod N^{R-1}) \cdot N+1}^{(i \mod N^{R-1}) \cdot N+N-1} \sum_{j=(i \mod N^{R-1}) \cdot N}^{(i \mod N^{R-1}) \cdot N+N-1} \alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1}) \beta_{t+1}(j)}$$

for $t = T - H + 1, ..., T - 1$, $i = 0, ..., N^R - 1$, $j = (i \mod N^{R-1}) \cdot N, ..., (i \mod N^{R-1}) \cdot N + N - 1$,

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

4.4. Compute $\gamma$:

$$\gamma_t(i) = \sum_{j=(i \mod N^{R-1}) \cdot N}^{(i \mod N^{R-1}) \cdot N+N-1} \xi_t(i, j)$$

for $t = T - H + 1, ..., T - 1$, $i = 0, ..., N^R - 1$,

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

4.5. Adjust $\pi$:

$$\overline{\pi_t} = \gamma_{T-H+1}(i)$$

- expected number of times the hidden state is $S_i$ ($i = 0, ..., N^R - 1$) at the initial time $t = T - H + 1$.

4.6. Adjust $A$:

$$a_{ij} = \frac{\sum_{t=T-H+1}^{T-1} \xi_t(i, j)}{\sum_{t=T-H+1}^{T-1} \gamma_t(i)}$$

- the probability of transition from hidden state $S_i$ to $S_j$, where $i = 0, ..., N^R - 1$ and $j = (i \mod N^{R-1}) \cdot N, ..., (i \mod N^{R-1}) \cdot N + N - 1$.

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

4.7. Adjust $B$:
\[
\sum_{j=T-H+1}^{T-1} \gamma_j(j) - \sum_{j=T-H+1}^{T-1} \gamma_j(j)
\]

- the probability of observation symbol \( V_k \)

\( k = 0, \ldots, M - 1 \) given that the model is in hidden state \( S_j \)

\( j = 0, \ldots, N^R - 1 \). The numerator is the expected number of times
the model is in hidden state \( S_j \) and the observation symbol is \( V_k \),
while the denominator is the expected number of times the model is
in hidden state \( S_j \).

4.8. \( c = c + 1; \)

if \( \log[P(O_{T-H+1} \ldots O_T | \hat{\lambda})] > \log[P(O_{T-H+1} \ldots O_T | \lambda)] \) and \( c < I \) then

go to 4.).

Since \( P \) would be out of the dynamic range of the machine [7], we
compute the log of \( P \), using the following formula [8]:

\[
\log[P(O_{T-H+1} \ldots O_T | \hat{\lambda})] = -\log \left( \frac{1}{\sum_{i=0}^{N^R} \pi_i \cdot b_i(O_{T-H+1})} \right) - \sum_{i=0}^{N^R} \log \left( \frac{1}{\sum_{i=0}^{N^R} \sum_{j=0}^{N^R} a_{i} \cdot a_{j} \cdot b_{j}(O_{i})} \right)
\]

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

- choose hidden state \( S_i \) at time \( T \), \( i = 0, \ldots, N^R - 1 \), maximizing \( \alpha_{T}(i) \);
- choose next hidden state \( S_j \) (at time \( T+1 \),
  \( j = (i \mod N^R) \cdot N, \ldots, (i \mod N^R) \cdot N + N - 1 \), maximizing \( a_{j} \);
- predict next symbol \( V_k \) (at time \( T+1 \), \( k = 0, \ldots, M - 1 \), maximizing
  \( b_{j}(k) \).

If the process continues, then \( T = T + 1 \) and go to 3.).
9.7 METHODOLOGY AND EXPERIMENTAL RESULTS

Our application predicts the next room based on the history of rooms, visited by a certain person moving within an office building. We evaluate these HMM predictors by some movement sequences of real persons developed by the research group at the University of Augsburg [6]. During this presentation we are interested in predicting the next room from all rooms except for the own office. The presented results are based on our previous work [11, 12].

We started with a HMM of order 1 with 2 hidden states (N=2), and we tested it on the fall benchmarks [6] without corridor. The following simulation methodology was used in order to predict each room from a certain benchmark: we predicted room at time $t$ based on the entire room sequence from that benchmark until time $t-1$. We compared a HMM without confidence automata with HMMs using different confidence automata. We denoted as $n$ rooms $m$ states conf an $m$-state confidence counter, associated to each sequence of the last $n$ rooms visited by the person. The 4-state automaton has 2 predictable states, while the 2-state automata has only one predictable state. In the case of using 4-state automata, a prediction is generated in each of the two predictable states. Table 9.2 shows that the best average prediction accuracy (AM) is obtained when we use a 4-state confidence for each sequence of 2 rooms:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>no conf</th>
<th>2 rooms 4 states conf</th>
<th>2 rooms 2 states conf</th>
<th>1 room 4 states conf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>91.89</td>
<td>92.68</td>
<td>90.8</td>
<td>93.33</td>
</tr>
<tr>
<td>Employee 2</td>
<td>77.9</td>
<td>84.06</td>
<td>86.41</td>
<td>79.24</td>
</tr>
<tr>
<td>Employee 3</td>
<td>65.66</td>
<td>73.82</td>
<td>71.05</td>
<td>69.35</td>
</tr>
<tr>
<td>Boss</td>
<td>79.07</td>
<td>84.72</td>
<td>84.07</td>
<td>84.35</td>
</tr>
<tr>
<td>AM</td>
<td>78.63</td>
<td>83.82</td>
<td>83.0825</td>
<td>81.5675</td>
</tr>
</tbody>
</table>

Table 9.2. Comparing a HMM without confidence with HMMs using different types of confidence

We continued our simulations varying the number of hidden states. We used again the fall benchmarks [6], and a HMM of order 1 with 4-state confidence automata associated to each sequence of two rooms. Table 9.3 shows how is affected the prediction accuracy of a HMM by the number of hidden states:
Table 9.3. Study of the number of hidden states using HMM with 4-state confidence automata

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N=1</th>
<th>N=2</th>
<th>N=3</th>
<th>N=4</th>
<th>N=5</th>
<th>N=6</th>
<th>N=7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>92.68</td>
<td>92.68</td>
<td>92.68</td>
<td>92.68</td>
<td>92.68</td>
<td>92.68</td>
<td>92.68</td>
</tr>
<tr>
<td>Employee 2</td>
<td>84.15</td>
<td>84.06</td>
<td>84</td>
<td>84.57</td>
<td>82.84</td>
<td>82.45</td>
<td>83.13</td>
</tr>
<tr>
<td>Employee 3</td>
<td>73.37</td>
<td>73.82</td>
<td>74.65</td>
<td>74.63</td>
<td>76.55</td>
<td>73.91</td>
<td>74.04</td>
</tr>
<tr>
<td>Boss</td>
<td>84.72</td>
<td>84.72</td>
<td>85.6</td>
<td>85.49</td>
<td>87.17</td>
<td>85.82</td>
<td>85.6</td>
</tr>
<tr>
<td>AM</td>
<td>83.73</td>
<td>83.82</td>
<td>84.2325</td>
<td>84.3425</td>
<td>84.81</td>
<td>83.715</td>
<td>83.8625</td>
</tr>
</tbody>
</table>

It can be observed in table 9.3 that for a HMM of order 1, the optimal number of hidden states is 5. Our scientific hypothesis in this particular research is that the hidden states are the five working days of week. We varied again the number of hidden states without using the confidence automata. Table 9.4 shows that in this case the optimal number of hidden states is 1.

Table 9.4. Study of the number of hidden states using HMM without confidence automata

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N=1</th>
<th>N=2</th>
<th>N=3</th>
<th>N=4</th>
<th>N=5</th>
<th>N=6</th>
<th>N=7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>91.89</td>
<td>91.89</td>
<td>91.89</td>
<td>91.89</td>
<td>91.89</td>
<td>91.89</td>
<td>90.09</td>
</tr>
<tr>
<td>Employee 2</td>
<td>79.77</td>
<td>77.9</td>
<td>76.77</td>
<td>76.02</td>
<td>75.65</td>
<td>74.53</td>
<td>75.65</td>
</tr>
<tr>
<td>Employee 3</td>
<td>70.18</td>
<td>65.66</td>
<td>64.9</td>
<td>64.15</td>
<td>65.28</td>
<td>61.13</td>
<td>64.9</td>
</tr>
<tr>
<td>Boss</td>
<td>79.49</td>
<td>79.07</td>
<td>72.38</td>
<td>71.12</td>
<td>72.8</td>
<td>71.54</td>
<td>71.12</td>
</tr>
<tr>
<td>AM</td>
<td>80.33</td>
<td>78.63</td>
<td>76.48</td>
<td>75.79</td>
<td>76.4</td>
<td>74.77</td>
<td>75.44</td>
</tr>
</tbody>
</table>

We compare now the best HMM of order one with different configurations of some “equivalent” NNs (Neural Networks) presented in our previous chapter and simple Markov predictors [9]. “Equivalent” means in this case that we compare all these different predictors considering that they have, however, the same inputs. We used the HMM of order one with 5 hidden states considering an attached 4-state confidence automata. We compared it with a statically pre-trained NN with a room history length of one (NN of order one) and 2 (NN of order 2), a learning rate of 0.2 and a threshold of 0.3 (having the same confidence automata). We also compared the HMM with some simple Markov predictors of order 1 and 2 (having the same confidence automata). The NN was statically trained on the summer benchmarks [6], and each predictor was tested on the fall benchmarks [6]. Table 9.5 presents the prediction accuracies obtained with the HMM, NN and Markov predictors, all with 4-state confidence automata associated to each sequence of two rooms:
Hidden Markov Models with Applications in Ubiquitous Systems. Next Location Prediction

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HMM of order 1 (N=5)</th>
<th>NN of order 1</th>
<th>NN of order 2</th>
<th>Markov of order 1</th>
<th>Markov of order 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>92.68</td>
<td>92.4</td>
<td>92.5</td>
<td>93.42</td>
<td>93.33</td>
</tr>
<tr>
<td>Employee 2</td>
<td>82.84</td>
<td>84.35</td>
<td>85.96</td>
<td><strong>83.43</strong></td>
<td>84.33</td>
</tr>
<tr>
<td>Employee 3</td>
<td>76.55</td>
<td>73.91</td>
<td>74.17</td>
<td>73.75</td>
<td>75</td>
</tr>
<tr>
<td>Boss</td>
<td>87.17</td>
<td>86.71</td>
<td>85.93</td>
<td>85.82</td>
<td>85.71</td>
</tr>
<tr>
<td>AM</td>
<td><strong>84.81</strong></td>
<td>84.34</td>
<td>84.64</td>
<td>84.1</td>
<td>84.59</td>
</tr>
</tbody>
</table>

**Table 9.5.** Comparing a HMM of order 1 with NN and Markov predictors using 4-state confidence automata

Employee 2 hasn’t got a behavior correlated with days (see table 9.3 and also table 9.5). It can be seen in table 9.5 too, because the Markov predictor of order one outperforms the HMM of order one. NNs are better than HMM on Employee 2, but we suspect that these results are too optimistic due to the NN’s pre-training (first case) and respectively due to the NN’s order 2 (second case). Table 9.6 presents the prediction accuracies obtained with the HMM, NN and simple Markov predictors (the same configuration) without confidence automata:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HMM of order 1 (N=1)</th>
<th>NN of order 1</th>
<th>NN of order 2</th>
<th>Markov of order 1</th>
<th>Markov of order 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee 1</td>
<td>91.89</td>
<td>89.18</td>
<td>90.09</td>
<td>81.98</td>
<td>75.67</td>
</tr>
<tr>
<td>Employee 2</td>
<td>79.77</td>
<td>77.15</td>
<td>76.4</td>
<td>74.15</td>
<td>70.03</td>
</tr>
<tr>
<td>Employee 3</td>
<td>70.18</td>
<td>62.26</td>
<td>67.54</td>
<td>64.9</td>
<td>56.98</td>
</tr>
<tr>
<td>Boss</td>
<td>79.49</td>
<td>76.56</td>
<td>75.73</td>
<td>73.64</td>
<td>62.34</td>
</tr>
<tr>
<td>AM</td>
<td><strong>80.33</strong></td>
<td>76.28</td>
<td>77.44</td>
<td>73.66</td>
<td>66.25</td>
</tr>
</tbody>
</table>

**Table 9.6.** Comparing a HMM of order 1 with NN and Markov predictors without confidence automata

In order to confirm or infirm our scientific hypothesis that, in the case of HMM with 4-state confidence automata, the five hidden states might be the working days of week, we implemented a predictor which consists of five simple Markov predictors. Each Markov predictor is associated to one of the 5 days (Monday, Tuesday, ..., Friday). Table 9.7 compares this predictor containing 5 simple Markov predictors with HMMs and simple Markov predictors, using the 4-state confidence automata, and shows that only Boss’ movements are confirming our hypothesis.
Prediction Techniques in Advanced Computing Architectures

Table 9.7. Comparing a HMM of order 1 with Markov of order 1 and a predictor consisting in 5 Markov predictors using 4-state confidence automata

Table 9.8 compares the same predictors (HMM, Markov, 5 Markov predictors) without using the confidence automata. In this case, however, the best number of hidden states was 1. We obtained (for all benchmarks) lower prediction accuracies when we used 5 simple Markov predictors, even related to the simple Markov predictor.

Table 9.8. Comparing a HMM of order 1 with Markov of order 1 and a predictor consisting in 5 Markov predictors without using confidence automata

In order to increase prediction latency, we pre-trained HMMs on the summer benchmarks [6] and after that we tested them on the fall benchmarks [6]. Table 9.9 presents comparatively the obtained results when we used HMM with 5 hidden states and 4-state confidence automata:

Table 9.9. Comparing simple HMMs with pre-trained HMMs, both with 4-state confidence automata, in terms of prediction latency
As it can be observed, with pre-trained HMMs we obtain better prediction latencies (about 85 ms) but for Employee 3 and Boss we obtained lower prediction accuracies (!). The same prediction accuracies were obtained using the untrained HMM with only one hidden state (see table 9.3). In our opinion, through pre-training, the HMM with 5 hidden states uses only one of the hidden states. We repeated the same simulations without using the confidence automata. Table 9.10 presents the results for HMM with 5 hidden states and table 9.11 presents the results for HMM with 1 hidden state:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HMM of order 1 (N=5)</th>
<th>Pre-trained HMM of order 1 (N=5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prediction accuracy [%]</td>
<td>Total time [ms]</td>
</tr>
<tr>
<td>Employee 1</td>
<td>91.89</td>
<td>118701</td>
</tr>
<tr>
<td>Employee 2</td>
<td>75.65</td>
<td>652768</td>
</tr>
<tr>
<td>Employee 3</td>
<td>65.28</td>
<td>620322</td>
</tr>
<tr>
<td>Boss</td>
<td>72.8</td>
<td>519497</td>
</tr>
<tr>
<td>AM</td>
<td>76.4</td>
<td>477822</td>
</tr>
</tbody>
</table>

Table 9.10. Comparing simple HMMs with pre-trained HMMs, both without confidence automata, in terms of prediction latency (N=5)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HMM of order 1 (N=1)</th>
<th>Pre-trained HMM of order 1 (N=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prediction accuracy [%]</td>
<td>Total time [ms]</td>
</tr>
<tr>
<td>Employee 1</td>
<td>91.89</td>
<td>471</td>
</tr>
<tr>
<td>Employee 2</td>
<td>79.77</td>
<td>2393</td>
</tr>
<tr>
<td>Employee 3</td>
<td>70.18</td>
<td>2333</td>
</tr>
<tr>
<td>Boss</td>
<td>79.49</td>
<td>1963</td>
</tr>
<tr>
<td>AM</td>
<td>80.33</td>
<td>1790</td>
</tr>
</tbody>
</table>

Table 9.11. Comparing simple HMMs with pre-trained HMMs, both without confidence automata, in terms of prediction latency (N=1)

As it can be seen in table 9.10, for a HMM with 5 hidden states through pre-training we obtained better prediction latencies and better prediction accuracies. It is interesting that we obtained the same prediction accuracies using the untrained HMM with one hidden state (see table 9.6, N=1). And it is more interesting that for HMM with one hidden state, after pre-training, the prediction accuracy is the same again (!). These results encourage our
previous conclusion, that the best number of hidden states is 1, when we don’t use confidence automata. Through pre-training, the HMM with 5 hidden states and without confidence automata, uses only one of the hidden states.

The next parameter we varied was the order of HMM. We studied HMMs of different orders and the same number of hidden states (N=2). As it can be observed in table 9.12, with a first order HMM were obtained the best prediction accuracies:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>With 4-state confidence</th>
<th>Without confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R=1</td>
<td>R=3</td>
</tr>
<tr>
<td>Employee 1</td>
<td>92.68</td>
<td>83.11</td>
</tr>
<tr>
<td>Employee 2</td>
<td>84.06</td>
<td>64.06</td>
</tr>
<tr>
<td>Employee 3</td>
<td>73.82</td>
<td>63.63</td>
</tr>
<tr>
<td>Boss</td>
<td>84.72</td>
<td>76.15</td>
</tr>
<tr>
<td>AM</td>
<td>83.82</td>
<td>71.73</td>
</tr>
</tbody>
</table>

Table 9.12. Studying HMMs of different orders

9.8 SOME CONCLUSIONS

In this chapter we analyzed machine-learning techniques based on HMMs, used in a context prediction in an ubiquitous computing application. Our concrete particular goal was to predict accurately the movements of persons within an office building. Two predictor types were analyzed: HMMs with confidence automata and respectively without confidence automata. The experimental results show that HMMs outperform other implemented prediction techniques such as Perceptrons and Markov predictors. The evaluations show that the simplest configuration of HMM (N=1 and R=1, equivalent with a simple Markov model of order 0!) is the most accurately for this application. We continued our study implementing a statically trained HMM and we obtained lower prediction latencies.

Predicting from all rooms except the own room and using a HMM with 4-state confidence automata, we obtained an average prediction accuracy of 84.81%, but the prediction accuracy measured on some local predictors grew up to over than 92%. As a further work we intend to use our NN and HMM developed predictors in other context prediction in ubiquitous computing
applications (e.g., next cell movements in GSM communications based on Nokia’s benchmarks or next web page prediction).

References


Professor Lucian N. Vintan, PhD, was born at August 27th 1962 in Cisnădie, Romania. He graduated University “Politehnica” of Timișoara, obtaining a M.Sc. in Computer Engineering, and a Ph.D. in Advanced Computer Architecture from the same university. At present he is a full-time Professor (2000) and a Ph.D. Supervisor in Computer Science (2001) at “Lucian Blaga” University of Sibiu, Romania. He published over 60 scientific papers in some prestigious reviews (ISI Thomson Journals, IEEE, IEE, ACM, etc.) and international top conferences from Romania, USA, UK, Portugal, Italy, Austria, Czech Republic, Hungary, Poland, Germany, China, Slovenia, Latvia, Spain, etc. Some of his papers were highly cited in a lot of valuable works published in world-top reviews (IEEE Trans. on Comp, ACM Trans. On CS, ACM Trans. On A&CO, IEEE Micro, IEE Proc. Comp. Digit. Techn., Journal of ILP, etc.) or conferences like ISCA, HPCC, MICRO, etc. (for example his “neural dynamic branch prediction”, “pre-computed branches” or “value prediction focused on CPU’s context” innovative proposals). Prof. L. N. Vintan is author of three Computer Architecture textbooks and coauthor of other two, published in Romanian Academy, Technical and Matrix Rom publishing houses (Bucharest). With one of these books he receives The Romanian Academy “Tudor Tănăsescu” Prize in 2005. He developed as a Project Manager 10 research grants obtained through competition. For his professional merits he was elected in 2005, Member of The Romanian Academy of Technical Sciences (ASTR). He is a European Commission DG Information Society Expert (FP6 NoE HiPEAC program’s reviewer), Visiting Research Fellow at University of Hertfordshire, UK, member of Romanian National Academic Research Council’s Engineering Committee, etc. Prof. Vintan is taking part in several peer-review committees and prestigious Computer Engineering conferences scientific boards (EuroPar, IASTED, HPC&S, ACSAC, ARCS, etc.) and also was invited professor at some universities from UK, Germany, Spain, NL, etc. Details at URL: http://webspace.ulbsibiu.ro/lucian.vintan