



**FEDERAL UNIVERSITY OF FRONTEIRA SUL
CAMPUS OF CHAPECÓ
COURSE OF COMPUTER SCIENCE**

DANIEL MELLO KONFLANZ

**INVESTIGATING HIERARCHICAL TEMPORAL MEMORY NETWORKS APPLIED
TO DYNAMIC BRANCH PREDICTION**

**CHAPECÓ
2019**

DANIEL MELLO KONFLANZ

**INVESTIGATING HIERARCHICAL TEMPORAL MEMORY NETWORKS APPLIED
TO DYNAMIC BRANCH PREDICTION**

Final undergraduate work submitted as requirement to
obtain a Bachelor's degree in Computer Science from the
Federal University of Fronteira Sul.
Advisor: Luciano Lores Caimi

CHAPECÓ
2019

Konflanz, Daniel Mello

Investigating Hierarchical Temporal Memory networks applied to dynamic branch prediction / Daniel Mello Konflanz. – 2019.

84 pp.: il.

Advisor: Luciano Lores Caimi.

Final undergraduate work – Federal University of Fronteira Sul, course of Computer Science, Chapecó, SC, 2019.

1. Dynamic branch prediction. 2. Neural branch predictor. 3. Hierarchical Temporal Memory. 4. HTM sequence memory. I. Caimi, Luciano Lores, advisor. II. Federal University of Fronteira Sul. III. Title.

© 2019

All rights reserved to Daniel Mello Konflanz. This work or any portion thereof may not be reproduced without citing the source.

E-mail: danielmk98@hotmail.com

DANIEL MELLO KONFLANZ

**INVESTIGATING HIERARCHICAL TEMPORAL MEMORY NETWORKS APPLIED
TO DYNAMIC BRANCH PREDICTION**

Final undergraduate work submitted as requirement to obtain a Bachelor's degree in Computer Science from the Federal University of Fronteira Sul.

Advisor: Luciano Lores Caimi

This final undergraduate work was defended and approved by the examination committee on: December 5, 2019.


EXAMINATION COMMITTEE



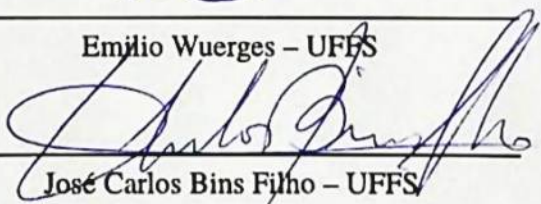
Luciano Lores Caimi – UFFS



Adriano Sanick Padilha – UFFS



Emilio Wuerges – UFFS



José Carlos Bins Filho – UFFS

ACKNOWLEDGEMENTS

I am very thankful to everyone who has somehow prevented me from not finishing this work.

ABSTRACT

As a consequence of the high application of instruction-level parallelism techniques in modern processors, the branch prediction area of study remains relevant after 40 years of research. This work applies neural networks based on the Hierarchical Temporal Memory (HTM) theory to the branch prediction task and explores their adequacy to the problem's characteristics. More specifically, the problem is faced as a sequence prediction task and tackled by the HTM sequence memory. Four traditional branch prediction schemes adapted to operate with an HTM system and two variations of the previous designs were evaluated on a slice of the traces provided by the 4th Championship Branch Prediction contest. The leading result was achieved by the HTM predictor based on the gshare branch predictor, that for 8 million instructions was able to improve the misprediction rate by 14.3% in comparison to its traditional 2-bit counters version when both used a 13-bit history length. However, high levels of aliasing were found to prevent the HTM system to scale and compete against larger conventional branch predictors.

Keywords: Dynamic branch prediction. Neural branch predictor. Hierarchical Temporal Memory. HTM sequence memory.

LIST OF FIGURES

| | |
|--|----|
| Figure 1 – A control hazard in a classic five-stage RISC pipeline | 12 |
| Figure 2 – Time-series graph of the number of papers that mention "branch prediction" or "branch predictor" in titles or abstracts of documents from the computing area in the Scopus database | 15 |
| Figure 3 – The 2-bit saturating counter branch predictor | 24 |
| Figure 4 – The two-level local branch predictor | 25 |
| Figure 5 – Example of code with correlated branches | 26 |
| Figure 6 – Example of in-path correlation | 27 |
| Figure 7 – The gshare branch predictor | 28 |
| Figure 8 – The path-based correlating branch predictor | 29 |
| Figure 9 – Hybrid predictor with a 2-bit selector table | 30 |
| Figure 10 – The operation of a perceptron predictor | 32 |
| Figure 11 – The TAGE branch predictor | 34 |
| Figure 12 – A set of bits paired with ranges of values that overlap with neighbors | 41 |
| Figure 13 – The encoded representation of the number 7.0 | 41 |
| Figure 14 – A mapping of active bits from an infinite SDR to a finite SDR | 42 |
| Figure 15 – A category encoding of parts of speech | 43 |
| Figure 16 – A pyramidal neuron with labeled structures | 44 |
| Figure 17 – The HTM Neuron Model | 45 |
| Figure 18 – Sequences represented in the HTM sequence memory | 48 |
| Figure 19 – Predictions in the HTM sequence memory | 49 |
| Figure 20 – Scheme of an HTM system for sequence learning tasks | 50 |
| Figure 21 – Artificial data set created to test HTM sequence memory properties | 51 |
| Figure 22 – The general HTM branch predictor design | 52 |
| Figure 23 – Essential operation of the CBP-4 evaluation framework | 55 |
| Figure 24 – Operation of the HTM global branch predictor | 59 |
| Figure 25 – Trend for the misprediction rate of the HTM global branch predictor | 60 |
| Figure 26 – Trend for the misprediction rate of the HTM gselect branch predictor | 61 |
| Figure 27 – Operation of the HTM gshare branch predictor | 62 |
| Figure 28 – Trend for the misprediction rate of the HTM gshare branch predictor | 63 |
| Figure 29 – Operation of the HTM streams branch predictor | 64 |
| Figure 30 – Trend for the misprediction rate of the HTM streams branch predictor | 64 |
| Figure 31 – Operation of the HTM gshare-streams branch predictor | 65 |
| Figure 32 – Trend for the misprediction rate of the HTM gshare-streams branch predictor | 66 |
| Figure 33 – Operation of the HTM gshare-block branch predictor with 2 predictions per iteration | 67 |
| Figure 34 – Trend for the misprediction rate of the HTM gshare-block branch predictor | 67 |

| | |
|--|----|
| Figure 35 – Trend for the misprediction rate of the HTM gshare branch predictor over 8 million instructions | 68 |
| Figure 36 – Trend for the misprediction rate of the HTM gshare-streams branch predictor over 8 million instructions | 69 |
| Figure 37 – Trend for the misprediction rate of the HTM gshare branch predictor with resetting | 70 |
| Figure 38 – Trend for the misprediction rate of the HTM gshare branch predictor with varying number of categories | 71 |
| Figure 39 – Trend for the misprediction rate of the HTM gshare branch predictor over 4 million instructions with a context switch | 72 |
| Figure 40 – Comparison of MPKI scores between HTM and non-HTM designs using a 13-bit history length representation of execution states, for 2 million instructions | 73 |

ABBREVIATIONS

ANNs Artificial Neural Networks.

BHR branch history register.

BP branch prediction.

BTB Branch-Target Buffers.

CBP Championship Branch Prediction.

HTM Hierarchical Temporal Memory.

ILP instruction-level parallelism.

MPKI mispredictions per thousand instructions.

PC program counter.

PHR path history register.

PHT pattern history table.

SDR Sparse Distributed Representation.

SP HTM Spatial Pooler.

SPEC Standard Performance Evaluation Corporation.

CONTENTS

| | | |
|--------------|--|-----------|
| 1 | INTRODUCTION | 12 |
| 1.1 | PROBLEM STATEMENT | 14 |
| 1.2 | OBJECTIVES | 14 |
| 1.3 | JUSTIFICATION | 14 |
| 1.4 | STUDY LIMITATIONS | 16 |
| 1.5 | DOCUMENT ORGANIZATION | 17 |
| 2 | LITERATURE REVIEW | 18 |
| 2.1 | BRANCH PREDICTION | 18 |
| 2.1.1 | Instruction-level parallelism and branch prediction | 18 |
| 2.1.1.1 | The importance of speculative execution and branch prediction | 18 |
| 2.1.1.2 | Out-of-order multiple-issue processors | 19 |
| 2.1.2 | Static and dynamic branch prediction | 20 |
| 2.1.2.1 | Static branch prediction | 20 |
| 2.1.2.2 | Dynamic branch prediction | 21 |
| 2.1.3 | Types of branch instructions and how to predict them | 21 |
| 2.1.4 | Challenges in the design of dynamic branch predictors | 22 |
| 2.1.5 | Main dynamic branch prediction techniques | 23 |
| 2.1.5.1 | Predicting branches from local history | 23 |
| 2.1.5.1.1 | <i>The two-level local branch predictor</i> | 24 |
| 2.1.5.2 | Correlating branch predictors | 26 |
| 2.1.5.2.1 | <i>The two-level global branch predictor</i> | 27 |
| 2.1.5.2.2 | <i>Global predictor with index selection</i> | 27 |
| 2.1.5.2.3 | <i>Global history with index sharing</i> | 28 |
| 2.1.5.2.4 | <i>Path-based correlating branch predictor</i> | 28 |
| 2.1.5.3 | Combining global and local branching histories | 29 |
| 2.1.5.4 | Combining predictors | 29 |
| 2.1.5.5 | Other techniques to improve prediction accuracy | 30 |
| 2.1.5.6 | State-of-the-art branch predictors | 31 |
| 2.1.5.6.1 | <i>The perceptron branch predictor</i> | 31 |
| 2.1.5.6.2 | <i>The TAGE branch predictor</i> | 32 |
| 2.1.6 | Tolerating prediction latency and making multiple predictions per cycle | 34 |
| 2.2 | HIERARCHICAL TEMPORAL MEMORY | 36 |
| 2.2.1 | Artificial Neural Networks and neuroscience | 36 |
| 2.2.2 | HTM theory overview | 36 |
| 2.2.2.1 | The structure of the neocortex | 37 |
| 2.2.2.2 | HTM principles | 37 |
| 2.2.2.3 | HTM applications | 38 |

| | | |
|--------------|--|-----------|
| 2.2.3 | Sparse Distributed Representations | 38 |
| 2.2.3.1 | Semantic meaning in SDRs | 39 |
| 2.2.3.2 | SDR capacity | 39 |
| 2.2.3.3 | Matching SDRs | 39 |
| 2.2.3.4 | Efficiently matching SDRs | 40 |
| 2.2.4 | Encoding data into SDRs | 40 |
| 2.2.4.1 | A simple scalar encoder | 41 |
| 2.2.4.2 | An unbounded scalar encoder | 42 |
| 2.2.4.3 | Category encoders | 42 |
| 2.2.5 | The HTM Neuron Model | 43 |
| 2.2.5.1 | Pyramidal neurons | 43 |
| 2.2.5.2 | Pyramidal neuron operation according to the HTM theory | 44 |
| 2.2.6 | HTM Spatial Pooler | 46 |
| 2.2.7 | HTM Sequence Memory | 46 |
| 2.2.7.1 | Network structure and operation | 47 |
| 2.2.7.2 | Learning transitions of SDRs | 49 |
| 2.2.7.3 | Network parameters | 49 |
| 2.2.7.4 | Extracting predictions from the network | 50 |
| 2.2.8 | An HTM system for prediction tasks | 50 |
| 3 | GENERAL OPERATION OF AN HTM BRANCH PREDICTOR | 52 |
| 3.1 | CONSTRAINTS OF OPERATION | 52 |
| 3.2 | GENERAL STRATEGY | 52 |
| 3.3 | DETAILS OF IMPLEMENTATION | 53 |
| 4 | RESEARCH METHODS | 54 |
| 4.1 | EVALUATION ENVIRONMENT | 54 |
| 4.2 | CHARACTERIZATION AND PATHWAY OF EXPERIMENTS | 55 |
| 4.3 | HTM SYSTEM IMPLEMENTATION | 56 |
| 4.3.1 | Parameters selection and tuning | 56 |
| 4.4 | SDR CLASSIFIER SELECTION MECHANISM | 57 |
| 5 | RESULTS | 59 |
| 5.1 | SINGLE PREDICTION DESIGNS | 59 |
| 5.1.1 | Global branch predictor | 59 |
| 5.1.2 | Gselect branch predictor | 60 |
| 5.1.3 | Gshare branch predictor | 62 |
| 5.2 | MULTIPLE PREDICTIONS DESIGNS | 63 |
| 5.2.1 | Streams branch predictor | 63 |
| 5.2.2 | Gshare-streams branch predictor | 65 |
| 5.2.3 | Gshare-block branch predictor | 66 |
| 5.3 | DESIGN EXPLORATION | 68 |

| | | |
|--------------|---------------------------------------|-----------|
| 5.3.1 | Longer traces | 68 |
| 5.3.2 | Network resetting | 69 |
| 5.3.3 | Limit of categories | 70 |
| 5.3.4 | SDR encoder seeds | 71 |
| 5.3.5 | Context switch recovery | 71 |
| 6 | DISCUSSION | 73 |
| 6.1 | PREDICTOR ANALYSIS | 73 |
| 6.2 | HARDWARE REQUIREMENTS AND ISSUES | 75 |
| 6.2.1 | Learning with delayed feedback | 76 |
| 7 | CONCLUSIONS | 77 |
| | REFERENCES | 78 |

1 INTRODUCTION

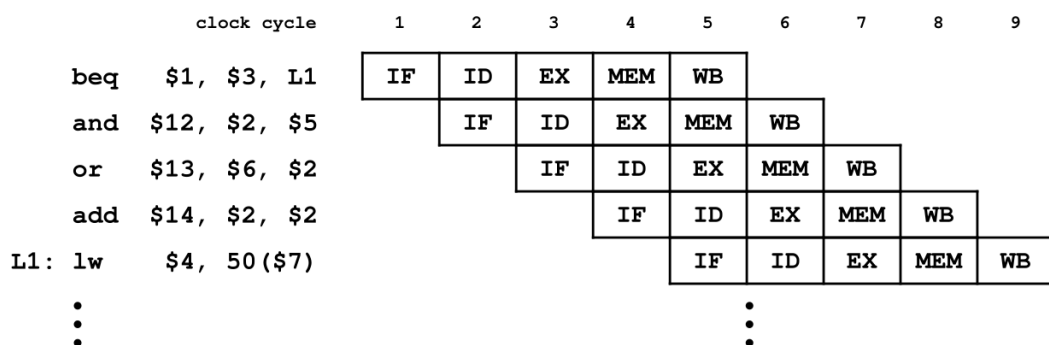
Processor architectures have evolved increasingly complex in their path to provide more computing power. One of the first explored design techniques to speed up computing was instruction-level parallelism (ILP), with early proposals being made in the 1940s and 1950s and initial commercial implementation in the 1960s. Such technique aims to give the processor the ability to execute individual instructions in parallel while keeping sequential behavior unchanged for developers (RAU; FISHER, 1993).

A vital ILP technique, ubiquitous in modern CPUs, is instruction pipelining. Its core idea is to overlap the execution of multiple instructions by breaking the processing of a single instruction into n sequential stages, thus allowing many instructions to be in execution simultaneously since each instruction occupies only one execution stage at a time. This partitioning can ideally allow an increase by a factor of n in the clock cycle rate, providing an equivalent acceleration in processing time (D. M. HARRIS; S. L. HARRIS, 2013).

The efficiency of pipelining, as well as the performance of other ILP techniques, depends on the possibility of maintaining the stages or execution units occupied most of the time. This parallelism, however, is limited by situations called *hazards*, that “prevent the next instruction in the instruction stream from executing during its designated clock cycle” (HENNESSY; PATTERSON, 2006) due to some dependency. Therefore, the implementation of such techniques is not trivial.

Hennessy and Patterson (2006) name three major classes of hazards: *structural hazards*, where the hardware cannot handle a combination of instructions due to the lack of execution units; *data hazards*, where the execution of an operation depends on the result of instructions not yet computed; and *control hazards*, where the following instruction address is unknown due to the ongoing execution of a branch instruction.

Figure 1 – A control hazard in a classic five-stage RISC pipeline



Source: Adapted from Santambrogio and Campanoni (2014)

Figure 1 shows an example of a control hazard in a pipelined processor with five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and Register Writeback (WB) (D. M. HARRIS; S. L. HARRIS, 2013). The *beq* instruction compares the values of two registers and skips the execution to *LI* if they are equal. Assume that this comparison is concluded only at the end of the 3rd clock cycle, although additional hardware could anticipate the result. Then, in the beginning of the 2nd and 3rd clock cycles the processor is not able to determine which instructions to push into the pipeline.

A simple solution to deal with control hazards is to simply *stall* the pipeline, i.e., to stop the fetching of new instructions and wait for the required result. This procedure, however, defeats some of the effectiveness of ILP (D. M. HARRIS; S. L. HARRIS, 2013). An often preferred strategy is to hypothesize about the result of branch instructions and to continue execution in a guessed direction. This method, referred as *hardware speculation*, present another set of problems since it requires the capacity of flushing in-execution instructions and undoing their effects in order to deal with flawed guesses (HENNESSY; PATTERSON, 2006). To justify the implementation of such hardware overhead, the correctness rate for these estimates is very relevant. Hence, methods for achieving better guesses were developed. These guidance techniques can be either static, when estimates for a given branch are based on fixed heuristics, or dynamic, when they are based on the code execution data. Dynamic techniques can also be classified as adaptive if they use data from the *current* execution to make better guesses (YEH; PATT, 1992). For dynamic and adaptive assistance, specialized hardware units called *branch predictors* have been designed. Usually, adaptability is implicit for dynamic branch predictors.

Dynamic branch predictors have been widely explored (MITTAL, 2018). The basic premise for building branch predictors is the existence of historical correlations of branch outcomes which can be identified and exploited. These correlations may exist either locally, i.e., when “branches can be predicted by their own execution pattern only” (MITTAL, 2018); globally, when “execution of previous branches can provide a hint about the outcome of a candidate branch” (MITTAL, 2018); or in a combination of both.

The most basic dynamic branch predictor scheme that works with local history is a *branch-prediction buffer*, a predictor that stores only one bit per branch representing the last outcome of that instruction (HENNESSY; PATTERSON, 2006). Even this simple strategy may achieve prediction accuracy of more than 90% in many scientific programs (J. E. SMITH, 1998). These good results, however, did not prevent the exploration and implementation of more efficient and complex predictors, like the neural inspired perceptron-based predictors (JIMÉNEZ; C. LIN, 2001), which are considered state-of-the-art techniques (GOPE; LIPASTI, 2014).

Encouraged by the successful results of the perceptron predictor, this work seeks to explore the use of biologically *constrained* artificial intelligence techniques for dynamic branch prediction. Notedly, it intends to study a neural branch predictor built with components of the Hierarchical Temporal Memory (HTM) theory, which aims to elucidate how biological

intelligence arises in the neocortex and has prediction as one of its core principles (HAWKINS; AHMAD; PURDY, et al., 2016).

1.1 PROBLEM STATEMENT

To the authors' knowledge, the Hierarchical Temporal Memory (HTM) theory had not yet been tested in the dynamic branch prediction task. Therefore, this study aimed to find the potential efficiency of HTM algorithms applied to the problem, since this was a basic research topic that needed to be explored in order to make further investigations of the proposal worthwhile. Primarily, it was expected that the HTM technology could offer good results as a consequence of its predictor component ability to remember large context chains and to quickly adapt to new patterns of data (CUI; AHMAD; HAWKINS, 2016). These properties, however, needed to be verified in the proposed environment.

As a secondary research question, this document intended to briefly analyze the implementation feasibility of HTM theory based branch predictors. The low latency requirement of the problem demands the neural mechanisms exploited by the HTM algorithms to be implemented in proper neuromorphic hardware, i.e., hardware with a brain-inspired architecture (SCHUMAN et al., 2017). Development in this area is still recent (BILLAUDELLE; AHMAD, 2015; PUENTE; ÁNGEL GREGORIO, 2016; STREAT; KUDITHIPUDI; GOMEZ, 2016; PUTIC; VARSHNEYA; STAN, 2017; WALTER et al., 2017; ZYARAH; KUDITHIPUDI, 2019a; 2019b), but progress in neuromorphic computing, such as in optical implementations (FELDMANN et al., 2019), could lead to the viability of increasingly larger networks.

1.2 OBJECTIVES

The main objective of this study was to investigate the use of HTM algorithms in the dynamic branch prediction problem. Particularly, the following specific objectives were defined:

1. To analyze strategies for the application of HTM algorithms in branch prediction;
2. To evaluate the accuracy and robustness of an HTM branch predictor selected among those considered;
3. To discuss issues related to hardware implementation of suggested predictors.

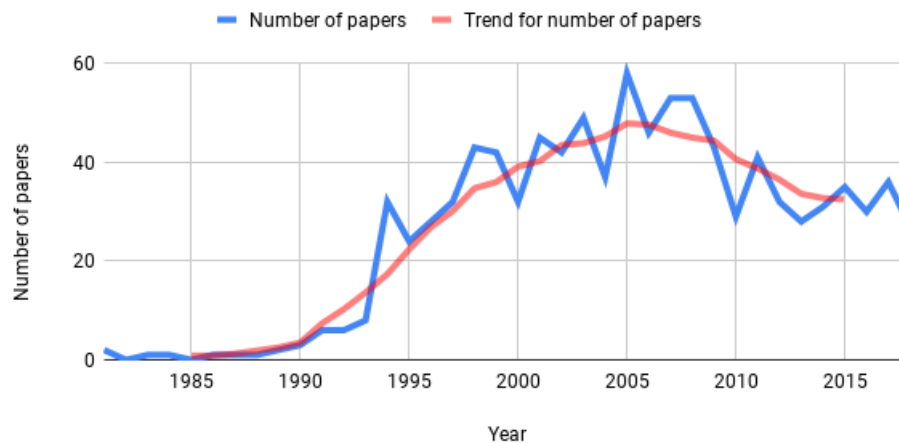
1.3 JUSTIFICATION

The research area of dynamic branch prediction has already been extensively studied for the past 40 years (MITTAL, 2018; J. E. SMITH, 1998), but not much progress was achieved in the last decade. The current state-of-the-art branch prediction (BP) schemes are improvements over the Perceptron and the TAGE predictors, first proposed in 2001 (JIMÉNEZ; C. LIN, 2001)

and 2006 (SEZNEC; MICHAUD, 2006), respectively (JILP, 2016b). After 2006, Sez nec (2011) states that “the trend for further prediction accuracy improvement is to associate side predictors targeting special categories of branches with a state-of-the-art main predictor” (SEZNEC, 2011, p. 3). Though the latter quote is from a 2011 work, as of 2019 the circumstances remain similar, and the TAGE predictor still takes the lead as a solo branch predictor (MITTAL, 2018). This situation indicates that the research field of branch prediction has achieved a stage of maturity.

Such observation can also be noticed in other metrics. The amount of papers around the branch prediction topic has considerably decreased in the last decade, likely due to increasingly marginal progress being accomplished. A search for the terms "branch prediction" or "branch predictor" in titles or abstracts of documents from the computing area in the Scopus database reveals a contraction of more than 30% in the number of papers published in the field between the years 2005 and 2015. This trend can be identified in the graph presented in Figure 2.

Figure 2 – Time-series graph of the number of papers that mention "branch prediction" or "branch predictor" in titles or abstracts of documents from the computing area in the Scopus database



This maturity, however, does not mean the area is not worthy of further research. Sez nec (2011) remarks that “more accuracy progress was achieved between 2001 and 2006 than between 1993 and 2001” (SEZNEC, 2011, p. 2), suggesting that the current stagnation might not be definitive. Additional advances in accuracy may still be possible, even if they are still hidden from prevailing techniques.

An indication of the above argument can be found in the results of the Championship Branch Prediction (CBP) (JILP, 2016a), a competition that brings together the main researchers in branch prediction. The second, fourth and fifth editions of the CBP included an unlimited size category in order to search for limits in branch predictability (SEZNEC, 2016). Through the idealistic category results it is possible to evaluate how much progress can still be made. The last CBP edition, occurred in 2016, showed that the predictor in the unlimited class can lower mispredictions per thousand instructions (MPKI) by 31.7% over the best 64KB maximum budget predictor (DUNDAS, 2016). These results demonstrate that there is still room for

improvement in the scope of viable predictors.

A motivation for further studies in branch prediction is its significant impact on processor performance. A study on a real processor has shown that “halving the average MPKI from 6.50 to 3.25 would improve [cycles per instruction (CPI)] by $13.0\% \pm 2.2\%$ ” (RAHMAN; Z. WANG; JIMÉNEZ, 2009). In future generation processors with higher ILP application, the benefit of improvements in BP accuracy may be even greater (JIMÉNEZ, 2011). Also, improving prediction accuracy can lead to energy savings, since wrong-path speculative work is avoided (MITTAL, 2018).

Artificial intelligence techniques have already demonstrated great results in branch prediction with the perceptron predictor (JIMÉNEZ; C. LIN, 2001), but the HTM algorithms had not yet been explored in the task, although there were indications that they could handle the problem well. Cui, Ahmad and Hawkins (2016) show that HTM networks achieve competitive accuracy in the continuous online sequence learning problem compared to other popular machine learning techniques, like the Long Short-Term Memory (LSTM) recurrent network (HOCHREITER; SCHMIDHUBER, 1997), declaring that “HTM networks learn complex high-order sequences [. . .], rapidly adapt to changing statistics in the data, [and] naturally handle multiple predictions and branching sequences” (CUI; AHMAD; HAWKINS, 2016, p. 3). HTM networks have a temporal and predictive aspect at their core and can identify time patterns with large context chains (HAWKINS; AHMAD; PURDY, et al., 2016; CUI; AHMAD; HAWKINS, 2016). Such properties are very desirable for the BP problem domain and thus made an HTM branch predictor a promising research topic. Moreover, the HTM theory is not a fixed one (HAWKINS; AHMAD; PURDY, et al., 2016), so further discoveries in neuroscience are likely to be incorporated by its technologies and potentially help to improve the results of this work.

It is hoped that conclusions from this study may ignite further investigations on new paths to tackle the branch prediction problem, especially with machine learning approaches still unexplored in the hardware environment.

1.4 STUDY LIMITATIONS

The HTM branch predictors proposed in this study have two main hardware related limitations associated to their examination:

1. They were not physically implemented, nor described through a hardware description language (HDL), nor examined according to their budget, latency, power consumption or area requirements, which profoundly limits any viability claim from this work;
2. They were not analyzed within a real nor virtual processor architecture environment, but isolated, and able to ideally interact with any desired part of the system.

Also, the simulation and evaluation of the predictors were done over execution traces only, thus replicating an ideal situation for all aspects of the scheme. Therefore, the accuracy results represent a very optimistic view of the solution.

The inaccuracies of the analysis, however, do not defeat the overall purpose of the study, which was to give insights about the possibilities offered by HTM systems and neuromorphic computing to the optimization of hardware mechanisms.

Lastly, this work evaluated a very narrow set of ways in which HTM algorithms can be applied to the dynamic branch prediction task. Thus, the low accuracy result does not imply that better results with other HTM components, parameters and inputs are not possible.

1.5 DOCUMENT ORGANIZATION

Chapter 2 discusses the main concepts needed for the understanding and development of this study, both for branch prediction and HTM. Chapter 3 integrates the ideas presented in the previous chapter and describes the overall operation of an HTM branch predictor. Chapter 4 details how the specific implementations of HTM predictors presented in chapter 5 were built and tested. Chapter 5 explains the operation of 6 variations of HTM branch predictors, shows their performance against a common evaluation framework and explores characteristics of the proposed systems. Chapter 6 discusses the results from chapter 5 and raises some issues for the implementation of HTM branch predictors in hardware. Chapter 7 concludes the study.

2 LITERATURE REVIEW

This chapter will explain the main ideas that are relevant to the present study, both in the areas of branch prediction (BP) and Hierarchical Temporal Memory (HTM). Related works will not be explicitly mentioned because this investigation, to the authors' knowledge, is the first to integrate the two research fields, making it considerably distant from other studies. However, associated researches that may guide the exploration process of this work are embedded in their respective sections.

2.1 BRANCH PREDICTION

The introductory chapter of this work briefly presented the branch prediction task. In short, the problem consists in predicting the outcome of branch instructions in order to make better use of speculative work on processors that implement instruction-level parallelism (ILP) techniques (D. M. HARRIS; S. L. HARRIS, 2013). This section will expose further motivations and characteristics of branch prediction, along with an introduction of the main challenges and solutions in its research field.

2.1.1 Instruction-level parallelism and branch prediction

The research in branch prediction evolved side by side with ILP techniques. This subsection explains how both areas correlate and why they are so important for modern computing.

2.1.1.1 The importance of speculative execution and branch prediction

The exploitation of ILP is ubiquitous in modern processor architectures. Hennessy and Patterson (2006) claim that at least since the 1990s all processors use pipelining to increase execution performance, showing that the benefit of such technique is widely known. What is not so obvious, in fact, is the influence of speculative work, and consequently branch prediction, in the efficiency of this approach. To support this last statement, it is appropriate to analyze both the impact generated by the use of speculation and the frequency of branch instructions, as it is in the presence of control dependencies that speculative execution may become an option.

Firstly, the use of parallelism inside *basic blocks*, i.e. entire blocks of instructions without any branches involved, is very limited since these regions are usually small due to the high frequency of branches. Some of the most important benchmarks for the evaluation of processors, the benchmarks of the Standard Performance Evaluation Corporation (SPEC), have the frequency of branch instructions ranging between 3% and 24%. Typical MIPS programs, however, hold this variation from 15% to 25%, which means that on average there are 4 or 5 instructions between successive branches. Thus, it is necessary to “exploit ILP across

multiple basic blocks” (HENNESSY; PATTERSON, 2006) to get meaningful performance improvements. However, this goal can not be achieved if the processors stall at every branch, hence the necessity of speculative execution.

Secondly, the deep pipelines used by modern processors imply that more clock cycles are wasted if the execution stalls in a control hazard. Therefore, as more ILP is exploited, speculative work and BP accuracy become increasingly important. In the introductory chapter a processor with only five stages was illustrated, but in the 1990s some processor architectures already required at least 21 clock cycles to complete the execution of a single instruction. In 2004, the Intel Pentium 4 with a clock frequency of 3.2 GHz demanded even more, with a minimum of 31 cycles (HENNESSY; PATTERSON, 2006). However, the trend of increasing pipeline depth has stopped or even reversed in the last decade, instead giving room to complementary ILP techniques which will be presented in the next subsection. As of 2019, most of the Intel processor architectures have 14 stages in the pipeline (INTEL CORPORATION, 2019). The regression in pipeline depth does not affect the argument being made since the additional ILP approaches increase speculative work and are also affected by BP accuracy.

Evidence of the importance of speculation became very apparent at the beginning of 2018 when the *Spectre* attacks revealed important security vulnerabilities caused by speculative execution (KOCHER et al., 2018). The obvious and quick solution of just avoiding speculation showed not to be a viable option since through that approach a slowdown of about 2.8x has been reported in benchmarks (MCILROY et al., 2019).

The final performance increment provided by speculation, though, depends on the accuracy of BP, since speculative work in wrong paths will only waste energy without any benefit. Hennessy and Patterson (2006, p. 120) warn that “[i]ncorrect speculation will not improve performance, but will, in fact, typically harm performance” due to the recovery delay from mis-predictions. Thus, BP accuracy is critical for high performance computing, and its importance grows with the pipeline length and the amount of speculative work exploited.

Other ILP techniques that expand pipelining will be presented next, evidencing more challenges that need to be considered in branch prediction.

2.1.1.2 Out-of-order multiple-issue processors

Although pipelining can provide a huge benefit to the performance of modern processors, it alone can not address the great limitation of fetching and executing instructions in the same order given by the program. This constraint means that if an instruction is stalled because of a data hazard, for instance, all the other instructions after it must also wait for the hazard to be cleared, even if they do not have any dependencies on their own. These types of limitations in ILP can be overcome by removing the requirement of an *in-order execution* (D. M. HARRIS; S. L. HARRIS, 2013).

Out-of-order execution can be achieved by using a type of queue where pending instructions can wait for the clearance of structural and data hazards. While the fetch of instructions is still done in-order, their execution begins as soon as they are free of dependencies. Out-of-order processors usually allow the execution of multiple instructions at the same time through the use of multiple functional units, in which case they are called *superscalar* processors. The *commit* of instructions, i.e., the definitive validation of the correctness in the execution of instructions, also happens in-order, but this fact does not invalidate the benefit of out-of-order execution. As a way of increasing performance even further, out-of-order schemes can also be extended to allow more than one instruction to be fetched and committed in each clock cycle. Processors that have this ability are called *wide-issue* or *multiple-issue* processors (HENNESSY; PATTERSON, 2006). The concept of multiple-issue processing may also be applied to in-order schemes, but it is more commonly used in out-of-order architectures.

Modern processors usually issue from 4 to 8 instructions in each clock cycle, but that number may be even larger. A wide-issue design, coupled with deep pipelining, highlights branch prediction accuracy even more, while also creates more challenges in the design of predictors. The fetching of multiple instructions per cycle means that more than one branch may also need to be predicted per cycle. Making only one prediction per cycle allows just one basic block to be fetched at a time, thus limiting the benefit of wide-issue schemes (HENNESSY; PATTERSON, 2006). This problem will be further discussed in subsection 2.1.6.

2.1.2 Static and dynamic branch prediction

One aspect that was made implicit from the beginning of this work is that branches are usually highly predictable. Branch predictors can only work because of that characteristic. If a given code has its branches working over random data only, for instance, branch predictors are made useless. Due to the misprediction recovery delay, the execution of such code can even be negatively affected by speculative execution (HENNESSY; PATTERSON, 2006). Fortunately, the vast majority of programs have well-behaved branches that can be predicted most of the time through the analysis of some of its features.

2.1.2.1 Static branch prediction

One of the cheapest methods of predicting branches is by analyzing only the data present in the instructions themselves, without using any execution history information. Since the instructions do not change, this type of prediction is called *static*. This technique may be useful in specific applications where branches can be predicted with a satisfactory accuracy at compile time, such as in some scientific programs. Other than that, it can also assist dynamic predictors (HENNESSY; PATTERSON, 2006).

The simplest static prediction scheme does not even check for any data: it always predicts branches as taken. This is a valid approach because in average branches are taken more often than not. In the SPEC benchmarks, however, the accuracy of this method varies between 91% and 41% (HENNESSY; PATTERSON, 2006). Other static approaches use some clever heuristics that examine the operation codes and target addresses of branches. For instance, forward branches, i.e., branches whose target address is greater than that of the branch address, are usually associated with *if* statements, that more often than not are satisfied. Thus, these branches are predicted not taken. On the other hand, backward branches are usually associated with loops, which in most cases are repeated several times. Therefore, these branches are predicted taken (BALL; LARUS, 1993).

2.1.2.2 Dynamic branch prediction

Firstly, note that in this study the term dynamic branch prediction will always implicitly refer to adaptive schemes. Thus, the core principle of this technique is the exploitation of ongoing execution patterns, which may come from different data sources. Most predictors learn patterns that exist in the branching history, as will be detailed later, but this is not the only available nor the only useful source of information. In fact, some branches may depend on particular conditions that can not be captured by branching behavior.

To overcome this limitation, some proposed schemes also correlate with the history of data values in the operands of branches, for instance (HEIL; Z. SMITH; J. E. SMITH, 1999). Although the most important predictors do not use this type of information, this data can be used by small predictors aimed at specific branches that work alongside more general approaches.

Dynamic branch prediction is the focus of this work, and several of its techniques will be presented in subsection 2.1.5.

2.1.3 Types of branch instructions and how to predict them

A branch instruction is able to change the normal execution flow of a program by altering the program counter (PC), which is the register responsible for storing the address of the next instruction to be executed by the processor (or, in some implementations, the address of the instruction being executed by the processor). When a branch is taken, it is said that a *jump* occurs (HENNESSY; PATTERSON, 2006).

Branches can be classified between conditional and unconditional, as well as between direct and indirect. Conditional branches are usually associated with *if* statements and loops, where a jump happens when a given condition is satisfied. On the other hand, unconditional branches will always make a jump. Most of unconditional branches are procedure returns, used to redirect the execution to the instruction that called a procedure (HENNESSY; PATTERSON, 2006).

Branches are direct if their targets are given by the instructions themselves, in which case the targets are fixed. Indirect branches, however, may have variable targets, since they do not directly point to a target, but to a register or a memory address where the target is stored (HENNESSY; PATTERSON, 2006).

Conditional branches are usually direct, while unconditional branches are mostly indirect. As unconditional and direct branches do not need prediction, since their behavior is fixed, branch prediction is commonly divided into conditional and indirect. The two types of prediction are done in different ways.

Conditional branches have their targets stored in a special buffer called Branch-Target Buffers (BTB). Therefore, conditional branch predictors only need to predict if the branch will be taken or not. If the prediction is taken, the value from the BTB is used as the new PC (HENNESSY; PATTERSON, 2006).

Indirect branch predictors, however, need to output targets directly. For example, if an address X is stored in a register R and an indirect branch instruction that points to R is fetched, then an indirect branch predictor should output X even before the decoding of the instruction completes. Most of the work in branch prediction is done in conditional branch predictors because conditional branches are more frequent than indirect branches (HENNESSY; PATTERSON, 2006). Anyhow, most of the techniques used by conditional branch predictors can be adapted to work with indirect branch predictors, as investigated by Driesen and Hlzl (1998).

2.1.4 Challenges in the design of dynamic branch predictors

The main problems that need to be considered in branch prediction design are listed below.

1. *Aliasing*: in order to predict the behavior of a given branch in a given context, branch predictors need to use a certain amount of data so that different situations can be appropriately differentiated from one another. If too little data is provided, the prediction scheme may see distinct circumstances as the same, thus allowing the occurrence of interference between them. This interference is often called aliasing. Because branch predictors are finite, aliasing can only be avoided up to a certain degree, but several works have studied ways to prevent it and increase BP accuracy (SPRANGLE et al., 1997).
2. *Storage budget*: the most recent edition of the Championship Branch Prediction, that happened in 2016, chose the storage budget of 64KB for the largest category of an implementable branch predictor (JILP, 2016a). This limit is usually not a constraint of the cost of implementation only, but also a limitation of *area* and *energy consumption*. Large branch predictors also usually require more time to learn patterns, i.e., to *warm-up*, and thus do not provide much improvement in accuracy (JIMÉNEZ; C. LIN, 2001).

3. *Latency*: the prediction latency is one of the most critical aspects of branch predictors. Even a perfect predictor with 2 cycles of prediction delay is not satisfactory, since an inaccurate predictor that is able to deliver a 1-cycle prediction may provide better performance (JIMÉNEZ; KECKLER; C. LIN, 2000). Thus, there is an important limit in the number of operations that may be used in series in a branch predictor design.

2.1.5 Main dynamic branch prediction techniques

This subsection will present the main approaches used in the dynamic and adaptive prediction of conditional branches, coupled with examples of designs that explore the referred ideas. Lastly, the main characteristics of the perceptron and TAGE state-of-the-art predictors will be highlighted. Take into consideration that each of the explained schemes may have multiple variations not mentioned here. For a broader view, see Mittal (2018).

2.1.5.1 Predicting branches from local history

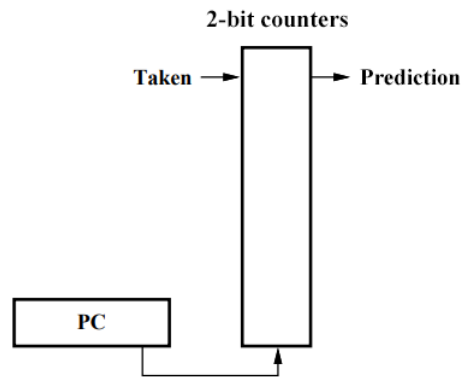
Branch predictors are called local if they estimate the direction of a given branch using only its own branching history, independently of other branches (MCFARLING, 1993).

The simplest local branch predictor estimates that branches will always behave in the same way they acted in the last time they were executed. This predictor is a small memory called a *branch-prediction buffer* or a *branch history table*. It is indexed by the least significant bits of the branch address and stores just one bit per entry (HENNESSY; PATTERSON, 2006). This scheme works well for biased branches, i.e., branches that evaluate to the same direction most of the time (MCFARLING, 1993), and since biased branches are very common in many programs, this technique can provide up to 98% of prediction accuracy for some scientific applications (J. E. SMITH, 1998).

In the 1-bit approach, however, even strongly biased branches will undergo two sequential mispredictions in the occurrence of an unusual change in the direction followed by a return to the previous behavior. To avoid that, 2-bit unsigned saturating counters can be used instead. In the 2-bit predictor, values 0 and 1 translates into a not taken prediction, and 2 and 3, a taken prediction. When the actual result of a branch is known, its respective predictor is incremented if the branch is taken and decremented otherwise, up to the saturating limits. Therefore, a predictor in a strongly biased state, i.e., with values 0 or 3, must miss twice before its prediction is inverted (J. E. SMITH, 1998; HENNESSY; PATTERSON, 2006; MCFARLING, 1993). Figure 3 shows this approach's design.

The SPEC's benchmarks show that the accuracy of a 1KB 2-bit predictor with 4096 entries varies between 82% and 99% for different programs (HENNESSY; PATTERSON, 2006), with the average across all benchmarks staying slightly above 93%. Increasing the size of the predictor reveals a saturation of precision at 93.5%, showing that even for this small number

Figure 3 – The 2-bit saturating counter branch predictor



Source: Adapted from McFarling (1993)

of entries the main source of mispredictions is not aliasing (MCFARLING, 1993). Once every branch is able to map to a particular entry in the table of predictors, the scheme shows its limits and makes it clear that additional data needs to be used in order to get more accurate estimates (HENNESSY; PATTERSON, 2006).

2.1.5.1.1 The two-level local branch predictor

Yeh and Patt (1992) explore 3 variations of a branch predictor that “uses two levels of branch history information to make predictions” (YEH; PATT, 1991), the first level being a branching history, and the second, the behavioral history of each pattern in the first level. The most effective version of this implementation improves accuracy by exploring long local branch histories. With such data, it is possible to capture more complex branching patterns, or specifically, to learn the behavior of branches with repetitive patterns, such as those involved in loops with a static number of iterations. For instance, if a *for* loop always execute 4 times before exiting, a branch instruction located at the end of the loop body will show the pattern $(1110)^k$, where k is the number of times the loop has executed. In this pattern, the outcome of the last 3 executions of the branch is enough to predict the following direction (111 \rightarrow 0, 110 \rightarrow 1, 101 \rightarrow 1, 011 \rightarrow 1) (MCFARLING, 1993).

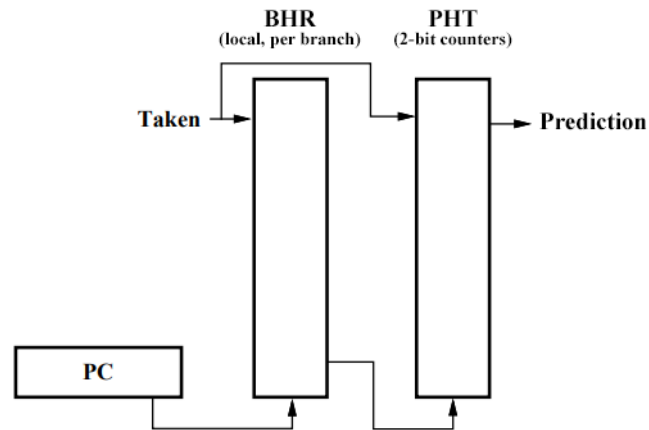
The design of the main local predictor proposed by Yeh and Patt (1992) requires the access of two tables in sequence, which makes its prediction latency more concerning.

The first table of the two-level local branch predictor is called the branch history register (BHR), equally indexed by the low-order bits of the branch address. The BHR stores the branching history of individual branch instructions up to a length of n , i.e., the outcome of each branch is shifted into its respective entry of the table (YEH; PATT, 1992).

The second table is called the pattern history table (PHT). It has 2^n entries and is indexed by the data in the BHR. Each of the entries in this table has an unsigned 2-bit saturating counter identical to that used in the previous scheme. The PHT is responsible for learning the behavior

of each of the branch histories in the BHR. Therefore, for a given history provided by the BHR, the PHT manages to predict what the next decision will be (YEH; PATT, 1992). This scheme is illustrated in Figure 4.

Figure 4 – The two-level local branch predictor



Source: Adapted from McFarling (1993)

The prediction and learning procedures of this design happen mostly in the same way as in the saturating counter predictor, differing in the addition of the BHR. When the prediction for a branch is requested, the lower portion of its address is used as an index to find the respective branching history in the BHR. This history is then used as an index to get a 2-bit counter from the PHT. The value of the counter gives the prediction of the branch, and its BHR entry is speculatively updated as if the prediction was correct. When the actual result of the branch is known, the 2-bit counter that gave the prediction is updated and in case of a misprediction the BHR is corrected (YEH; PATT, 1992).

Note that in this design the PHT works globally, storing patterns for branching histories of all branches. Although this approach might reduce accuracy because of interference between branches, the huge storage increment required by a scheme with particular PHTs per branch or per set of branches makes the global design more efficient for a given predictor size (YEH; PATT, 1992).

A large two-level local branch predictor with 64KB of storage achieves an average accuracy of about 97% in the same SPEC's benchmarks used for the 2-bit predictor. This precision means that the number of mispredictions is reduced in about 50% in comparison with the latter design, which translates into a significant improvement in performance (MCFARLING, 1993; YEH; PATT, 1992).

Although the accuracy of the two-level local branch predictor is high, it is clear that there is room for improvements since the predictive ability of its design is limited by the lack of a global view of the code execution, thus making the branches unable to correlate with others (HENNESSY; PATTERSON, 2006). This topic will be further discussed next.

2.1.5.2 Correlating branch predictors

A lot of codes have branches that are somehow related to each other. Some branches are so strictly correlated that the knowledge about the outcome of a set of previous branches is enough to precisely predict the behavior of another one (HENNESSY; PATTERSON, 2006). Figure 5 shows an example of code with an exploitable correlation among branches. If the first and the second branches are both satisfied, then *aa* and *bb* are set the value of 0, hence always making the third branch not satisfied. Therefore, looking at the results of preceding branches may be a valuable source of information to improve the accuracy of branch predictors. Predictors that do use the behavior of other branches to make predictions are called *correlating predictors* (HENNESSY; PATTERSON, 2006).

Figure 5 – Example of code with correlated branches

```

if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {

```

Source: Hennessy and Patterson (2006)

Evers et al. (1998) remark the different reasons that make branches correlate and contribute to render the global execution history useful for branch prediction.

One of their observations refers to the type of global history used by branch predictors, that might be composed of *directions* or *path* data. Predictors that use directions look for patterns only in the outcomes of the branches, without identifying which branch instruction was responsible for which result (EVERS et al., 1998). These are named *pattern-based* correlating predictors (NAIR, 1995).

Although tracking only directions provide good results, some correlations can be better identified by predictors that analyze the path to a given branch (EVERS et al., 1998). An example of code with an *in-path* correlation can be seen in Figure 6. The outcome of branch V is not correlated with the result of branch X, but the fact that the code has reached branch V already means that branch X will be satisfied.

Branches belonging to subroutines that have their outcome influenced by the part of the code that called the subroutine can also be better predicted by *path-based* predictors (EVERS et al., 1998).

Three designs of pattern-based and one of path-based correlating branch predictors will be presented next.

Figure 6 – Example of in-path correlation

```

branch Y: if (NOT(cond1)) ...
branch Z: else if (NOT(cond2)) ...
branch V: else if (cond3) ...
...
branch X: if (cond1 AND cond2)

```

Source: Evers et al. (1998)

2.1.5.2.1 *The two-level global branch predictor*

The pattern-based two-level global branch predictor is one of the variations proposed by Yeh and Patt (1992) in the same paper that presents the two-level local branch predictor described above. This global design has both the BHR and the PHT working globally, i.e., the BHR is not a table indexed by branch addresses, but a single shift register with n bits that stores the outcomes of the n last executed branches. Thus, unlike the local version, this scheme does not require the sequential access to two tables.

Each time a branch instruction is encountered, the global branching history in the BHR is used as the index to the PHT, that has 2^n entries, each with a 2-bit counter. Thus, the address of the branch being predicted is made irrelevant to the prediction.

At the same budget size, this design outperforms the simple 2-bit counter predictor for budgets above 1KB, but the two-level local variant has a higher accuracy for all budgets up to 64KB. The weakness of the two-level global branch predictor is its poor ability to identify individual branches, which leads to high levels of aliasing since multiple branches share the same global history (MCFARLING, 1993). The following two schemes aim to combat this flaw.

2.1.5.2.2 *Global predictor with index selection*

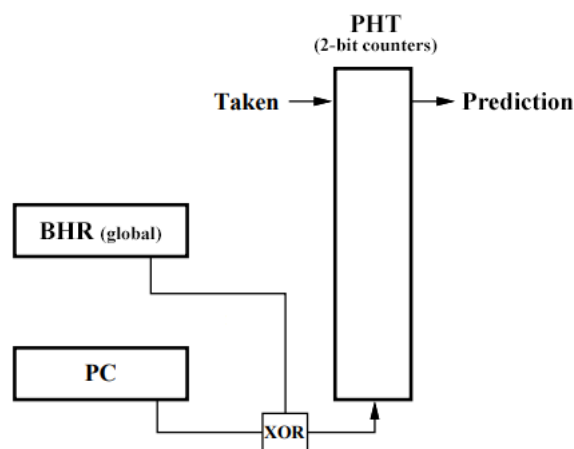
The global predictor with index selection, as known as *gselect*, was proposed by Pan, So and Rahmeh (1992). It extends the prior global design by simply building the index used to access the PHT from the concatenation of the global branching history with some number of the least significant bits from the branch address. Such technique enables a better identification of branches, reducing aliasing upon the latter global approach.

Although the optimal number of bits from each source must be empirically found through tests with benchmarks, the *gselect* predictor generally outperforms the accuracy of the two-level local branch predictor for budgets up to 1KB and compares performance for greater budgets, while requiring a single table access (MCFARLING, 1993).

2.1.5.2.3 Global history with index sharing

McFarling (1993) suggests a better way of merging the global branching history and the branch address through a predictor named *gshare*. This predictor uses the bitwise exclusive OR operation between the two data values in order to maximize the amount of information carried by the index value while maintaining its size. Figure 7 illustrates this scheme. The bitwise XOR operation may be considered a hash function that is able to better separate the branches in a particular position of the global branching history, thus decreasing aliasing over the *gselect* predictor.

Figure 7 – The *gshare* branch predictor



Source: Adapted from McFarling (1993)

The *gshare* predictor may also be tuned on the amount of bits used in the XOR operation. Its best version slightly improves over *gselect* from a budget of 256 bytes (MCFARLING, 1993).

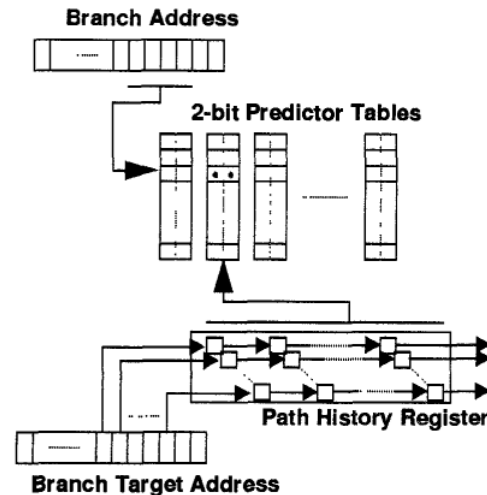
2.1.5.2.4 Path-based correlating branch predictor

The first path-based correlating branch predictor was introduced by Nair (1995). His design uses the same PHT from the previous presented predictors, but the BHR is replaced by a path history register (PHR) that stores path data. More specifically, it saves q of the least significant bits from “the addresses of the targets of the immediately preceding p conditional branches” (NAIR, 1995). Thus, the total length of the PHR in bits is $p \times q$. The indexing of the PHT is done with the concatenation between some bits from the PC and the path data in the PHR. Note that neither p nor q may be too large because of the PHT size limits. However, the author mentions that a smart hash function could help to use more path information within the same amount of bits.

Another important detail is that the PHR stores bits from the *targets* of the last branches, not from the addresses of the branches themselves. Nair (1995) explains that in some cases the

targets provide information not contained in branch addresses and thus make the predictor work better.

Figure 8 – The path-based correlating branch predictor



Source: Nair (1995)

Figure 8 shows the path-based predictor's design with the PHT divided into several columns in order to increase the access parallelization. Overall, with the right tuning in the parameters p and q , this path-based scheme is comparable to the gselect predictor (NAIR, 1995).

2.1.5.3 Combining global and local branching histories

The predictors introduced so far use either local or global data only, while achieving similar accuracy results. Mixing local and global methods is a natural improvement idea. Chang and Chou (2002) improve over the gshare predictor with the *LGshare*, a predictor that indexes the PHT with the XOR hash between the lower part of the branch address and the concatenation of local and global branching histories. Lu et al. (2003) concurrently suggest *mshare*, a predictor that follows the same approach of LGshare, except that the local branch history is excluded from the XOR operation. Both papers show that their design outperforms gshare.

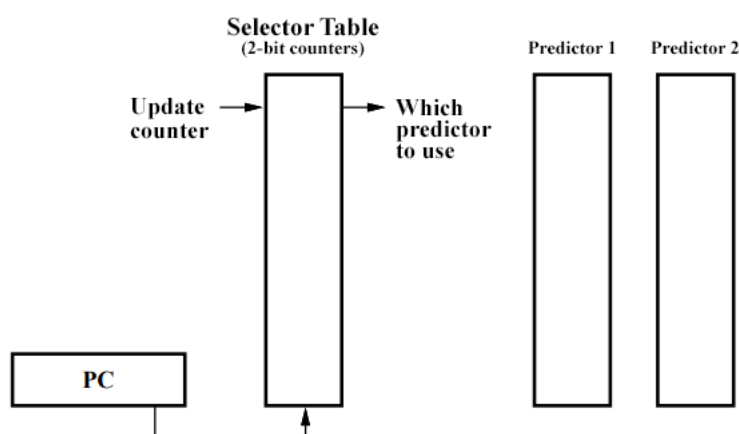
2.1.5.4 Combining predictors

The tournament or hybrid predictors were first proposed by McFarling (1993) as a solution to the cited problem of local and global history advantages integration. The proposal of an hybrid predictor is to have two or more distinct predictors working simultaneously and to combine their estimates into a more precise one (HENNESSY; PATTERSON, 2006).

The hybrid predictor scheme offered by McFarling (1993) is capable of working with any two predictors. His design is a simple table of 2-bit saturating counters acting as selectors to choose which of the two components to use for the final prediction. The method allows each branch to use the predictor that works best for it in a given moment (HENNESSY; PATTERSON, 2006).

The 2-bit selector table is indexed by the lower-order bits from the branch address. If the selected entry has a value of 0 or 1, the prediction from predictor 1 is used. Otherwise, the prediction from predictor 2 is used. The 2-bit counter selector learns the most effective predictor for a branch by tracking the correctness of the components and adjusting its value accordingly to favor the most accurate predictor. If both predictors estimate the same outcome, nothing is changed in the selector. If predictor 1 is correct and predictor 2 is incorrect, the respective counter is decremented. Analogously, if predictor 1 is incorrect and predictor 2 is correct, the respective counter is incremented (MCFARLING, 1993). See the approach's design in Figure 9.

Figure 9 – Hybrid predictor with a 2-bit selector table



Source: Adapted from McFarling (1993)

Hybrid predictors with more than two components may use more complex integration solutions like majority voting, most confident prediction or majority voting with confidence values (MITTAL, 2018).

Generally the hybrid predictors integrate at least one local and one global predictor due to their distinct benefits (HENNESSY; PATTERSON, 2006). McFarling (1993) suggests the combination of the two-level local and the gshare predictors and achieve, with a budget of 64KB, an accuracy of 98.1%, compared to 97.1% for the two-level local predictor alone with the same budget (MCFARLING, 1993).

2.1.5.5 Other techniques to improve prediction accuracy

Several specific methods for improving prediction accuracy have already been explored. These techniques include many ways of reducing aliasing, branch history filtering, history length

adaptation and data correlation usage (MITTAL, 2018).

Some predictors, called *side predictors*, target special situations or classes of branches and are meant to act like correctors for the more general predictors. An example of such component is a loop termination predictor, that can predict the end of regular loops with 100% of accuracy (SHERWOOD; CALDER, 2000). Side predictors only override the main predictor when they have a high confidence about their correctness, and therefore are used to improve state-of-the-art techniques (SEZNEC, 2011).

2.1.5.6 State-of-the-art branch predictors

The overall design and functioning of the original Perceptron and TAGE branch predictors, considered state-of-the-art methods (GOPE; LIPASTI, 2014), will be described below. Many follow-up papers improve over these techniques, but this subsection will focus its exposition on the reason behind the improved accuracy demonstrated by such predictors.

2.1.5.6.1 *The perceptron branch predictor*

The perceptron branch predictor was proposed by Jiménez and C. Lin (2001) as one of the first steps in the application of machine learning to branch prediction. They show how the simplest neural network model, the perceptron, can be implemented in hardware for low delay operation and high accuracy in conditional branch prediction.

Jiménez and C. Lin (2001) remark that most of the research done in branch prediction up to 2001 focused in removing aliasing from the indexing of branches to a 2-bit counters PHT, but few of them tried to improve the prediction mechanism itself. Therefore, they introduce the Perceptron predictor as a substitute to the traditional 2-bit scheme.

The perceptron is a simple model of a neuron that is able to learn a function that maps inputs to outputs through examples (BLOCK, 1962). The single-layer perceptron used by Jiménez and C. Lin (2001) is a simple summation unit that receives a list of inputs multiplied by respective weights. The perceptron weights are signed values that represent the importance of each particular input to the final result.

The list of inputs to the perceptron predictor is given by a global BHR similar to that used by pattern-based predictors, with the difference that the value -1 is used to represent the not taken decision. An extra bit of bias, always set to 1, is also used to help predict biased branches that do not depend on branching history. Each branch instruction is mapped by its address bits to a table of weights that are used for the computation in the perceptron. The prediction of this design is determined by the sign of the perceptron output. Non-negative values mean predict taken, while negative values mean predict not taken. The weights are adjusted by a learning rule after the actual outcome of a branch is known (JIMÉNEZ; C. LIN, 2001). Figure 10 exemplifies the perceptron prediction mechanism.

Figure 10 – The operation of a perceptron predictor

| bit | 0 | 1 | 2 | 3 | Bias |
|----------------|---|----|----|-----|-------|
| result | NT | T | T | NT | Input |
| Branch history | -1 | 1 | 1 | -1 | 1 |
| Weights | 1 | 30 | -2 | -20 | 10 |
| Prediction | $-1 + 30 - 2 + 20 + 10 = 57 \geq 0 \rightarrow$ Predict Taken | | | | |

Source: Jiménez and Calvin Lin (2002)

The perceptron predictor performs better than gshare for all budgets above 1KB and can be combined in an hybrid design to deliver even higher accuracy. At a budget of 4KB, the perceptron predictor shows its best result over gshare, decreasing the misprediction rate by 10.1% (JIMÉNEZ; C. LIN, 2001). According to Jiménez and C. Lin (2001), the good results provided by their predictor come from the use of longer history lengths. In branch predictors like gshare the number of entries in the PHT grows exponentially with the number of bits used in the BHR, thus significantly limiting the amount of history that can be used to the search of correlations. Moreover, even very big storage sizes could not help these predictors, since longer histories also mean a longer time to train. Jiménez and C. Lin (2001) show that more than 18 bits of history actually harms gshare performance.

On the other hand, the resources required by perceptrons increase linearly with the number of bits in the BHR since they only need to store additional weights. Also, the training time of perceptrons is independent from the history length. Therefore, the perceptron predictor can work with history lengths of more than 60 bits, allowing the identification of correlations between very distant branches (JIMÉNEZ; C. LIN, 2001).

An evolution of this design that correlates with path history in addition to pattern history was proposed two years later (JIMÉNEZ, 2003). Besides achieving a 7% decrease in the misprediction rate over the original perceptron predictor, the scheme was optimized to reduce latency, which was still an important issue for the adoption of the first proposal.

2.1.5.6.2 The TAGE branch predictor

The term TAGE stands for *TA*gged *GE*ometric *history length*, a name that makes reference to the way that the history information is used in the design of the predictor. The TAGE predictor uses a set of independent prediction tables, each of them indexed by a hash value computed over a different length of the global execution history. The history lengths in each table follow a geometric series. Each table may give its own prediction, and thus, TAGE is an hybrid predictor by itself. The main strength of the TAGE predictor is its ability to exploit an even larger execution history than the perceptron predictor. Because of the use of a geometric series

in the list of history lengths, the predictor is able to find correlations in the range of hundreds of bits (SEZNEC; MICHAUD, 2006).

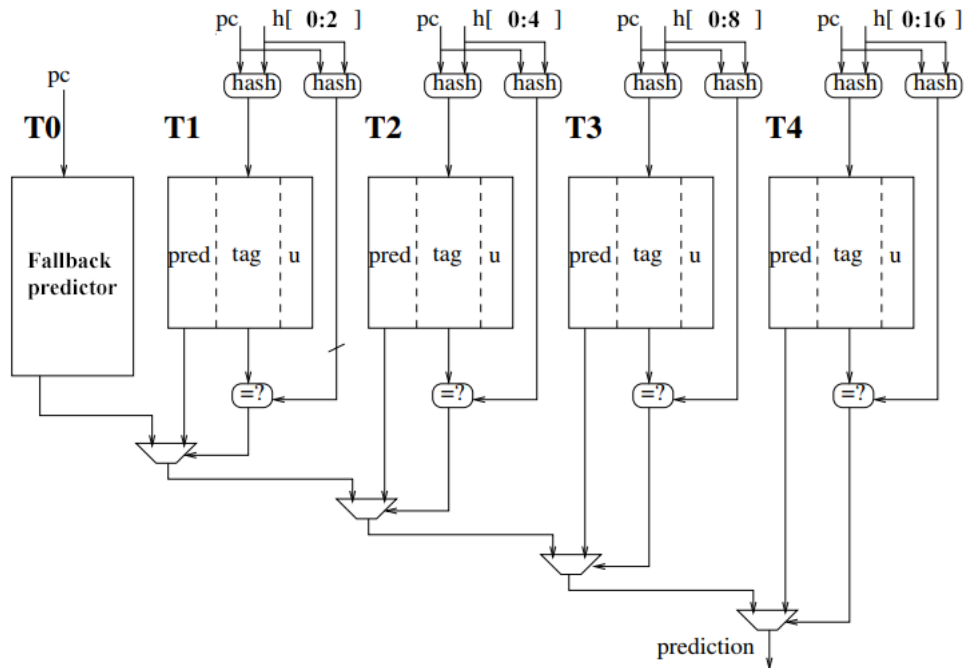
The first predictor to use the idea of a geometric series was the GEometric History Length (GEHL) branch predictor, proposed by Seznec (2005), the researcher who led the study on the TAGE predictor a year later. The GEHL predictor is similar to TAGE, but simpler, and therefore will be explained first. The GEHL predictor implements from 4 to 12 prediction tables, each with, typically, 4-bit signed saturating counters. A possible geometric series of history lengths of a GEHL implementation with 8 components is [0, 2, 4, 8, 16, 32, 64, 128]. Thus, the first component will be indexed only by the branch address; the second component will be indexed by a hash value computed from the branch address and 2 bits from a global history; and so on. The global history may be either a pattern history, a path history, or a combination of both. When a prediction is requested, all tables are accessed simultaneously, each using its own hash function, and the output values are added together. As in the perceptron predictor, the sign of the summation is used to give the final prediction. In a misprediction or when a low confidence result is given, the counters in all components are updated to contribute with the right outcome, since all of them participate in the decision (SEZNEC, 2005).

Note that the use of a geometric series in the history lengths makes most of the storage to be used in recent events, while still allowing distant correlations to be captured. Another important observation on the GEHL predictor is that because of the long histories used, most local correlations are already captured by the predictor in the global data, thus making the use of local histories not very useful (SEZNEC, 2005).

The TAGE predictor improves over GEHL by using a more cost-effective solution for the selection of the final prediction, while also improving the update algorithm. Instead of adding estimates from all components, TAGE uses an additional counter in each entry to store the usefulness of the prediction given by the component. Also, the entries are partially tagged to decrease aliasing, i.e., extra information is stored in each entry to help avoiding that a branch uses predictions determined by another branch. In this design, when a prediction is requested the partial tag is first checked to invalidate components that already use the entry for other purposes. From the suited entries, only the one with maximum usefulness and that uses the larger history length is selected to define the prediction of the branch. If no component has these characteristics, a fallback predictor is used. In the case of a misprediction, the usefulness of the entries are adjusted (SEZNEC, 2005). See a more detailed specification of the TAGE predictor design in Figure 11.

Besides lower prediction latency over GEHL, the TAGE predictor can also achieve better accuracy. At a budget of 64KB, the TAGE predictor can decrease the misprediction rate in more than 30% over gshare (SEZNEC, 2011). Moreover, the scheme can also be adapted to work with indirect and conditional branches together, reusing part of the hardware (SEZNEC, 2005).

Figure 11 – The TAGE branch predictor



Source: Adapted from Seznec and Michaud (2006)

2.1.6 Tolerating prediction latency and making multiple predictions per cycle

As mentioned in subsection 2.1.4, the prediction latency is critical to high processor performance. Complex, large and very accurate predictors that can not be optimized to give predictions in one clock cycle are rendered worthless by much simpler designs (JIMÉNEZ; KECKLER; C. LIN, 2000). Moreover, modern wide-issue processors that are able to fetch 4 or more instructions per clock cycle can be significantly benefited by the ability of fetching more than 1 basic block per cycle, a capacity that naturally requires multiple predictions to be made in a single cycle (SEZNEC; JOURDAN, et al., 1996). Some methods to address these issues will be described below.

Although a 2 cycle delay for a prediction can not be tolerated in most cases, slow predictors may be efficiently used if its latency can be hidden somehow. One of the techniques proposed by Jiménez, Keckler and C. Lin (2000) is *overriding*, an approach that amortizes the impact of a slow predictor by using an auxiliary fast predictor. In this method, a less-accurate fast predictor is always requested to deliver a 1 cycle prediction while the slow predictor works in parallel. When the second predictor completes, its prediction is compared against the first one, and in the case of disagreement, the situation is handled as a misprediction and the instructions fetched after the respective branch are flushed, i.e., the second predictor overrides the first one. Although this approach can avoid the cost of a full misprediction caused by the fast predictor, it does not ideally use the more accurate predictor.

A technique that more efficiently hides the delay of a slow predictor is *ahead pipelining*. Most conditional branch predictors try to estimate the behavior of a branch in order to find, with the help of the BTB, the address of the first instruction in the immediate following basic block that should be executed. In the ahead pipelining method, the branch predictor is altered to predict not the following basic block, but the basic block that is n branches away, with n usually equals 2. In this scheme the predictor can be pipelined into n stages and still provide its prediction in time. The method can also be used to deliver multiple predictions in a single cycle. In both use cases, the prediction accuracy of a 2 block ahead predictor is equivalent to that of a traditional 1 block ahead predictor (SEZNEC; JOURDAN, et al., 1996). The state-of-the-art TAGE predictor implements ahead pipelining (SEZNEC; MICHAUD, 2006).

Another way of predicting multiple branches at a time, although not necessarily in a single clock cycle, is by decreasing the granularity of the prediction scheme. Instead of predicting the outcome of each individual branch, it's possible to define batches of branches and to predict the following batches themselves. Ramirez et al. (2002) propose the utilization of a type of batch named instruction *stream* or *dynamic block*, which “is a sequential run of instructions, from the target of a taken branch, to the next taken branch” (RAMIREZ et al., 2002). However, a given instruction stream can only provide multiple predictions if there is at least one not taken branch between its beginning and the next taken branch. An alternative to streams are *traces* of instructions, which are “dynamic sequences of instructions, containing [implicitly] embedded predicted branches” (JACOBSON; ROTENBERG; J. E. SMITH, 1997). Traces are cached in some processor designs to accelerate the fetching of non-sequential execution flows, thus making their use as prediction units more straightforward than streams. Note that streams and traces may ramify into multiple paths, rather than just two, as in conditional branches. Therefore, next-stream and next-trace predictors are more similar to indirect target branch predictors. Both streams and traces approaches are shown to outperform overriding methods (SANTANA; RAMIREZ; VALERO, 2003).

2.2 HIERARCHICAL TEMPORAL MEMORY

This section will introduce the Hierarchical Temporal Memory (HTM) theory with a focus on the HTM sequence memory, which is the prediction component that will be exploited for branch prediction in this work. A complete explanation of the use of the HTM sequence memory for prediction tasks will be shown after the presentation of a set of concepts that will support the understanding and use of the prediction algorithm.

2.2.1 Artificial Neural Networks and neuroscience

The research field of Artificial Intelligence (AI) was dominated by Artificial Neural Networks (ANNs) as soon as, in 2012, Krizhevsky, Sutskever and Hinton (2017) applied a Deep Convolutional Neural Network, later known as AlexNet, to image classification in the ImageNet competition and achieved an unprecedented accuracy rate. In recent years, traditional ANNs have shown remarkable successes, from beating humans at the ancient game of Go (SILVER et al., 2017) to generating natural speech (SHEN et al., 2018) or incredibly detailed faces (KARRAS; LAINE; AILA, 2018). One point to be made, however, is that the models responsible for such accomplishments, despite being biologically inspired, operate very differently from biological brains (HAWKINS; AHMAD; PURDY, et al., 2016).

While most AI researchers are not concerned with the use of simplified, non-biologically accurate models of neurons, others prefer to adhere to neuroscience. These scientists argue that traditional ANN systems are highly optimized for specific tasks and therefore lack the learning flexibility found in animals, which for them is the core of true intelligence (HAWKINS; AHMAD; PURDY, et al., 2016). Thus, they aim to address the limits of classical machine learning algorithms by studying the only known general intelligence system: the human brain.

2.2.2 HTM theory overview

The Hierarchical Temporal Memory (HTM) is a theory of intelligence focused on the understanding of the human neocortex. The HTM theory is not just biologically inspired, but a biologically *constrained* model of intelligence (HAWKINS; AHMAD; PURDY, et al., 2016). It belongs, therefore, to the AI research group that develops side by side with neuroscience.

According to Hawkins, Ahmad, Purdy, et al. (2016, p. 5), HTM is more broadly the name used to describe either “the overall theory of how the neocortex functions”, “the technology used in machines that work on neocortical principles”, or “a theoretical framework for both biological and machine intelligence”.

2.2.2.1 The structure of the neocortex

On the path to understanding what makes human intelligence, the anatomy of the brain is a good starting point to look for hints. The human brain can be seen by an evolutionary standpoint as a stack of structures that developed on top of each other and are capable of producing progressively complex behavior. It means that mammals still have a reptilian brain which interfaces with more recently evolved brain structures (HAWKINS; AHMAD; PURDY, et al., 2016). The most recently developed part of the brain in mammals is the neocortex, which is responsible for everything that is often called intelligent behavior (HAWKINS; AHMAD; PURDY, et al., 2016).

Hawkins, Ahmad, Purdy, et al. (2016, p. 10) describe the human neocortex as “a sheet of neural tissue about the size of a large dinner napkin (2,500 square centimeters) in area and 2.5mm thick” that “wraps around the other parts of the brain” and mention that one of the most remarkable things about it is its homogeneity. The cortical regions responsible for hearing, vision, language or planning are all very similar, sharing a common structure that repeats for every part of the neocortex. This fact suggests that there are also common algorithms running everywhere, and therefore every brain function is handled in the same way. Thus, what makes the function of a cortical region are its inputs (HAWKINS; AHMAD; PURDY, et al., 2016).

These observations indicate that learning what a small section of the neocortex is doing can unwrap most of what is needed to understand intelligence, and this is what HTM theory intends to do (HAWKINS; AHMAD; PURDY, et al., 2016).

2.2.2.2 HTM principles

The HTM theory carries in its name some of its main principles. Hawkins, Ahmad, Purdy, et al. (2016) explain that first,

[. . .] it is best to think of the neocortex as a "memory" system. The neocortex must learn the structure of the world from the sensory patterns that stream into the brain. [. . .] Second, the memory in the neocortex is primarily a memory of time-changing, or "temporal", patterns. [. . .] And finally, the regions of the neocortex are connected in a logical "hierarchy".

HTM systems learn in an online manner, i.e. they're continuously learning from their sensing or interaction with the environment around them. Their learning is not oriented by external labels, but through their own predictions on what is about to be sensed (HAWKINS; AHMAD; PURDY, et al., 2016). Therefore, HTM relies on streaming data in order to build “a predictive model of the world” (HAWKINS; AHMAD; PURDY, et al., 2016, p. 14). Prediction is at the core of HTM, as it “postulates that every excitatory neuron in the neocortex is learning transitions of patterns” (HAWKINS; AHMAD; PURDY, et al., 2016, p. 13). Hawkins, Ahmad, Purdy, et al. (2016, p. 14) present the Temporal Memory or HTM Sequence Memory learning algorithm as “a memory of transitions in a data stream” that lies at the heart of HTM theory, and

suggest that it “is probably the biggest difference between HTM theory and most other artificial neural network theories”.

HTM is not a fixed but rather an evolving theory. There are several properties and parts of the neocortex that have not yet been fully explored despite their importance (HAWKINS; AHMAD; PURDY, et al., 2016), in a way that new findings on sequential learning can positively impact the results of this work.

2.2.2.3 HTM applications

The current applications of the HTM theory reside in the prediction area, and consequently in anomaly detection (AHMAD; LAVIN, et al., 2017; LAVIN; AHMAD, 2015; RODRIGUEZ; KOTAGIRI; BUYYA, 2018; C. WANG et al., 2018). Recent work has also been done on the use of HTM principles to increase noise tolerance in traditional ANNs (AHMAD; SCHEINKMAN, 2019).

Moreover, recent HTM research include sensorimotor integration, the representation of locations through grid cells and the definition of a broader framework for intelligence (KLUKAS; LEWIS; FIETE, 2019; HAWKINS; AHMAD; CUI, 2017; LEWIS et al., 2019; HAWKINS; LEWIS, et al., 2019), which are studies that may impact much richer application areas, such as robotics.

2.2.3 Sparse Distributed Representations

The representation of knowledge is still a major challenge in the development of intelligent systems. There is a virtually infinite amount of information that can be extracted from even the most basic real-world task. Furthermore, relationships throughout this knowledge can be very complex and abstract. For these reasons, traditional computer data structures are not well suited for the construction of intelligent agents. Therefore, inspired by how brains deal with such demands, HTM uses Sparse Distributed Representations (SDRs) for all of its data representation (HAWKINS; AHMAD; PURDY, et al., 2016, p. 15).

In HTM, a Sparse Distributed Representation (SDR) is essentially a vector of sparsely activated bits usually representing a set of neurons in the brain and their respective states (AHMAD; HAWKINS, 2015). An usual size for a SDR is 2048 bits. The suggestion for this kind of representation comes from the neocortex, as “no matter where you look, the activity of neurons is sparse, meaning only a small percentage of them are rapidly spiking at any point in time” (HAWKINS; AHMAD; PURDY, et al., 2016, p. 11). SDRs are also said to be “distributed because although each active neuron contributes information, it is the set of active neurons that determine what is being represented” (AHMAD; HAWKINS, 2016). This way of representing data leads to very powerful properties that are exploited by HTM algorithms.

2.2.3.1 Semantic meaning in SDRs

An essential characteristic of SDRs is the existence of semantic meaning in every one of its bits, in a way that “the representation and its meaning are one and the same” (HAWKINS; AHMAD; PURDY, et al., 2016, p. 12). Thus, if two SDRs share an active bit, they also share some semantic property. The more shared active bits, the more similar the information represented in the SDRs. Although it is possible to previously assign the meaning of particular bits, this information is often learned by the system itself. Such property is fundamental for the generalization of concepts (HAWKINS; AHMAD; PURDY, et al., 2016).

2.2.3.2 SDR capacity

The length and percentage of active bits in a SDR define the amount of distinct information it is able to represent. The capacity, i.e., number of possible representations of a SDR of length n and quantity of active bits w is n choose w :

$$\binom{n}{w} = \frac{n!}{w!(n-w)!}$$

Although this capacity is substantially smaller than that of a regular bit vector of the same size, common values for n and w in HTM systems make this difference irrelevant. In HTM, the length of a SDR is at least a thousand bits, while the number of active bits is usually 2% or less (HAWKINS; AHMAD; PURDY, et al., 2016). For instance, a SDR with $n = 2048$ and $w = 40$ has a number of available representations greater than the amount of atoms in the observable universe (AHMAD; HAWKINS, 2015).

2.2.3.3 Matching SDRs

Two SDRs can be compared to each other with a *match* operation to calculate similarity and execute pattern recognition. Computing the match of two SDR encodings means performing a bitwise AND operation between them and comparing its final population of active bits to a given threshold. A match occurs if the population is at least the threshold (AHMAD; HAWKINS, 2015).

If the threshold equals w , the operation is looking for an exact match. Inexact matches, however, are more useful since they can handle noise in the input. The high sparsity of SDRs enable them to be reliably used for classification under extremely noisy environments with up to 50% of interference (AHMAD; HAWKINS, 2015). This property has been explored in traditional ANNs by Ahmad and Scheinkman (2019). Inexact matches of SDRs are used throughout HTM algorithms for spatial and temporal pattern recognition (AHMAD; HAWKINS, 2015).

2.2.3.4 Efficiently matching SDRs

Verifying that a given SDR matches any other SDR from a given set of SDRs is an operation usually computed in HTM systems for pattern recognition. This operation would normally require as much matching operations as the number of SDRs in the set. However, a property of SDRs called the "union property" allows the execution of this process in a single match if a certain degree of false positives is acceptable. The "union property" is “the ability to reliably store a set of patterns in a single fixed representation by taking the OR of all the vectors” (AHMAD; HAWKINS, 2015). Then, verifying if a SDR encoding is part of this set can be done by computing a match against the result of the union operation. The precision of this match is remarkably accurate. For instance, say M is the number of vectors to be united and consider the

[...] SDR parameters $n = 1024$ and $w = 2$. Storing $M = 20$ vectors, the chance of a false positive is about 1 in 680. However, if w is increased to 20, the chance drops dramatically to about 1 in 5.5 billion. [...] In fact, if increasing M to 40, the chance of an error is still better than 10^{-5} .

(AHMAD; HAWKINS, 2015)

2.2.4 Encoding data into SDRs

Since HTM algorithms rely on SDRs as inputs, there is a need to convert a variety of data types into this standard data structure. This transformation is performed by encoders, which are the artificial equivalents of sensory organs in biological bodies (PURDY, 2016).

As a result of the massive capacity offered by high dimensional SDRs (AHMAD; HAWKINS, 2015), a single value can be translated into several different representations. A proper encoder, however, should be able to capture and represent the most important semantic information present in the data according to the problem to be solved (PURDY, 2016). Generally, an encoder should respect the following principles:

1. Semantic meaning shall be preserved, which means that “[s]emantically similar data should result in SDRs with overlapping active bits” (PURDY, 2016), while semantically dissimilar data should result in SDRs with few or no overlapping active bits (PURDY, 2016);
2. Encoding must be a deterministic process, i.e., “[t]he same input should always produce the same SDR as output” (PURDY, 2016);
3. Output dimensionality shall be fixed, i.e., “an encoder must always produce the same number of bits for each of its inputs” (PURDY, 2016);

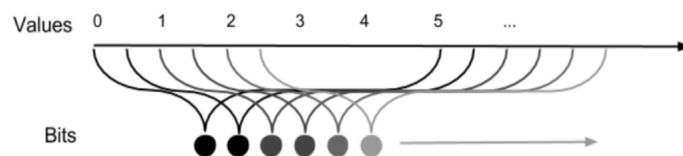
4. Output sparsity should be similar. Although “sparsity for encoders can vary from around 1% to 35% [, it] should be relatively fixed for a given application of an encoder” (PURDY, 2016);
5. Output sparsity should ensure at least 20-25 active bits, as “there must be enough one-bits to handle noise and subsampling” (PURDY, 2016).

Encoders for some data types may incorporate complex ideas to benefit from SDR properties. An encoder for geospatial information paired with motion velocity, for instance, needs to find a balance in spatial semantics when at different speeds (PURDY, 2016). In this subsection, however, the focus will be on simpler encoding ideas that may be useful for the current study.

2.2.4.1 A simple scalar encoder

A simple fixed-range scalar encoder can be built by assigning different fixed size overlapping ranges to positions of an also fixed-size SDR. Each bit shall be active if the value to be encoded matches its given range (PURDY, 2016). An illustration of this behavior with overlapping intervals in neighboring positions can be seen in Figure 12.

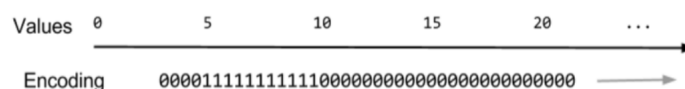
Figure 12 – A set of bits paired with ranges of values that overlap with neighbors



Source: Purdy (2016)

A more detailed example can be seen in Figure 13. Note that if the values 7.0 and 10.0 were represented with the same encoding parameters of 100 total bits, minimum-value 0, value range per bit of 5.0, and increase per bit of 0.5, their representations would share many active bits. This overlap might be desirable if 7.0 and 10.0 are semantically similar values.

Figure 13 – The encoded representation of the number 7.0 (parameters: 100 total bits, minimum-value 0, value range per bit of 5.0, and increase per bit of 0.5)



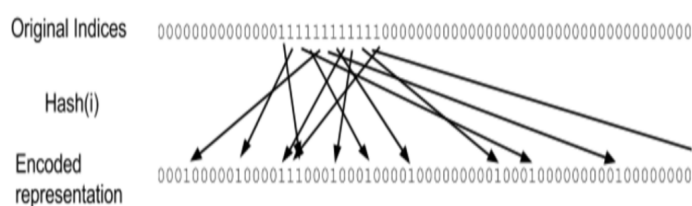
Source: Purdy (2016)

Such a method of scalar encoding requires the prior definition of both lower and upper range bounds. The encoding of a value smaller than the minimum can be represented in the same way as the minimum, thus keeping a semantic meaning in the representation. Likewise, the encoding of a value greater than the maximum can be represented in the same way as the maximum (PURDY, 2016).

2.2.4.2 An unbounded scalar encoder

Easing the requirement of making semantically dissimilar data result in SDRs with few or no overlapping active bits allow the definition of an unbounded scalar encoder. In this approach, a scalar encoder as presented above is defined, but without lower or upper limits, assuming a SDR representation with an infinite number of bits. Next, a hash function is used in the index of each of the active bits in that ideal SDR, mapping them to a finite output SDR (PURDY, 2016).

Figure 14 – A mapping of active bits from an infinite SDR to a finite SDR



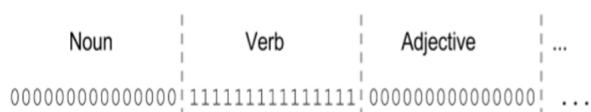
Source: Purdy (2016)

Figure 14 illustrates the described procedure. Notice that two or more active bits might map to the same index in the final SDR, making the representation more sparse than initially planned. In addition, a single bit can be used to represent very different input values, leading to erroneous semantic correlation in the final SDR. This issue, though, will not cause harm to the system provided a proper hash function is chosen and a high dimensional output space is used (PURDY, 2016).

2.2.4.3 Category encoders

Except for ordered categories, which can be interchanged with scalar values, categories will not share any semantic meaning. Thus, a category encoder should avoid the use of common bits in the representation of any two distinct categories. This property can be achieved by “[dedicating] some number of bits to each option” (PURDY, 2016) and making them exclusively active in the representation of its category (PURDY, 2016). Figure 15 shows the encoding of parts of speech, which is a categorical type of data.

Figure 15 – A category encoding of parts of speech. Each category is unrelated to the others and therefore no bits are shared among their representations.



Source: Purdy (2016)

Considering a sparsity of 2%, requiring exclusive bits for each category would limit the number of categories to 50, which may be impractical for a variety of applications. Instead, category encoders can also make use of the hashing procedure described for scalar encodings, although unwanted collisions may occur more frequently than in the context of scalars.

2.2.5 The HTM Neuron Model

The operation of neurons is still not well understood (HAWKINS; AHMAD; PURDY, et al., 2016). Despite experimental evidence showing how different inputs affect the cell behavior (AHMAD; HAWKINS, 2016), the overall mechanisms by which neurons communicate and learn together are unknown. Some theorists interpret the neuron spike rate as a scalar output value, while others consider a neuron spike as an entirely digital data (HAWKINS; AHMAD; PURDY, et al., 2016) (as implicitly noted in the discussion about SDRs, HTM theory is an adept of the latter idea). This subsection will briefly introduce the interpretation of the HTM theory on the functioning of neurons.

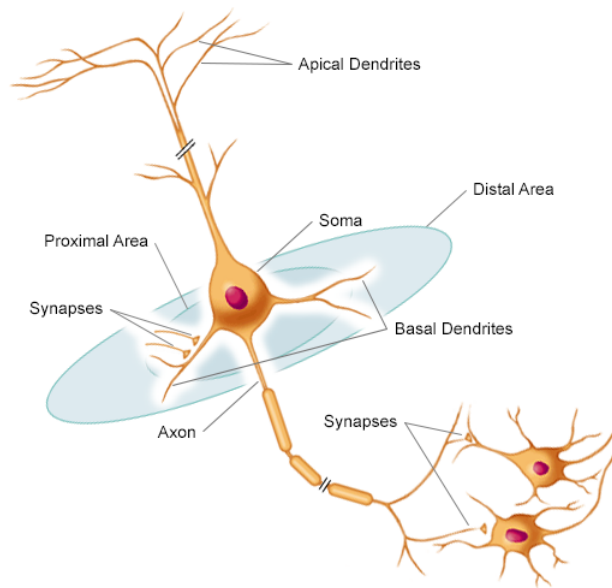
2.2.5.1 Pyramidal neurons

The pyramidal neuron is the primary type of excitatory cell found in the neocortex. Although this type of neuron varies substantially in its number of connections across different cortical regions, the HTM theory believes that they work similarly everywhere (HAWKINS; AHMAD, 2016). Therefore, they are one of the main focuses of research.

The pyramidal cell is composed mainly of three sections: the body, called soma; the input connections, called dendrites; and an output connection, called axon. Dendrites can be categorized as basal if they grow from the base of the soma, or apical if they grow from the top end. Moreover, basal dendrites can be divided into proximal or distal according to their proximity to the cell body (HAWKINS; AHMAD, 2016).

The axon has many branches which connect to other neurons through thousands of signaling structures called synapses. On the other hand, each dendrite segment may have several hundred synapses connected to it (HAWKINS; AHMAD, 2016). Figure 16 illustrates a pyramidal cell and show labels for each of the cited structures.

Figure 16 – A pyramidal neuron with labeled structures



Source: Adapted from CSLS/The University of Tokyo (2010)

The firing or activation of a pyramidal neuron, called an "action potential", sends a spike through the axon to the neuron's synapses. On the other hand, when a set of spatially close synapses receive temporally close signals in a dendritic segment, they generate a dendritic spike that travels in the direction of the soma and depolarizes the cell. Normally, a dendritic spike generated in a proximal connection will also lead to an action potential, thus carrying on the process (HAWKINS; AHMAD, 2016; SPRUSTON, 2008). Despite the knowledge about these behaviors, however, an explanation for what a neuron is really doing is still incomplete.

2.2.5.2 Pyramidal neuron operation according to the HTM theory

The HTM theory defines different roles for each type of dendritic connection and offers an interpretation of the proposed behaviors. According to HTM's research, each dendritic segment is a pattern detector of sparse neuronal activity, i.e., an SDR detector (AHMAD; HAWKINS, 2016). The learning process in a neuron occurs only in its dendrites, that through synapse growth and decay can learn to identify hundreds of activity patterns in a network (HAWKINS; AHMAD, 2016). The detection of a pattern, though, affect the cell in distinct ways depending on the type of segment responsible for the recognition.

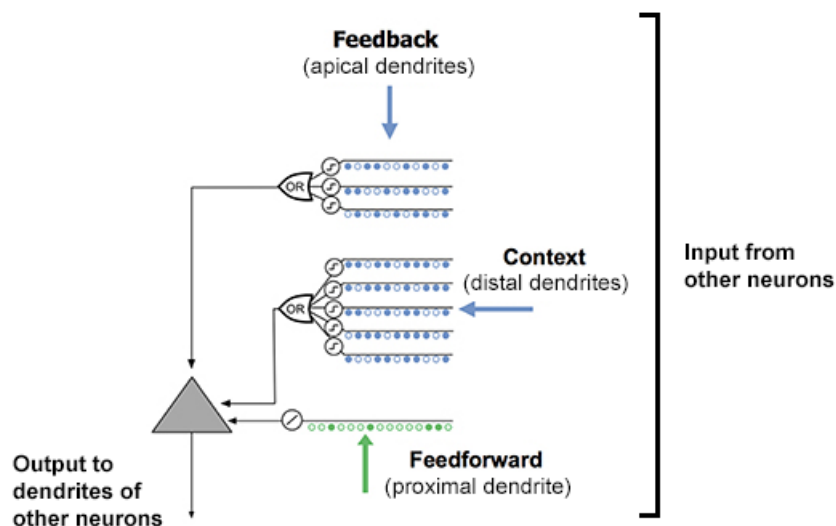
Proximal dendrites are responsible for detecting feedforward patterns (HAWKINS; AHMAD, 2016). Because of its proximity with the soma, proximal segments highly impact the occurrence of a somatic spike, i.e., cell activation (SPRUSTON, 2008). Thus, the event of a dendritic spike from a proximal connection will generate a somatic spike unless this behavior is inhibited by some other neuron (HAWKINS; AHMAD, 2016). Inhibition will be presented in subsection 2.2.7. The HTM neuron defines only one proximal dendrite.

Distal dendrites “of a neuron recognize patterns of cell activity that precede the neuron firing, in this way [. . .] learn[ing] and stor[ing] transitions between activity patterns” (HAWKINS; AHMAD, 2016). Dendritic spikes from distal connections will not generate somatic spikes but will depolarize the cell. HTM theory interprets such condition as a prediction state, in which the cell is more prone to activation for a small period of time (HAWKINS; AHMAD, 2016). Distal dendrites, therefore, are responsible for delivering temporal context to the neuron. The prediction state is very important for sequence memory and is also going to be explored in subsection 2.2.7.

Spikes from apical dendrites cause a more complex effect in the cell body. In many cases, such dendritic activation will also lead to the depolarization of the soma, and HTM theory proposes that this event is used to “establish a top-down expectation, which can be thought of as another form of prediction” (HAWKINS; AHMAD, 2016). So, apical dendrites are said to detect feedback. Since this kind of connection is not used in the current implementation of the HTM sequence memory algorithm, further details about it will be omitted.

The complete model of the so-called HTM neuron, therefore, defines proximal, distal and apical dendrites as three independent pattern detectors that influence the cell behavior when a threshold of active synapses in the segment is met (HAWKINS; AHMAD, 2016). The representation of this neuron is shown in Figure 17.

Figure 17 – The HTM Neuron Model



Source: Adapted from Hawkins and Ahmad (2016)

HTM neurons can be classified into one of four states: they can either be active, inactive, predicted or active after predicted (HAWKINS; AHMAD; PURDY, et al., 2016). The HTM spatial pooler and sequence memory algorithms define how such states are transitioned in a learning HTM network.

2.2.6 HTM Spatial Pooler

When encoders were discussed in subsection 2.2.4, it was quoted that they should produce HTMs with a fixed sparsity between 1% and 35%. However, the sparsity level of 35% is too high for the proper working of the HTM's prediction component. A typical value used by Cui, Ahmad and Hawkins (2017) and Cui, Ahmad and Hawkins (2016) is 2%. The HTM Spatial Pooler (SP) is an important component of the HTM theory which has as one of its responsibilities to solve this problem.

The SP represents a set of neurons learning feedforward patterns in an HTM network. The HTMs generated by encoders are used as inputs to the SP, that adjusts the synapses in the proximal dendrite of each cell in order to convert the original activation patterns into new HTMs that meet a series of desirable properties (CUI; AHMAD; HAWKINS, 2017). Cui, Ahmad and Hawkins (2017) lists such characteristics:

[. . .] (1) preserving topology of the input space by mapping similar inputs to similar outputs, (2) continuously adapting to changing statistics of the input stream, (3) forming fixed sparsity representations, (4) being robust to noise, and (5) being fault tolerant.

(CUI; AHMAD; HAWKINS, 2017)

The SP will not be further discussed as it's not the intention of this project to employ this algorithm in the branch predictor. This decision comes after the following reasons: (1) the currently considered predictors incorporate categorical data only, so topology is not an issue; (2) encoders can be built to generate fixed sparsity representations at the desired level; (3) there is no noise in the input data, since the encoder is deterministic and the data is categorical; (4) hardware complexity can be significantly reduced by skipping this algorithmic step; and (5) fault tolerance and continuous adaptation to input data is also handled by the HTM prediction component (CUI; AHMAD; HAWKINS, 2016). Hence, the SP is not a requirement in this work. The use of additional topological data for further correlation possibilities, though, might require the use of the SP to achieve the quoted properties and improve prediction accuracy.

2.2.7 HTM Sequence Memory

Hawkins and Blakeslee (2004) propose that the most fundamental activity performed by the neocortex, "a necessary component of sensory inference, prediction, language, and motor planning" (HAWKINS; AHMAD, 2016), is "learning and recalling sequences of patterns" (HAWKINS; AHMAD, 2016). The HTM sequence memory or Temporal Memory algorithm (TM) explains how the HTM theory believes such action happens in the brain.

The HTM sequence memory receives input from the HTM spatial pooler and learns sequences of SDRs. Through the growth and decay of synapses in the distal dendrites of each cell, the network is able to discover temporal patterns in the presented input and make predictions about the following time step (HAWKINS; AHMAD, 2016).

The algorithm has the following properties, which are required by biological implementations:

1. On-line or continuous learning;
2. High-order predictions: “the ability to incorporate contextual information from the past [and] dynamically determine how much temporal context is needed to make the best predictions” (HAWKINS; AHMAD, 2016);
3. Multiple simultaneous predictions for “overlapping and branching sequences” (HAWKINS; AHMAD, 2016);
4. Local learning rules “in both space and time” (HAWKINS; AHMAD, 2016);
5. Robustness “to high levels of noise, loss of neurons, and natural variation in the input” (HAWKINS; AHMAD, 2016).

2.2.7.1 Network structure and operation

The HTM sequence memory is implemented as a network of HTM neurons divided into columns called *mini-columns* (HAWKINS; AHMAD, 2016). These columns should not be mistaken as layers as in traditional ANNs. There is no hierarchy in the network since all neurons in all mini-columns work simultaneously with no bias towards any coordinated behavior across columns.

The HTM sequence memory network has as many mini-columns as the number of bits in the SDRs coming as input from the SP or directly from an SDR encoder, and there is a one to one correspondence between bits in the SDR and mini-columns. Each mini-column has the same number of HTM neurons, being that an usual value is 32 cells per mini-column (CUI; AHMAD; HAWKINS, 2016).

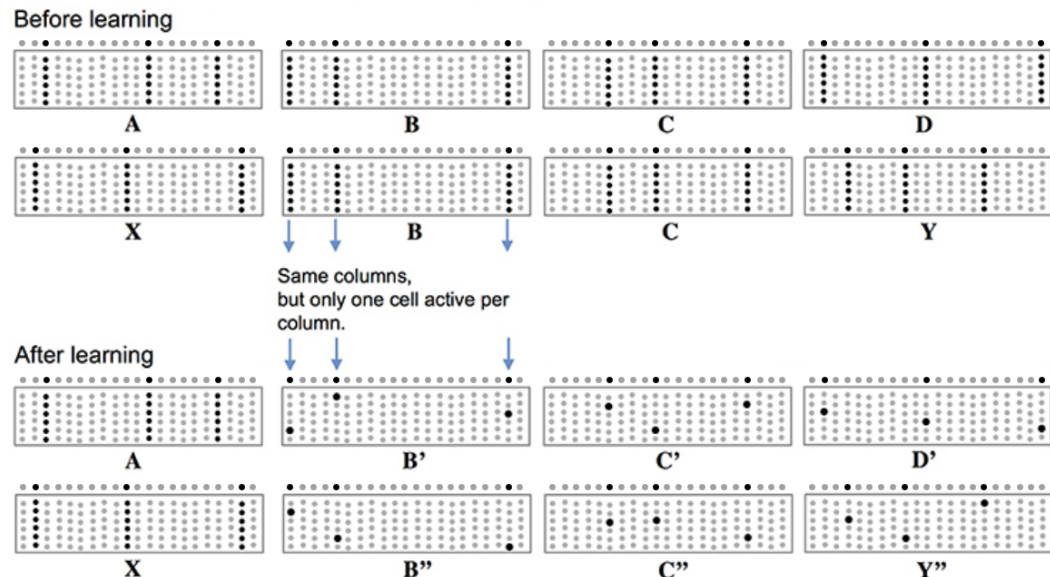
This structure allows the representation of the input SDR into many different temporal contexts. While the activation of a pattern of mini-columns represent some information, the activation of specific neurons within the mini-columns represent that information in a given sequence (HAWKINS; AHMAD, 2016).

If a given bit in the SDR is active, then all the neurons in its corresponding mini-column will receive a proximal dendritic spike. Biologically, this organization means that “neurons in a mini-column share the same feedforward receptive fields” (HAWKINS; AHMAD, 2016). The proximal dendritic spike will cause the firing of either all or some neurons in the mini-column, depending on the neuron states. Unexpected patterns will have all neurons in the column in an inactive state, thus leading to the firing of all cells. Inputs in a known context, however, will have depolarized or predicted cells within their activated columns that

[. . .] will be the first to generate an action potential, inhibiting the other cells nearby. Thus, a predicted input will lead to a very sparse pattern of cell activation that is unique to a particular element, at a particular location, in a particular sequence.

(HAWKINS; AHMAD, 2016)

Figure 18 – Sequences represented in the HTM sequence memory



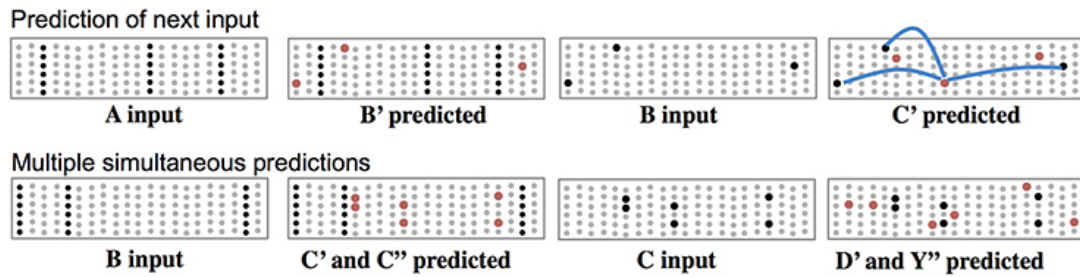
Source: Adapted from Hawkins and Ahmad (2016)

Figure 18 illustrates the exposed behavior. Consider the patterns $ABCD$ and $XBCY$. Before learning, the SDRs representing each input will trigger the entire mini-column. After learning, the same inputs in different contexts will have different representations within their active columns. Inputs A and X will still fire all cells in their mini-columns since they are the first inputs presented to the algorithm and thus have no temporal context (HAWKINS; AHMAD, 2016).

The HTM sequence memory works with data streams, continuously receiving SDRs and predicting the subsequent SDRs it's going to receive. In each time step, a set of neurons fire and cause dendritic spikes in distal connections with a sufficient amount of synapses connected to them. These spikes depolarize another set of cells, which form the following predicted state (HAWKINS; AHMAD, 2016).

Figure 19 shows the network ability to make one or multiple predictions at the same time. When A is fed into the trained network, it predicts B' , which is the only transition it has learned about. The confirmation of the pattern B triggers C' and the sequence continues. However, if B is the first presented input, the network has no temporal context, thus predicting both C' and C'' . If C is presented to the network, the confirmation of both C' and C'' will result in the prediction of D' and Y'' . The blue lines highlight the synapses in a distal segment that were responsible for the prediction of a cell in C' (HAWKINS; AHMAD, 2016).

Figure 19 – Predictions in the HTM sequence memory



Source: Adapted from Hawkins and Ahmad (2016)

2.2.7.2 Learning transitions of SDRs

Each neuron has many distal dendrites that learn a set of neuronal activity patterns in the network by adjusting the permanence of the synapses connected to them, being that a threshold on the permanence of the synapse defines if it will be considered as connected or not. Each distal dendrite acts like a SDR detector, setting its neuron in a predictive state when a matching pattern is detected (HAWKINS; AHMAD, 2016).

The neurons learn their synapse connections with other cells through a Hebbian-like rule (HAWKINS; AHMAD, 2016), a policy that can be summarized by the quote “neurons wire together if they fire together” (LOWEL; SINGER, 1992). The specifics of this rule won’t be explained in this work. Though, the general aim of the policy is to reinforce the permanence of synapses that contribute for a correct prediction while decrementing the permanence of synapses that lead to incorrect predictions. As already noted, this learning is applied locally and doesn’t need a global view of the network (HAWKINS; AHMAD, 2016).

2.2.7.3 Network parameters

Cui, Ahmad and Hawkins (2016) lists 11 parameters that can be adjusted in an HTM network. Among them, it is possible to highlight the number of columns; the number of cells per column; the maximum number of segments, i.e, distal dendrites, per cell; the maximum number of synapses per segment; and the rates of increase and decrease for the synapses permanence. It is substantial to note, however, the high flexibility that HTM networks present without any parameter tuning. Cui, Ahmad and Hawkins (2016) achieves state-of-the-art results for two very different prediction problems with the same set of parameters. Hawkins and Ahmad (2016) confirm that the HTM network is insensitive for changes in many settings, but observe that “[t]he most critical parameters are the dendritic spike threshold and the number of synapses stored per pattern”.

2.2.7.4 Extracting predictions from the network

Unlike ANNs, HTM networks do not have an output layer from which predictions can be easily extracted. Instead, predictions are also represented as SDRs spread throughout the network, i.e., all the neurons in a predictive state. Thus, a complete HTM system also requires a special classifier component to obtain the predictions from the HTM sequence memory (HAWKINS; AHMAD, 2016; CUI; AHMAD; HAWKINS, 2016). Cui, Ahmad and Hawkins (2016) present two solutions in a paper in which the HTM system is tested and analyzed in two prediction problems.

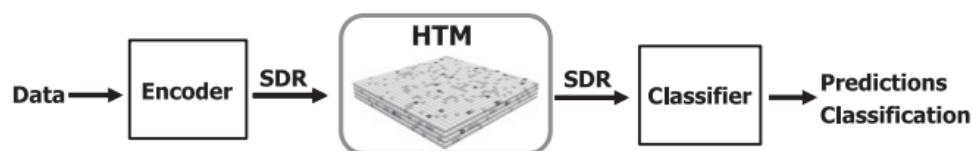
First, the best prediction can be derived by matching the set of predictive cells against every SDR seen by the network and selecting the element with the highest overlap score (HAWKINS; AHMAD, 2016). Although effective, this method quickly becomes very computationally intensive and may be impractical for continuous noisy data.

The second approach makes use of a simple ANN to overcome the limitations of the above procedure. In this method, every neuron in the HTM network is used as input in an ANN with no hidden layers. In the output layer, every result category must have its own neuron with a softmax activation function which allows the ANN to tell the likelihood for each class (HAWKINS; AHMAD, 2016). Cui, Ahmad and Hawkins (2016) also remark that the sparsity of the input compensate for its dimensionality, making the training of the network less computationally expensive.

2.2.8 An HTM system for prediction tasks

Cui, Ahmad and Hawkins (2016) describe an HTM system designed for prediction tasks that incorporates a SDR encoder, an HTM sequence memory and a SDR classifier. Figure 20 illustrates the scheme.

Figure 20 – Scheme of an HTM system for sequence learning tasks

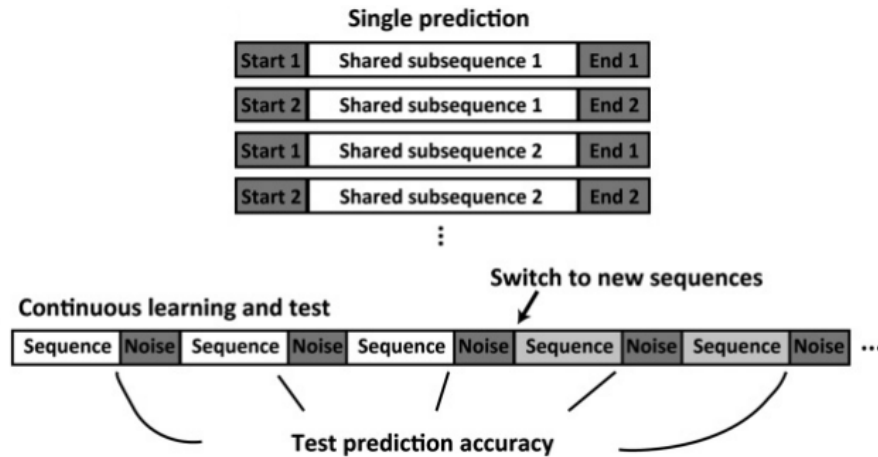


Source: Cui, Ahmad and Hawkins (2016)

Cui, Ahmad and Hawkins (2016) evaluates the system with artificial and real data. The artificial data set is a series of categorical elements, i.e., pairwise non-related items, with a predefined structure. On the other hand, the real data set describes the taxi passenger demand in New York City, which is made up of scalar and date-time values. The resolution for the latter task does use the HTM spatial pooler algorithm along with scalar SDR encoders in order to build proper SDRs to the HTM sequence memory. Therefore, this subsection will focus on the

former problem only as it is more similar to the problem of branch prediction as approached in the following section.

Figure 21 – Artificial data set created to test HTM sequence memory properties



Source: Adapted from Cui, Ahmad and Hawkins (2016)

Figure 21 exposes part of the structure of the artificial data set. Sequences are composed of related openings and endings with a shared subsequence in the middle. The data set joins several sequences with noisy representations in between. Learning and testing are performed at every step.

The SDR encoder for categorical data used by Cui, Ahmad and Hawkins (2016) in this problem randomly assigned 40 active bits out of a total of 2048 to the representation of each element. That is, it generated SDRs with about 2% sparsity. This procedure does not prevent collisions of active bits across different SDRs, but it does generate representations with a small amount of shared active bits, which is sufficient for the application.

The HTM sequence memory in Cui, Ahmad and Hawkins (2016)'s work had 2048 mini-columns, matching length of the encoder output SDR. Each of these columns had 32 neurons, adding up to 65536 neurons in the network. Unlike many machine learning algorithms used for prediction, HTM networks do not require data buffers, as only one SDR is fed into the network at a time, in an online manner. After a number of examples are shown to this component, it learns to identify the temporal context of each SDR. Then, at each step, a set of neurons is able to identify their role in a given pattern and move to a predictive state.

The classifier extracted the predictive neurons from the network and created a 2048 bit SDR with active bits corresponding to the mini-columns where there was at least one neuron in a predictive state. This SDR was then matched against every SDR shown to the network, and the representation with the highest overlap was considered as the final predicted SDR, resulting in a predicted category.

Without hyperparameter tuning, Cui, Ahmad and Hawkins (2016) has shown that this scheme achieves comparable accuracy to other machine learning algorithms, like Long Short-Term Memory (LSTM) networks.

3 GENERAL OPERATION OF AN HTM BRANCH PREDICTOR

This section will explain the overall approach used in this work to apply the HTM theory principles and components to the branch prediction task. More specifically, it will examine the use of the HTM sequence memory to predict branch outcomes.

3.1 CONSTRAINTS OF OPERATION

While most traditional branch predictors use the execution history to create an index and look for predictions in a different component, an HTM branch predictor that operates with an HTM sequence memory must work with a sequence of SDRs alone. Thus, building an HTM branch predictor requires defining a sequence that allows predictions to be extracted at each time step.

3.2 GENERAL STRATEGY

The HTM branch predictors explored in this study, detailed in chapter 5, borrow the principles used by classical branch predictors and adapt them for use in an HTM system. In general, slices of the branching history and branch addresses, or the combination of both, are faced as categories and fed into the HTM sequence memory. In this way, the network learns to predict the branching history or branch address for the following step, from which a prediction can be extracted.

Figure 22 – The general HTM branch predictor design

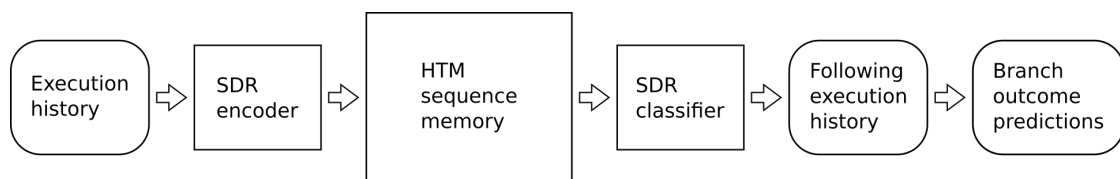


Figure 22 describes the general design of an HTM branch predictor. A slice of recent execution history, which may be formed from branch outcomes, instruction addresses, register values or any data that may help describing the current execution state, is sent to a SDR encoder. This piece of data is then translated into a sparse distributed representation and fed into the HTM sequence memory, causing the firing of a set of neurons. These activations put another set of neurons in a predictive state, whose columns will form the predicted SDR. The SDR classifier will receive the predicted SDR and compare it against already known SDRs in order to identify it. Once recognized, a future execution history might be determined. The last step is extracting the predictions, that can be encountered either in the transition between the current execution history and following execution history or other predefined logic that works with the employed sequence.

An execution history represented with n bits allows 2^n different patterns, which means that 2^n different SDRs are also required to represent the original elements. Since the problem will be handled by the HTM network as a sequence of *categories*, this is how each pattern will be referred to in this work.

The SDR encoder considers every execution history as a category because even if two slices of execution history represent the same branch in the same context and only distinguish themselves by their prediction, e.g., 1 bit of difference, they carry a totally different semantic information in the context of branch prediction, whose aim is to predict exactly this single bit. So, despite sharing a lot of information, execution histories should not share active bits in their SDRs.

3.3 DETAILS OF IMPLEMENTATION

The SDR encoder from this HTM branch predictor faces the same problem as the encoder mentioned in subsection 2.2.8, where SDRs with 2% of sparsity can not represent more than 50 totally disjoint categories. Using only 50 representations, however, would lead to much aliasing in a branch predictor due to its large amount of possible states. This study solves this limitation through a random encoder, which is also used by Cui, Ahmad and Hawkins (2016) and is mentioned in subsection 2.2.4.3. Furthermore, the encoder used in this work, despite being random, assures that every bit activates to the same number of categories and that every category has a SDR with the same number of active bits. Variations in the seed for this encoder are evaluated in subsection 5.3.4.

In order to reduce hardware complexity and latency, the SDR encoder should be implemented as a fixed mapping from slices of execution history to SDRs. Physically, it means that each bit in the SDR must have a circuitry that allows it to activate when the encoder is given a category that has this bit as active in its representation. If a dynamic mapping was to be used, the SDR classifier would also be affected by increased complexity.

The SDR classifier should also be kept simple. A similar mapping from SDRs to categories could be used, but not all categories should be compared to each other in order to find the most probable following execution history. Instead, the classifier must focus on a set of categories which are already expected to win. This is possible because some sequences can only continue in a small number of ways, while others have a broader range of possibilities, but some of the paths are more likely than others. Tiebreakers and a default behavior should also be defined.

4 RESEARCH METHODS

HTM systems are very flexible and can be adapted to accommodate a variety of prediction methods, including most of the algorithms used in branch predictors. This study has chosen some of the branch prediction techniques presented in subsection 2.1.5 and subsection 2.1.6, adapted their main ideas to work with an HTM branch predictor as described in chapter 3 and evaluated them in a common set of execution traces.

The adapted and evaluated branch prediction techniques were chosen according to their potential importance to the understanding of the advantages and limitations of an HTM branch predictor. Insights from partial results guided further exploration of algorithms and designs.

4.1 EVALUATION ENVIRONMENT

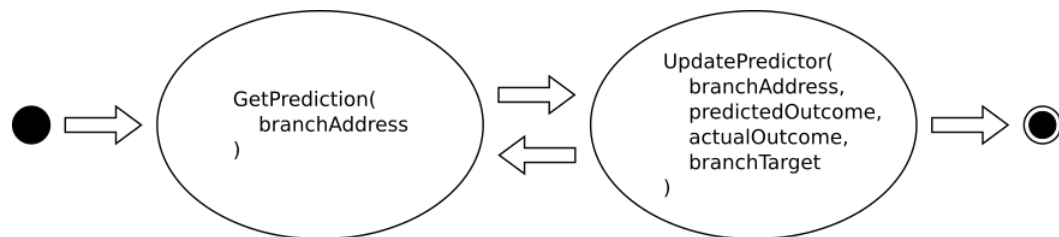
The implemented predictors were tested within the evaluation framework provided by the 4th Championship Branch Prediction, hosted in 2014 (JILP, 2014a). The Championship Branch Prediction (CBP) competition brings together some of the main branch prediction researchers to compare new branch prediction (BP) designs under a common evaluation scheme (JILP, 2016b), providing a simple interface to estimate the accuracy of branch predictors in a shared set of execution traces.

Forty execution traces from diverse environments are included in the CBP-4 kit. Twenty of them have about 30 million instructions, derived from the CBP-1 competition, and other twenty have about 150 million instructions, derived from benchmarks of the Standard Performance Evaluation Corporation (SPEC) 2006 (JILP, 2014b).

The evaluation framework has a very straightforward interface and operation. It works with conditional branches only, asking for the implementation of just 3 functions. The first function, called *GetPrediction*, receives a branch address and must return an outcome prediction, either *taken* or *not taken*. The second function, called *UpdatePredictor*, is always called immediately after the first function and gives feedback about the previous prediction, informing about the actual direction taken by the branch and the branch target address. The third function, called *TrackOtherInst*, gives the address and operation code of other instructions in the trace, offering the possibility of using this information to help making better predictions. However, the only information given by the third function that changes throughout execution is the result of unconditional branches, which means its data is of limited utility for conditional branch prediction. For that reason, most branch prediction implementations just ignore *TrackOtherInst* and work with only the first 2 functions. In this work, *TrackOtherInst* is also left blank as the traditional predictors on which the HTM branch predictors are based do not use data from unconditional branches. Figure 23 shows the basic operation of the CBP-4 evaluation kit.

Note that the framework simplifies the update of the predictor by informing the real outcome before asking for a new prediction. This characteristic is specially relevant for HTM

Figure 23 – Essential operation of the CBP-4 evaluation framework



predictors, since they need the confirmation of a prediction in order to make further predictions. This topic is further discussed in subsection 6.2.1.

The CBP-4 kit also comes with an already implemented gshare branch predictor of 32KB that is used as reference in all charts throughout the results chapter.

After the last instruction in each trace, the evaluation results are given by the framework in mispredictions per thousand instructions (MPKI). An independent script gives the final score to the branch predictor by averaging all 40 results.

4.2 CHARACTERIZATION AND PATHWAY OF EXPERIMENTS

The CBP-4 evaluation framework was expanded with a robust experimentation structure and direct communication with Python code in order to facilitate iteration of designs. Not only for ease of development, the interaction with Python programs was a requirement because of the chosen HTM implementation.

The integration of the framework with Python allowed the addition of functionalities without changing much of its original code, keeping the solutions modular and clean. The main capabilities added to the evaluation tool were: running experiments across multiple cores and computers in a distributed manner; logging every prediction made by every algorithm; and defining multiple prediction criteria for the same network, effectively creating multiple slightly different versions of a predictor while computing the network states only once (more on this in section 4.4). The performance of each predictor version was obtained by matching their logged predictions against the ground truth outcomes.

As HTM networks are very compute-intensive to run in a CPU because of their huge amount of synapses, only a small slice of each of the original traces was evaluated. After empirical analysis, it was observed that running only the first 2 million instructions for every trace offered a good trade-off between quality of results and feedback speed. The logging of predictions allowed the plotting of graphs from which the trend in mispredictions for each predictor design could be inspected. Additional tests with 8 million instructions, detailed in subsection 5.3.1, also show that the HTM predictors stabilize and do not significantly improve performance after 2 million instructions. Such configuration allowed the examination of results within a day, with execution of traces distributed across 4 personal computers.

The experiments started with the implementation of the most promising approaches first, followed by variations in parameters and techniques driven by observations and insights from previous tests. Simpler branch prediction schemes followed after. The design with the best results was then explored in aspects that could be relevant to the understanding of its characteristics and limits.

4.3 HTM SYSTEM IMPLEMENTATION

While the SDR encoder and classifier were independently implemented for this study, the HTM sequence memory was taken as given from an official implementation. This project did not intend to optimize or adjust any HTM algorithm in order to facilitate its use in a branch prediction scheme. Instead, it has worked with standard interfaces, options and parameters available in regular implementations of HTM components.

This study uses the HTM algorithms implemented in the *Numenta Platform for Intelligent Computing* (NuPIC), a code base aimed mainly for research purposes that is owned by Numenta, a privately held company behind the leading research in HTM theory (NUMENTA, 2018a).

The library is written in Python 2 and has been put in maintenance mode, meaning that only critical bugs and internal research needs can drive updates to the code. During the initial tests with the library, however, an error was encountered when the option to punish synapses that cause an incorrect prediction was activated. The bug was spotted, corrected and sent for evaluation in the repository. This was the only modification made to the HTM sequence memory algorithm.

4.3.1 Parameters selection and tuning

The parameters for the HTM sequence memory were initially selected to be equal to the model used by Cui, Ahmad and Hawkins (2016), whose work also addresses sequence learning tasks. The following step was shrinking the model down to half of its columns, so the scheme would be quicker to simulate and more realistically implementable in hardware. At this point it was important to remind that shrinking the design too much would defeat the SDR properties presented in 2.2.3, making the network much less useful.

After defining the network size, a batch of experiments with small changes in parameters was executed in the interest of looking for meaningful improvements in accuracy. The following parameters were tested: number of active bits in SDRs; rate of increment and decrement in synapses permanences; and total number of categories, which is determined by the length of the execution history used. In the performed tests it was found that, on average: using SDRs with less than of 2% of active bits produces similar or better results than using representations with 2% of sparsity; a higher rate of adjustments in synapses allow the network to learn new patterns faster and improve accuracy; and increasing the total number of categories lead to

diminishing returns in accuracy in a rate much steeper than in traditional branch predictors with 2-bit counters. Note that such results are specific to this data set and may differ greatly for other sequence learning tasks.

Table 1 – Parameters used in the SDR encoder and HTM sequence memory

| Parameter name | Value |
|---|-------|
| Number of columns | 1024 |
| Number of active bits in the SDR | 10 |
| Total number of categories | 8192 |
| Number of cells per column | 32 |
| Dendritic segment activation threshold | 4 |
| Initial synaptic permanence | 0.21 |
| Connection threshold for synaptic permanence | 0.5 |
| Synaptic permanence increment | 0.3 |
| Synaptic permanence decrement | 0.1 |
| Synaptic permanence decrement for predicted inactive segments | 0.06 |
| Maximum number of segments per cell | 64 |
| Maximum number of synapses per segments | 128 |
| Maximum number new synapses added at each step | 32 |

Such findings guided the definition of a set of parameters that were used throughout all HTM network experiments. Table 1 specifies all parameters used in the SDR encoder and HTM sequence memory. The network was defined with 1024 columns and 32 cells per column, resulting in a total of 32768 neurons. The choice of 8192 categories means that a history length of 13 bits was selected as the most appropriate to represent the execution state.

4.4 SDR CLASSIFIER SELECTION MECHANISM

As aforementioned, the SDR classifier in a branch predictor does not need to find the most likely match of category to a SDR among all possible categories. Instead, most designs allow only a few categories to follow after a known state of execution. Hence, the SDR classifier only need to inspect those possibilities. The selection of the right category among the possible or probable choices was tested with two main criteria: *most-probable* and *cascaded*.

The most-probable criteria compares the number of matches between the active bits in the SDR of each category and the active bits in the predicted SDR. The category with the greatest overlap is chosen. It also defines an order to the categories to use as a tiebreaker.

The cascaded criteria defines an order to the categories and a threshold to the number of active bits that match the active bits in the predicted SDR. The mechanism selects the first category in the designated order that meets the threshold. The last member of the order works as a fallback option. For instance, if only 2 categories are expected and the SDR of the first one in the defined order does not meet the threshold for the minimum overlap with the predicted

SDR, then the second category is chosen without further calculations. This technique is simpler as it does not require a second level of comparisons like the most-probable criteria.

The selection mechanisms significantly affect the prediction given by the system, but they do not affect the HTM network operation. Thus, it was possible to compute many variations of SDR classifiers in parallel in each experiment. The results chapter shows only the performance of predictors with the best SDR classifier.

5 RESULTS

This chapter will present and compare results from many different algorithms and designs of HTM and non-HTM branch predictors, highlighting the main aspects that differentiate the HTM approach from traditional methods.

The results and graphs in this chapter show the average MPKI score for simulations executed over the first 2 million instructions of every trace provided by the 4th CBP. Exceptions will be made clear in the text. The MPKI trend shown on the charts is calculated as a moving average over the last 100 thousand instructions, with 100 data points in each image. Note that not every chart will have the same scale in the MPKI axis.

5.1 SINGLE PREDICTION DESIGNS

This section compares three BP schemes that predict only one outcome at a time. All three techniques are correlating branch predictors that work in a global level, as local correlations would require either one HTM branch predictor per branch, which is senseless, or a complex combination of local histories working within a single network. Prioritization of interests left the latter idea out of this study.

5.1.1 Global branch predictor

The HTM version of the two-level global predictor, named here as the HTM global predictor, is modeled as a sequence of slices of recent branch outcomes. That is, instead of using the global history of outcomes as an index to a 2-bit saturating counters table, as detailed in subsection 2.1.5.2.1, this data is fed into the HTM branch predictor. Only two predictions are expected from the network: the current global history without the oldest outcome added with either a zero or a one in the newest position. The new bit is the prediction itself.

Figure 24 – Operation of the HTM global branch predictor

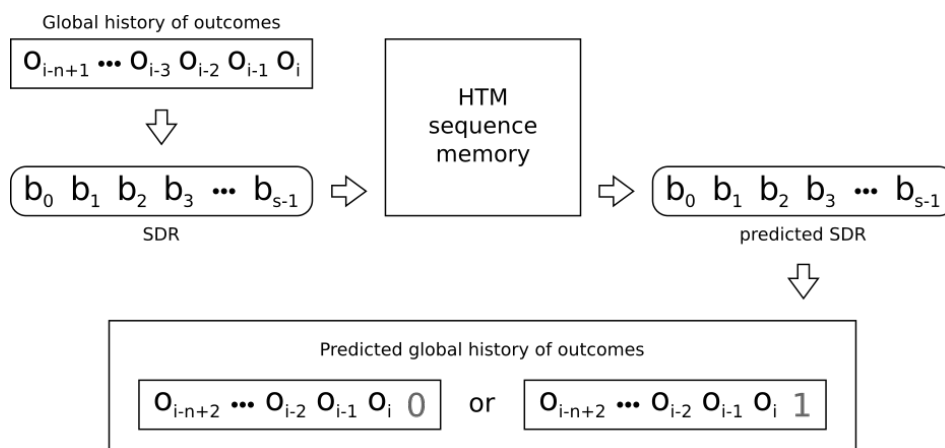


Figure 24 illustrates the operation of the scheme. The length of the global history and SDRs is n and s , respectively. When the outcome for branch i is given, the system predicts the outcome for branch $i+1$.

Figure 25 – Trend for the misprediction rate of the HTM global branch predictor

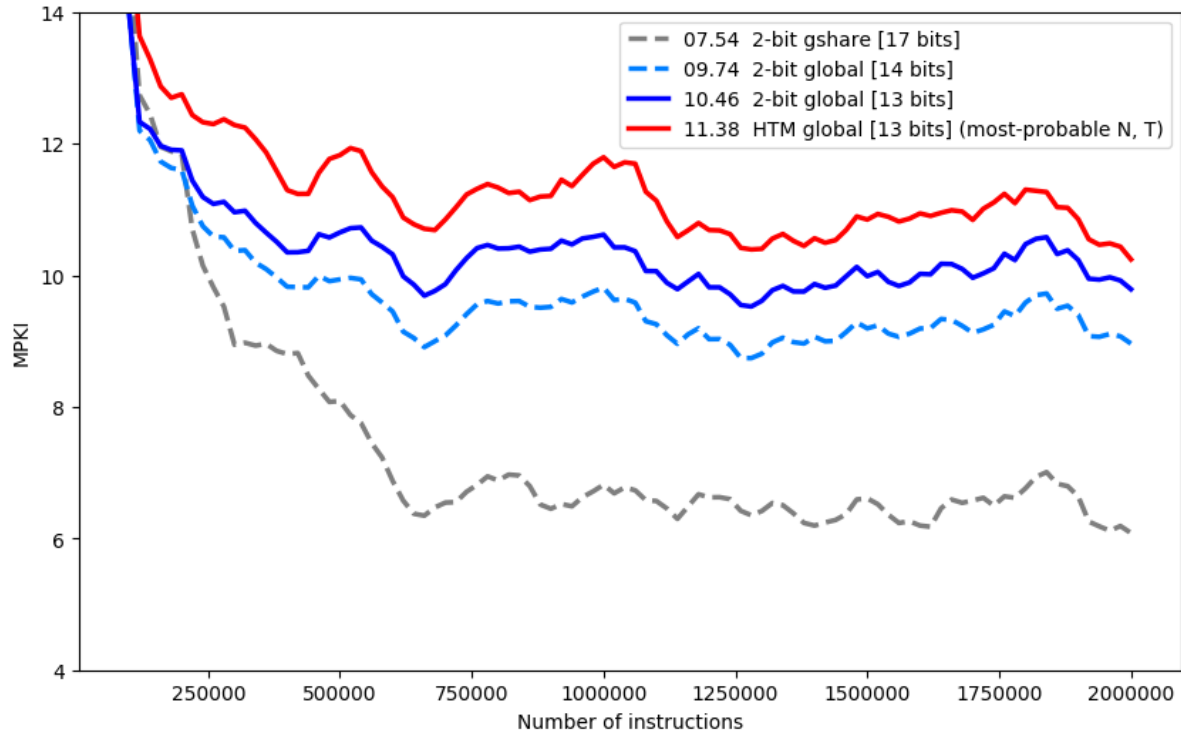


Figure 25 shows the performance of the HTM global predictor compared to the default two-level global predictor with the same history length for both. The best SDR classifier selection mechanism is most-probable, with *not taken* preferred over *taken* in the case of a draw.

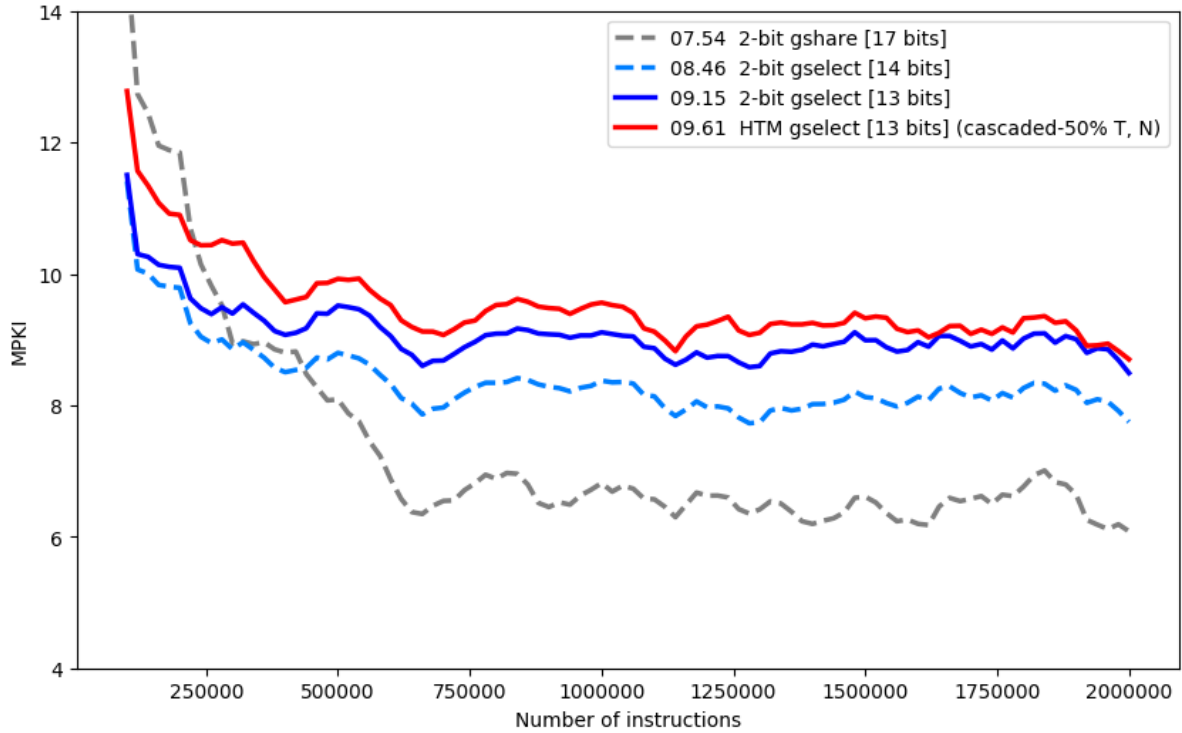
Although the HTM network could potentially build a chain of history slices and have access to more context information than the regular predictor, it still performs worse with a MPKI score 8.8% higher. The low performance possibly happens because the sequence is not stable enough for the system to learn long correlations (see chapter 6 for more details). Meanwhile, 2-bit saturating counters can adapt faster in this scenario. Still, the gap between the predictors' accuracy gets smaller over time.

5.1.2 Gselect branch predictor

The HTM gselect predictor is the HTM version of the two-level global predictor with index selection, also known as gselect. It works similarly to the HTM global predictor, but with a portion of the oldest outcomes of the global history replaced with the least significant bits of the current branch address (see subsection 2.1.5.2.2 for details on the traditional implementation). The following experiment has used 7 bits for the address and 6 bits for the global history.

The additional data from addresses, aside from reducing aliasing, gives the HTM network the ability to explore path information from traces. Both characteristics help the HTM predictor perform better.

Figure 26 – Trend for the misprediction rate of the HTM gselect branch predictor



As a result, the gap between the designs gets smaller than in the approach without addresses data. The HTM gselect design almost catches up with regular gselect when both use the same input length, as can be seen by the trend in figure 26. Overall, however, the HTM predictor has a MPKI score about 5% higher than its traditional counterpart for 2 million instructions.

Note that in this experiment the best SDR classifier selection mechanism changes to cascaded, where *taken* is chosen if its category has a SDR with at least 50% of active bits overlapping with the predicted SDR. In practice, this mechanism also makes the *not taken* decision preferable over the *taken* prediction when the sequence is unknown.

5.1.3 Gshare branch predictor

The HTM gshare predictor, like the traditional gshare predictor discussed in subsection 2.1.5.2.3, hashes the global history of outcomes together with the current branch address, causing even less aliasing than in the gselect scheme and giving the network more information while maintaining the same number of total categories.

Figure 27 details the operation of the HTM gshare predictor. Unlike previous designs, the SDR classifier of the gshare scheme requires the address of the branch for which the prediction is being computed in order to determine the winner category. Extracting the final prediction also requires unhashing the category in order to find the original bit added to the global history. The processes of identifying the winner category and determining the prediction, however, can be merged together in order to remove unnecessary computations.

As a consequence of less aliasing and greater availability of data from which path information can be extracted (more details in the discussion chapter), the HTM gshare predictor can finally benefit from the HTM sequence memory abilities and reach a MPKI score 5.5% smaller than its traditional version for 2 million instructions. The chart in figure 28 also shows that in the last few data points the HTM predictor even surpasses the regular gshare predictor with one extra bit of information.

Figure 27 – Operation of the HTM gshare branch predictor

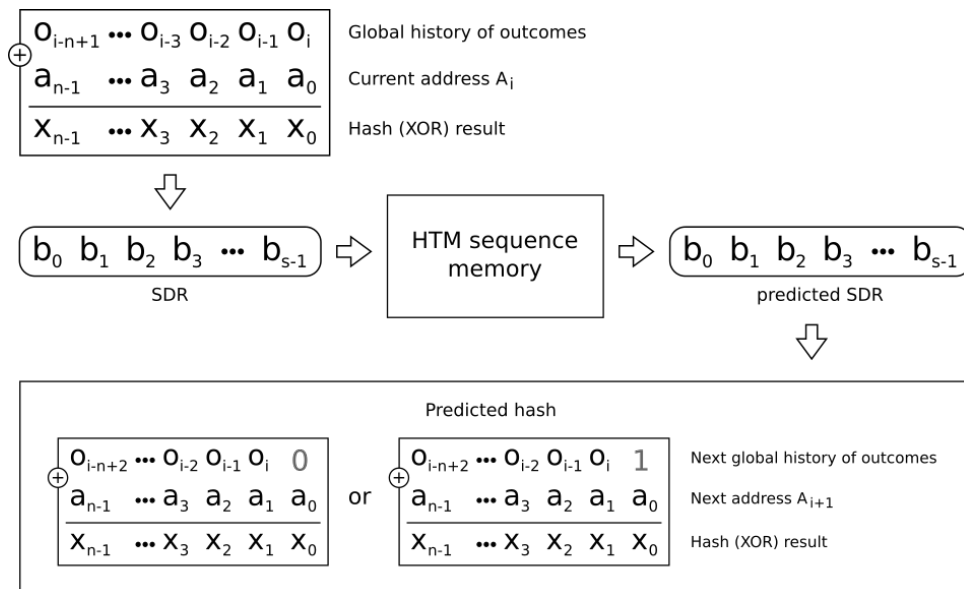
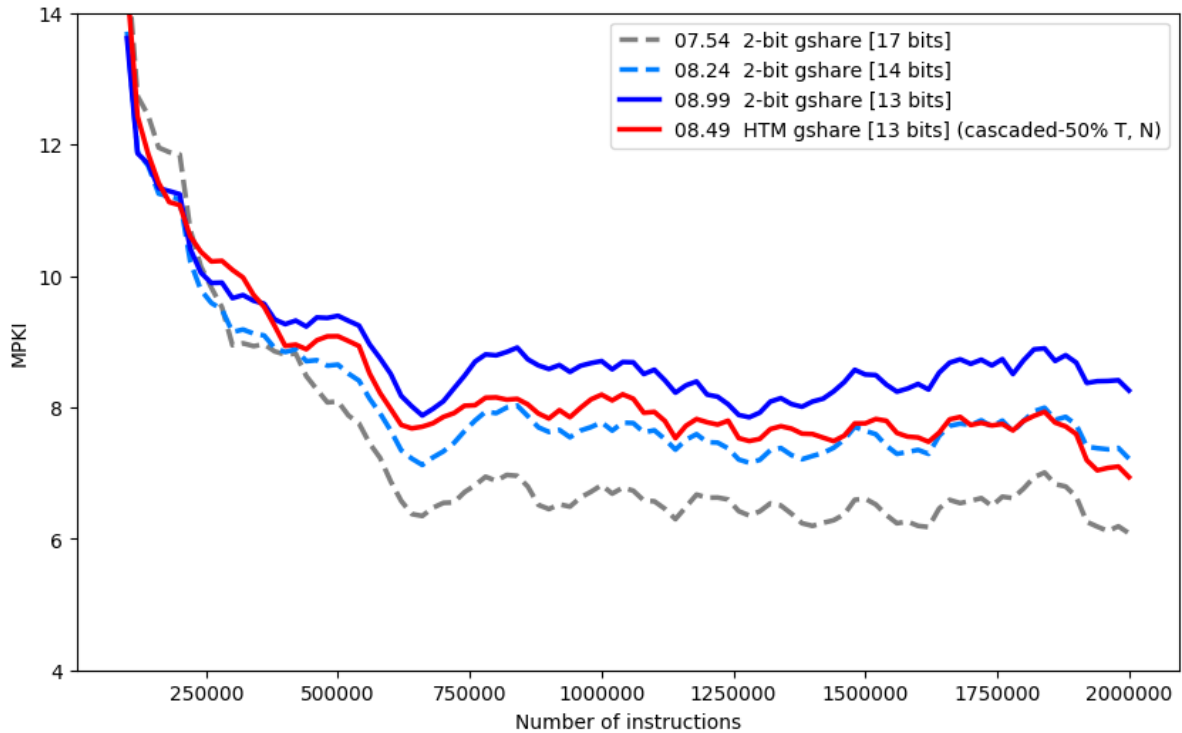


Figure 28 – Trend for the misprediction rate of the HTM gshare branch predictor



5.2 MULTIPLE PREDICTIONS DESIGNS

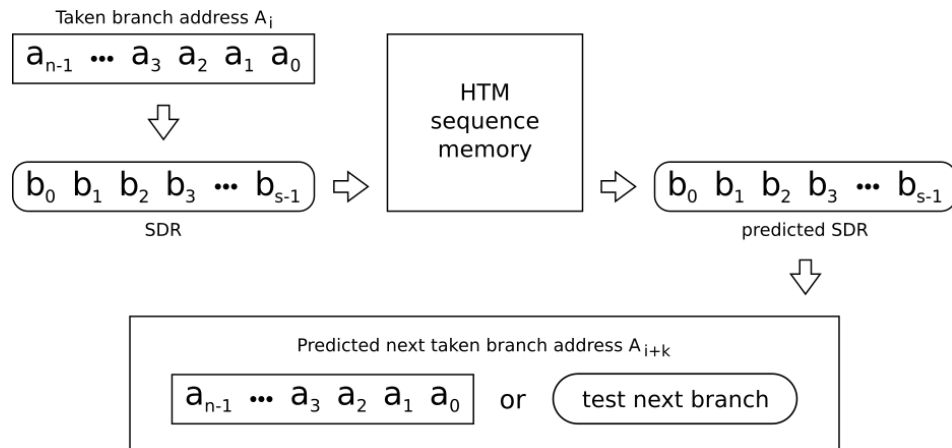
This section evaluates three additional HTM designs that are able to deliver one or more predictions per iteration.

5.2.1 Streams branch predictor

The HTM streams predictor works by predicting the next branch that will be taken, which means that the number of predictions per iteration can vary a lot depending on the program that is executing. Regardless of not always delivering multiple predictions, it's capable of doing so.

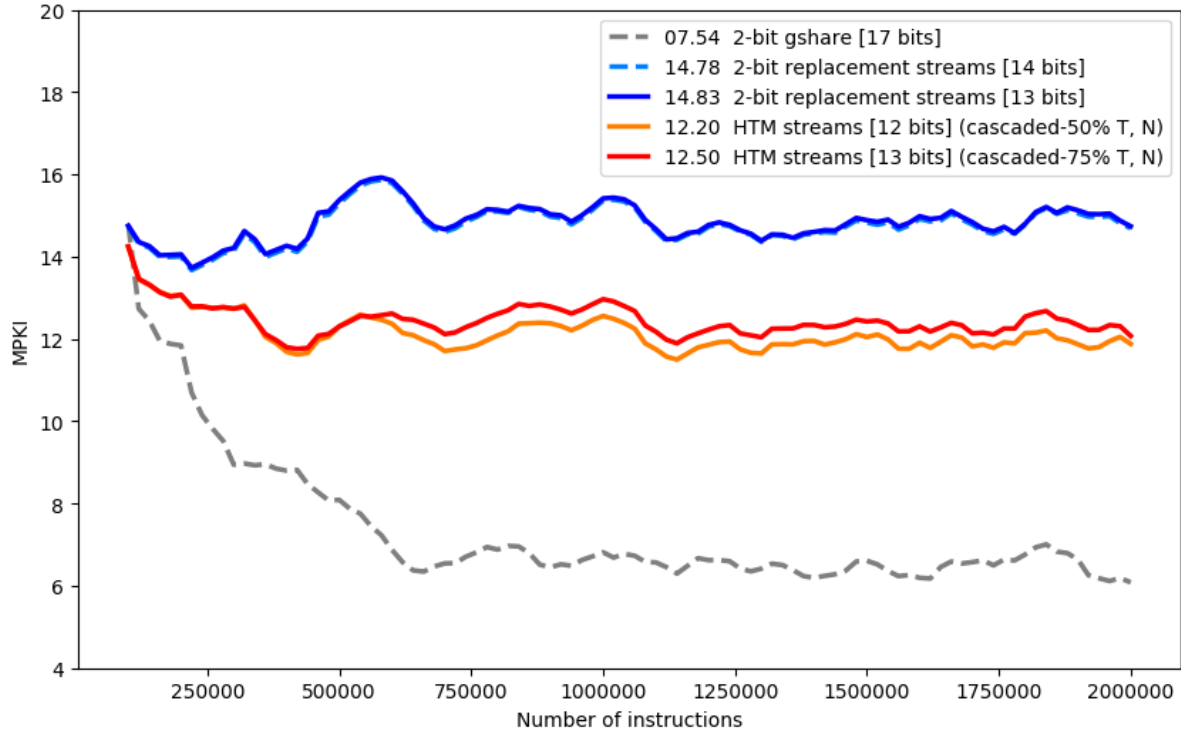
The operation of the predictor is shown in figure 29. For every encountered branch, the prediction is *taken* if the branch category's SDR meets the threshold of overlap against the predicted SDR. Otherwise, the prediction is always *not taken* and the following branches are tested until a match occurs. Only taken branches are fed into the network, creating a sequence of addresses of taken branches.

Figure 29 – Operation of the HTM streams branch predictor



A traditional, non-HTM implementation for this scheme would use a table of transitions where given an index, which could be created from one or more addresses of recent taken branches, the scheme would return the address of the next branch expected to be taken. An implementation of such design with an index formed from a single address and a 2-bit counter policy for replacement is compared to the HTM streams predictor in figure 30.

Figure 30 – Trend for the misprediction rate of the HTM streams branch predictor



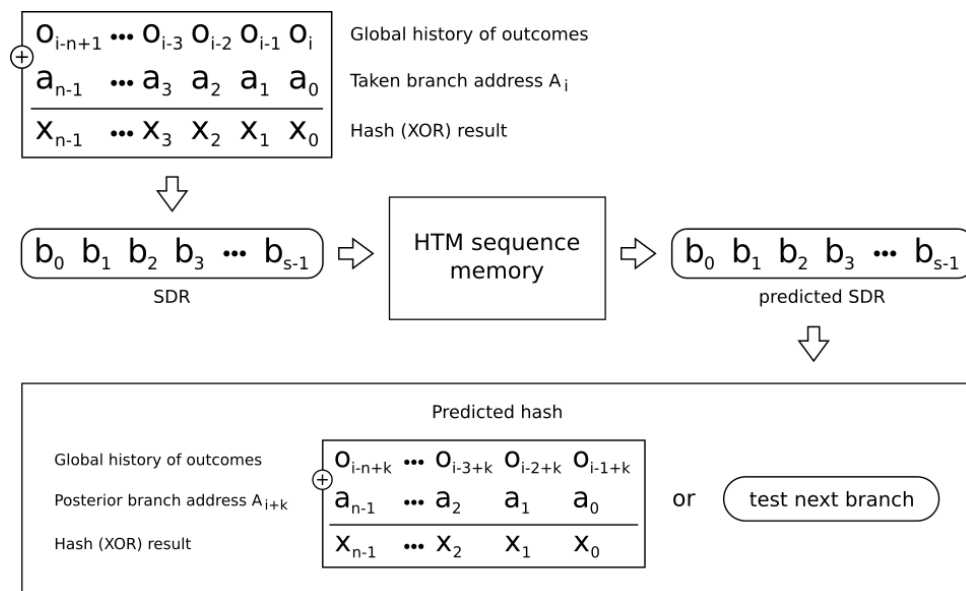
The HTM version naturally performs better given its ability to explore path information. Unusually the version of the HTM streams predictor that uses less bits from the branches' addresses achieves higher accuracy, which possibly happens due to less aliasing in the SDRs. The best HTM approach achieves a MPKI score 17.7% smaller than the non-HTM approach. Note

that increasing the number of bits in the traditional implementation does not impact accuracy in a significant way since its problem is not aliasing, but lack of contextual information. In general, this approach works poorly in comparison to other HTM schemes that have context data embedded into their categories.

5.2.2 Gshare-streams branch predictor

The HTM gshare-streams predictor combines the regular HTM gshare predictor with the idea of tracking only taken branches, while implicitly giving not taken predictions to branches that do not continue the sequence. Figure 31 shows the general operation of this design.

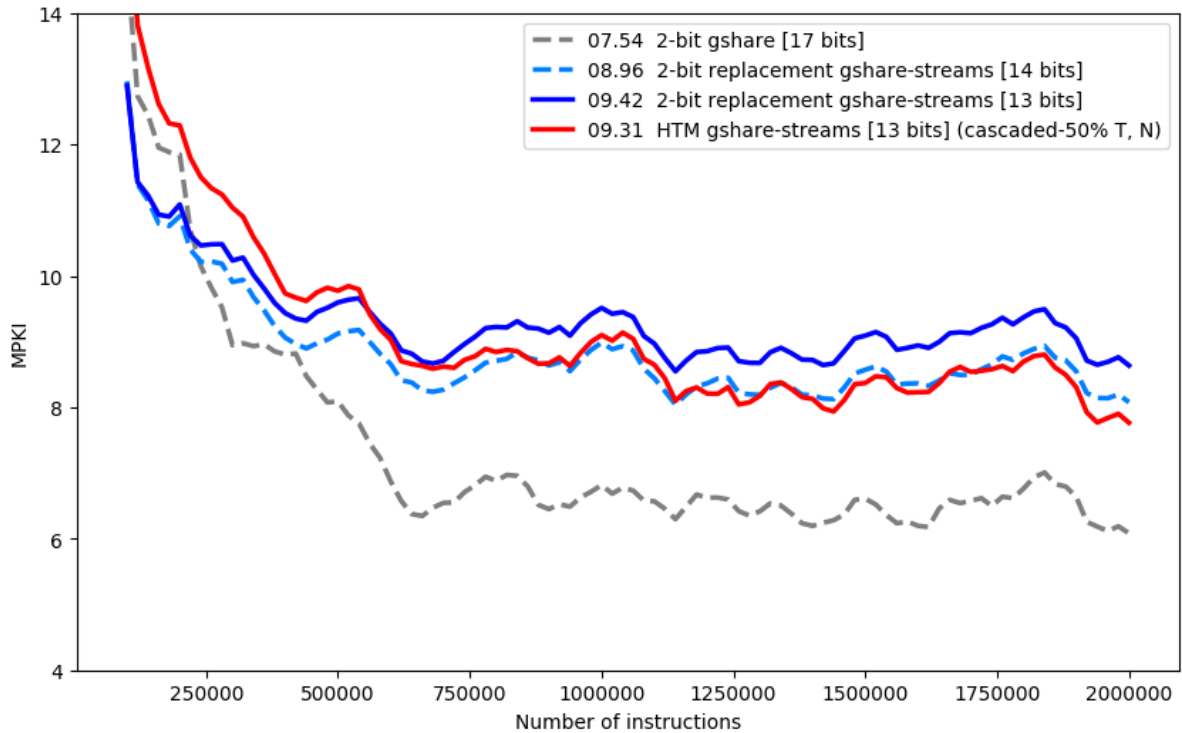
Figure 31 – Operation of the HTM gshare-streams branch predictor



The non-HTM implementation of the gshare-streams predictor is analogous to the design of the traditional streams predictor as described in the subsection above. The comparison of performance between the two approaches is shown in figure 32.

This approach is significantly superior to the regular streams scheme as it is able to access more contextual data from past execution. Similarly to the non-streams gshare predictor, the HTM gshare-streams predictor is able to perform better than its non-HTM counterpart, even though it takes longer to catch up and finishes with a MPKI score just 1.16% smaller for 2 million instructions. In the last few data points, the HTM gshare-streams also beats the non-HTM version with one additional bit of data.

Figure 32 – Trend for the misprediction rate of the HTM gshare-streams branch predictor



5.2.3 Gshare-block branch predictor

Unlike the streams-based branch predictors, the here named HTM gshare-block predictor allows a guaranteed fixed amount of predictions per iteration. It does so by simply shifting the global history of outcomes behind and adding a block of $b > 1$ predictions to the front of it before applying the hash function, thus allowing the predictor to skip $b - 1$ branches between iterations.

Figure 33 details the working of an HTM gshare-block branch predictor with b equals two. Note that the number of categories that need to be handled by the SDR classifier grows exponentially with the size of b .

Figure 34 shows the loss of accuracy in the predictor for up to 4 predictions per iteration. The trade-off between accuracy and number of predictions is sharp, since increasing the number of predictions from 1 to 2 already makes the MPKI 24% worse for 2 million instructions. In this experiment, only the numerical orders for tiebreaking were used in the SDR classifiers due to the high number of possible arrangements.

Figure 33 – Operation of the HTM gshare-block branch predictor with 2 predictions per iteration

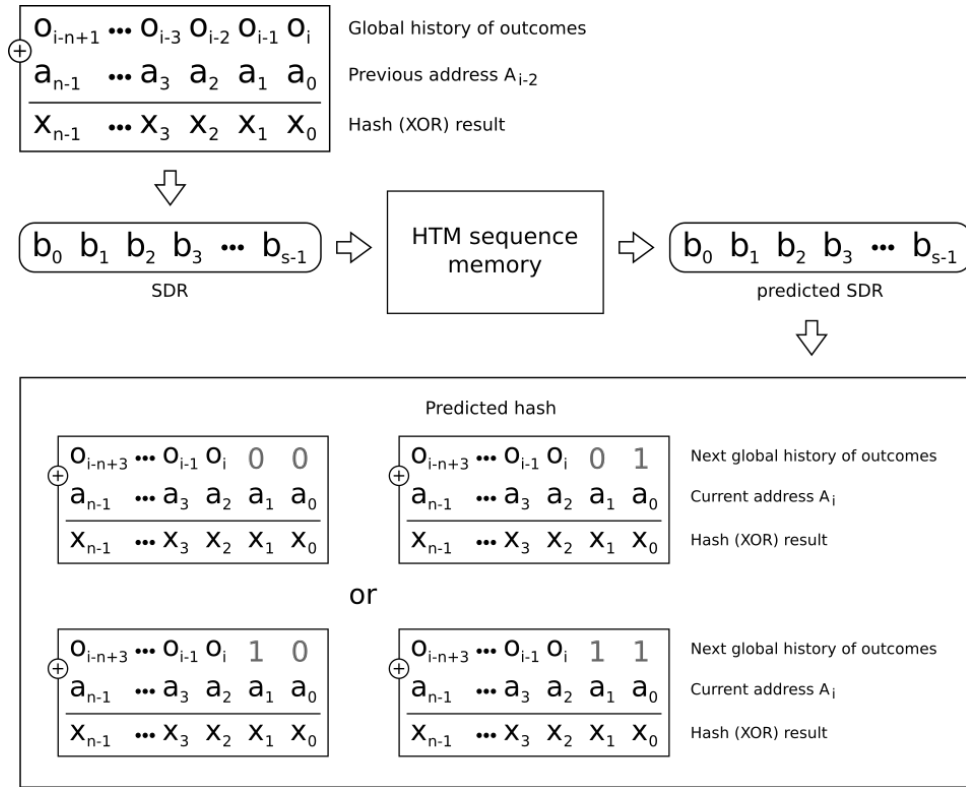
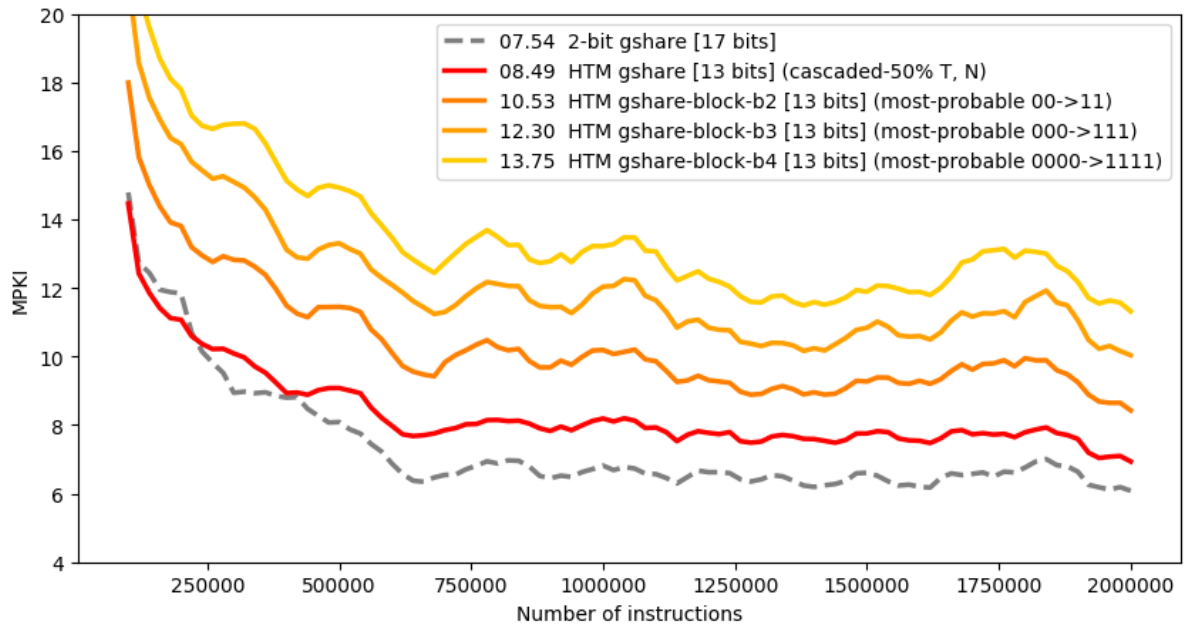


Figure 34 – Trend for the misprediction rate of the HTM gshare-block branch predictor



5.3 DESIGN EXPLORATION

After testing the above approaches, the best performing designs were explored further in order to inspect the result of small variations in the networks and verify their main limitations and behavior in different situations.

5.3.1 Longer traces

The first additional test to the HTM predictors intended to inspect the accuracy that can be achieved by its best designs both in the single e multiple predictions modalities. Therefore, the HTM gshare and HTM gshare-streams predictors were simulated for the first 8 million instructions of all traces provided by the 4th CBP.

Figure 35 – Trend for the misprediction rate of the HTM gshare branch predictor over 8 million instructions

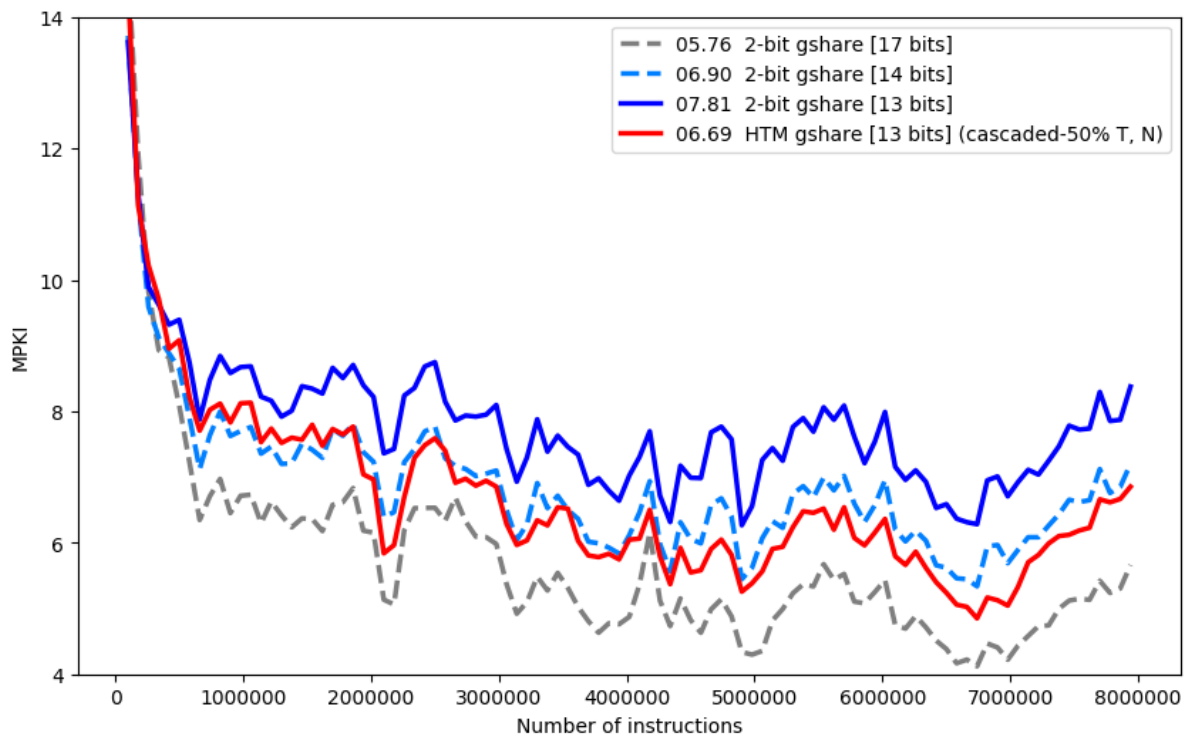
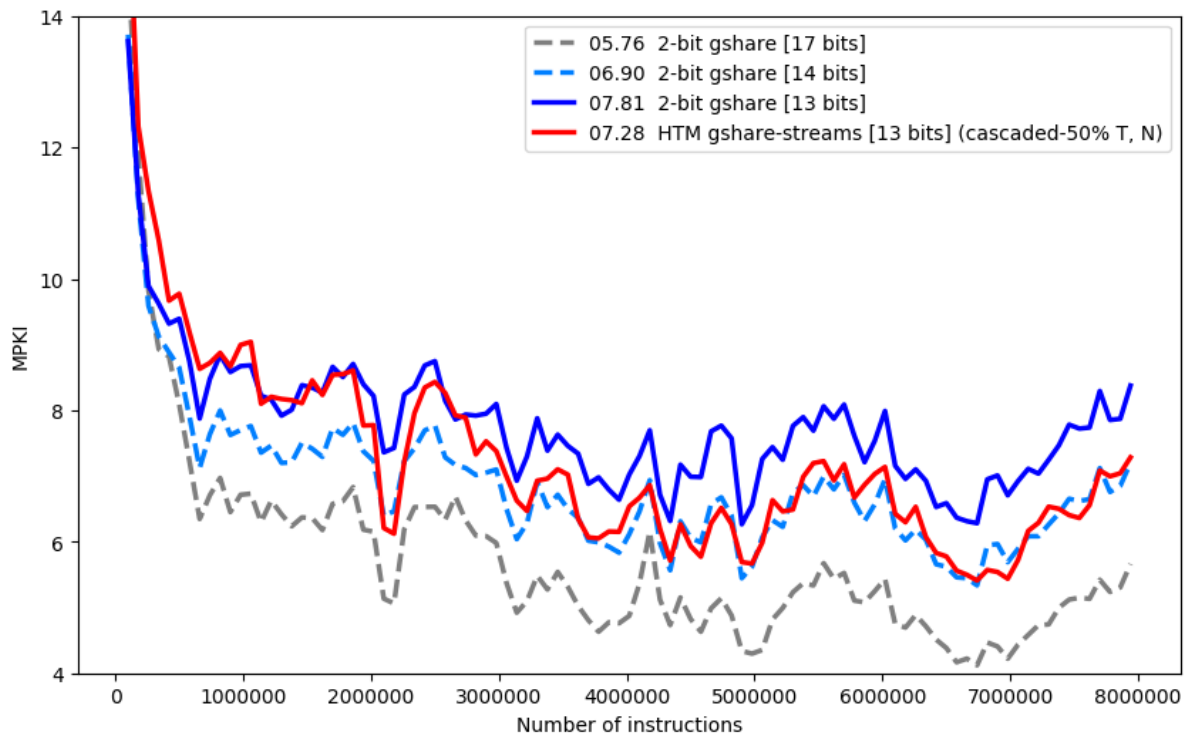


Figure 35 shows that the HTM gshare predictor consistently overcomes the accuracy of the traditional gshare predictor both with the same length and with one additional bit of data, decreasing MPKI scores by 14.3% and 3%, respectively. In comparison with the gshare implementation of 32KB, however, the HTM predictor still performs worse, with a MPKI score 16.1% higher.

Figure 36 – Trend for the misprediction rate of the HTM gshare-streams branch predictor over 8 million instructions



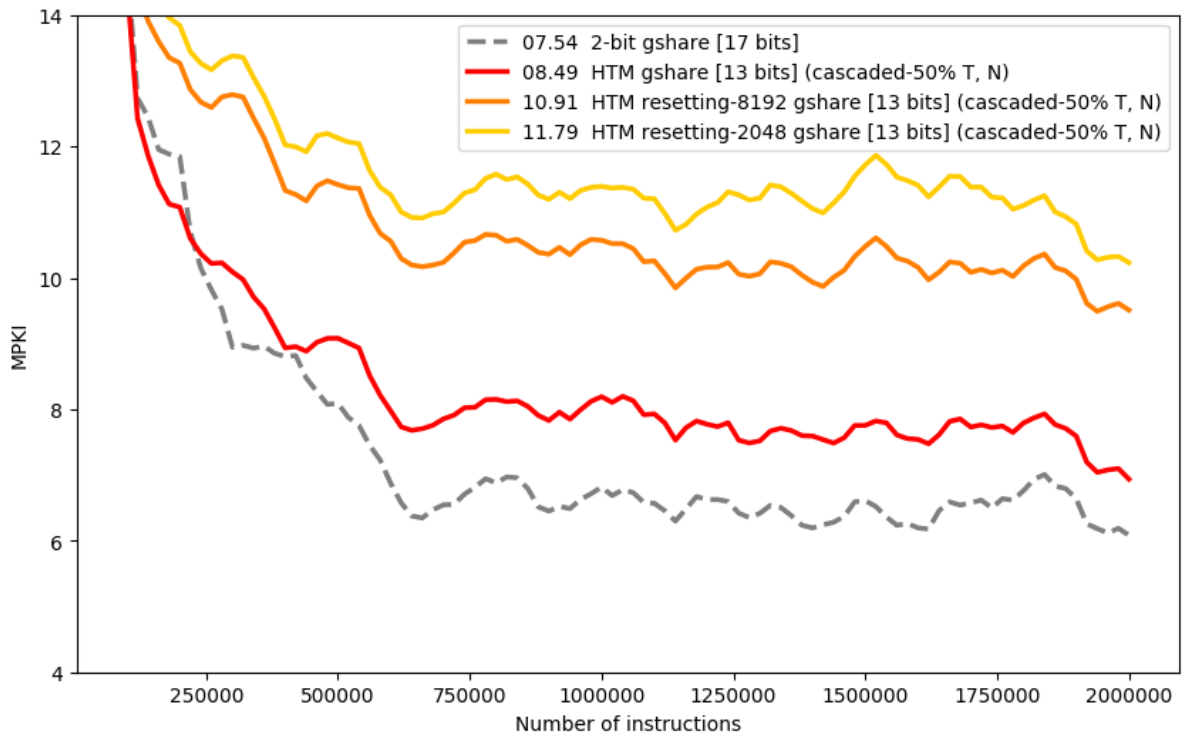
The HTM gshare-streams predictor also surpasses the traditional gshare predictor for the same length of data, yielding a misprediction rate 6.78% lower for 8 million instructions. As can be seen in figure 36, the HTM predictor capable of multiple predictions also stay in par with the gshare predictor with one extra bit of information as of 4 million instructions.

5.3.2 Network resetting

Resetting an HTM sequence memory means clearing all predicted neurons from the network in order to define the start of a new sequence, yet preserving all synapses permanences from previous training (NUMENTA, 2018b). Through resetting, the network can learn many unrelated sequences or even a single sequence quicker, since the system is told the starting and ending points of the series. This experiment has tried to improve the HTM branch predictor performance by using this technique.

Whenever a jump of branch addresses in the instructions flow exceeded a specified threshold, the network was reset. The procedure proved to be unappropriated in the context of branch prediction. As can be seen in figure 37, the accuracy is harmed when the method is applied.

Figure 37 – Trend for the misprediction rate of the HTM gshare branch predictor with resetting



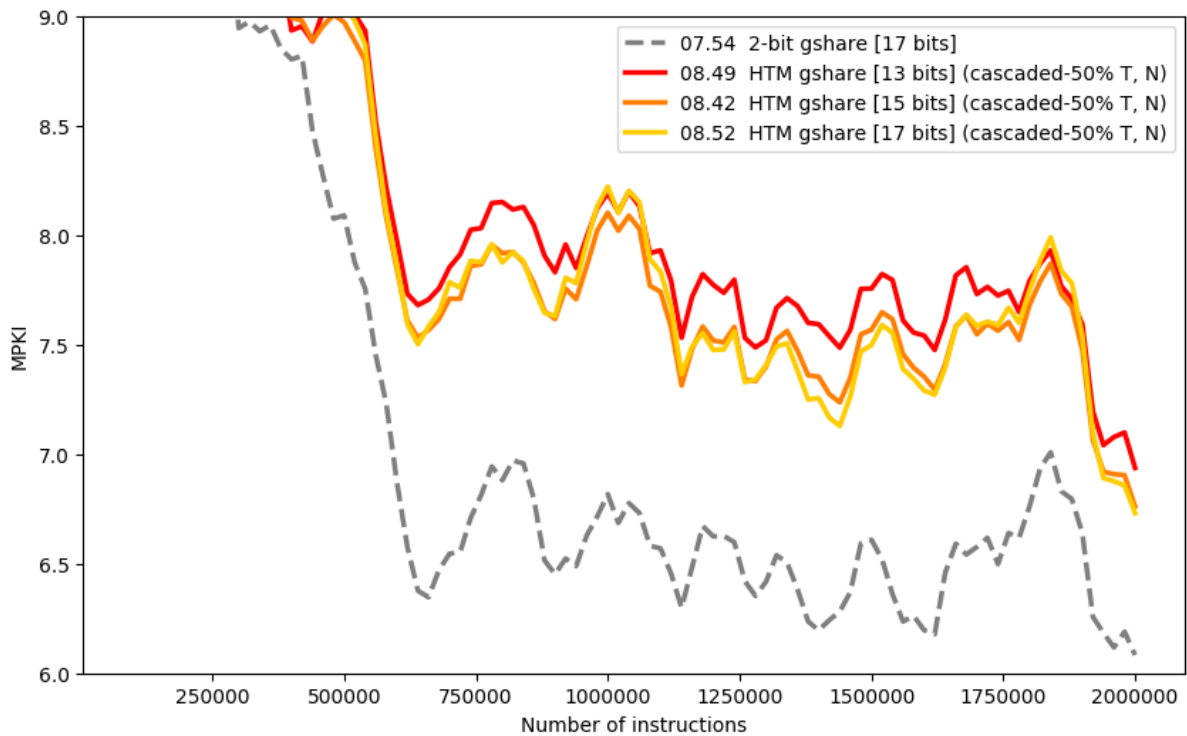
5.3.3 Limit of categories

Although small collisions in active bits of SDRs with no semantic relations are acceptable and the amount of unique SDRs in a network with 1024 columns is high, the HTM sequence memory can only identify up to a certain limit of categories until aliasing starts harming network performance more than the additional information provided to the network can improve it.

The experiments above have used 8192 total categories, each with a SDR of 10 active bits, which means that every column in the HTM sequence memory needs to represent 80 different categories. If only one extra bit of information is used to represent new categories, the total number of categories doubles, also doubling the amount of categories that use each of the bits in the SDRs space.

This experiment attempted to find the maximum amount of aliasing that the network in the described configurations is capable of handling before the occurrence of an inflexion in the predictor accuracy rate (more details in the next chapter). It is difficult to pinpoint the exact limit in the number of categories that can lead to increased performance, since using more categories also slows down learning, but figure 38 gives a hint: around 32768.

Figure 38 – Trend for the misprediction rate of the HTM gshare branch predictor with varying number of categories



5.3.4 SDR encoder seeds

The mapping from categories to SDRs, which is responsibility of the SDR encoder, is very important to avoid high amounts of aliasing in the representations. As this study has used a random encoder to create the mapping, this experiment intended to test the importance of the pseudo-random mapping generator seed to the results of the predictor. Ten simulations of 2 million instructions were run using different seeds for the random SDR encoder with the HTM gshare predictor. It was found that the different seeds do not significantly affect the performance of the predictor, since the coefficient of variation of MPKI scores stayed at 0.077%.

5.3.5 Context switch recovery

Branch predictors' adaptation to new execution profiles is important both within the same program and between different programs, as in the case of a context switch. In this experiment, slices of 2 million instructions from different execution traces were strapped together in order to test the ability of adaptation of the HTM predictor when execution traces change.

Figure 39 – Trend for the misprediction rate of the HTM gshare branch predictor over 4 million instructions with a context switch

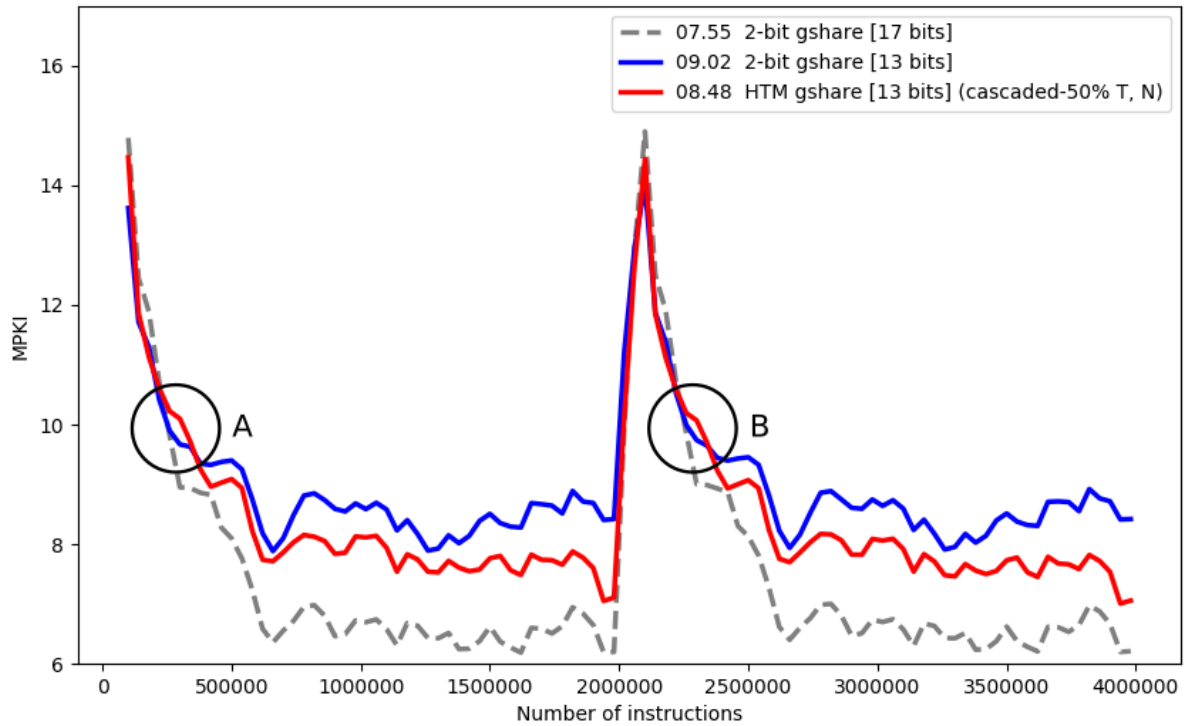


Figure 39 reveals that the HTM predictor can recover slightly faster than the traditional gshare predictor, which can be more easily noticed in the circled regions *A* and *B*. For the same area on the scoring curve, the misprediction rate of the HTM predictor approaches the misprediction rate of the traditional predictor faster in *B* than it does in *A*. While the traditional gshare predictor learns slower after a context switch, compared to its first run when all counters are initialized with a value of 2, the HTM network learns at the same rate both when the synapses' permanences are initialized at 0.21 (see table 1) and when the network has already learned a sequence.

6 DISCUSSION

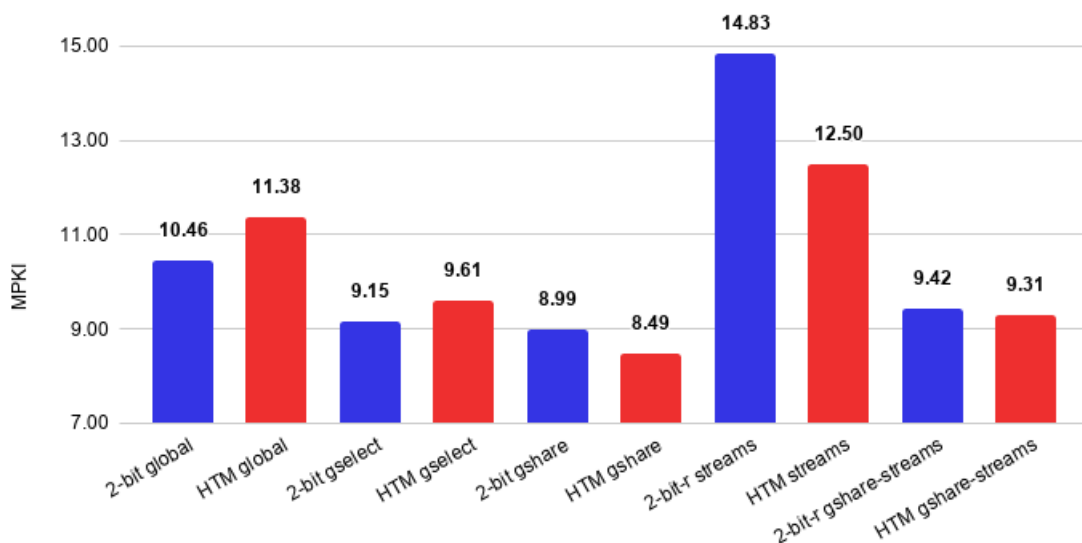
Although the results chapter has shown that HTM branch predictors, as described in this work, are able to handle the branch prediction task and deliver a reasonable accuracy, it is important to analyze what the HTM systems were really doing, what properties differentiate them from traditional branch predictors, which characteristics of the problem prevented better performance and what are the main limitations of the scheme related both to operation and implementation. This chapter discusses such topics.

6.1 PREDICTOR ANALYSIS

Working on improvements in the operation of HTM branch predictors requires understanding their true capabilities, flaws and advantages. The following analysis introduces some hypotheses to explain the operation of HTM networks in the tested environment.

Figure 40 shows a comparison of MPKI scores between HTM and traditional 2-bit counter branch predictors. It can be observed that for the global and gselect predictors, results show that the HTM network is able to learn transitions no better than an indexed table of 2-bit counters. That is, the learned transitions likely take into account only the current input of the network in order to predict the following SDR. This means that, for these schemes, the network was not able to identify consistent temporal contexts for the inputs and to create a chain of network activity that leads to improved predictions.

Figure 40 – Comparison of MPKI scores between HTM and non-HTM designs using a 13-bit history length representation of execution states, for 2 million instructions



However, for the gshare, streams and gshare-streams predictors, the charts demonstrate that the HTM systems are doing, indeed, more than just a short-sighted decision based only on the last entry provided to the network. Specially the 14.3% improvement in the misprediction rate

reported in subsection 5.3.1 evidences that some degree of temporal correlation was successfully detected and exploited by the HTM network. Two reasons might explain these distinct behaviors in apparently similar tasks.

As aforementioned in the results chapter, adding branch address information to the input data allows the network to use path-based correlations to improve prediction accuracy. However, it seems unlikely that this factor can fully explain why the HTM global branch predictor performs significantly worse than the traditional global predictor while the HTM gshare branch predictor surpasses the accuracy of the traditional gshare predictor. Instead, it is expected that the cause for the different behaviors among the predictors may be mainly aliasing.

It is important to distinguish between two kinds of aliasing that hurt the HTM predictors' performance. The first type is the same faced by traditional branch predictors, where the slice of recent execution history utilized is not enough to differentiate between execution states. The second level of aliasing faced by HTM branch predictors is encountered in the SDRs space, where categories share bit activations.

The former type of aliasing could be the responsible for most of the accuracy disparity. For instance, as using only the global history of outcomes incurs in more aliasing, many categories end up being repurposed along unrelated portions of the execution traces. Repurposing a category in an HTM network is, however, slower than repurposing a 2-bit counter. Moreover, the process can also destroy sequences that had already been learned. In the HTM gshare branch predictor this process occurs less often, giving the network more opportunities to learn longer sequences.

The main advantage expected from HTM branch predictors was the ability to learn very long term dependencies of branch outcomes that could work with richer information than traditional branch predictors use, like less compressed path data, while at the same time naturally adapting the history length needed for each prediction. Although only a maximum of 8 million instructions has been tested, results suggest that this strength could not manifest itself in a meaningful way for the task of branch prediction.

One of the reasons for the poor results of the HTM approach may be the presence of noise in the sequences used, which is a huge problem for the HTM sequence memory as reported by Cui, Ahmad and Hawkins (2016). The noise in this context refers to changes in the outcomes caused by reasons that were not exposed to the network, like values in registers. If an outcome changes because of circumstances not accessible by the network, the alteration may seem like it happened at random. Although HTM algorithms can tolerate high amounts of spatial noise through the HTM spatial pooler (CUI; AHMAD; HAWKINS, 2017), they can not handle temporal noise. Also, as the learning of long sequences in the network is a slow process, even a perfect training process could provide little value if the pattern is not very frequent.

Another limitation of the system resides in the maximum number of categories that the network can work with due to aliasing in the SDRs space, as evidenced in subsection 5.3.3. A high collision score degrades and possibly collapses previous learned transitions whenever a

new sequence is being learned. Since the mapping from categories to SDRs is fixed, even if not all possible categories are entered into the system, many columns of the network might be used by more than 32 different categories, making it difficult to uniquely identify a category even after combining activations from all columns.

Thus, fighting aliasing in SDRs is very important, although it is not a simple problem to solve. The easiest way to reduce this kind of aliasing is by decrementing the total number of categories, but that solution clashes against the objective of lowering the aliasing also in the representation of execution states. In a similar way, increasing the length of the slice of execution history to decrease the traditional kind of aliasing will cause more aliasing in the level of SDRs. There is, however, the option to dynamically assign SDRs only to utilized categories in a way that collisions are minimized. This procedure, though, adds complexity both to the SDR encoder and to the SDR decoder. Therefore, there is not a straightforward way of solving either kind of aliasing, and both of them were balanced together in this work to yield better results.

6.2 HARDWARE REQUIREMENTS AND ISSUES

The hardware implementation of HTM algorithms is still in its infancy, and only a few studies implement the HTM sequence memory with all of its characteristics. The simulated networks are still tiny, with only hundreds of neurons, and at least two orders of magnitude slower than what is needed for a branch predictor (LI; FRANZON, 2016; ZYARAH; KUDITHIPUDI, 2019b). The learning and prediction processes of an HTM branch predictor need to take place within one clock cycle, thus requiring an implementation with a massive parallelization of computations in a neuromorphic architecture very similar to that of biological brains. Ideally, all synapses should exist physically. That, however, is a big challenge because of the high connectivity of HTM networks. If every possible synapse existed as a 4 bit register in a network with 1024 columns, 32 cells per column and 64 segments per cell, then only maintaining their values would require a total of 32GB of storage. As most of the synapses permanences would store a 0 value, this scheme could be severely optimized by routing synapses, but the system would still need at least dozens of megabytes of storage.

One way of accelerating computations and, at the same time, eliminating the huge storage requirement for synapses permanences is to use new methods for building such connections. Instead of using registers to represent synapses weights digitally, analog hardware could be exploited. In special, memristive systems could greatly benefit HTM hardware implementations, along with any kind of cognitive hardware, as they could “store the synaptic weights as their conductance states and [...] perform the associated computational tasks in place” (BOYBAT et al., 2018). The development of this type of hardware is, however, also in its inception.

6.2.1 Learning with delayed feedback

A still not discussed issue faced by an HTM predictor implemented in a real environment would be learning and recovering from mispredictions. Unlike most branch predictors, an HTM branch predictor is completely serial. Therefore, predictions can not be accessed simultaneously as in a pattern history table (PHT), but rather the system needs to run speculatively in order to deliver predictions ahead of non-committed branches. This characteristic can make the process of learning in the network a complex task, as recovering from mispredictions should be considered as an important part of the total process. Four options for handling learning in the network are listed next.

1. Speculative learning: if learning was done speculatively, somehow all modifications to synapses should be stored for a given number of recent iterations. Then, if a misprediction occurred, it would be possible to rollback the speculated alterations. That, however, requires control over individual synapses at any point in time, which is not a guaranteed property for efficient neuromorphic hardware. Also, the amount of modifications in the permanence of synapses can be huge, thus causing a very significant overhead for the system.
2. Learning after commit with synapse modifications storage: synapses' alterations could be stored and applied to the network only when branch outcomes were committed. This approach, however, has the same problem as the first one, requiring control over individual synapses and having to store a large history of interactions.
3. Learning after commit with neuron activations storage: storing could be done at the level of neurons instead of the level of synapses. Instead of storing modifications to specific connections, only the sequence of neuron activations could be used. In this approach the learning could occur through a delayed replay of activations in the network, solving both of the problems described for the previous solution. The only factor to pay attention in this scheme would be conflicts with other activities happening in parallel in the network.
4. Duplicated networks: two networks could be run in parallel, with one of them learning speculatively and the other learning after commit. The predictions would come from the network who is running ahead. When the ahead predictor mispredicts a branch, the state of the network who is behind is entirely copied to the other predictor, recovering its correctness. In this approach the necessary hardware would double in size and the action of copying synapses states would be required. While not exactly the same, this scheme is equivalent to the first one presented.

While option 3 seems to be the best approach, it can also be the most complex to implement. Further research in the area must be done before the topic can be properly examined.

7 CONCLUSIONS

This study has explored the application of new artificial intelligence techniques in the branch prediction task by proposing and evaluating predictors built with HTM networks. More specifically, the problem was faced as a sequence prediction task and tackled by the HTM sequence memory.

Comparing HTM and non-HTM designs using a 13-bit history length representation of execution states has shown that the proposed HTM branch predictor can, in certain conditions, exploit temporal correlations and improve prediction accuracy over traditional 2-bit counter schemes. The solution, however, besides requiring a special and impractical amount of hardware compared to traditional schemes, could not scale presumably because of problems with aliasing and temporal noise.

Overall, in the proposed design the HTM networks have shown not to work well with the characteristics of the branch prediction problem. Still, many BP methods that use HTM algorithms as part of their operation remain to be tested, and it is hoped that results from this work can offer insights to new exploratory work both in the areas of branch prediction and applications for the HTM theory.

Despite unpromising results, there are several ways in which this study could be expanded. The main expected future works are: to test the hypotheses presented in the discussion chapter for the failing of the proposed approach; to integrate the HTM spatial pooler in the system, which could allow more data to be efficiently used by the predictor; and to explore techniques to decrease aliasing. Minor improvements to this study include testing more predictor designs; optimizing the HTM network parameters further; decreasing the network size; and simulating the HTM predictors for the entire set of execution traces provided by the 4th CBP in order to compare the results with other studies.

In the long run, it is important to note that the HTM theory defines biologically constrained components that are meant to constitute the building blocks of general intelligence. The flexibility of such systems, combined with a relatively constrained network size and incomplete operation, e.g., lack of hierarchy interactions, makes the approach perform in a poor way in comparison to simpler handcrafted methods that incorporate specific domain knowledge about the problem. However, HTM might still solve the branch prediction problem.

REFERENCES

- AHMAD, Subutai; HAWKINS, Jeff. How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. **arXiv e-prints**, arxiv:1601.00720, 2016.
- _____. Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory. **arXiv e-prints**, arxiv:1503.07469, 2015.
- AHMAD, Subutai; LAVIN, Alexander, et al. Unsupervised real-time anomaly detection for streaming data. **Neurocomputing**, vol. 262, pp. 134–147, 2017. DOI: 10.1016/j.neucom.2017.04.070.
- AHMAD, Subutai; SCHEINKMAN, Luiz. How Can We Be So Dense? The Benefits of Using Highly Sparse Representations. **arXiv e-prints**, arxiv:1903.11257, 2019.
- BALL, Thomas; LARUS, James R. Branch prediction for free. **ACM SIGPLAN Notices**, vol. 28, no. 6, pp. 300–313, 1993. DOI: 10.1145/173262.155119.
- BILLAUELLE, Sebastian; AHMAD, Subutai. Porting HTM Models to the Heidelberg Neuromorphic Computing Platform. **arXiv e-prints**, arxiv:1505.02142, 2015.
- BLOCK, H. D. The Perceptron: A Model for Brain Functioning. I. **Reviews of Modern Physics**, vol. 34, no. 1, pp. 123–135, 1962. DOI: 10.1103/revmodphys.34.123.
- BOYBAT, Irem et al. Neuromorphic computing with multi-memristive synapses. **Nature Communications**, Springer Science and Business Media LLC, vol. 9, no. 1, 2018. DOI: 10.1038/s41467-018-04933-y.
- CHANG, M.-C.; CHOU, Y.-W. Branch prediction using both global and local branch history information. **IEE Proceedings - Computers and Digital Techniques**, vol. 149, no. 2, p. 33, 2002. DOI: 10.1049/ip-cdt:20020273.
- CSLS/THE UNIVERSITY OF TOKYO. **To What Extent Is the Brain Understood?: Neurons**. 2010. Address: <http://csls-text2.c.u-tokyo.ac.jp/inactive/06_03.html>. Visited on: June 21, 2019.
- CUI, Yuwei; AHMAD, Subutai; HAWKINS, Jeff. Continuous Online Sequence Learning with an Unsupervised Neural Network Model. **Neural Computation**, vol. 28, no. 11, pp. 2474–2504, 2016. DOI: 10.1162/neco_a_00893.
- _____. The HTM Spatial Pooler—A Neocortical Algorithm for Online Sparse Distributed Coding. **Frontiers in Computational Neuroscience**, vol. 11, p. 111, 2017. DOI: 10.3389/fncom.2017.00111.
- DRIESEN, Karel; HLZLE, Urs. Limits of Indirect Branch Prediction, Jan. 1998.

- DUNDAS, James. **Announcement of results and awards - Championship Branch Prediction (CBP-5)**. June 18, 2016. Address: <<http://www.jilp.org/cbp2016/slides/CbpOrgChairPresentation.pptx>>. Visited on: May 28, 2019.
- EVERS, Marius et al. An analysis of correlation and predictability. **ACM SIGARCH Computer Architecture News**, vol. 26, no. 3, pp. 52–61, 1998. DOI: 10.1145/279361.279368.
- FELDMANN, J. et al. All-optical spiking neurosynaptic networks with self-learning capabilities. **Nature**, vol. 569, no. 7755, pp. 208–214, 2019. DOI: 10.1038/s41586-019-1157-8.
- GOPE, Dibakar; LIPASTI, Mikko H. Bias-Free Branch Predictor. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. [S.l. S.n.], 2014. pp. 521–532. DOI: 10.1109/micro.2014.32.
- HARRIS, David Money; HARRIS, Sarah L. **Digital design and computer architecture**. Waltham, MA: Morgan Kaufmann, 2013. ISBN 9780123944245.
- HAWKINS, Jeff; AHMAD, Subutai. Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex. **Frontiers in Neural Circuits**, vol. 10, p. 23, 2016. DOI: 10.3389/fncir.2016.00023.
- HAWKINS, Jeff; AHMAD, Subutai; CUI, Yuwei. A Theory of How Columns in the Neocortex Enable Learning the Structure of the World. **Frontiers in Neural Circuits**, vol. 11, p. 81, 2017. DOI: 10.3389/fncir.2017.00081.
- HAWKINS, Jeff; AHMAD, Subutai; PURDY, Scott, et al. Biological and Machine Intelligence (BAMI). Initial online release 0.4. [S.l.], 2016. Address: <<https://numenta.com/resources/biological-and-machine-intelligence/>>.
- HAWKINS, Jeff; BLAKESLEE, Sandra. **On Intelligence: How a New Understanding of the Brain Will Lead to the Creation of Truly Intelligent Machines**. [S.l.]: Times Books, 2004. ISBN 0805074562.
- HAWKINS, Jeff; LEWIS, Marcus, et al. A Framework for Intelligence and Cortical Function Based on Grid Cells in the Neocortex. **Frontiers in Neural Circuits**, vol. 12, p. 121, 2019. DOI: 10.3389/fncir.2018.00121.
- HEIL, Timothy H.; SMITH, Zak; SMITH, James E. Improving Branch Predictors by Correlating on Data Values. In: PROCEEDINGS of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture. [S.l. S.n.], 1999. pp. 28–37.
- HENNESSY, John L.; PATTERSON, David A. **Computer Architecture, Fourth Edition: A Quantitative Approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 0123704901.
- HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-Term Memory. **Neural Computation**, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.

INTEL CORPORATION. **Intel® 64 and IA-32 Architectures Optimization Reference Manual**. [S.l. S.n.], 2019.

JACOBSON, Quinn; ROTENBERG, Eric; SMITH, James E. Path-based Next Trace Prediction. In: PROCEEDINGS of the 30th Annual ACM/IEEE International Symposium on Microarchitecture. [S.l. S.n.], 1997. pp. 14–23.

JILP. **Championship Branch Prediction (CBP-4)**. 2014. Address: <<https://www.jilp.org/cbp2014/>>. Visited on: June 9, 2019.

_____. **Championship Branch Prediction (CBP-5)**. 2016. Address: <<https://www.jilp.org/cbp2016/>>. Visited on: May 28, 2019.

_____. **Championship Branch Prediction Program**. June 18, 2016. Address: <<https://www.jilp.org/cbp2016/program.html>>. Visited on: May 28, 2019.

_____. **How to Use the Branch Prediction Championship Kit**. 2014. Address: <<https://www.jilp.org/cbp2014/framework.html>>. Visited on: June 9, 2019.

JIMÉNEZ, D. A. An optimized scaled neural branch predictor. In: 2011 IEEE 29th International Conference on Computer Design (ICCD). [S.l. S.n.], 2011. pp. 113–118. DOI: 10.1109/ICCD.2011.6081385.

_____. Fast path-based neural branch prediction. In: 22ND Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449). [S.l. S.n.], 2003. DOI: 10.1109/micro.2003.1253199.

JIMÉNEZ, D. A.; KECKLER, S.W.; LIN, C. The impact of delay on the design of branch predictors. In: PROCEEDINGS 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000. [S.l. S.n.], 2000. DOI: 10.1109/micro.2000.898059.

JIMÉNEZ, D. A.; LIN, C. Dynamic branch prediction with perceptrons. In: PROCEEDINGS HPCA Seventh International Symposium on High-Performance Computer Architecture. [S.l. S.n.], 2001. pp. 197–206. DOI: 10.1109/hpca.2001.903263.

JIMÉNEZ, D. A.; LIN, Calvin. Neural methods for dynamic branch prediction. **ACM Transactions on Computer Systems**, vol. 20, no. 4, pp. 369–397, 2002. DOI: 10.1145/571637.571639.

KARRAS, Tero; LAINE, Samuli; AILA, Timo. A Style-Based Generator Architecture for Generative Adversarial Networks. **arXiv e-prints**, arxiv:1812.04948, 2018.

KLUKAS, Mirko; LEWIS, Marcus; FIETE, Ila. Flexible representation and memory of higher-dimensional cognitive variables with grid cells. **bioRxiv**, Cold Spring Harbor Laboratory, 2019. DOI: 10.1101/578641.

KOCHER, Paul et al. Spectre Attacks: Exploiting Speculative Execution. **arXiv e-prints**, arxiv:1801.01203, 2018.

- KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet classification with deep convolutional neural networks. **Communications of the ACM**, vol. 60, no. 6, pp. 84–90, 2017. DOI: [10.1145/3065386](https://doi.org/10.1145/3065386).
- LAVIN, Alexander; AHMAD, Subutai. Evaluating Real-Time Anomaly Detection Algorithms – The Numenta Anomaly Benchmark. In: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA). [S.n.], 2015. pp. 38–44. DOI: [10.1109/icmla.2015.141](https://doi.org/10.1109/icmla.2015.141).
- LEWIS, Marcus et al. Locations in the Neocortex: A Theory of Sensorimotor Object Recognition Using Cortical Grid Cells. **Frontiers in Neural Circuits**, vol. 13, p. 22, 2019. DOI: [10.3389/fncir.2019.00022](https://doi.org/10.3389/fncir.2019.00022).
- LI, Weifu; FRANZON, Paul. Hardware implementation of Hierarchical Temporal Memory algorithm. In: 2016 29th IEEE International System-on-Chip Conference (SOCC). [S.l.]: IEEE, 2016. DOI: [10.1109/socc.2016.7905453](https://doi.org/10.1109/socc.2016.7905453).
- LOWEL, S; SINGER, W. Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity. **Science**, vol. 255, no. 5041, pp. 209–212, 1992. DOI: [10.1126/science.1372754](https://doi.org/10.1126/science.1372754).
- LU, Zhijian et al. Alloyed Branch History: Combining Global and Local Branch History for Robust Performance. **International Journal of Parallel Programming**, vol. 31, no. 2, pp. 137–177, 2003. DOI: [10.1023/a:1022669325321](https://doi.org/10.1023/a:1022669325321).
- MCFARLING, Scott. **Combining Branch Predictors**. [S.l. S.n.], 1993.
- MCILROY, Ross et al. Spectre is here to stay: An analysis of side-channels and speculative execution. **arXiv e-prints**, arxiv:1902.05178, 2019.
- MITTAL, Sparsh. A survey of techniques for dynamic branch prediction. **Concurrency and Computation: Practice and Experience**, vol. 31, no. 1, e4666, 2018. DOI: [10.1002/cpe.4666](https://doi.org/10.1002/cpe.4666).
- NAIR, Ravi. Dynamic Path-based Branch Correlation. In: PROCEEDINGS of the 28th Annual International Symposium on Microarchitecture. [S.l. S.n.], 1995. pp. 15–23.
- NUMENTA. **Numenta Platform for Intelligent Computing**. [S.l.]: GitHub, 2018. <https://github.com/numenta/nupic>.
- _____. **Sequence Memory**. 2018. Address: <https://nupic.docs.numenta.org/1.0.4.dev0/api/algorithms/sequence-memory.html>. Visited on: Nov. 18, 2019.
- PAN, Shien-Tai; SO, Kimming; RAHMEH, Joseph T. Improving the accuracy of dynamic branch prediction using branch correlation. **ACM SIGPLAN Notices**, vol. 27, no. 9, pp. 76–84, 1992. DOI: [10.1145/143371.143490](https://doi.org/10.1145/143371.143490).
- PUENTE, Valentin; ÁNGEL GREGORIO, José. CLAASIC: a Cortex-Inspired Hardware Accelerator. **arXiv e-prints**, arxiv:1604.05897, 2016.

- PURDY, Scott. Encoding Data for HTM Systems. **arXiv e-prints**, arxiv:1602.05925, 2016.
- PUTIC, Mateja; VARSHNEYA, A.J.; STAN, Mircea R. Hierarchical Temporal Memory on the Automata Processor. **IEEE Micro**, vol. 37, no. 1, pp. 52–59, 2017. DOI: 10.1109/mm.2017.6.
- RAHMAN, Shah Mohammad Faizur; WANG, Zhe; JIMÉNEZ, D. A. Studying microarchitectural structures with object code reordering. In: PROCEEDINGS of the Workshop on Binary Instrumentation and Applications - WBIA '09. [S.l. S.n.], 2009. pp. 7–16. DOI: 10.1145/1791194.1791196.
- RAMIREZ, A. et al. Fetching instruction streams. In: 35TH Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings. [S.l. S.n.], 2002. DOI: 10.1109/micro.2002.1176264.
- RAU, Bantwal Ramakrishna; FISHER, Joseph A. Instruction-Level Parallel Processing: History, Overview, and Perspective. **The Journal of Supercomputing**, vol. 7, pp. 9–50, 1993. DOI: 10.1007/978-1-4615-3200-2_3.
- RODRIGUEZ, Maria A.; KOTAGIRI, Ramamohanarao; BUYYA, Rajkumar. Detecting performance anomalies in scientific workflows using hierarchical temporal memory. **Future Generation Computer Systems**, vol. 88, pp. 624–635, 2018. DOI: 10.1016/j.future.2018.05.014.
- SANTAMBROGIO, Marco D.; CAMPANONI, Simone. **Branch Hazards and Static Branch Prediction Techniques**. 2014. Address: <<https://slideplayer.com/slide/9372894/>>. Visited on: June 13, 2019.
- SANTANA, O.J.; RAMIREZ, A.; VALERO, M. Latency tolerant branch predictors. In: INNOVATIVE Architecture for Future Generation High-Performance Processors and Systems, 2003. [S.l. S.n.], 2003. DOI: 10.1109/iwia.2003.1262780.
- SCHUMAN, Catherine et al. A Survey of Neuromorphic Computing and Neural Networks in Hardware, 2017.
- SEZNEC, André. A new case for the TAGE branch predictor. In: PROCEEDINGS of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11. [S.l. S.n.], 2011. pp. 117–127. DOI: 10.1145/2155620.2155635.
- _____. Analysis of the O-GEometric History Length Branch Predictor. In: 32ND International Symposium on Computer Architecture (ISCA'05). [S.l. S.n.], 2005. DOI: 10.1109/isca.2005.13.
- _____. Exploring branch predictability limits with the MTAGE+SC predictor *. In: 5TH JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5). [S.l. S.n.], 2016.

- SEZNEC, André; JOURDAN, Stéphan, et al. Multiple-block ahead branch predictors. In: PROCEEDINGS of the seventh international conference on Architectural support for programming languages and operating systems - ASPLOS-VII. [S.l. S.n.], 1996. DOI: 10.1145/237090.237169.
- SEZNEC, André; MICHAUD, Pierre. A case for (partially) TAgged GEometric history length branch prediction. **Journal of Instruction-level Parallelism (JILP)**, vol. 8, 2006.
- SHEN, Jonathan et al. Natural TTS Synthesis by Conditioning Wavenet on MEL Spectrogram Predictions. In: 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). [S.l. S.n.], 2018. DOI: 10.1109/icassp.2018.8461368.
- SHERWOOD, Timothy; CALDER, Brad. Loop Termination Prediction. In: LECTURE Notes in Computer Science. [S.l.]: Springer Berlin Heidelberg, 2000. pp. 73–87. DOI: 10.1007/3-540-39999-2_8.
- SILVER, David et al. Mastering the game of Go without human knowledge. **Nature**, vol. 550, no. 7676, pp. 354–359, 2017. DOI: 10.1038/nature24270.
- SMITH, James E. A study of branch prediction strategies. In: 25 years of the international symposia on Computer architecture (selected papers) - ISCA '98. [S.l. S.n.], 1998. pp. 202–215. DOI: 10.1145/285930.285980.
- SPRANGLE, Eric et al. The agree predictor: a mechanism for reducing negative branch history interference. In: PROCEEDINGS of the 24th annual international symposium on Computer architecture - ISCA '97. [S.l. S.n.], 1997. DOI: 10.1145/264107.264210.
- SPRUSTON, Nelson. Pyramidal neurons: dendritic structure and synaptic integration. **Nature Reviews Neuroscience**, vol. 9, no. 3, pp. 206–221, 2008. DOI: 10.1038/nrn2286.
- STREAT, Lennard; KUDITHIPUDI, Dhiresha; GOMEZ, Kevin. Non-volatile Hierarchical Temporal Memory: Hardware for Spatial Pooling. **arXiv e-prints**, arxiv:1611.02792, 2016.
- WALTER, Florian et al. Towards a neuromorphic implementation of hierarchical temporal memory on SpiNNaker. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS). [S.l. S.n.], 2017. pp. 1–4. DOI: 10.1109/iscas.2017.8050983.
- WANG, Chundong et al. A Distributed Anomaly Detection System for In-Vehicle Network Using HTM. **IEEE Access**, vol. 6, pp. 9091–9098, 2018. DOI: 10.1109/access.2018.2799210.
- YEH, Tse-Yu; PATT, Yale N. Alternative implementations of two-level adaptive branch prediction. In: PROCEEDINGS of the 19th annual international symposium on Computer architecture - ISCA '92. [S.l. S.n.], 1992. DOI: 10.1145/139669.139709.
- _____. Two-level adaptive training branch prediction. In: PROCEEDINGS of the 24th annual international symposium on Microarchitecture - MICRO 24. [S.l. S.n.], 1991. DOI: 10.1145/123465.123475.

ZYARAH, Abdullah M.; KUDITHIPUDI, Dhireesha. Neuromemrisitive Architecture of HTM with On-Device Learning and Neurogenesis. **ACM Journal on Emerging Technologies in Computing Systems**, vol. 15, no. 3, pp. 1–24, 2019. DOI: [10.1145/3300971](https://doi.org/10.1145/3300971).

_____. Neuromorphic Architecture for the Hierarchical Temporal Memory. **IEEE Transactions on Emerging Topics in Computational Intelligence**, vol. 3, no. 1, pp. 4–14, 2019. DOI: [10.1109/tetci.2018.2850314](https://doi.org/10.1109/tetci.2018.2850314).