Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

11-18-2008

Hardware Architectures and Implementations for Associative Memories : the Building Blocks of Hierarchically Distributed Memories

Changjian Gao
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Electrical and Computer Engineering Commons
Let us know how access to this document benefits you.

Recommended Citation

Gao, Changjian, "Hardware Architectures and Implementations for Associative Memories : the Building Blocks of Hierarchically Distributed Memories" (2008). *Dissertations and Theses.* Paper 6173.
https://doi.org/10.15760/etd.8033

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

HARDWARE ARCHITECTURES AND IMPLEMENTATIONS FOR ASSOCIATIVE MEMORIES – THE BUILDING BLOCKS OF HIERARCHICALLY DISTRIBUTED MEMORIES

by

CHANGJIAN GAO

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
2008

DISSERTATION APPROVAL

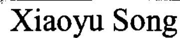
The abstract and dissertation of Changjian Gao for the Doctor of Philosophy in Electrical and Computer Engineering were presented November 14, 2008, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

Dan Hammerstrom, Chair

Malgorzata Chrzanowska-Jeske

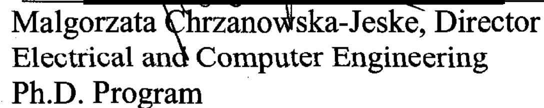
Douglas Hall



Xiaoyu Song

Fei Xie
Representative of the Office of Graduate Studies

DOCTOR PROGRAM APPROVAL:



Malgorzata Chrzanowska-Jeske, Director
Electrical and Computer Engineering
Ph.D. Program

ABSTRACT

An abstract of the dissertation of Changjian Gao for the Doctor of Philosophy in Electrical and Computer Engineering, presented November 14, 2008.

Title: Hardware Architectures and Implementations for Associative Memories – the Building Blocks of Hierarchically Distributed Memories

During the past several decades, the semiconductor industry has grown into a global industry with revenues around $300 billion. Intel, no longer relies on only transistor scaling for higher CPU performance, but instead, focuses more on multiple cores on a single die. It has been projected that in 2016 most CMOS circuits will be manufactured with 22 nm process. The CMOS circuits will have a large number of defects. Especially when the transistor goes below sub-micron, the original deterministic circuits will start having probabilistic characteristics. Hence, it would be challenging to map traditional computational models onto probabilistic circuits, suggesting a need for fault-tolerant computational algorithms. Biologically inspired algorithms, or associative memories (AMs) – the building blocks of cortical hierarchically distributed memories (HDMs) discussed in this dissertation, exhibit a remarkable match to the nano-scale electronics, besides having great fault-tolerance ability. Research on the potential mapping of the HDM onto CMOL (hybrid CMOS /

nanoelectronic circuits) nanogrids provides useful insight into the development of non-von Neumann neuromorphic architectures and semiconductor industry.

In this dissertation, we investigated the implementations of AMs on different hardware platforms, including microprocessor based personal computer (PC), PC cluster, field programmable gate arrays (FPGA), CMOS, and CMOL nanogrids. We studied two types of neural associative memory models, with and without temporal information.

In this research, we first decomposed the computational models into basic and common operations, such as matrix-vector inner-product and *k*-winners-take-all (*k*-WTA). We then analyzed the baseline performance/price ratio of implementing the AMs with a PC. We continued with a similar performance/price analysis of the implementations on more parallel hardware platforms, such as PC cluster and FPGA. However, the majority of the research emphasized on the implementations with all digital and mixed-signal full-custom CMOS and CMOL nanogrids.

In this dissertation, we draw the conclusion that the mixed-signal CMOL nanogrids exhibit the best performance/price ratio over other hardware platforms. We also highlighted some of the trade-offs between dedicated and virtualized hardware circuits for the HDM models. A simple time-multiplexing scheme for the digital CMOS implementations can achieve comparable throughput as the mixed-signal CMOL nanogrids.

TABLE OF CONTENTS

List of Tables iv

List of Figures v

List of Abbreviations ix

1. Introduction 1
   1.1. Research Background and Motivations 1
   1.2. Objectives and Contributions 14
      1.2.1. Objectives 14
      1.2.2. Contributions 15
   1.3. Thesis Organization 20
2. AM Algorithms and Hardware Architectures 23
   2.1. Introduction 23
   2.2. Algorithm 24
      2.2.1. HDM 24
      2.2.2. Bayesian Memory and Associative Memory 29
   2.3. Virtualization in Hardware Resources 48
   2.4. Hardware Architectures for AM 56
      2.4.1. WPNAM-like Model Hardware Implementations Review 56
      2.4.2. Hardware Architectures Used in This Work 63
   2.5. Methodology 68
3. Design with General-Purpose Architectures 72
   3.1. Introduction 72
   3.2. Implementation with PC 73
      3.2.1. CSIM Introduction 73
      3.2.2. Network Configurations and Vtune Introduction 74
      3.2.3. Average Memory Access Time for Data (AMATD) 76
      3.2.4. Column-wise Inner-product 77
      3.2.5. Simulation Results 80

3.3. Implementation with PC Cluster 85
3.4. Performance Comparison between PC and PC Cluster 89
4. Design with Reconfigurable Architectures 91
4.1. Introduction 91
4.2. A Case Study of FPGA Implementation 92
4.2.1. FPGA Implementation 94
4.2.2. Simulation Results 99
4.3. Implementation Analysis with the Relogix FPGA Board 101
4.3.1. FPGA System Architecture 102
4.3.2. Performance Analysis 110
4.3.3. Performance Comparison between the P4 and the Relogix FPGA Board 113
4.4. Conclusion 115
5. Design with Special-Purpose Architectures (CMOS) 117
5.1. Introduction 117
5.2. Implementation with Digital CMOS 118
5.2.1. Non-spiking AM Model 118
5.2.2. Spiking AM Model 121
5.3. Implementation with Mixed-signal CMOS 132
5.3.1. Non-Spiking Mixed-signal CMOS Design 132
5.3.2. Spiking Mixed-signal CMOS design 134
6. Design with CMOL 136
6.1. Introduction 136
6.1.1. Introduction of Nanoscale Devices 138
6.1.2. Circuits 146
6.1.3. Other Nanoarchitecture 155
6.1.4. Performance Modeling of Nano-structures 157
6.2. Implementation with Digital CMOL 170
6.2.1. Non-spiking AM Model 170
6.2.2. Spiking AM Model 171

6.3. Implementation with Mixed-signal CMOL 171
6.3.1. Non-spiking AM Model 171
6.3.2. Spiking AM Model 178
6.4. Performance/price Comparisons between CMOS and CMOL 180
6.5. Results and Discussion 182
7. Summary and Future Work 190
7.1. Summary and Conclusions 190
7.2. Future Work 195
8. References 197

LIST OF TABLES

Table 2-1: Neocortex neuron counts in several mammals. 26
Table 2-2: Data precisions for AM. 43
Table 3-1: Configurations of the PALM associative neural networks. 75
Table 3-2: The overall performance results for the sparsely represented matrix with release compiler mode and speed optimizations enabled. 81
Table 3-3: The overall performance results for the fully represented matrix with release compiler mode and speed optimizations enabled. 81
Table 3-4: The hotspot OutGenObj::Execute() performance results for the sparsely represented matrix with release compiler mode and speed optimizations enabled. 82
Table 3-5: The hotspot OutGenObj::Execute() for the full-matrix representation with release compiler mode and speed optimizations enabled. 83
Table 3-6: PC cluster (8 processors) simulation results. 88
Table 4-1: Performance comparison of the WPNAM implementations on PC and D2 FPGA board. 99
Table 4-2: The time for the inner-product operation for the full-weight matrix representation. The weights are obtained from the SDRAM, the test vector is stored inside the FPGA. 111
Table 4-3: Time for k-WTA and the total time for the inner-product and k-WTA. 113
Table 6-1: Components for different systems of non-spiking AM model. 180
Table 6-2: Components for different systems of spiking AM model. 181
Table 6-3: Circuit performance/price scaling. 182
Table 6-4: Performance/price comparison for the non-spiking AM model. 183
Table 6-5: Performance/price comparison for spiking AM model. 187

LIST OF FIGURES

Figure 1-1: Parallelism increases with the evolution of Intel® microarchitecture.............. 5
Figure 2-1: Hierarchically distributed memory structure in the Mountcastle sense. 27
Figure 2-2: The "decoder" model of association... 30
Figure 2-3: Communication channel.. 31
Figure 2-4: Hawkins and George's HTM structure... 33
Figure 2-5: A three-BM network (adapted from [72]). 34
Figure 2-6: With light emitted from C through B to A, the original spots on A are generally brighter.. 36
Figure 2-7: The architecture of the Willshaw and Palm NAM model. 37
Figure 2-8: Spiking neuron model.. 45
Figure 2-9: A possible RC circuit model for an I&F neuron. 47
Figure 2-10: Hardware spectrum for the implementation of biologically inspired networks. 51
Figure 2-11: Processing nodes (PNs) time-multiplex neuron emulation. 55
Figure 2-12: The architecture of the BACCHUS III chip. 57
Figure 2-13: Neuron unit block diagram of the Porrmann's FPGA design for the WPNAM model.. 58
Figure 2-14: CNAPS system architecture. 60
Figure 2-15: CNAPS PN architecture. 61
Figure 2-16: CrossNets.. 63
Figure 2-17: Methodology flow chart used in this work, i.e., performance/price ratio comparisons for AM algorithm hardware implementations. 68
Figure 3-1: Memory hierarchy test for Pentium 4 machine (DELL Dimension 8100).. 77
Figure 3-2: The row-wise inner-product. 79

Figure 3-3: The column-wise matrix-vector inner-product. 80
Figure 3-4: The relationship between the number of network nodes (vector elements) and the inner-product memory bandwidth for the Pentium 4. 83
Figure 3-5: Vector size versus node update rate for Pentium 4. 84
Figure 3-6: Illustration of the weight matrix distribution for the PC cluster (4 PCs example). 88
Figure 3-7: Node update rate for the P4 and the PIII cluster. 89
Figure 4-1: EVS associative memory application. 93
Figure 4-2: System organization and data flow for the EVS system with the WPNAM algorithm implemented on the D2 FPGA board. 96
Figure 4-3: FPGA functional blocks - D2 FPGA board implementation. 97
Figure 4-4: Weight matrix data structure. 98
Figure 4-5: The data structure in the "result vector" unit. 98
Figure 4-6: Node update rate (node outputs computed per second) for P4 and FPGA versus the vector dimension, *n*. 101
Figure 4-7 The basic Relogix FPGA board. 103
Figure 4-8: FPGA board components block diagram. For the performance analysis, the FPGA_x is Xilinx XC2V1000–5, the DRAM_x is a 64-bit 133 MHz DDR SDRAM DIMM. 105
Figure 4-9: Weights distribution in the DRAMs. 106
Figure 4-10: Relogix FPGA functional block diagram. 107
Figure 4-11: Node update rate for P4 and Relogix FPGA. 114
Figure 5-1: The functional partitioning of the four configurations for CMOS and CMOL implementations. 119
Figure 5-2: CMOS column processor system and functional blocks in a single PN for the spike-timing-dependent AM model. 124
Figure 5-3: Memory architectures inside CP. 125
Figure 5-4: Average waiting time in terms of firing rate λ, according to (5.1). 129
Figure 5-5: Normalized (weight read) time of multiple-weight read with the network size of 16,384. 131

Figure 5-6: Schematic view of the *k*-WTA circuit. 134

Figure 6-1: Two different views of a nanoscale crossbar. 140

Figure 6-2: Nanoscale molecular-switch crossbar circuit from Chen [34]. 141

Figure 6-3: The crossbar as a 64-bit random access memory. 142

Figure 6-4: Schematic diagram illustrating the basic function of a latch used to transform a bidirectional switch (memory) into a voltage (logic) representation for representing logical data values. 143

Figure 6-5: Two-terminal latching switch. 145

Figure 6-6: The generic CMOL circuit. 147

Figure 6-7: The system architecture of CMOL memory. 148

Figure 6-8: Logic blocks implemented with a complementary/symmetry array. 149

Figure 6-9: Implementing two logic functions by selectively configuring the nanodevice-based junctions. 150

Figure 6-10: CMOL FPGA architecture. 151

Figure 6-11: Kogge-Stone adder and its NOR gate synthesis. 153

Figure 6-12: Schematic diagrams of hybrid circuits. The left is CMOL. The right is FPNI. 154

Figure 6-13: A hypercell consisting of four three-input NAND/AND gates, one flipflop and 26 buffers. 155

Figure 6-14: A design paradigm involving nanoelectronics on a CMOS IC. 156

Figure 6-15: Schematic of crossbar arrays. 158

Figure 6-16: Calculation of nanowire capacitance with back gate. 159

Figure 6-17: Nanowire arrays with square-like cross-section NWs. 160

Figure 6-18: Schematic of NOR gate with CMOL FPGA. 162

Figure 6-19: Equivalent circuit of transmission line for the circuit in Figure 6-18. 162

Figure 6-20: Schematic of current flow and R, C parameters for CMOL memories. 164

Figure 6-21: Equivalent circuit of Figure 6-20. 164

Figure 6-22: Simplified equivalent circuit of Figure 6-21. 165

Figure 6-23: Schematic *I-V* curve of a two-terminal nanodevice 167

Figure 6-24: Current (the arrowed line) flows from the input pin via an input nanowire through the nanodevice and output nanowire to the output pin. 169
Figure 6-25: A structural view of the mixed-signal CMOL design. 172
Figure 6-26: CMOL nanogrid interface between nanowires and CMOS. 173
Figure 6-27: CMOL nanogrid weight bits. 175
Figure 6-28: 1-D asymmetric multi-bit weight nanogrid implementation. 176
Figure 6-29: CMOS weighted multi-bit implementation, with complexity of $O(N)$. 176
Figure 6-30: (a) Programming nanodevices with multi-bits. (b) Operation of CMOL nanogrids with multi-bits. 177
Figure 6-31: Mead *k*-WTA CMOS circuit. 178
Figure 6-32: Schematic view of an analog integrate-and-fire neuron circuit. 179
Figure 6-33: A log-log plot of the input spiking rate of the digital CMOS design for an 858 mm^2 chip with three different levels of connectivity. 184
Figure 6-34: A log-log plot of the input spiking rate of the digital CMOL design for an 858mm^2 chip with three scenarios of connectivity. 185

LIST OF ABBREVIATIONS

Abbreviation	Meaning
AER	Address Event Representation
AI	Artificial Intelligence
AM	Associative Memory
AMATD	Average Memory Access Time for Data
ANN	Artificial Neural Network
ASIC	Application Specific Integrate Circuit
BBP	Bayesian Belief Propagation
BCPNN	Bayesian Confidence Propagation Neural Network
BIC	Biologically Inspired Computation
BM	Bayesian Memory
CAM	Content-Addressable Memory
CB	CrossBar
CB-D/R	CrossBar Diode/Resistor
CB-FET	CrossBar Field Effect Transistor
CMOL	hybrid CMOS / nanoelectronic circuit
CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip MultiProcessor
CP[1]	Column Processor

CP^2	Control Processor
CPI	Clocks Per Instruction
CPS	Connections Per Second
CSIM	Cortex SIMulation
EBS	Event Based Sampling
eDRAM	embedded DRAM
EVS	Enhanced Vision System
FET	Field-Effect Transistor
FPGA	Field-Programmable Gate Array
FPNI	Field-Programmable Nanowire Interconnect
HC	Hyper-Column
HC-WPNAM	Hyper-Column WPNAM
HDD	Head-Down Display
HDM	Hierarchically Distributed Memory
HTM	Hierarchical Temporal Memory
HUD	Head-Up Display
I&F	Integrate and Fire
ILP	Instruction-Level Parallelism
k-WTA	*k* Winners-Take-All
LUT	Look-Up Table
LWIR	Long Wave InfRared
MIPS	Million Instructions Per Second

MMX	MultiMedia eXtensions
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MP	Membrane Potential
MPI	Message Passing Interface
NW	NanoWire
OCR	Optical Character Recognition
P4	Pentium® 4
PC	Personal Computer
PLA	Programmable Logic Array
PN	Processing Node
PSEM	PreSynaptic Events Memory
PSP	PostSynaptic Potential
RC	Resistor-Capacitor
SIMD	Single Instruction and Multiple Data
SSE	Streaming Single-instruction multi-data Extension
STDP	Spike-Timing-Dependent synaptic Plasticity
SWIR	Short Wave InfRared
VLSI	Very-Large Scale Integration
WPNAM	Willshaw and Palm Neural Associative Memory
WTA	Winner-Take-All

1. INTRODUCTION

1.1. RESEARCH BACKGROUND AND MOTIVATIONS

In this chapter we state the research problem pursued in this dissertation. In addition, we present the motivation for solving this problem, along with background information on the research domain. We also provide a brief introduction of the concepts and technologies used in this work. However, most of the details are presented only in their corresponding chapters. For example, previous work on hardware implementations of artificial neural networks is discussed in Chapter 2, where we also discuss algorithms, virtualization, and potential hardware platforms studied in this work.

This dissertation deals with the hardware implementation of certain families of artificial neural network algorithms. These chosen algorithms are biologically-inspired, and the hardware implementations are based on CMOS (complementary metal-oxide-semiconductor) technology augmented in some cases with speculative nanogrid-based computing structures. This dissertation constitutes a comprehensive evaluation of a wide range of hardware platforms for implementing the hierarchically-distributed memories (HDM) that we believe will constitute a core component in the artificial cognitive systems of the future. Both spiking and non-spiking forms of the algorithms are studied for a range of hardware platforms, such as microprocessors, cluster of microprocessors, FPGAs (field-programmable gate arrays), digital/mixed-

signal CMOS, and digital/mixed-signal CMOL (hybrid CMOS / nanoelectronic circuits) [1, 2].

The semiconductor industry has been following Moore's law, where the number of transistors doubles every 18-24 months, for more than 30 years. It is not a real physical law, but one of faith; the fruits of a hyper-competitive $300 billion global semiconductor industry. And then there is Moore's lesser known 2nd law that states that sustaining the 1st law requires exponentially increasing investment. There is also what we refer to as Moore's 3rd law, where the 1st law results in exponentially increasing design errata (design errors).

At the time of this writing, Intel is manufacturing some of its chips with 45 nm process [3], with physical gate length of 18 nm, and is preparing to move to 32 nm process. Transistors of this size are no longer acting like ideal switches. And there are other problems outlined by Hammerstrom [4] as follows:

- Projected power density – the main limiter of processor performance today;
- Performance overkill – the highest volume segments of the market are no longer performance (clock frequency) driven;
- Density overkill – How do we use all these billion transistors [3]?
- The end of Moore's law – scaling will continue, though at a decreasing rate, asymptotically approaching 22 nm in 7-12 years – the current business model based on shrinks and compactions will change dramatically;

- Complexity – how do we create a 100% guaranteed correct design of several billion transistors?

Because of power and interconnect limitations, ever increasing processor performance will need to come more from parallel execution and not from raw clock speed improvements, and we see this already happening with the offering of multi-core chips from Intel® and AMD®. Figure 1-1 shows the evolution of Intel®'s microarchitecture. The parallelism starts with the instruction pipeline, a fundamental parallelism in organizing and executing multiple tasks. It then goes to instruction-level parallelism (ILP), which allows super-scalar, out of order execution. With additional, wider registers, the microarchitecture achieves data-level parallelism and thread-level parallelism with MMX (multimedia extensions) and SSE, SSE2, SSE3 (streaming single-instruction multi-data extensions) technology.

Due to power limitations and diminishing returns in ILP, Intel® is embracing chip-level multiprocessing (or chip multiprocessor – CMP) or "multi-core" architectures to increase computational performance [4]. However, with the exception of multi-media and perhaps games, there are still few opportunities to leverage parallelism in volume market desktop applications. Furthermore, writing parallel code remains one of the most difficult programming tasks, with only a few programmers able to do it well.

As CMOS scaling slows, researchers are now looking at molecular scale / nanoelectronics to continue Moore's law. Will nanoelectronics be economical? What will we do with it? Can we design it reliably? Can we tolerate the expected high levels

of faults and defects with our "fault intolerant" applications? Will it enable new applications? Or will it be more of the same? What should the research agenda be?

Although nanoelectronics will probably take many different forms, the most likely configuration, at least initially, will be simple nanogrids fabricated on top of traditional CMOS circuitry in a hybrid configuration. Simplistically, a nanogrid consists of:

- A roughly horizontal group of nanowires;
- A single molecule layer of some specialized material (with the electrical properties of rectification and bi-stable resistance);
- Another roughly vertical group of nanowires;
- Connections of both groups of nanowires to CMOS metal lines (the CMOL self-aligning skewed pattern developed by Likharev [1] is the most promising).

Figure 1-1: Parallelism increases with the evolution of Intel® microarchitecture.
The horizontal axis is the degree of parallelism. The vertical axis is the overall performance (adapted from [5]).

At the intersections (cross points or junctions) of vertical and horizontal nanowires, the molecules or other nanoelectronic devices ideally form a bistable, rectifying device that connects a horizontal wire to a vertical wire. Currently, researchers are fabricating wires from silicon and other materials that are ~15 nm in diameter, eventually going to < 10 nm, with lengths up to 10 μm [6, 7]. These ultra-thin wires have very high resistance, limiting their speed, but they have significantly high density.

Standard photolithography will not be possible at these very small dimensions, and electron beam lithography remains too expensive for commercial production.

However, several groups have demonstrated reasonably effective manufacturing of nanowires using a template method called nanoimprint lithography [8, 9].

Unfortunately, of the various problems facing the semiconductor industry, nanoelectronics does not solve very many. In fact, it addresses the end of Moore's law scaling, and perhaps the memory bandwidth problem. It also severely aggravates the design complexity problem. If we are currently struggling with the complexity of designing chips with several billion switches, then how will we manage designs of several trillion switches in the future?

Nanoelectronic systems will be built from modules, which, in turn, are implemented by nanodevices connected into nanocircuits. The collection of modules creates a computing structure – the specific modules and their connections constitute a "nanoarchitecture [10]."

As discussed above, one likely form that early nanoelectronics will take is nanogrids fabricated on top of CMOS, as illustrated in Figure 6-25. In such a configuration, nanoscale devices perform their limited, but very dense computing. At the same time, the CMOS part provides the remaining "enabling" functionality such as I/O and signal restoration; and some logic operations such as AND, NOT, buffers, and flip-flops. One important example of this is the CMOL (hybrid CMOS / nanoelectronic circuits) concept, developed by Likharev [1, 2] at SUNY Stony Brook.

The effective use of nanoelectronics will require solutions to more than just increased density; we need to consider total system solutions [4]. We cannot create chip-

architectures without some sense of the applications they will execute [10], since a chip-architecture is not an end in itself, but a tool to solve a problem.

Any paradigm shift in applications and architecture will have a profound impact on the tools and the entire design process. Radically new technologies create opportunities for radically new capabilities. However, generally, those capabilities require radically new computational models and structures.

One direction to take computing architecture involves a large class of problems that computers still do not solve well. These problems involve the interaction of a system with the real world, which, in part, involves the transformation and interpretation of data at the boundary between the real world and the digital world [4]. These problems occur wherever a computer is interacting with the real world. Examples include: speech recognition, computer vision, textual and image content recognition, robotic control, unmanned vehicles, sensor data collection, intelligent power management, and OCR (optical character recognition) [4].

These are difficult problems that require the computer to find complex structures and relationships in massive quantities of low precision, ambiguous, and noisy data [4]. In spite of phenomenal increases in processor speed and memory capacity, true machine intelligence still eludes us. Our inability to adequately solve these problems constitutes a significant barrier to computer usage [4]. Neither AI (artificial intelligence), ANNs (artificial neural networks), fuzzy logic, nor Bayesian networks (see Section 2.2.2) have yet led to robust solutions [4].

One potential source of inspiration, which also acts as an existence proof for solutions to these challenges, is the biological system. Though there is a wide range of "natural" computation from genes to cells, the one that is most relevant here deals with the *neural circuits*. Our research is inspired by the *computational* or "*systems*" [4, 11] *neuroscience*, and *cognitive science*.

There is a growing set of fascinating models and techniques that deserve a closer look by main-stream computer engineers and scientists. It is a gold-mine waiting for us to dig and refine the ore. The goal of this work is not to build a brain nor to contribute to neuroscience, but to use biology as a source of ideas for building new kinds of chip architectures, especially architectures that consist, in part, of nanoscale components.

Do we know how the brain works? No. However, we believe that enough is known to begin to utilize information from neuroscience as inspiration for new techniques and algorithms. In general, *neural circuits*

- are fine-grained and massively parallel, consisting of networks of sparsely connected nodes,
- are built from slow, low-power, asynchronous, low precision, unreliable components,
- demonstrate a level of design error tolerance,
- are robust in the presence of faulty and failing hardware, including hardware manufacturing defects,

- degrade gracefully,
- adapt instead of being programmed,
- constitute systems involved in the interaction of an organism / system with the real world, and
- self-organize – in fact, system design becomes more the provisioning of organizing principles (Prof. Christoph von der Malsburg, FIAS, www.organic-computing.org) than the specification of all operational aspects of the models.

In short, neural models could not be more different from most existing computational models – and yet they are a remarkable match to nanoelectronics.

The ultimate cognitive processor is the cerebral cortex (neocortex), which is a folded 2D planar sheet. When stretched out, the human neocortex is about 2/3's of a square meter. It is 3-4 millimeters thick, and consists of roughly 3×10^{10} neurons. It has a consistent six-layer structure and is remarkably uniform; not only across all different parts of human neocortex, but across almost all mammalian species. Although we are far from understanding the details of what it does and how, some of the basic computations are beginning to take shape. Nature, so it appears, has produced a general purpose computational device in the neocortex that is a fundamental component of higher level intelligence [4]. Consequently, several groups are looking to create increasingly more sophisticated models of neocortex and then to apply these models to real applications [12-20]. In Section 2.2, we will introduce the meaning and

structure of the core terminology, HDM (Hierarchically-Distributed Memory) and AM (Associative Memory) – the building block of HDM, used in this dissertation. For now, we will assume AM algorithm as the biologically-inspired cognitive building block.

The problem with using traditional computers (microprocessors) for simulating AM algorithms is that, for the most part, instruction execution is sequential with only modest amounts of parallelism. Even though the current generation of microprocessors is using multiple cores in a single chip, parallelism is still limited compared to neural models. If there are millions of nodes in a neural network algorithm for a real system, the traditional PC (personal computer) cannot emulate such a network in real time (in the sense of biology). In fact, as we understand more about how to scale these algorithms, the lack of efficient implementations is starting to be a major impediment to the use of AM in real applications. Therefore, our research goal is to study the hardware platforms with the best performance/price ratio for those models, using both existing technology and, in the long term, hybrid CMOS nanoscale structures.

There has been some research over the last 15-20 years on special-purpose hardware for implementing traditional artificial neural network algorithms. However, the rapid advances in microprocessor performance, coupled with the poor scalability of these algorithms [4], have resulted in little research being done on relevant hardware development.

Now, however, we are looking at very large, scalable, biologically-inspired algorithms that are extremely compute intensive. They are also massively parallel and can absorb just about any parallelism we can provide at the hardware level.

Ideally, the range of hardware that can be used to implement AMs could go from microprocessors to FPGAs (field-programmable gate arrays), full-custom silicon, and nanoelectronic architectures. Figure 2-10 shows a qualitative description of the hardware spectrum for artificial neural network implementations. As we move from general to specialized computing structures, we can get greater performance/price from the implementations, but such customization also tends to reduce general flexibility.

However, the efficiency of parallel architectures is dependent on Amdahl's law, which is represented as "$Total\ Speedup = 1 / [(1 - Fraction) + Fraction / Speedup_{Fraction}]$" [21]. Thus, the total speedup of a system with some parallel operations tends to be limited by the sequential part of the algorithm, i.e. the $(1 - Fraction)$ in the previous equation. We could think of a simple example: for a given system, if *Fraction* (parallel part) = 0.5, $Speedup_{Fraction} = 1000$, the system's *Total Speedup* will be only about 2×. However, if we increase the *Fraction* (parallel part) to 0.95, the system's *Total Speedup* will be almost 200×, which is much better than the previous case. This shows that the more computational time for the sequential part in the algorithm, the less speedup the enhanced hardware could possibly achieve. As a result, by being difficult to parallelize a large portion of an application, Amdahl's law favors the flexibility in designing a hardware computing platform. We should also be aware that trading off

flexibility with parallelism is a critical aspect of designing hardware platforms for the AM application presented here.

For different applications and constraints, computer engineers use various kinds of performance/price ratios to measure the effectiveness of different hardware implementations. Performance is generally a measure of the speed of a system. So for neural network models, performance may be measured by the number of connections computed per second. Price has traditionally been measured by silicon area, though power consumption is becoming more important. Comparing the performance/price for different hardware implementations will give us an intuitive view of how to choose the optimal hardware platform, and the optimal hardware architecture.

In looking at hardware implementations of artificial neural networks, a wide range of neuromorphic (coined by Carver Mead [22, 23] in the 1980s to describe the analog VLSI systems that mimic the neural circuits) VLSI (very-large scale integration) has been explored. Since most computational models were massively parallel, designers sought to explore this characteristic to achieve cost-effective speedup. Additionally, because many network algorithms were very compute intensive, specialized neuro-hardware was even thought to be necessary for the field to progress [4].

Neuromorphic hardware took the form of special computers based on general-purpose microprocessors, such as BSP400 [24] and COKOS [25], or systems that used specialized silicon. However, Amdahl's law coupled with the rapid increase in speed in mainstream microprocessor technology compromised most of the

performance/price value these systems had. The same can be said for many of the digital neurochips such as CNAPS [26] and SYNAPSE-1 [27]. Palm *et al.* [17, 28] also developed BACCHUS chips for their neural associative memory models. We will introduce those prior works in Section 2.4.

Carver Mead [22] developed the idea of using analog circuits to implement a number of neural inspired algorithms. One of Mead's innovations was to operate MOS FETs in the sub-threshold region, creating slow, but ultra-low power designs. This technology is now often referred to as aVLSI (analog VLSI).

Analog based neurochips can be both extremely fast due to massive parallelism, yet very inexpensive and consume very little power. However, they are essentially algorithms wired into silicon and are inflexible. As shown later in this dissertation, under certain conditions and for a certain family of algorithms, they do not provide an improved performance/price. They are intrinsically good for front end applications or low-precision cognitive models [15], but lack the flexibility and precision of higher level cognitive models.

Because of the storage requirements of BIC (Biologically-Inspired Computational) models, and because nanogrid technologies promise very high density on-chip storage, we believe that the CMOL technology developed by Likharev [1] is a promising paradigm to include into our search for cost-effective AM implementations. There are a number of other nanodevice crossbar architectures [29-35]. However, the electronics-based CMOL technology appears to be more practical to manufacture with

a reasonable cost [2, 36, 37] by the use of nanoimprint lithography [8]. For these reasons, we include CMOL as an implementation option for our AM models. CMOL is probably the densest charge-based computing technology that we will see any time over the next decade or so [38].

When we use analog circuits to implement AM models at the nanoscale, under certain assumptions power density can become a significant limiting factor. One promising approach to reducing power requirements is the use of spiking models. Although they are not well developed, we also investigate what is possible if we assume simple spiking neuron models [39]. From the algorithmic perspective, in spiking models, time is the additional dimension of information. This temporal information is not present in the WPNAM family. Spiking neural network models are intrinsically closer to the neocortex than non-spiking neural network models [39]. Although they are much more complex than WPNAM model, they lend themselves to some interesting optimizations.

1.2. OBJECTIVES AND CONTRIBUTIONS

1.2.1. Objectives

The goal of this dissertation is to compare different hardware solutions for a certain class of artificial neural network algorithms, specifically associative networks (or associative memories) as might be used to implement a node in an HDM, to determine the best performance/price solution among several reasonable alternatives. Although dynamic, incremental learning in the neural associative memory is important, it is

complex, and, hence, was deemed to be beyond the scope of this dissertation. Consequently we assume that the connection weights are pre-determined and downloaded into the network prior to network operation. Consequently, the hardware only deals with the network's retrieval of the stored patterns.

In this dissertation, there are two neural associative memory models implemented by the specialized hardware platforms. One is a non-spiking (often called a "rate" code), binary, neural associative memory model, derived from WPNAM. This dissertation focuses on a single WPNAM network, which would be a node in a larger HDM model and does not consider multiple nodes and inter-node communication. The other model is a spiking neural associative memory model. The spiking neuron model is based on the simple spiking neuron model proposed by Gerstner [39] and a variation of Gerstner's simple spiking neuron model, the leaky integrate and fire (I&F) neuron model (details in Chapter 2).

1.2.2. CONTRIBUTIONS

Here we show the contributions of the work undertaken in this dissertation, including all the major problems addressed by this work.

- Algorithm decomposition

To implement the target algorithms, we first need to understand their properties. We started by decomposing the AM models into a number of simple computations that have reasonable functional coverage. For example, the simple WPNAM model includes a matrix-vector inner-product and *k*-WTA operations. The simple spiking

neuron model includes the post-synaptic potential, accumulation, and threshold operations. The leaky I&F (integrate and fire) model includes accumulation and threshold operations. From these models, we extract the basic computations: multiplication, summation (accumulation), and threshold. The hardware architectures should handle these computations efficiently and at some reasonable "optimal" performance/price ratio. The HDM consists of large numbers of relatively independent modules and decomposes fairly nicely. These modules are typically modeled as "Bayesian Memories" [40]. In this dissertation, we studied how to physically implement associative memories as an approximation to a Bayesian Memory.

- Explore possible hardware platforms

For most people, using a stand-alone workstation to simulate AM algorithms seems very practical. However, in order to run some applications in real time or on small portable embedded platforms, hardware accelerators will still be required. In addition, we are interested in the best performance/price solution. In this dissertation, we explore several different hardware configurations for implementing associative networks, including a desktop PC (commercial microprocessor), PC cluster, FPGA, custom CMOS, and nanodevice hybrid circuits (CMOL).

There is also another factor that motivates the use of custom hardware: the need to handle very large, scalable networks. More than anything else, the inability to scale has been the biggest problem with traditional AI (artificial intelligence) and ANN algorithms [4], which is one reason that specialized hardware was ultimately not cost

effective in the ANN boom of the late 80s and early 90s. BMs are scalable and consequently will demand scalable hardware.

As a part of this work we have developed an architectural analysis methodology, which includes an approach to analyzing the performance/price ratio of different hardware implementations, and the comparisons for those performance/price ratios to determine the best configuration from a range of hardware platforms. This exploration of hardware platforms for AM algorithms is new and unique. It not only establishes the equations to analyze the performance/price ratio with different hardware platforms, but it also creates a set of hardware building blocks that can be used for a wide range of related data parallel algorithms.

- FPGA architectural design, analysis, and implementation

It is generally accepted that the finer grain the parallelism, and the lower the precision, the more effective an FPGA implementation will be [41, 42]. Consequently, an FPGA design should exhibit a better performance/price ratio compared to a PC. In the work described here we developed a method for the design of FPGA functional blocks for the non-spiking algorithm. We then compared the performance/price ratio with the previous results of PC and PC clusters, and showed the validity of the parallelism assumption. We also present an example FPGA design from a low-end FPGA development board to demonstrate the feasibility of using an FPGA as a hardware platform for our non-spiking AM model. We demonstrated this system at NIPS'03 (*Neural Information Processing Systems 2003*) and published a paper at IJCNN'04

(*International Joint Conference on Neural Networks 2004*) [43]. The FPGA implementation for the AM algorithm is an important step in this work. FPGAs tend to be used to accelerate the applications that the PC cannot execute in real time, or where the PC is too expensive of a platform. Our results proved the validity of this assumption for the algorithms studied here.

- Using CMOL to implement AM algorithms

Although the concept of CMOL [1] is relatively new, HP [44, 45], for example, has already been working to commercialize nanogrid structures. They have proposed a possible application FPNI (field-programmable nanowire interconnect) based on CMOL. And while Türel and Likharev [46] have proposed neuromorphic CMOL structures, we are the first to study the use of CMOL for implementing HDM-like cortical models [47-49]. Our implementations not only include non-spiking, but also the spiking AM algorithms. The mixed-signal CMOL structure implements the multiplication and accumulation directly in the nanowire, which saves silicon area and power. Consequently, mapping the multiplication-summation, i.e., inner-product computations, onto CMOL is an excellent candidate technology for implementing neural associative memories. We will show detailed equations to estimate CMOL's real estate, time delay (according to Elmore's time delays for resistor-capacitor network [50]), and power.

- Hardware parallelism

In order to explore hardware parallelism, we studied both mixed-signal CMOS and mixed-signal CMOL as potential hardware technologies for implementing non-spiking and spiking AM algorithms. As illustrated in Figure 2-10, the finest-grained parallelism literally has a PN (processing node) for each synapse. Only mixed-signal CMOL can achieve such hardware parallelism. We will show (in Chapter 6) that this ultimate parallelism gives us the best performance/price ratio compared with other hardware candidates for both spiking and non-spiking AM algorithms. This also demonstrates the significant parallelism of neural associative memory models and the intrinsic properties of parallelism of mixed-signal CMOL.

- Hardware virtualization

Although this ultimate parallelism gives us the best performance, it may not necessarily be the most efficient. We present a theoretical analysis of hardware virtualization in the spiking AM algorithm implementations, especially for digital CMOS. Because of the sparse input activation and sparse connectivity, the most parallel (finest grained) structure of digital CMOS is not necessarily the most efficient implementation. Instead, we can increase efficiency by multiplexing many neurons over a single physical processing node. Though it is possible to multiplex analog and digital computations, digital circuits are easier to implement and can actually be less expensive. They also tend to be more fault-tolerant and they put fewer requirements on the fabrication process. Consequently sparse activation tends to significantly increase efficiency in digital implementations of these algorithms. Hardware

virtualization then provides guidance to other similar designs that have the properties of sparse input events or sparse connectivity.

1.2.2.1. Contribution to knowledge

However, the contributions listed above mainly represent the detailed work we have done, such as using FPGA to implement associative memory algorithm for the EVS application (see Section 4.2). However, the results of our methodology and work could provide insights to people from different fields. For the neuromorphic engineering, our results provide "insights into brain's high-level computational principles" [51], which pave the road to novel scalable cognitive systems; and also provide guidance to steer future research trends in computational neuroscience. The study of associative memory hardware architectures will lead to novel non-von Neumann computing architectures, which could change the future state of computer industry. Moreover, it provides the semiconductor industry one of the potential architectural solutions for the scaling challenges faced by the design of nanoelectronic circuits in the future [52].

1.3. THESIS ORGANIZATION

The purpose of the work presented here is to analyze the performance/price characteristics of a range of hardware platforms for implementing cortical-like models. The general organization of this thesis is to first present the models, and then to describe the selected hardware options. Finally the results of this analysis are presented. Because of its increased complexity, learning, i.e., dynamic, real-time modification of the network parameters is not addressed here. However, for non-

learning models we believe that this work constitutes one of the more thorough analysis to date of the various hardware implementation options for a range of cortical-like algorithms, including the addition of nanogrid structures that may be available in the next 7-12 years.

In Chapter 2 – "AM Algorithms and Hardware Architectures", we introduce the hierarchically-distributed memory models, the Bayesian memory, and associative memory models that approximate Bayesian memory. There are both non-spiking and spiking variations of these associative memory models. We explain the most important theory, virtualization, for our work in this chapter. We also introduce possible hardware architectures for the neural associative memories, and the methodology for conducting the performance/price comparisons among selected hardware architectures.

In Chapter 3 – "Design with General-Purpose Architectures", we investigate the implementation method of using a desktop computer for the non-spiking neural associative memories. This chapter describes a baseline, the PC, for the hardware implementation of neural associative memories. This baseline is used to compare with different hardware architectures in later chapters. We also investigate the implementation with PC clusters, and compare the performance/price ratios of those two solutions (PC and PC cluster).

In Chapter 4 – "Design with Reconfigurable Architectures", we analyze the performance/price ratio of using FPGAs to implement the non-spiking neural

associative memory algorithm. We give an example of a simple FPGA design with an available FPGA development board to implement the AM algorithm. We also compare the performance/price ratios of the PC and FPGA implementations.

In Chapter 5 – "Design with Special-Purpose Architectures (CMOS)", we introduce a detailed design method and analyze the performance/price ratios with the digital/analog CMOS technology for the non-spiking and spiking AM algorithms.

In Chapter 6 – "Design with CMOL", we introduce previous work on nanodevice and nanoarchitecture proposed or developed by other people. We give the equations for the performance/price modeling of CMOL. We compare the performance/price benchmarks for non-spiking algorithm implementations with PC, CMOS, and CMOL, and spiking algorithm implementations with CMOS and CMOL.

Finally, in Chapter 7 – "Summary and Future Work", we summarize our work and point out the important conclusions. We also point out some potential future work.

2. AM ALGORITHMS AND HARDWARE ARCHITECTURES

2.1. INTRODUCTION

In traditional computing, data are stored in memory using an address-based scheme, which requires that the program knows the location of the desired data. For some applications, such as the storage of numerical arrays, creating the necessary address is not particularly onerous. However, for some applications, such as the EVS (Enhanced Vision System) association engine (in Section 4.2), the address is unknown, or only the part of the data is known and the remainder is sought. In these circumstances, content addressing [42] is used.

The most common kind of content addressing is *exact match*, where part of the data is known and is matched exactly to any location with these contents. Exact match addressing is commonly used in traditional computing hardware and software, and a number of implementation techniques, such as hashing, have been developed. However, with exact match association, any search using slightly corrupted data will return the wrong data. For this reason, some applications use a more complex form of association, called *best match*, where the memory finds the content that has the "closest" match according to some metric. There are many metrics that can be used in best-match search. For the networks studied here, we use a simple Hamming distance

for binary vectors, and the Euclidean distance[1] for non-binary (fixed or floating point) vectors. Presenting a noisy vector as input to a best match associative memory returns the location that has the smallest metric difference. Although best-match association is quite useful, there are no real computational short-cuts to checking the distance to every item in memory. For this reason, best match association has rarely been used. At the same time, we are aware of some techniques for VQ (Vector Quantization) [53], such as *k*d-tree (*k*-dimensional tree) [54] and LSH (Local Sensitive Hash) [55], which are approximations and can work well in some cases and not so well in others.

There are biologically inspired associative memory models, such as the Willshaw [18-20] and Palm [17, 56] neural associative memory models (WPNAM), and the Hopfield [57] neural associative model, which find the best-matched content in a computationally efficient manner. The WPNAM model forms the basis of the work reported here.

2.2. ALGORITHM

2.2.1. HDM

Neocortex appears to implement efficient Bayesian-like inference over sparsely distributed representations of data (only a few bits in the data are ones, the others are zeros), using columns to represent invariant lower level "features" in the data [58]. We believe that distributed representations are a kind of factorization, allowing massively

[1] For the higher levels in an HDM system, we would ideally want a metric that had meaning in the application space where the HDM is being used. However, such measures are beyond the scope of the research proposed here and so simple vector distance metrics are used.

parallel inference. Distributed representations also diffuse information, topologically localizing it to the area where it is needed, and reducing global connectivity requirements. Those columns are connected with each other to form hypercolumns. We call this hierarchical structure of correlated information, or HDM. As mentioned earlier, a number of people in Computational (or "Systems") Neuroscience view HDM-like models as being a promising model for some aspects of mammalian cerebral cortex.

Based on the current understanding of the biology of the mammalian neocortex, Braitenberg and Schüz [59] proposed a statistical approach to cortical neuroanatomy. One finding was that the number of neurons in the mouse neocortex is at least 10× greater than the number of input elements. In human neocortex, this relationship between neuron counts and input fiber counts can exceed 1000×.

Johansson and Lansner [60] considered the mammalian neocortex as a kind of associative memory. When we look at the vast number of homogeneous neurons and synapses in neocortex, the electrical charge accumulation and ion concentration variations determine the state of the neurons and synapses, with many learning rules following the basic Hebb model [61].

Johansson and Lansner [60] present a nice summary of the neuron and synapse counts in some mammals, as illustrated in Table 2-1. It shows a consistent synapses/neuron proportion (around 7000) prevailing among those of several different mammalian species. This consistency gives us a hint that there may be a fundamental

computational unit or module inside mammalian neocortex. This also partly proves Mountcastle's proposal of columnar structure introduced in the following paragraph.

Table 2-1: Neocortex neuron counts in several mammals.
(Adapted from [60].)

	Human	Macaque	Cat	Rat	Mouse
Cortex Area (mm^2)	2.4×10^5	2.5×10^4	8.3×10^3	6.0×10^2	2.5×10^2
Neurons	2.0×10^{10}	3.0×10^9	6.0×10^8	5.0×10^7	2.0×10^7
Neurons (mm^{-2})	8.3×10^4	1.2×10^5	7.2×10^4	8.4×10^4	8.0×10^4
Synapses	1.5×10^{14}	2.2×10^{13}	4.5×10^{12}	4.0×10^{11}	1.6×10^{11}
Synapses/Neuron	7500	7300	7500	7900	8000

Mountcastle [62] proposed a columnar organization of the neocortex. In the book "Perceptual neuroscience – the cerebral cortex" [62], based on the physiological studies of some heterotypical cortical areas, such as somatic sensory cortex, visual cortex, auditory cortex, motor cortex, and the physiological studies of some other homotypical cortical areas, he concluded that the basic unit of the neocortex is a minicolumn. The minicolumn is a vertically organized group of about 80-100 neurons that traverses the thickness of the gray matter (~3 mm) and is about 50 μm in diameter. The neocortex also has a distinct six layer organization. Neurons in a minicolumn tend to communicate vertically with other neurons on different layers in the same minicolumn.

Mountcastle then proposed that minicolumns are grouped into larger units, variously referred to as columns, macrocolumns, or hypercolumns, as shown in Figure 2-1. The existence of this larger structure is more controversial in the neuroscience community. However, Braitenberg and Schüz [59] showed that there are spatially close groups of neurons that are tightly connected with each other, but sparsely, and more randomly

connected to other groups. For convenience we loosely use the term "column" for these tightly connected groups, but do not necessarily imply a true column in the Mountcastle sense.

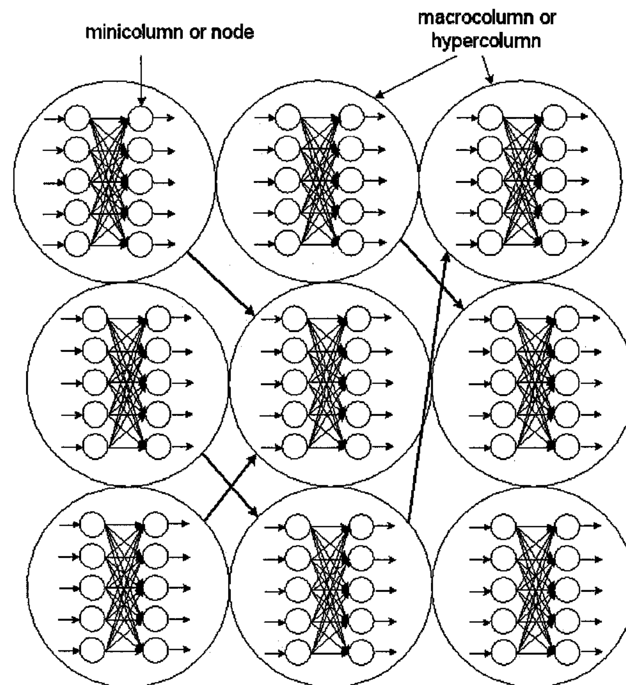


Figure 2-1: Hierarchically distributed memory structure in the Mountcastle sense.
Each big circle represents a hypercolumn. Hypercolumns are connected loosely. Located inside each hypercolumn are several the minicolumns, which have tightly connected neurons.

In the early days of neural networks, simple associative memory models were considered a first step towards modeling the neocortex. It is now clear that the early models fell far short [63], among other things they did not scale. However, they still are useful as models for the smaller cortical "modules", such as the cortical column. A number of advanced models [64-66] have been developed to create cortical-like

structures by loosely connecting such modules into larger arrays. Several of these models assume columns are implemented as associative networks or Bayesian networks (in Section 2.2.2). Since the majority of connections and computation are within a column, which will be the focus of this dissertation, that is the hardware implementation of a single associative memory column. We also call the hardware that emulates the single associative memory column – Column Processor (CP), details in Section 2.4.1.

Once we have an efficient implementation of a column, the next step is to connect the cortical columns together into a large array, which creates the HDM, as illustrated in Figure 2-1. In many of these models, the columns are configured into a two-dimensional grid. Connectivity is typically nearest neighbor with a few random, longer-range, point-to-point connections. The entire structure creates a higher-order, scalable, large capacity Association Memory (AM) or Bayesian Memory (BM), which are explained in Section 2.2.2.

Analysis of such structures is more complex and is not addressed here, but there are several successful approaches, including the work of Lansner [67], Fulvi-Mari [68], Granger [69], Hecht-Nielsen [64], and Anderson [65], George and Hawkins's HTM [70], Hecht-Nielsen's Cortronics [14, 64, 71]. Zhu [72] demonstrated that a group of smaller-size networks that are sparsely interconnected can be less costly than much larger networks, without sacrificing basic network performance. Also, several of these researchers have proposed that at the cortical level, the columns do a kind of Bayesian Belief Propagation (BBP). It is our belief that a more complex associative memory,

called a BAM (Bidirectional Associative Memory) can approximate BBP [14, 73]. However, this functionality is beyond the scope of this dissertation and is not addressed here. The interested reader is referred to [40, 74, 75].

2.2.2. Bayesian Memory and Associative Memory

To demonstrate how an HDM model works, a communication channel is a useful analogy. In speech recognition, e.g., when we say a word, we have a specific word in mind. We then encode that word into a form (speech) that is transmitted over a noisy channel (sound waves in air) to a computer, which then decodes the received wave back into the original word. The channel, as shown in Figure 2-2, is our mouth, tongue, vocal chords, the physical environment, as well as the computer's microphone and signal processing. This channel is noisy because there is significant variation in the way the word is encoded – people say the same things very differently from one instant to the next, and there are speaker variations and ambient noise in the environment. The decoding process uses digital signal processing algorithms at the front end and complex "Intelligent Signal Processing (ISP)" algorithms that model grammatical rules and higher order dependencies at the back end (more about this in Section 2.4.2).

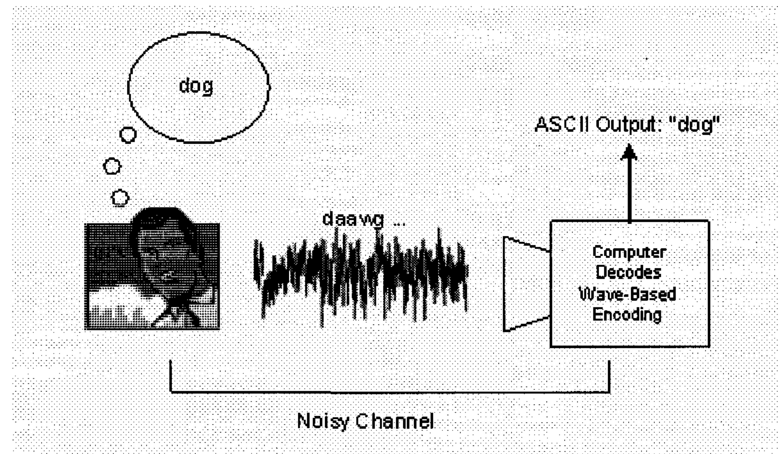


Figure 2-2: The "decoder" model of association.

By casting pattern recognition into a communication channel model, we can then apply both information theory, and Bayesian decision theory [76] to the decoding process. In simple pattern recognition, we collect statistics on the occurrence of certain features. During decoding, we then use those statistics to answer the question: "if we received a certain input, what was the most likely word or phrase that was sent?"

An HDM, or specifically speaking, the building block of HDM, the BM[2], can be modeled as the decoder in a simple communication channel, as illustrated in Figure 2-3. An input generates a message y that is encoded by the transmitter as x. The message is then sent over a noisy channel and x' is received. The decoder decodes x' into y', which is the most likely y that has been sent.

[2] We refer to BM as the building block, or module / node of HDM.

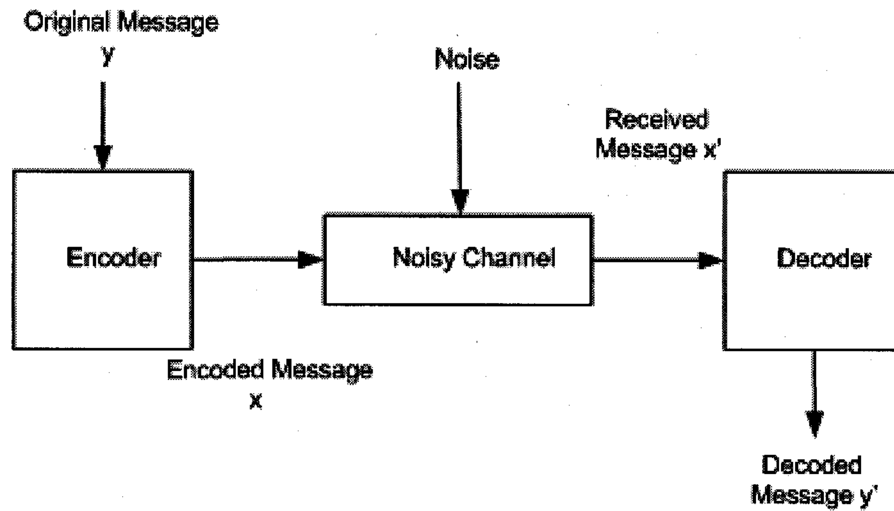


Figure 2-3: Communication channel.

Messages are generated with probability $p(y)$, these are encoded into a transmitted message x (a bit stream) over a noisy channel, a received message (another bit stream, the transmitted message with errors) x' is input into the decoder. The decoder has, via many examples learned the probabilities $p(y)$ and $p(x'|y)$, as well as the probabilities $p(x'|x)$ and $p(x)$. In this case, the network is given the received bit vector, and it outputs the most likely message to have been sent. It then uses these data to determine the most likely y given that it received x' as shown below:

$$p(y|x') = \frac{p(x'|y)p(y)}{p(x')} = \frac{p(x'|y)p(y)}{\sum_x p(x'|x)p(x)} \tag{2.1}$$

The basic node of an HDM implements what we call a Bayesian Memory (BM). So BM is a single node or module of HDM. We call our network a Bayesian Memory,

since it basically stores information and then, in response to some input, retrieves the "most likely" input in a Bayesian sense, based on inference on probabilities estimated from statistics collected during the write (training) process (2.1). A large number of BM modules are connected into a layered hierarchy (HDM), where each module connects to a subset of the modules in the preceding layer. Based on Pearl's Bayesian networks and Bayesian Belief Propagation (BBP) [77, 78], Hawkins and George proposed HTM [12]. As illustrated in Figure 2-4, the HTM's output is a set of probabilities of learned causes. The input to the HTM is a set of sensory data. The distribution of the causes is called belief. Actually, HTM learns a hierarchy of causes as illustrated in Figure 2-4. Each block or node in Figure 2-4 learns causes and forms beliefs according to Section 4 in [12]. In addition, Zaveri [40] discussed the implementation of a more abstract Bayesian Memory functionality based on the original Bayesian Belief Propagation [78], however, did not incorporate the temporal information, which is required in HTM. Our work, however, uses Associative Memory (AM) to approximate the BM, the details of Hawkins' HTM and Zaveri's BM CMOS and nanoelectronics implementation will not be explained in this dissertation. Interested readers are referred to [40, 70, 79].

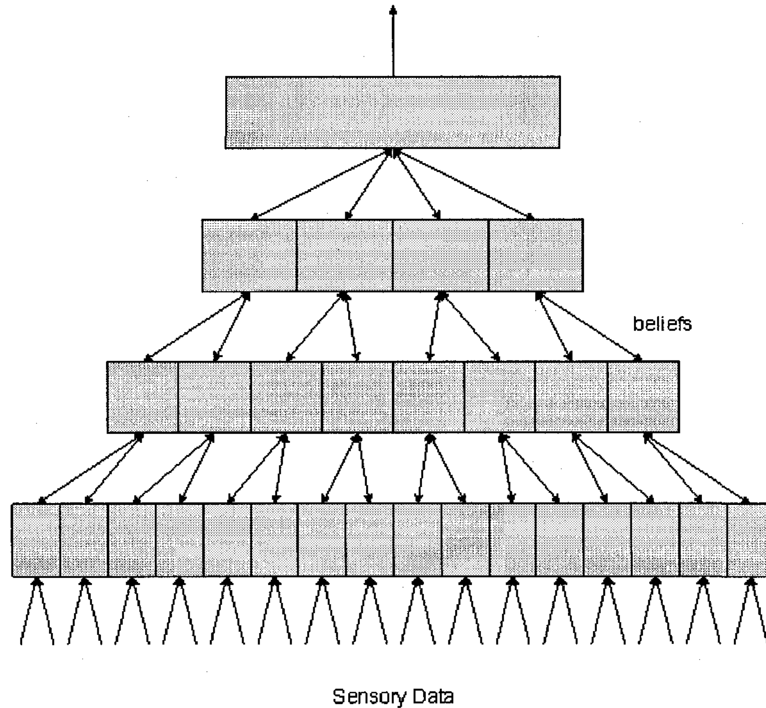


Figure 2-4: Hawkins and George's HTM structure.
The sensory data are fed into the lowest layer of nodes. The arrows in between two layers are called belief (adapted from [12]).

Different from Hawkins and George's HTM or Zaveri's simplified BBP as the building block of BM, Zhu [80] showed that under certain circumstances AM models approximate Bayesian inference or BM. Zhu [80] gave a simplified version of HDM with two layers of BMs. As illustrated in Figure 2-5, each BM is a WPNAM (Willshaw and Palm Neural Associative Memory) network (see Section 2.2.2.1). The arrows in Figure 2-5 are input and output vector-pairs. This hierarchy of BMs also resembles a hierarchy of Bidirectional Associative Memories (BAMs) [73], because it has the input and output vectors (e.g., V_{Lin1}, V_{Hout1}, V_{Hin1}, V_{Lout1}) from both directions. Zhu used one WPNAM model to map BM in one direction, and used another

WPNAM model to map the other direction's BM. Zhu [72] demonstrated that with this three-BM network, the noisy input vectors (i.e., V_{Lin1} and V_{Lin2}) were recovered to the output vectors (i.e., V_{Lout1} and V_{Lout2}) with the best performance according to Zhu's metrics. Zhu [72] also demonstrated that the use of such hierarchical structures (HDM) can lead to large and scalable systems based on their preliminary work.

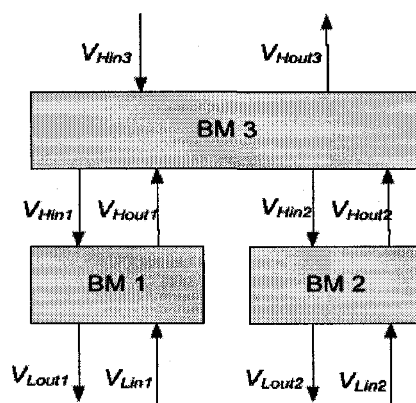


Figure 2-5: A three-BM network (adapted from [72]).

Our primary objective, therefore, is to develop an associative memory that performs such probabilistic inference in the BM in real-time over very large data sets using sophisticated metrics. We also believe that a large capacity version of a BM, implemented in an inexpensive chip, has significant commercial value.

For performing such Bayesian inference, there is always the brute force approach: a simple processor per record (or per a small number of records), all computing the match in parallel, then with a competitive "run-off" to see which processor has the best score. Though inefficient, this implementation of best-match guarantees the best

results, and can easily be used to generate optimal performance criteria. Unfortunately, it is too compute intensive for most real applications.

In Figure 2-3, message x_i is transmitted, message x' is received. For simplicity, assume that all vectors are N bits in length, so the noisy channel only causes substitution errors. The Hamming Distance (HD) between two vectors, x_i and x_j, is $HD(x_i, x_j)$. Assume that the noisy channel is binary symmetric with the probability of a single bit error being ε, and the probability that a bit is transmitted intact is $(1 - \varepsilon)$. The error probabilities are independent and identically distributed and $\varepsilon < 0.5$. Under these circumstances it can be shown that the Palm memory approximates a Bayesian inference [72]. That is, the Palm associative memory will, under certain conditions, recall the most likely original message, x_i, from a noisy received message x' [72]. Since Bayesian inference is compute intensive (NP-Hard) [72], the distributed representations used by associative memory and the modular organization of the BM have the potential to perform inference quickly and efficiently.

In 1969, Willshaw [18] published a paper about the theory of neural associative memory, and Palm [56] gave a detailed explanation of associative memories in 1980. They both proposed associative memory models based on a binary weight matrix, Hebbian learning, and threshold activation functions. Their model is a particularly efficient implementation of BM. We call their model WPNAM (Willshaw and Palm Neural Associative Memory), and we will use WPNAM extensively in this work. In this dissertation, WPNAM and AM are interchangeable. When we mention AM in this

work, we mean WPNAM-based AM. Now, we will start introducing the WPNAM-based HDM model in the following sections.

2.2.2.1.Non-spiking AM model

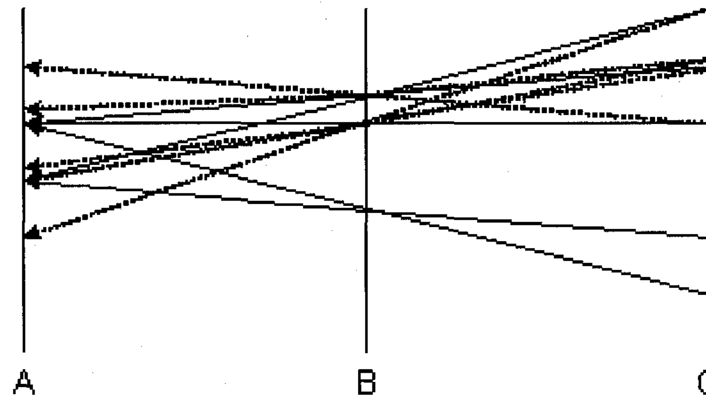


Figure 2-6: With light emitted from C through B to A, the original spots on A are generally brighter.
(Adapted from [18].)

To understand the associative memory, a good place to start is Willshaw's work back in the 1960s. Willshaw [18] used a physical experiment to investigate the analogy of associative memory. In this experiment, a light was emitted from a source on board A. The light passed through holes in board B and struck board C. The experiment was then reversed, by emitting light from board C, through holes in board B and striking board A. During the reversed experiment, Willshaw found that the brightest spots on A were at the positions where the light was being emitted in the forward experiment. The reversed process is illustrated in Figure 2-6. This experiment gives us a hint: the light spots on C are the correlation results of holes on B and A. When we are given patterns of C and B, we can find the original pattern of A. That is to say, the C-pattern

is the training result of A-pattern and B-pattern. When given the B-pattern and C-pattern, we can retrieve A-pattern. These training and retrieval processes are somewhat similar to biological associative processing.

Based on the Willshaw NAM model, Palm proposed different types of neural associative memories using a binary weight matrix, Hebbian learning, and a transfer function with a global threshold [17, 56]. Figure 2-7 shows the basic network structure of the WPNAM model.

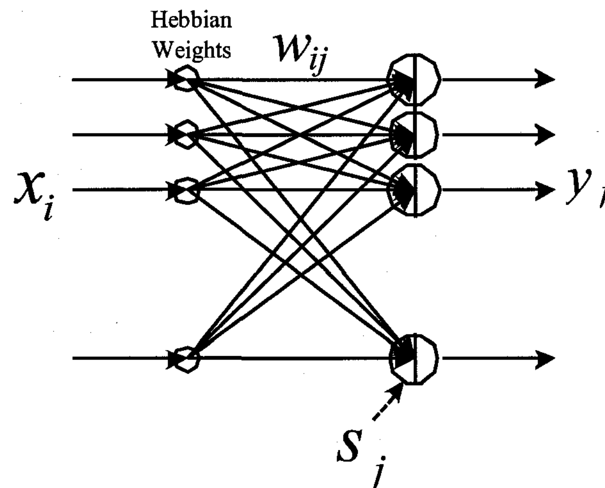


Figure 2-7: The architecture of the Willshaw and Palm NAM model.

For the training stage of the network, Palm uses a "clipped" Hebbian learning rule:

$$w_{ij} = \bigvee_{\mu=1}^{M}(x_i^{\mu} \wedge y_j^{\mu}) \text{ or } w_{ij} = \sum_{\mu=1}^{M}(x_i^{\mu} \wedge y_j^{\mu}) \quad (2.2)$$

where w_{ij} is the binary value 0 or 1 (or multi-bit value) of the weight between the i-th input neuron and the j-th output neuron; x_i^μ is the binary value of the i-th input neuron; y_j^μ is the binary value of the j-th output neuron; μ is the μ-th training pattern index; M is the total number of training patterns. The training patterns (vectors x_i^μ and y_j^μ) we used in Chapter 3 were generated with constant number of active neurons (nodes) uniformly distributed among all input neurons.

For the retrieval stage of the network, when an input pattern x propagates through the network, the output neuron's somatic potential is given by:

$$s_j = \sum_i w_{ij} x_i \tag{2.3}$$

where s_j is the j-th output neuron's somatic potential; x_i is the value of the i-th input neuron's output value, which is a noisy version of the training vector (how to generate such test vector was discussed in [72]); w_{ij} is the weight connecting the i-th input neuron and the j-th output neuron. This somatic potential then goes through a transfer function with a global threshold θ, which activates k different output neurons (i.e., activated neurons are set to one, others to zero). The value of the j-th output neuron is

$$f(S_j) = \begin{cases} 1 & (s_j \geq \theta) \\ 0 & (s_j < \theta) \end{cases} \tag{2.4}$$

The computations in (2.3) are the Boolean AND, and integer additions of the weight matrix and the input vector. Equation (2.3) is a matrix-vector inner-product operation.

Equation (2.4) is the activation function, a non-linear step function (a threshold based Heaviside step function), which makes the output layer have only k active nodes. The operation specified by Equation (2.4) is generally referred to as k-WTA (k Winners-Take-All). These two operations, matrix-vector inner-product and k-WTA, are the primary computations for the retrieval phase of the WPNAM model. It is not clear from biology whether a column simulation needs only be WTA or whether k-WTA is required ($k > 1$). Obviously the WTA is simpler, but it also reduces capacity [17]. We used the complex k-WTA for analysis since it is more generic. Those two computations (matrix-vector inner-product and k-WTA) also represent a group of operations in AM algorithms that use binary data representations, a clipped Hebbian learning process, and excitatory-inhibitory output activation functions. Any hardware implementations of the WPNAM-based HDM model must perform these two main computations.

Hopfield proposes an associative memory model with a recurrent network structure [57], where he uses the analogy of an associative memory network with the collective properties common in the physics of "spin-glass" systems. This network operates to minimize its "energy", where the locally stable states of the network are essentially energy minima ("attractors"). Ideally, these energy minima correspond to the "training" vectors, i.e., the data entries into the associative memory. The main drawback of the Hopfield NAM model is that it is not particularly robust. It also requires an asynchronous update of network nodes, when emulating a Hopfield network on a synchronous hardware platform, not all the network nodes update at one

time. This causes the Hopfield NAM updates to be slower than those of the WPNAM model, which can be synchronous or asynchronous. The WPNAM model in its simplest form is a simple linear inner-product with a non-linear competitive output. It also works very well with binary input and weight vectors, leading to significant hardware savings. Generally, the network updates all the nodes for one retrieval cycle, and is more robust than Hopfield NAM model [17, 18, 56, 57]. In addition, compared with the WPNAM model, the Hopfield associative memory has less capacity [17].

As an example of how associative memories might be used in real applications, we present an association driven Enhanced Visual System (EVS) in Chapter 4 and in [43]. Association networks or AMs have much promise as a component in building systems that perform Intelligent Signal Processing (ISP) [4, 81]. However, there are a number of problems that need to be solved before distributed representation, best match associative memories can find widespread usage. Perhaps the most serious concern is the scaling to very large networks. Even though the vectors used by the WPNAM are sparse (only very few bits in the vector are one, the others are all zero), the weight matrix becomes decidedly non-sparse as we add training vectors to the memory. Palm [17, 82] has shown that maximum capacity occurs at 50% connectivity, where there are an equal number of ones and zeros in the weight matrix. At the systems level this is not biological and does not scale. For example, the law of large numbers tells us that as network size increases each neuron gets almost the exact same input. This level of connectivity causes significant implementation problems as we scale to relatively large networks.

If we randomly delete connections from a Palm network, performance degrades gradually to a point where the network suddenly fails completely. However, there is no existing algorithm that achieves the neocortex's association performance (the ability to retrieve information from the associative memory) with only about 0.0001% connectivity (shown in Table 2-1). Clearly we are doing something wrong.

In the early days of neural modeling, scaled up versions of simple auto-associative networks (same input and output training vectors) were proposed as models of neocortex. Such networks have been studied extensively [17, 57, 83, 84]. However, it quickly became clear that these networks do not scale well enough to be a model of the neocortex [63].

Many in the research community [16, 64, 65] are beginning to study large associative networks that are organized more like cortical columns and are scalable. Column in those networks is defined as a single associative network, or a WPNAM network. These columns are then connected together into an array. The layout is generally a two-dimensional grid, and connectivity is typically nearest neighbor with a few random, longer-range, point-to-point, connections. The entire structure creates a higher order association memory, which can store hierarchical and distributed data structures with a large capacity, and has the ability to do inference over those structures.

In this dissertation we will refer to this type of model as an HDM. However, the exploration of HDM models is beyond the scope of this dissertation. Consequently, for

the work discussed here, we concentrate on the physical implementation of a single HDM node, or an individual column, which can be modeled as an auto-associative memory (WPNAM) as presented above. Also, since these columns have the densest connectivity and constitute the majority of the computational load, they consume most of the silicon real estate and power of the HDM hardware implementation.

In an HDM, the cortical columns are connected together into a two-dimensional array. The hardware structure to do this would most likely consist of a set of column processors interconnected by some form of routing network. We call the processor to implement the column algorithm or a single WPNAM a column processor. There are a number of architectural issues involved in the hardware requirements of such multi-column networks that are not addressed by this paper. Other important issues concern the control signals, general clocking, and synchronization of the system. These are ignored here, though our final goal is to design column processors that will most likely be asynchronous [85] and execute in roughly the same time as their biological counterparts.

Since these models are naturally parallel, a column can be implemented by several processors, or a column processor can implement one or more columns. This "virtualization" (see Section 2.3) ratio can be chosen to maximize density and meet performance requirements. Though the column processors may have a hybrid CMOS / nanoelectronic circuits implementation, the larger inter-column network would be most likely implemented strictly in CMOS. From the perspective of an individual

column, the external, incoming connections constitute additional inputs. Likewise, out-going connections are taken from the column processor outputs.

For the CMOS and CMOL implementation analysis performed in Chapter 5 and 6, we have assumed the values for each column as shown in Table 2-2. In the "Range" column, we list the range of possible values for each parameter type. The column node size range is determined from Mountcastle's proposal [62]. The ranges of other parameters are decided by Equations (2.2), (2.3), and our assumptions for some particular parameters, e.g., the number of active nodes in each column is defined as log_2(column node size). However, we used single weight bit in the PC, PC cluster, and FPGA implementations and analysis (Chapter 3 and 4), due to the early stage of our research. We also swept the column node (input vector) size from 1 K to 64 K (some cases 32 K) in the PC and FPGA analysis to compare the performance/price ratios under different column node size scenarios. Because the inter-parameter dependencies can make these kinds of performance/price analysis and comparisons much more complex in CMOS and CMOL cases, we did not sweep column node size in the CMOS and CMOL implementation analysis as we did in the PC and FPGA analysis. Instead, we chose to the "Typical Values" for the analysis in Chapter 5 and 6.

Table 2-2: Data precisions for AM.

Parameters	Range	Typical Value
Column node size	128 ~ 128 K	16 K
Weight matrix size (single-weight-bit)	$2^{14} \sim 2^{34}$ bits	2^{28} bits
Weight matrix size (multi-weight-bit)	$2^{18} \sim 2^{51}$ bits	2^{42} bits
Weight bits	1 ~ 17 bits	4 bits
# Active nodes in column	7 ~ 17	14
Inner-product result bits (single-weight-bit)	3 ~ 5 bits	5 bits
Inner-product result bits (multi-weight-bit)	11 ~ 21 bits	18 bits

2.2.2.2. Spiking AM model

As discussed later, our preliminary analysis [49] of the hardware architectures for the non-spiking auto-associative memory model showed significant power density problems in the mixed signal CMOL implementations. In addition, it is becoming clear that cortical-like models leverage the time domain as a fundamental organizing principle [69, 70]. Consequently, in this dissertation we also look at the implementation requirements of spiking models, which operate in both space and time domains. An additional benefit is that these models also have a limited duty cycle which leads to a reduction in estimated power consumption, and potentially a more efficient use of hardware. However, in this dissertation, we do not study how to use spiking models in real applications as we do with the Palm model, e.g., EVS system in Section 4.2. The spiking models are more complex, so such an exercise is beyond the scope of this dissertation. Though there has been work on spiking associative models, such as models in [86, 87].

Spiking or pulse-based models actually lead to an important principle: *computation proceeds by incremental change in response to spikes to a baseline state, where incremental data are represented by the inter-pulse timing*. Traditional signal processing and neural models generally consist of sums of products. By using pulse-based models, the entire sum needs not be computed at any one time, rather only sparse incremental updates are processed. In this approach, then, the membrane potential of the neuron is updated by the sparse arrival of spikes. This characteristic leads to significantly increased efficiency in implementation, especially due to the use of resource multiplexing as we will show.

Consequently, for the analysis performed here, we expand our associative memory to use neurons based on spiking neuron models. As illustrated in Figure 2-8, a neuron's dendrites receive pre-synaptic spikes (or PreSynaptic Events – PSE) from other neurons' axons. The soma or neuron body accumulates the neurotransmitters (or PostSynaptic Potential – PSP) from the dendrites. When a neuron is sufficiently depolarized and passes a voltage threshold, the axon will fire and send spikes to other neurons. After a neuron fires, it will be hyperpolarized (undershoot) to a voltage lower than the resting potential. It will go back to the normal voltage after a refractory period. During this refractory period, neuron cannot fire again.

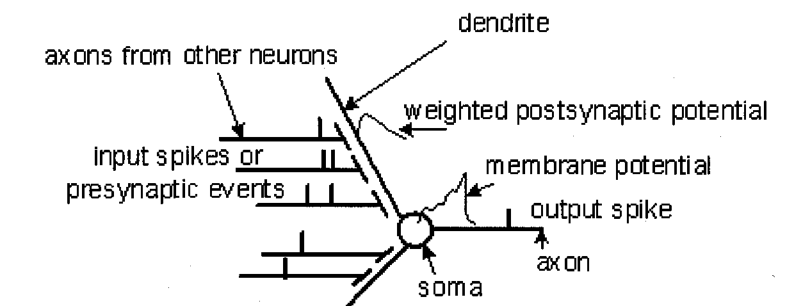


Figure 2-8: Spiking neuron model.

Suri [88] proved that all information in the spiking neuron model is determined by the time of the spike's occurrence, not by its shape. Hence, this gives us the freedom to choose the spiking neuron models that favor our hardware implementations. For the digital versions of the implementations studied in Chapter 5 and 6, we use the Gerstner spiking neuron model [39]. The reason that we chose this model is because it satisfies our criteria that the neuron model can represent the time domain with spiking or

limited duty cycle model. It is also fairly simple, has a solid mathematical foundation, and is widely used in the computational neuroscience community. In this model, the membrane potential (MP) $u_i(t)$ of neuron i ($1 \leq i \leq N$) at time t is given by

$$u_i(t) = \sum_{j=1}^{N} w_{ij} \varepsilon_{ij}(t - t_j) + \eta_i(t - t_i), \tag{2.5}$$

where w_{ij} is the efficacy (weight) of the connection from neuron j to neuron i; $\varepsilon_{ij}(t - t_j)$ is the postsynaptic potential (PSP) of neuron j contributing to neuron i; and $\eta_i(t - t_i)$ is the refractory function, which, in our model, is a negative contribution that reduces the likelihood of additional output for some period of time τ_r as soon as the MP reaches the threshold value θ. The threshold value can be static or dynamic.

The PSP function is given by

$$\varepsilon_{ij}(t) = [\exp(-\frac{t - \tau_a}{\tau_m}) - \exp(-\frac{t - \tau_a}{\tau_s})] H(t - \tau_a), \tag{2.6}$$

where τ_m and τ_s are time constants; $H(t - \tau_a)$ is the Heaviside function; and τ_a is the axonal transmission delay.

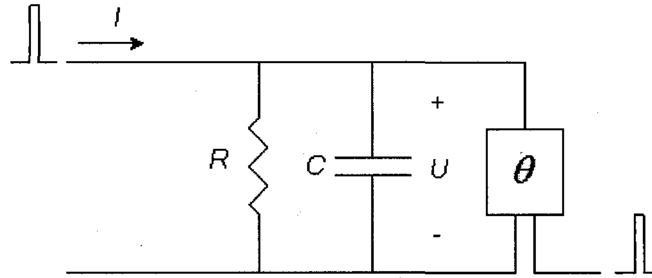


Figure 2-9: A possible RC circuit model for an I&F neuron.

However, when using analog circuit in the mixed-signal CMOS / CMOL implementations (in Chapter 5 and 6) to emulate the spiking neurons, the RC circuit illustrated in Figure 2-9 becomes more appealing to us, especially in the sense of saving much more power and silicon real estate compared with the possible analog circuits for the equations (2.5) and (2.6). Thus, for the mixed-signal CMOS / CMOL implementations, we chose to use one of Gerstner's simple spiking neuron model variations, the leaky integrate-and-fire (I&F) neuron model. This I&F model can be represented by a first-order linear differential equation: $\tau_m du/dt = -u(t) + RI(t)$, where $\tau_m = RC$ is the time constant of the current $I(t)$ leaky integrator, with the neuron's equivalent resistance *R* and capacitance *C*. As soon as the membrane potential reaches the threshold θ, the membrane potential will go to zero with time constant τ_d and kept at zero for a time of τ_r. A detailed circuit by Indiveri [89, 90] for this I&F model is shown in Figure 6-32.

Palm *et al.* [86, 87] used Gerstner's simple spiking neuron model in their modeling of primary visual cortex, which is a part of the neocortex. This model is based on the

WPNAM model for the functionality of associative memory. Rao [91] used the I&F neuron model to implement Bayesian inference for graphical models. However, in this work, we did not give an application example using either Palm's spiking associative memory model, or Rao's spiking Bayesian network model. In Chapter 5 and 6, we focus on the analysis of implementing the CMOS / CMOL column processors with Gerstner's simple spiking neuron model and I&F model, rather than the details of the spiking AM models.

A number of learning schemes exist for the spiking neuron model, such as competitive Hebbian learning through spike-timing-dependent synaptic plasticity (STDP) [92]. However, due to the significant increase in implementation complexity required to do learning, especially in spiking models we do not address it in this dissertation.

2.3. VIRTUALIZATION IN HARDWARE RESOURCES

When looking at massively parallel models, such as WPNAM, Bailey *et al.* [93, 94] used *c-graph* to represent the connectivity graph of the network being emulated and *p-graph* to represent a physical realization. When synthesizing a logic network to a set of standard cells, which are placed into a silicon design, the c-graph and p-graph are the same. But the two graphs need not be the same and can be different if there is a sharing (multiplexing) of physical resources by virtual resources. For example, when synthesizing to an FPGA where physical connections are configured dynamically, the c-graph and p-graph are not necessarily isomorphic.

One of the key contributions of the work presented here is the definition and use of the principle of virtualization and its application to our computational models and assumed implementation options (CMOS and CMOL). In the context of this dissertation, we define *virtualization* to be the degree of time-multiplexing of computations and communication tasks over hardware resources [47]. Virtualization is typically not an issue in most applications of computer technology, since almost all computation and communication are virtualized. In our sense, virtualization is about time-multiplexing a hardware resource by a number of different operations, which can be on a space-available demand driven basis, such as a bus. Alternatively, it can be explicitly scheduled (by instructions), as in the case of computational hardware.

The scheduling of the hardware resources can become very complicated as one sees with out-of-order and superscalar executions by high end general-purpose processors. One of the best examples of virtualization is a computer network. Rather than having dedicated point-to-point connections between every computer on a network, these connections are virtualized, via time multiplexing over a global interconnect structure (bus).

Virtualization then is about taking advantage of usage demands to share expensive resources. Virtualization involves space, generally hardware costs. However, it can also include power usage, time, system performance, and related trade-offs. Increasing virtualization can reduce performance, but it improves utilization. That is, the efficient use of a resource results in how to leverage performance and hardware costs of non-shared resources. Generally, virtualization implies a digital representation of the data

in this dissertation, since it is usually more difficult and costs more to multiplex analog signals. However, it is possible to multiplex analog computation so it should not be thought of as an exclusively digital technique.

Although virtualization is ubiquitous in general purpose computing, this has not been the case in many proposed hardware implementations of neural algorithms. These algorithms, and many other kinds of signal processing algorithms, have a natural massive parallelism that allows a wide range of parallel implementation options. Often, they also have low precision requirements, so the individual processors turn out to be very inexpensive. Consequently, analog implementations can be used effectively in emulating many of these algorithms in silicon. This approach certainly makes sense when computation is being done continuously, as might be the case at the sensory front end. However, as we move further back into the system, activations normally become sparser, which favors virtualization.

One way to conceptualize the implementation options for neural algorithms is to imagine a "virtualization" spectrum. At one end of the spectrum (in Figure 2-10) we have an off-the-shelf single microprocessor (or say PC) that emulates all components, computation, and communication of the model in a mostly sequential fashion [95]. At the other end of the spectrum, we literally implement the algorithm in custom silicon. This can be thought of as having a processor for each individual parallel computation, which at the finest grain is the individual synapse. Obviously, minimizing virtualization increases performance. However, it can also introduce significant inefficiency, and is more expensive. Also, the virtualization of hardware tends to make

the processor more general purpose, enhancing flexibility to be used to more applications. Figure 2-10 shows a qualitative hardware virtualization spectrum.

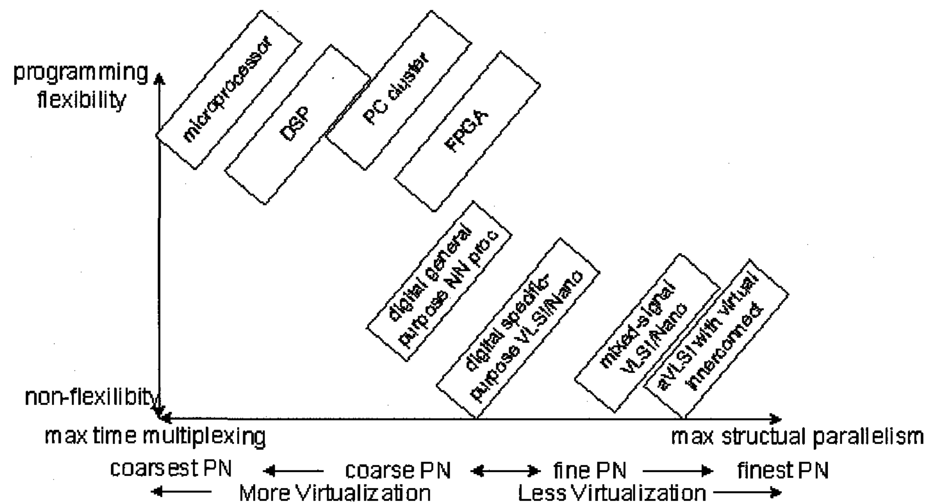


Figure 2-10: Hardware spectrum for the implementation of biologically inspired networks.
Finer-grained processing (less virtualization) means more structural parallelism, but less efficiency and flexibility.

In the neural network community, there has been a diversity of opinion on whether such hardware for emulating Biologically-Inspired Computations (BIC) should be analog or digital [96]. Quite a bit of the research into hardware for emulating neural models over the years involved implementing most, if not all, of an algorithm directly in silicon, with minimal virtualization. And over the years many groups have done that [90, 97-99]. Computing in the analog domain provides significant computational density, performing complex calculations with just a few transistors and very little power. On the other hand, computing in the digital domain is easier to design, more flexible, and easier to move to higher precision calculation [96].

However, even the analog community has accepted that there are significant interconnect limitations when silicon is compared to biological circuits, and it is difficult to communicate analog signals over a long distance. Neurons use spikes to transmit information. Consequently, the aVLSI community developed the Address Event Representation (AER) [100], which is based on spiking models. As we will show here, depending on the dynamic behavior of the network, virtualization can actually be more cost-effective for the computational models we are using [47].

Another implementation option concerns the representation of data. The spiking neuron models in this work use timing to represent data. During actual physical implementations of such computations, however, there are other options, including digital (discrete) and analog (continuous) circuits, that can use voltage or current representations [39].

It has also been argued [22] that analog circuits implement a "truer" version of the computation, and that the digital implementation is fundamentally lacking. We have not seen such an effect in the simple models we are emulating, though it is possible that, as we scale to larger systems, this could become a problem. Also, neural circuits are very noisy, so the presence of digital quantization noise may not be a problem.

The cost of virtualization in hardware is the sum of the non-shared hardware, plus the shared hardware and the hardware required to do the actual multiplexing. Those pieces of hardware that cannot be multiplexed are, in most cases, memories that store computational states or synaptic weights that are unique to each computation.

In this dissertation we use the term Processing Node or Processor Node (PN) somewhat loosely. In general, it is a simple, low precision digital processor that emulates a single neuron. Though the PN is digital, mixed signal computation can also be used. For our analysis, we studied a single PN for each neuron within a column processor at the minimum level of virtualization, as well as at the maximum level of virtualization, where one PN is assumed to emulate all the neurons assigned to a column. Lesser and greater levels of virtualization are possible, but, for the models and parameters discussed here, these tend not to be as cost-effective, and are not included in this work.

As stated in Section 2.2.2.2, the primary reason for moving from level sensitive to spiking, or pulse-based, models was to control the power dissipation in the mixed-signal CMOL. Additionally, the spiking models allow us to capture the time dimension of signals more effectively, although they are more complex. However, the other big advantage of spiking-based models is that they tend to be sparsely activated, which can be leveraged for significant savings in more effective hardware sharing. In the Palm model, only $O(log_2N)$ neurons are active at a time, which means that, on average, only $O(log_2N)$ connection computations need to be computed during the general time to update the network. One way to virtualize the neuron computation is by treating the incoming spikes as incremental modifications to an accumulated sum. This is particularly efficient if the spikes arrive, as they would on an AER bus, in a serialized manner.

In a virtualized spiking neuron, the cost per connection includes the storage of the synaptic facilitation (i.e., weight); some timing information for "active" synapses that have recently experienced some postsynaptic potential; and the shared computation; and communication hardware, which is amortized over all the connections. If spikes occur sparsely in time and the degree of virtualization is balanced properly, then there is little performance lost in waiting for the availability of the compute hardware. Likewise, the idle time for the shared resource is minimized, thus maximizing the performance/price ratio.

Although there is a global system clock, computations are assumed to be asynchronous in the sense of a global timing signal and to occur in real time. As Carver Mead once said, "we let time be its own representation" [22]. Most system times will be predictable within some jitter noise, something that also occurs in real neural circuits as well. The PNs do have a need for some internal timing, for example, post-synaptic potentials and refractory periods require predetermined times to run their course. Though we assume synchronous circuits to simplify the analysis, it is most likely that the implementation of such processors will be done almost exclusively in asynchronous logic [85].

Figure 2-11 shows different degrees of time-multiplexing spiking neuron models onto PNs, from the coarsest-grained PN (one PN multiplexes all computations) to the finest-grained PN (without multiplexing). Each column processor can have a single or multiple PNs to emulate a single cortical column (WPNAM). Many column processors, in turn, emulate a much more complex cortical function.

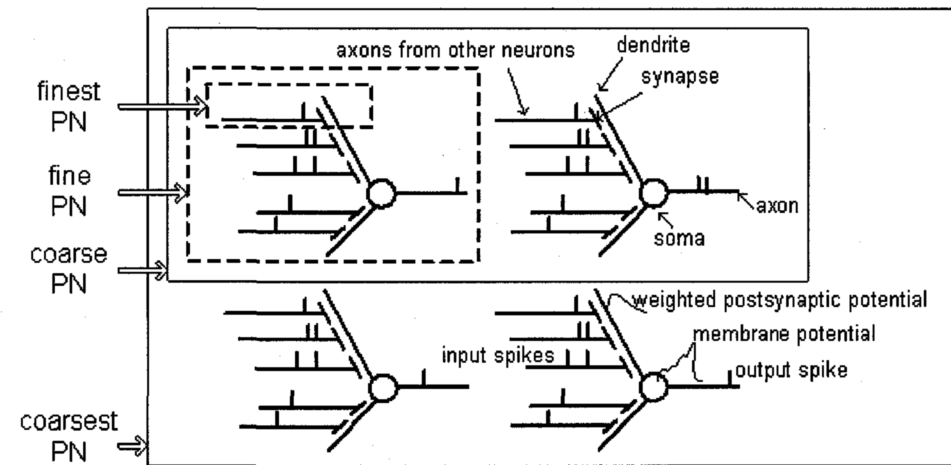


Figure 2-11: Processing nodes (PNs) time-multiplex neuron emulation.
The finest-grained PN computes a single postsynaptic potential and does not multiplex other postsynaptic potentials. A coarser-grained PN time-multiplexes computations from multiple neurons. The coarsest PN time-multiplexes all the computations required by the network.

The traditional view of neural emulation was that a small number of transistors were dedicated to an analog, non-multiplexed implementation of each synapse. However, the sparse communication and the sparse activation of our models appear to compromise the effectiveness of such an approach. In other words, depending on the dynamics of the network, dedicated, non-multiplexed compute hardware, whether it is analog or digital, does not appear to be the most efficient use of silicon area.

Although learning is not addressed here, multiplexed computational hardware looks to be an even more efficient way to utilize silicon real estate when dynamic, incremental learning is added to the model.

2.4. HARDWARE ARCHITECTURES FOR AM

2.4.1. WPNAM-LIKE MODEL HARDWARE IMPLEMENTATIONS REVIEW

Before we discuss our own hardware architectures and implementations, we look at the previous work people have done to implement the WPNAM-like cortical models, or ANN models. Actually, over the years, a number of hardware implementations of WPNAM-like algorithms have been proposed and built. Strey and Palm implemented the WPNAM model with a special-purpose digital IC (BACCHUS) [17, 28] to leverage the parallel features of their model. Figure 2-12 shows the architecture of the BACCHUS III chip. They also showed how to use this chip to implement the neural associative memory for an image recognition application [17]. Although implemented in older technology, the chip architecture is still relevant to the exploration presented here.

Each BACCHUS chip emulates 32 neurons in the WPNAM model in parallel. The corresponding weights are stored off-chip in standard DRAM memory chips. The BACCHUS chip executes Boolean OR operations and the threshold operations (for the output function). Because all the neurons in the WPNAM model execute the same instruction simultaneously, the parallel structure of Palm's hardware platform is a single-instruction multiple-data (SIMD) computational model. The CP (Control Processor) broadcasts the input vector to all BACCHUS chips, which perform the inner-product and threshold comparison operations, and send the output vector to the CP.

Figure 2-12: The architecture of the BACCHUS III chip.
(Adapted from [28].)

Rückert's research group in Germany also developed digital hardware implementations of the WPNAM model in the 1990s [101-104]. They used digital VLSI technology to build the associative memory processing nodes. In 2002, Porrmann, Rückert, and their colleagues proposed a reconfigurable hardware accelerator RAPTOR2000 with Xilinx Virtex FPGA, to implement the WPNAM model [101].

The RAPTOR2000 has several serially-connected daughter-boards. Each daughter-board has one Xilinx Virtex XCV1000 FPGA, which implements an associative memory with 512 neurons. The SDRAM controller and a neuron processing unit (for the output neurons' computations) are implemented on the FPGA (Figure 2-13). The SDRAM controller provides a 512-bit register that buffers the data to and from the

SDRAM. Each output neuron processes one row of the memory matrix. Each bit 1 in the input vector selects one element of the memory matrix and thus one component in each row. The activated elements of the memory matrix are processed sequentially. The activated element is read by all neurons in parallel. Those neurons that receive a zero will decrement their internal counters and stop firing if and when the counter reaches zero. The synthesis results show that about 3200 Virtex CLBs (Configurable Logic Blocks) were required for 512 neurons and 250 Virtex CLBs for the SDRAM controller. The neurons operate at 50 MHz, and the SDRAM controller uses a clock frequency of 100 MHz. The time for the association of a vector depends mainly on the SDRAM access time. Porrmann claimed their implementation requires about 5 μsec for one association. However, neither the neuron (node) update rate nor the connection update rate for this FPGA implementation was provided.

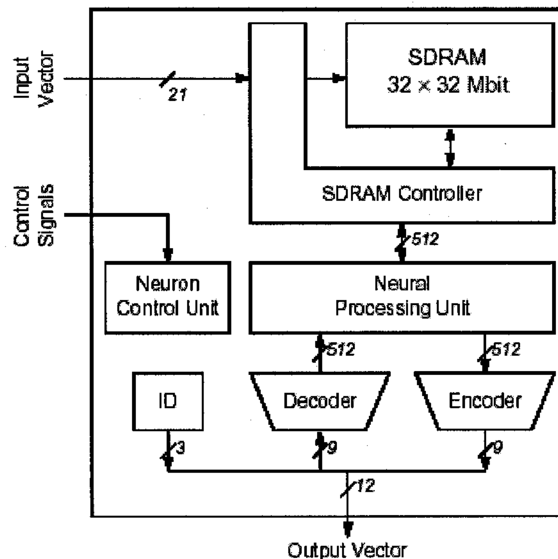


Figure 2-13: Neuron unit block diagram of the Porrmann's FPGA design for the WPNAM model.
(Adapted from [101].)

The BACCHUS chip used sparse representation for the input and output vectors, and implemented learning algorithm on chip. The RAPTOR2000 reconfigurable accelerator did not use sparse representation for the input and output vectors, and did not implement learning algorithm. Furthermore, the functionality provided by those two implementations was limited to specific AM algorithm, without much flexibility. The architectures for those two implementations did not use virtualization, meaning maximum performance was pursued, but not the efficiency. Although they could, they did not use pipelining in their designs either.

During the late 1980s and early 1990s, Hammerstrom (Adaptive Systems Inc.) [105] proposed and taped out CNAPS chips for more general ANN algorithms and other non-ANN applications, such as Back-Propagation (BP), image processing, and Optical Character Recognition (OCR). In each CNAPS-1064 chip, there are 80 PNs during fabrication. Before packaging, only 64 working PNs are tested and enabled. The rest 16 PNs are disconnected from the bus and the power grid. This redundancy of PNs during manufacturing guaranteed a high yield. Because of this 64-parallel-working PNs, the CNAPS-1064, for example, can store and train the entire NetTalk [106] network in about 7 seconds [105]. CNAPS is a single instruction, multiple data stream (SIMD) architecture. As shown in Figure 2-14, in CNAPS, the PNs are connected in a one-dimensional array. Each PN can only communicate with its right or left neighbors. The sequencer broadcasts each instruction and data to all PNs, which execute the same

instruction at each clock. The PNs transmit output data to the sequencer (with arbitration).

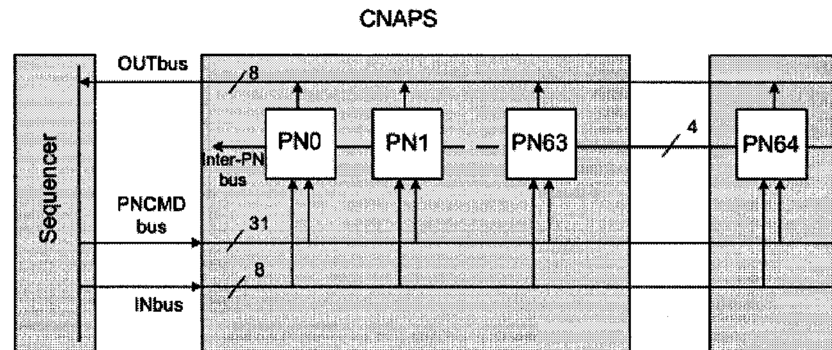


Figure 2-14: CNAPS system architecture.
(Adapted from [105].)

Figure 2-15 shows that each PN in CNAPS has a local memory, a multiplier, an adder/subtracter, a shifter/logic unit, a register file, and a memory addressing unit. During each clock, the computational units can work at the same time. With pipelining, the memory accessing, instruction fetching, and message passing can also execute in the same clock cycle. The 16 × 16 fixed-point multiplier is sufficient (in the sense of precision) for most ANN algorithms. The non-linear activation function (e.g., sigmoid function) is realized by Look-Up Table (LUT) with the local memory.

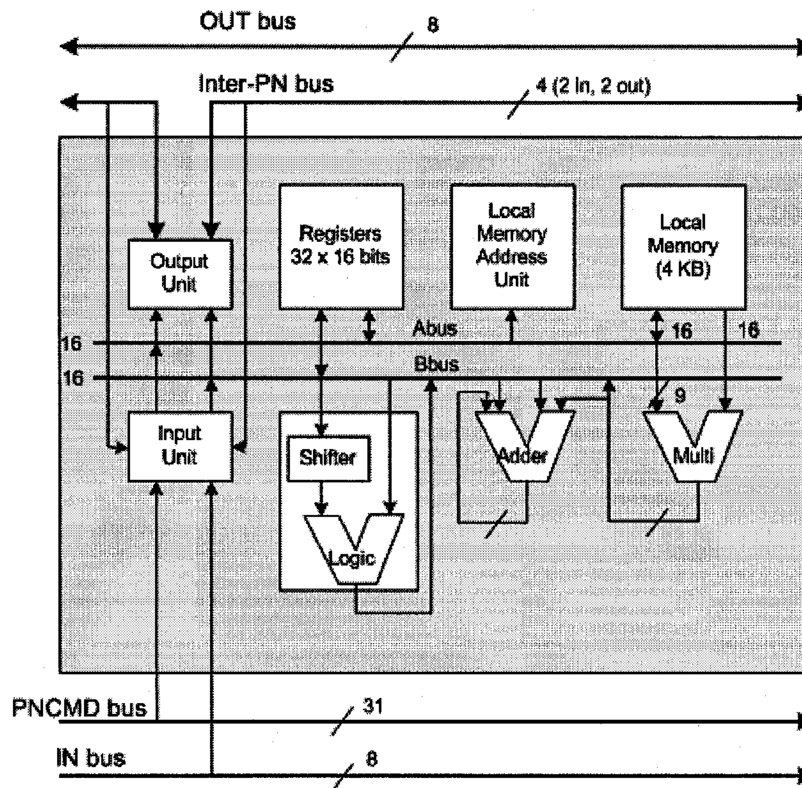


Figure 2-15: CNAPS PN architecture.
(Adapted from [105].)

Because most ANN algorithms use matrix-vector inner-product operations, the multiply-accumulate unit (i.e., the multiplier and adder in the PN) is very helpful for ANN algorithms. Hammerstrom [4] claimed that the maximum compute rate for CNAPS was 1.2 billion multiply-accumulates per second per chip (at 25 MHz and 6 watts worst case power consumption), which was about 1000× faster than the fastest workstation at that time, about a decade ago.

In addition to the digital designs explained above, Türel *et al.* proposed a possible implementation of the Hopfield model using mixed-signal CMOL, or, more

specifically, CMOL CrossNets [107]. CrossNets can be described by the fire-rate equation $\tau_0 dx_i / dt + x_i = g\sum_j w_{ij} f(x_j)$, where τ_0 is a constant, $x_i = GV_i / V_0$ is normalized dendritic voltage, w_{ij} are synaptic weights, g is the effective somatic gain, and G is the voltage gain of a somatic amplifier. The output function is $f(x) = \begin{cases} x & (|x| < 1) \\ \text{sign}(x) & (|x| > 1) \end{cases}$. As illustrated in Figure 2-16, the red lines are output signal nanowires from neurons, and the blue lines are input signal nanowires to the neurons. The green circles represent nano-switches. The soma calculation is implemented in CMOS. However, the synapses are implemented with the nanodevices. The nanowires represent the dendrites and axons in the CrossNets. Türel [107] reported that the CMOL CrossNets should be able to emulate a Hopfield model with high defect tolerance.

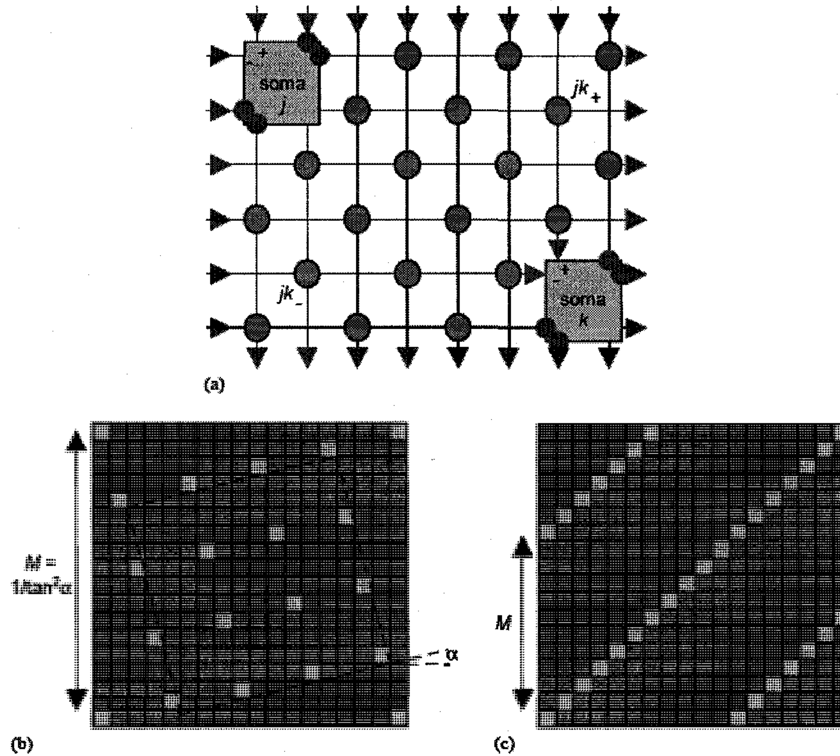


Figure 2-16: CrossNets.
(a) Aeneral structure, and two varieties of (b) InBar and (c) FlossBar. (Adapted from [107].)

2.4.2. Hardware Architectures Used in This Work

One can think of an abstract neuron model as consisting of three components:

- State information;
- Local computation that takes inputs, updates local state, and computes outputs;
- Connection mechanisms to allow neurons to communicate.

People can disagree about whether the state and local computation should be analog or digital. Complex dynamics, functional density, and low power favor analog circuits, while sparse activation, complex learning algorithms, and large, simplistic models favor digital circuits. However, for large cortical-like association models, multiplexed connectivity is the only alternative – Why?

Assume a rectangular array of silicon neurons where each neuron receives input from its N nearest neighbors. Each such connection consists of a single metal line (and the number of metal layers is much less than n). The area required by the metal interconnect is of $O(n^3)$ [93, 94]. The non-local interconnect problem manifests itself in simulation costs as well. It and the global WTA (Winner-Take-All) were the major time sinks in all of our simulation experiments (see Chapter 3, 4, 5, and 6).

Again, as illustrated in Figure 2-10, the hardware platforms we could choose to implement the AM algorithms can vary from the off-the-shelf PC, to aVLSI (analog VLSI). In between those two, there are PC cluster, DSP (Digital Signal Processor), FPGA, digital general-purpose NN processor, digital special-purpose CMOS / CMOL, and mixed-signal CMOS / CMOL architectures. However, we could not choose all of them as the hardware architecture candidates in this work. The five hardware architectures we used to implement the AM algorithms or estimate the possible implementations for such algorithms are PC, PC cluster, FPGA, digital special-purpose (full custom) CMOS / CMOL, and mixed-signal CMOS / CMOL. In the following paragraphs, we will explain why we chose those architectures, and why not the other architectures.

The PC or microprocessor is the traditional computing architecture for people to run simulations for AM. We did not invent new microprocessor architecture in this work, however, we used Intel®'s off-the-shelf CPU to run the non-spiking AM simulations. The PC implementation is a baseline for us to compare the performance/price ratios across all hardware architectures in this work. PC is the most flexible way to program for all kinds of AM models, as shown in Figure 2-10. PC virtualizes all computations and communications within the traditional von Neumann's computer architecture. The reason we did not run spiking AM algorithms on PC (and FPGA) was because we did not have a fully working spiking-AM algorithms. What all we had for the spiking AM algorithms is the basic spiking neuron model, along with possible memory and interconnect requirements. The spiking AM algorithms are only applied to CMOS and CMOL architectures (in Chapter 5 and 6).

The reason that we did not choose aVLSI is because aVLSI implementation will not virtualize any computations. The benefit is the upmost performance we could expect from using aVLSI. However, the disadvantage of aVLSI is the inefficiency in implementing sparsely activated AM model explained in Section 2.2.2.1and 2.2.2.2. This inefficiency diminishes the performance gain from using aVLSI. In aVLSI implementation, each synapse and neuron's computations have their own dedicated circuits, except some of the pulse communications between different neurons with the help of AER [100]. Also, aVLSI is front-end oriented for emulating biologically-inspired models (BIC). Hammerstrom [4] summarized the differences between front end versus back end operations, the two domains for emulating the BIC models. Front

end operations tend to involve direct access to raw signals from the front end sensors (Figure 4 in [4]). The operations include filtering and extracting features, thus, aVLSI fits well into emulating such kind of sensory signal processing. The back end operations deal more about Intelligent Signal Processing (ISP) [81] in the BIC domain. Hammerstrom [4] said: "As we move from front end to back end, the computation becomes more interconnect driven, leveraging ever larger amounts of diffuse data at the synapses from the connections." Hammerstrom [4] also argued that the analog-based circuits are primarily focused on the front end. So, there is not much room for aVLSI to emulate the robust back end ISP computations. This is also the reason we chose the more virtualization-oriented digital and mixed-signal CMOS / CMOL architectures to emulate the AM. The mixed-signal CMOS / CMOL designs are the finest-grained PN hardware architectures in this work as shown in Figure 2-10.

The digital general-purpose NN processor, such as CNAPS [105], can be one of the hardware candidates in this work. However, we need more dedicated circuits for the AM algorithms to compete with the general-purpose microprocessor in the sense of performance. Although such dedicated circuits (or digital and mixed-signal special-purpose NN processors) lose programming flexibility to the more general-purpose NN processors, the huge performance benefit from those dedicated circuits for AM algorithms could enable the digital / mixed-signal CMOS and CMOL designs to race well against PC and FPGA. The reason we chose CMOL as the nanoarchitecture to emulate AM can be found in Section 1.1, and 6.1.

We also chose FPGA and PC cluster to be the other two hardware candidates to implement the AM algorithms. PC cluster can be as flexible as PC (shown in Figure 2-10), and can also modestly move toward the less-virtualization direction in the hardware spectrum (to the right hand side of Figure 2-10) to improve performance. However, it (PC cluster) also increases the cost proportionally (see Chapter 3). In Figure 2-10, FPGA has less flexibility than PC and PC cluster, however, less virtualization too. That means FPGA should have better performance than PC and PC cluster when power and silicon cost are considered into this comparison. FPGA now becomes an important tool for people to run simulations or build prototypes for AI, ANN, and non-ANN applications (details in Section 3.1).

Although we did not put multi-core or CMP (Chip Multi-Processor) into the hardware architecture spectrum in Figure 2-10, it is worth for us to study CMP's performance/price on emulating HDM algorithms in the future. We put the possible future work in Chapter 7.

2.5. METHODOLOGY

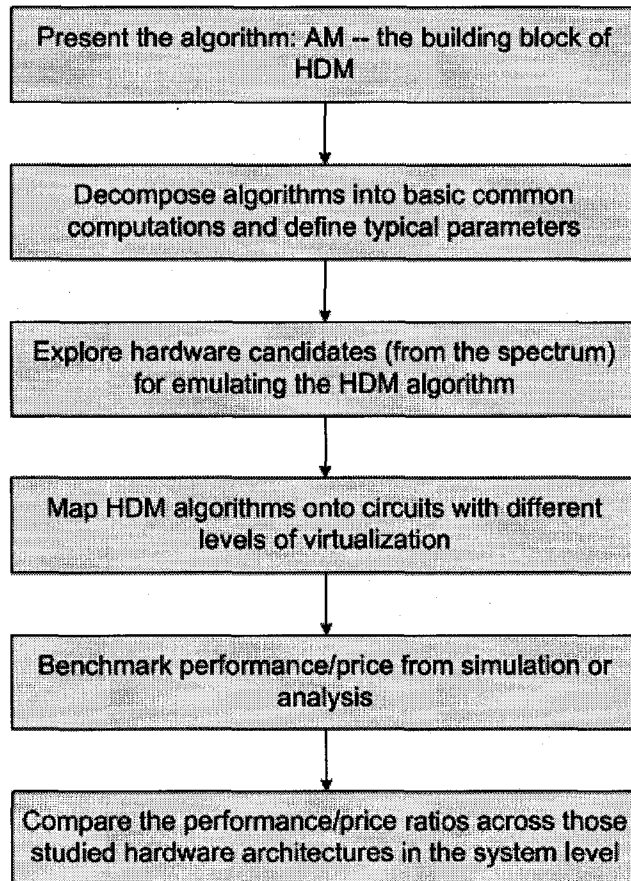


Figure 2-17: Methodology flow chart used in this work, i.e., performance/price ratio comparisons for AM algorithm hardware implementations.
(Adapted from [40].)

The objective of this research is to conduct a methodical search of the design space for different hardware architectures to emulate a single HDM column network. Zaveri [40] from our group proposed a formal design flow for implementing abstract cortical models. When conducting this methodical search, we list six major steps in Figure 2-17 to show the methodology flow chart used in this work.

The first step in the methodology in Figure 2-17 is to present the algorithm, the non-spiking and spiking AM algorithm as the building block of HDM. We have presented this in Section 2.2. The second step is to decompose the algorithms into basic common computations, which are matrix-vector inner-product and *k*-WTA operations. Also, we realized that compared to those two operations, multiplication, addition, and comparison, which are used for the two operations, are fundamental computations one level lower. The second step was also presented in Section 2.2.

To implement the AM algorithm with suitable hardware architectures, the third step we did was to explore the hardware candidates, which were explained in Section 2.4. In later chapters (Chapter 3, 4, 5, and 6), we covered the next three steps in the methodology flow chart. For example, for the fourth step, the mapping of computations to hardware is a complex issue since it concerns the degree of multiplexing of the computational hardware. Because of the significant parallelism available in an AM network, we could spend significant computational hardware to do massive speedup of each node (neuron) in the network. At the other end of the spectrum in Figure 2-11 (the coarsest PN, which multiplexes all the computations required by the network), we could use one arithmetic unit to sequentially compute all the nodes in a column, essentially multiplexing the computational hardware over the required computations. Each point on the spectrum in Figure 2-11 brings its own set of performance/price trade-offs. Mapping of the computation to hardware is a critical problem for FPGA, CMOS, and CMOL implementations, as explained in Chapter 4, 5, and 6, respectively. However, for the PC implementations, the mapping is to

maximally virtualize the PC system for the computations (see Chapter 3). As we have explained in Section 2.4.2, we only mapped the non-spiking AM algorithms on the PC, PC cluster, and FPGA, while both the non-spiking and spiking AM algorithms on the CMOS and CMOL architectures.

The fifth step in our methodology is to benchmark performance/price from simulation or analysis of implementing AM with different hardware architectures. For PC and PC cluster implementations, this benchmarking or measuring performance is straightforward, because we could record the simulation's running time (see Chapter 3). We did this for one of the FPGA implementations in Chapter 4 too. However, for the optimal FPGA, CMOS, and CMOL architectures, we did not have the physical implementations, so that benchmarking the performance/price ratio was achieved through estimate. We used ideal performance/price numbers or numbers from literatures for this estimate. We also scaled down the CMOS transistor size to 22 nm technology according to ITRS [108], since 22 nm is the projected feature size for mass production in 2016. The details of this estimate and scaling are in Section 4.3.2, 6.1.4.4, and 6.4.

The final step in this work is to compare the performance/price ratios across those studied hardware architectures at the system level. The performance for the non-spiking AM implementations is measured by Connections Per Second (CPS) that the hardware could compute for the AM networks. The results are listed in Section 3.2.5, 3.4, 4.2.2, 4.3.3, and 6.4. For spiking AM implementations with CMOS and CMOL, we used the maximum input spiking rate as the performance measurement (see

Section 6.4). For estimating the price, we used power and silicon area as the measurement.

3. DESIGN WITH GENERAL-PURPOSE ARCHITECTURES

3.1. INTRODUCTION

Although traditional general-purpose microprocessors could provide different kinds of parallelism, such as instruction level parallelism or multi-thread, we could still think of a single microprocessor chip as mostly a sequential computational platform. If we do not care about the performance and real time[3] requirements, writing software implementations of biologically inspired algorithms on a PC or workstation is sufficient. An overview of simulators of some artificial neural networks based on such general-purpose PCs can be found in [109].

Our study of association algorithms begins with implementations on a PC, primarily with Intel's Pentium 4 microprocessor as our baseline of performance comparisons. Having a working simulator and an understanding of the performance bottleneck of a PC implementation is as important as finding the most efficient way to implement the AM algorithm on specialized hardware platforms.

More recently PC clusters, typically with distributed memory, are becoming more and more popular in scientific and engineering applications [13, 51, 110]. Such clusters are an easy way to increase the performance of computing the AM algorithm. However,

---

[3] Here, real time refers to either running video/audio applications or real world applications that must be shown/played in the real time manner, or emulating an artificial "scaled-up" brain with response time approaching that of biological systems, which requires huge amount of computations that are beyond all but the very largest parallel computing systems.

PC cluster is a limited solution for real applications. This is particularly true if the inter-PC interconnect bandwidth is a performance bottleneck, in which case we could not expect the ideal performance scaling from a PC cluster. Thus, we need to investigate the performance/price ratio of PC cluster implementations as well as single PC implementations.

This chapter will look at the implementation methods of WPNAM on a PC and a PC cluster, compare their performance/price ratios, and use the results as a baseline to compare with FPGA, CMOS, and CMOL implementations of the same AM models (see Chapter 4 and Chapter 6).

3.2. IMPLEMENTATION WITH PC

3.2.1. CSIM Introduction

Our PC simulation is based on a C++ software system – CSIM (Cortex SIMulation). The primary objective of CSIM is to allow researchers to build simulators of large neuro-like models quickly and run them on a variety of parallel machines. CSIM is not actually a simulation system so much as it is a collection of library modules and utilities that allow a rapid prototyping capability for a range of simulators for modeling neuro-computational models. The models implemented by CSIM are fairly simple, and more oriented toward easing the next step to silicon implementation (FPGA or full-custom VLSI). There are both C++ and MATLAB® versions of CSIM. Using CSIM, we ran several experiments measuring average memory bandwidth and node-update rate per second [41]. CSIM also uses the MPI (Message Passing

Interface, version 1.1) [111, 112] standard for inter-process communication in multi-thread (and multiprocessor) systems. The WPNAM algorithm that CSIM implemented is not optimal. Both for the sparse and full representations (explained in the following paragraph), the weight matrix is stored by row. The PC reads in one row at a time and performs an inner product with the input (test) vector. Because the input vectors are very sparse, they are stored as a set of indices. Another approach to the matrix-vector inner-product with very sparse data structures is to store the weight matrix by columns and then only read the columns that correspond to a non-zero element in the input vector. We used the latter version (column-wise weight matrix) on the PC and PC cluster implementations.

The weight representation in CSIM is implemented in two different ways: *full matrix* and *sparse matrix*. Full-matrix weight representation is defined that all the bits of the synapses (weights) are explicitly stored in the matrix, no matter there is a connection or not for the corresponding weight. A sparse matrix weight representation is defined that only the indices and values (both of type *unsigned int*) of the synapses with connections are stored in the weight matrix. The performance of the different CSIM options was collected.

3.2.2. Network Configurations and VTune Introduction

The PC version of the WPNAM model is implemented on a DELL Dimension 8100 (P4 1.8GHz) with CSIM. The network configurations are listed in Table 3-1.

Table 3-1: Configurations of the PALM associative neural networks. *vector_size* is the number of input neurons, which is also equal to the number of the output neurons; *number_train_vector* is the number of training vectors; *active_nodes* is the number of active nodes in each of the training vectors (it equals $\log_2$(*vector_size*)); the Hebb matrix fullness is the measurement of the sparseness of the Hebb weight matrix of the network (the percent of non-zero entries).

	vector_size	number_train_vector	active_nodes	Hebb matrix fullness
Conf 1	1024	706	10	10.00%
Conf 2	2048	1412	11	3.50%
Conf 3	4096	2826	12	1.75%
Conf 4	8192	5652	13	0.72%
Conf 5	16384	11304	14	0.51%

For our experiments, we used vectors that are randomly generated internally to CSIM. The time for generating the vectors was not measured. The most time consuming operations were the inner-product and *k*-WTA.

Intel's VTune Performance Analyzer 6.1 is used to evaluate the code performance on PCs with Intel processors. With VTune's Event Based Sampling (EBS), we can view the performance of our code in relation to processor events, and obtain performance measures such as CPI (Clocks Per Instruction), L1 cache load misses, and L2 cache load misses. Additionally, we can drill down to any hot spots in the program. In order minimize the performance impact of the compiler, we set it to maximum speed optimization. Since VTune is sampling based, it still provides a useful view of the causes of the varying performance by different configurations of the associative network.

3.2.3. Average Memory Access Time for Data (AMATD)

To help us understand the specific performance characteristics of the associative memory algorithm, we defined a quantitative method to measure the memory performance within the simulation program, the Average Memory Access Time for Data or AMATD:

$$\begin{aligned} \text{AMATD} &= \text{L1 data cache hit time} + \text{L1 data cache miss rate} \times \text{miss penalty} \\ &= \text{L1 data cache hit time} + \text{L1 data cache load miss rate} \times (\text{L2 cache hit time} \\ &\quad + \text{L2 cache load miss rate} \times \text{main memory hit time}) \end{aligned} \tag{3.1}$$

In order to obtain the L1 data cache hit time, L2 cache hit time, and main memory hit time, we used the method by Andrea Dusseau of U. C. Berkeley [21] to run the memory hierarchy test (using different data array size and different data read and write offset, or stride, to test the PC's average read and write time for each word) for the DELL 8100. The memory hierarchy test code was also compiled with speed optimizations.

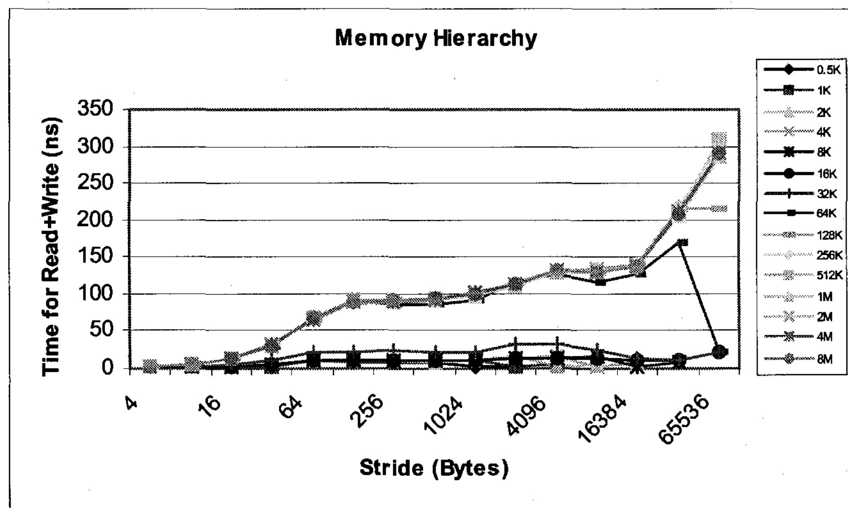


Figure 3-1: Memory hierarchy test for Pentium 4 machine (DELL Dimension 8100).
The x-axis is the stride (offset) to read and write array elements from the example in [21]. The y-axis is the time for reading and writing an integer. The different curves show different data array sizes from 0.5 KBytes to 64 MBytes.

As illustrated in Figure 3-1, the L1 data cache hit time is about 0.8 ns (or 1.6 ns/2), the L2 cache hit time is about 4 ns (or (9.6 ns - 1.6 ns) / 2), and the main memory hit time is about 45.2 ns, or (100 ns - 9.6 ns) / 2.

3.2.4. COLUMN-WISE INNER-PRODUCT

Over the last ten years there have been a number of hardware implementations of the WPNAM model. Since the development of the first Palm NAM silicon chip [28], the performance of commercial microprocessors has increased dramatically according to Moore's Law. Although the memory bandwidth lags behind the CPU clock rate, the Pentium 4 CPU can compensate to some degree by having 1 MB on-chip cache storage, which was not possible even ten years ago. For providing a baseline

performance number, it is important that we evaluate the PC's performance in implementing the WPNAM algorithm. With the baseline performance of the Pentium 4, we can then evaluate the implementation on a PC cluster to see the speedup obtained using multiple processors, and the influence of inter-processor communication overhead.

When implementing the matrix-vector inner-product, there are several issues to be considered. The first is whether we have sparse and full weight matrix representations. The second issue is that we can do row-wise inner-product or column-wise inner product. Figure 3-2 shows a traditional row-wise inner-product method. Figure 3-3 shows the method of column-wise inner-product.

As shown in Figure 3-2 for the row-wise inner-product, the computer reads in one word in the i-th weight row according to one active node in the input vector V_{in}, and examines the corresponding bit in the word to see if it is 1 or 0. If it is 1, then the computer accumulates the result value in the i-th row of the somatic-potential vector S. Assuming that the memory word width is 32 bits, and the input vector V_{in} is sparsely coded, this method of row-wise inner-product wastes too much memory bandwidth, approximately 96.9%, for the bits that are zeros. This is because we use 32 bits to represent a one bit weight, so for each weight matrix word that is read, there is only one bit out of the 32 bits to be tested for a one or zero. We do not need the remaining 31 bits. In addition, even if we could optimize the 32 bits to represent 32 weight connections, because of the sparseness of the input vector V_{in}, a lot of weight bits are discarded after being read into the CPU from external memory.

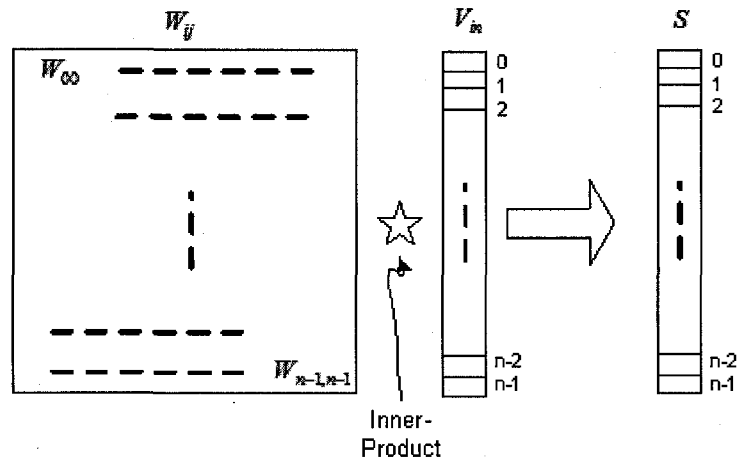


Figure 3-2: The row-wise inner-product.

W_{ij} is the weight matrix. The i-th element in the somatic-potential vector S is the inner-product vector of the weight matrix and the input vector V_{in}.

Because of the sparseness of the input vector V_{in}, we propose a much more efficient method for inner-product, namely the column-wise inner-product [113]. In this operation, we only read the columns of the weight matrix W_{ij} according to the active nodes in the input vector V_{in}. We then do a column-wise add of these column vectors to get the result vector, which is the somatic-potential vector S. For example, in Figure 3-3 the input vector V_{in} has three active nodes, 0, 2, and n-1. We read in the 0-th, 2nd, and (n-1)-th columns of the weight matrix W_{ij}, and add all three columns together to get the result vector S. Because the columns of the weight matrix that are read in from memory do not contain as many redundant bits as the row-wise inner-

product, the column-wise inner-product method could save us more time on reading the weight matrix as compared to the row-wise inner-product method.

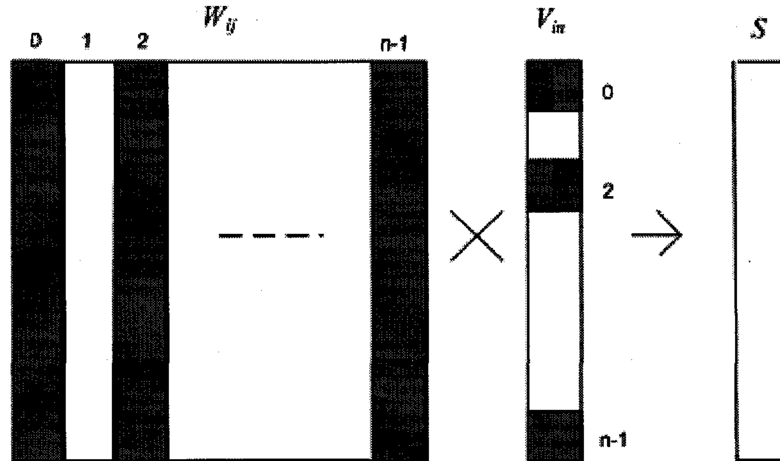


Figure 3-3: The column-wise matrix-vector inner-product.
This diagram shows an example of a sparse input vector V_{in} only with the 0th, 2nd, and $(n\text{-}1)^{th}$ nodes active.

3.2.5. Simulation Results

We ran CSIM on a DELL 8100, and used VTune Performance Analyzer 6.1 to evaluate the L1 cache load miss rate, L2 cache load miss rate, and CPI, as illustrated in Table 3-2 and Table 3-3.

When we drilled down to the hotspot of the program CSIM( ), we found that the OutGenObj::Execute( ) (the procedure for inner-product) has the highest CPI for all functions. We can also see the detailed data from the VTune Performance Analyzer and for Equation (3.1).

From Table 3-2 to Table 3-5, we find that the sparsely represented weight matrix has a greater AMATD and requires more computational time than the full-weight matrix representation. We also find that the sparsely represented weight matrix has a smaller CPI than the full-weight matrix representation, both for the overall program and the hotspot function. This indicates that the sparse-weight matrix representation requires more instructions. That is obvious, since each connected synapse should have a 32-bit *unsigned int* index and a 32-bit *unsigned int* value for the sparse-weight matrix representation. For a full-weight matrix representation, however, each input/output node pair has a one-bit weight to represent connectivity. If the connectivity is very sparse, the sparse-matrix will be a better solution. From a memory bandwidth perspective, the break-even point is at a sparseness of about 3%.

Table 3-2: The overall performance results for the sparsely represented matrix with release compiler mode and speed optimizations enabled.

	L1 Cache Load Miss Rate (%)	L2 Cache Load Miss Rate (%)	Average memory access time for data AMATD (ns)	# (loads + stores) / # (total instructions)	Time (s)	CPI
Conf 1	1.03	15.97	0.92	0.51	2.3	1.20
Conf 2	1.04	16.67	0.92	0.51	5.6	1.20
Conf 3	1.06	17.27	0.92	0.51	13.5	1.22
Conf 4	1.08	19.05	0.94	0.51	31.9	1.23
Conf 5	1.08	22.92	0.96	0.51	75.1	1.24

Table 3-3: The overall performance results for the fully represented matrix with release compiler mode and speed optimizations enabled.

	L1 Cache Load Miss Rate (%)	L2 Cache Load Miss Rate (%)	Average memory access time for data AMATD (ns)	# (loads + stores) / # (total instructions)	Time (s)	CPI
Conf 1	0.45	10.00	0.84	0.51	0.5	1.49
Conf 2	0.58	15.63	0.86	0.51	1.1	1.53
Conf 3	0.89	11.90	0.88	0.51	3.0	1.53
Conf 4	0.76	16.27	0.89	0.51	8.0	1.50
Conf 5	0.58	16.57	0.87	0.51	11.2	1.42

Table 3-4: The hotspot OutGenObj::Execute() performance results for the sparsely represented matrix with release compiler mode and speed optimizations enabled.

	L1 Cache Load Miss Rate (%)	L2 Cache Load Miss Rate (%)	Average memory access time for data AMATD (ns)	# (loads + stores) / # (total instructions)	Ratio of clockticks for the total vtune event samplings	CPI
Conf 1	0.23	1.11	0.81	0.4	0.39	1.16
Conf 2	0.28	1.27	0.81	0.4	0.38	1.14
Conf 3	0.25	5.18	0.82	0.4	0.37	1.13
Conf 4	0.29	15.53	0.83	0.4	0.37	1.13
Conf 5	0.37	33.63	0.87	0.4	0.36	1.14

3.2.5.1. Vector size impact to the memory bandwidth

A major concern is that these programs exhibit little locality, which compromises cache performance and puts greater dependence on memory bandwidth. Consequently, we examine the bandwidth issue in more detail.

From Figure 3-4, we can see that, as the vector size increases (the number of training vector numbers used is approximately 0.69 of the vector size, and the number of active nodes is $\log_2$(*vector size*)), the inner-product memory bandwidth decreases. For the sparse-matrix representation, the bandwidth goes to 140 MB/sec when the vector size is about 32 K. For the full-matrix representation, the bandwidth goes to 100 MB/sec when the vector size is about 32 K.

Table 3-5: The hotspot OutGenObj::Execute() for the full-matrix representation with release compiler mode and speed optimizations enabled.

	L1 Cache Load Miss Rate (%)	L2 Cache Load Miss Rate (%)	Average memory access time for data AMATD (ns)	# (loads + stores) / # (total instructions)	Ratio of clockticks for the total vtune event samplings	CPI
Conf 1	0.04	23.08	0.81	0.57	0.10	1.43
Conf 2	0.08	0.00	0.80	0.56	0.11	1.89
Conf 3	0.11	7.19	0.81	0.55	0.08	1.77
Conf 4	0.11	9.36	0.81	0.52	0.06	1.49
Conf 5	0.14	13.51	0.81	0.55	0.05	1.69

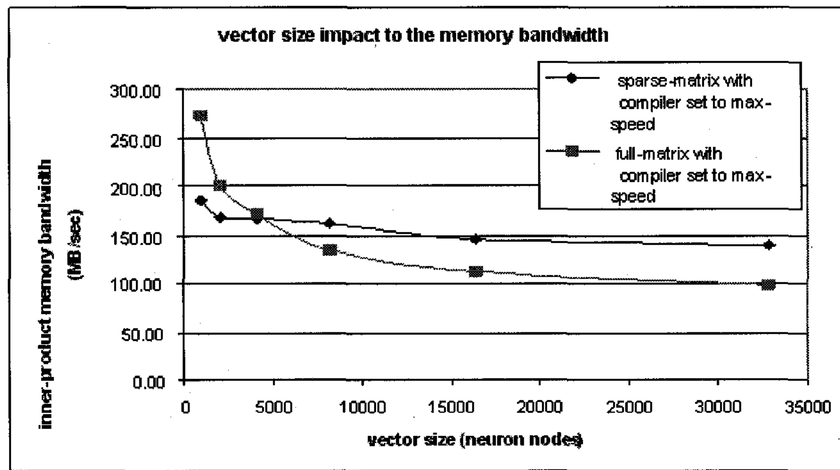


Figure 3-4: The relationship between the number of network nodes (vector elements) and the inner-product memory bandwidth for the Pentium 4.

In Figure 3-4, the horizontal axis is the vector size, and the vertical axis is the inner-product memory bandwidth (MB/sec). The diamond-dotted curve denotes the sparse weight matrix representation. The square-dotted curve denotes the full binary weight matrix. As shown in Figure 3-4, when the vector size increases, the inner-product memory bandwidth is reduced, both for the full-matrix and for the sparse-matrix representations, where the number of training vector numbers used is 0.69 of the

vector size, n, and the number of active nodes, *k*, is approximately $\log_2 n$. For the sparse-matrix representation, the memory bandwidth goes to 140 MB/sec when the vector size, n, is about 32 K. For the full-matrix representation, the bandwidth decreases to 100 MB/sec when the vector size is about 32 K. Figure 3-4 also shows that the full binary weight matrix representation has a smaller memory bandwidth requirement than the sparse weight matrix representation when the vector size is greater than 5 K. Intuitively, we prefer the method with a lower memory bandwidth requirement. From Table 3-2 to Table 3-5, the greater AMATD in sparse weight matrix representation proves that the full binary weight matrix representation is favorable in our WPNAM simulation cases.

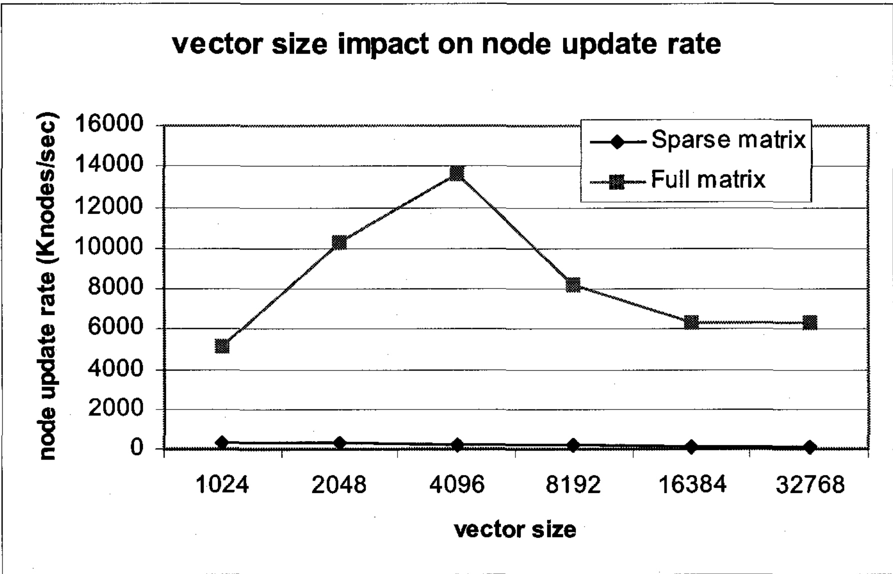


Figure 3-5: Vector size versus node update rate for Pentium 4.

Figure 3-5 shows that the node update rate with full-matrix is much greater than that of sparse-matrix. The horizontal axis is the vector size. The vertical axis is the node update rate (Knodes/sec). The diamond-dotted curve is the sparse-matrix with compiler set to maximum speed. It goes to 110 Knodes/sec when the vector size is about 32 K. The square-dotted curve is the full-matrix representation with compiler optimization set to maximum speed. It goes to 6000 Knodes/sec when the vector size is about 32 K. Because sparse matrix wastes much more bandwidth on reading the indices of the active weight bits than the full matrix that just reads the weight column bits corresponding to the non-zero elements in the input vector, the full-matrix has less of a memory bandwidth requirement than the sparse-matrix. For the sparse-matrix representation, each active weight bit has a 32-bit row-index and a 32-bit column-index.

3.3. IMPLEMENTATION WITH PC CLUSTER

One common technique to get more performance using the basic PC platform is with a multiple PC configuration, generally referred to as a PC cluster [114]. A common cluster configuration is to use Linux based PCs in what is referred to as a "Beowulf cluster" [115], where a number of PCs are connected to a common broadband network. Software is available that allows these PCs to share tasks and communicate data and results.

In addition to system support there are programming environments for developing parallel programs for such clusters, the most commonly used is MPI (message passing

interface) [111]. MPI consists of a set of calls (in C, C++, or Fortran) for implementing parallel programs. CSIM has a parallel processing option that is based on MPI. In this section, we present results of executing CSIM on a small Beowulf cluster. In these experiments, a single association network is spread across the processors as shown in Figure 3-6.

The purpose of this experiment was to understand the overhead required to execute a single association network across multiple processors. However, real implementations will probably use multiple association networks, with each module assigned to a processing node.

There are two components in the WPNAM algorithm that require different approaches to parallelization. The first is the matrix-vector inner-product. Here, the weight matrix is equally divided into p groups of r rows ($n = p \times r$), each processor is assigned a group of r rows. The entire input vector is broadcast to all processors so that they can perform a matrix-vector inner-product on those rows of the weight matrix that are assigned to that processor.

The second part of the algorithm involves the k-WTA. Each processor computes a k-WTA on the portion of the output vector for which it is responsible (those k nodes allocated to it). These $(p-1)$ k-element vectors are then sent to processor 0 (the root process), which performs another k-WTA over the entire vector. Since pk is usually much smaller than the vector dimension, n, this approach is reasonably efficient and

guarantees the final k-WTA is correct. Also, the k-WTA is only performed after all the inner-products are complete, which generally require a much longer time.

The cluster gets its performance by dividing the computation into p equal parts that can be computed concurrently. This also increases the effective memory bandwidth by p. Ideally the speedup would be p. Unfortunately, the speedup is smaller than p, since parallelism creates additional overhead. In our simulator this overhead has two components: 1) the broadcast of the input vector to all the processes, and 2) the broadcast of the k local winners to the root processor and the computation of the final k-WTA.

To know the relationship between the speedup and the number of processors, p, we performed experiments of implementing the WPNAM model on the PC cluster. For these experiments, we used a small PC cluster environment with 8 DELL Dimension L1000R computers. Each node includes an Intel® PIII 1.0 GHz CPU, 512 MB PC133 memory. The OS is RedHat® Linux 7.1. The compiler is G++ 2.96.

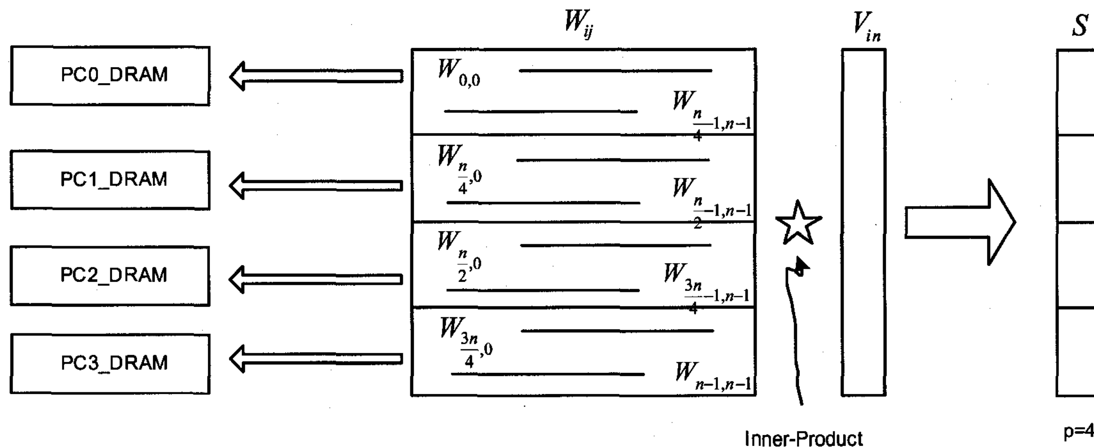


Figure 3-6: Illustration of the weight matrix distribution for the PC cluster (4 PCs example).

Table 3-6: PC cluster (8 processors) simulation results.

Number of PCs	Vector Size, n	Knodes update per second	Normalize nodes update rate by 2-PC number
2	4K	410	1.00
4	4K	683	0.83
8	4K	1024	0.63
2	8K	390	1.00
4	8K	745	0.95
8	8K	1170	0.75
2	16K	356	1.00
4	16K	683	0.96
8	16K	1170	0.82
2	32K	312	1.00
4	32K	607	0.97
8	32K	1130	0.91
2	64K	289	1.00
4	64K	560	0.97
8	64K	1057	0.92

A summary of the results is shown in Table 3-6. The number of training vectors, M, is 0.69 of the vector size, n. The number of active nodes, k, is $\log_2 n$. The most

important result is that for the larger vectors, increasing parallelism does not add significant overhead. For the 64 Knode network, only about 8% of the performance is lost in going from two PCs to eight PCs. In these experiments we are only 8% short of a linear performance improvement of four times.

3.4. PERFORMANCE COMPARISON BETWEEN PC AND PC CLUSTER

Figure 3-7 shows the node update rate for the P4 and the PIII cluster. The x-axis is the node size. The y-axis is the node update rate (Knodes/sec). The square-dotted curve is the cluster implementation. The diamond-dotted curve is the simulation result from the P4 for full-matrix vector.

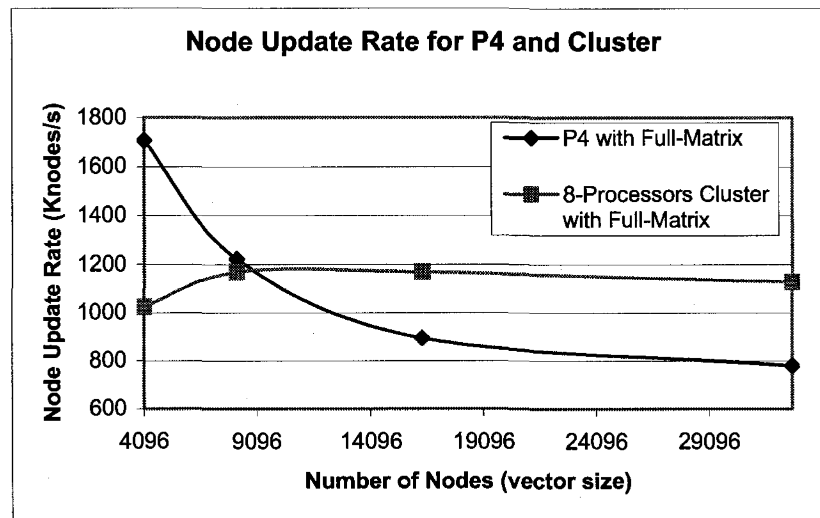


Figure 3-7: Node update rate for the P4 and the PIII cluster.

From Figure 3-7, we can see that the PC cluster has better performance for the node update rate when the vector size grows beyond 10 K. However, the entire PIII cluster

implementation did not do significantly better than the P4 implementation. This is because the computers in the PIII cluster are much slower than the P4, and PC133 memory in the PIII cluster has much lower bandwidth than the Rambus memory in the P4.

4. DESIGN WITH RECONFIGURABLE ARCHITECTURES

4.1. INTRODUCTION

FPGA is a useful hardware platform for implementing certain kinds of computationally intensive algorithms. The use of FPGAs has become a trend not only in the artificial neural network field, but also in fields such as computer graphics [116], computer vision [117], speech recognition [118], bioinformatics and computational biology [119], financial analysis [120], and physics [121]. In the recent seven years, FPGAs have benefited from Moore's law with steadily reduced pricing and increased capacity and speed. Consequently, the performance (speed) gap between FPGAs and microprocessors has been shrinking dramatically. People are not only using FPGAs as pre-silicon prototypes, but also in more and more situations as the final component listed in the bill of materials, due to shorter development time, shorter product life time, and field reconfiguration.

Generally speaking, with our data-parallel, low precision AM model, an FPGA should have a competitive performance/price ratio when compared to a PC and PC cluster due to more parallel computing capability of the FPGA, and the specialized design of FPGA implementations over the general-purpose software implementation of the PC and PC cluster. Therefore we believe that an FPGA implementation of the AM model should have a better performance/price over the PC and PC cluster, without considering the flexibility of mapping and programming AM onto FPGA or PC platforms. As a part of the FPGA implementation exercise, we introduce the EVS

(Enhanced Vision System) application with the WPNAM model and how to implement the WPNAM model onto an existing Xilinx development board (D2) [43]. We then study the potential for implementing the WPNAM model onto a virtually designed FPGA board Relogix, which is optimized for maximum external memory capacity and speed, and analyze the performance of such design.

4.2. A CASE STUDY OF FPGA IMPLEMENTATION

We first introduce the FPGA implementation of the AM model for a real application. This application is an association engine[4] in the EVS for commercial aircraft [43], which has been developed in collaboration with Max-Viz Inc. A pilot guidance system [122, 123], which makes use of fused, visible-band, infrared (short wave and long wave) and radar imaging sensors, provides guidance to pilots when landing in low visibility conditions. Short wave and long wave infrared (SWIR and LWIR) are suitable for night operation whereas radar is useful in foggy conditions. However, since these sensors have differential sensitivities, their image characteristics can be very different and even opposite in some cases. Partial polarity reversal for local contrast, for instance, is very common for visible-band and infrared images. Also, features reported in one sensor might not be seen by the other sensors. More importantly, this application has critical time constraints and requires a maximum likelihood image fusion to be performed in real time, in the sense that the time lag between processing of each single image frame has to be small enough so that it does

[4] The author acknowledge Luk who did most of feature extraction and MATLAB prototype work.

not cause any visual discrepancy. All these requirements pose a big challenge to the image fusion task.

Figure 4-1 shows the EVS association engine, which uses the WPNAM model we have discussed in Chapter 2. The EVS takes these various images and fuses them into a WPNAM model, which generates the most-likely object image and projects this image (video) onto a Head-Up Display (HUD) in front of the pilot [43].

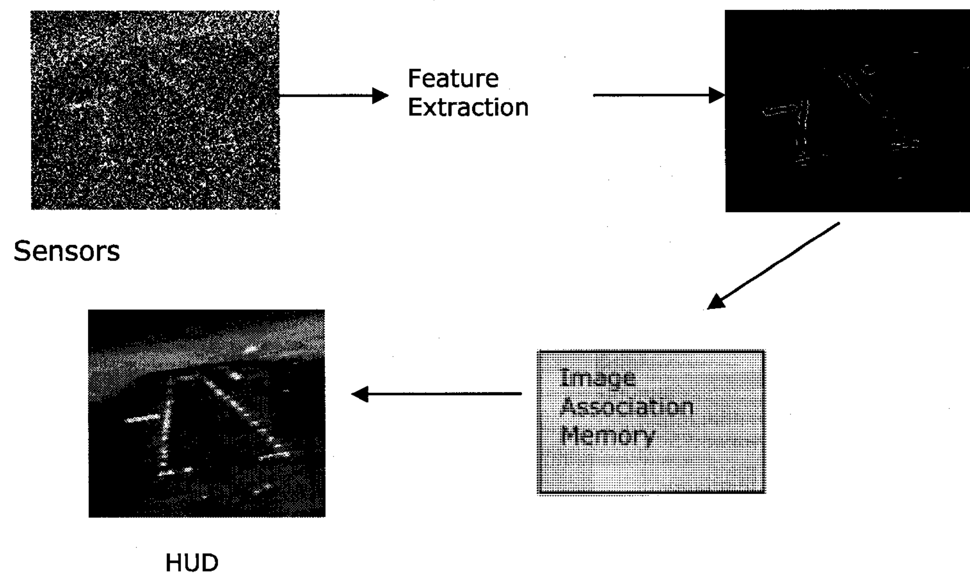


Figure 4-1: EVS associative memory application.

In this system, simple visual processing is used on each image to find key elements of the background, generally the edges of the runway. We utilize feature extraction based on the model of the early visual processing of primate visual cortex, V1 [124, 125]. We have followed Olshausen's approach [124] to approximating V1 function, making

use of the receptive fields of V1 simple cells. This biologically-inspired feature extraction is invariant over subtle shifts in feature position and size [126, 127].

Since these features tend to be consistent across all images from the various sensors, the feature vectors are just added together, then thresholding (sparsification) is performed to reduce the number of features, creating a sparse representation of the strongest features. This cumulative feature vector is then input to an associative memory that returns the features of a stored image.

Association involves storing mappings of specific input representations to specific output representations. When an obscured, noisy or incomplete input image is presented to the memory, it will recall the best match in its database. For auto-association, the input representations are the same as that of the output and allows for easy recursion (using the previous output for the next input). We used an associative memory based on the WPNAM model from equations (2.2), (2.3), and (2.4). This network was trained by presenting the visual approaches to the field in clear visual conditions [41].

4.2.1. FPGA Implementation

Since the associative memory model in the EVS system has a massively parallel computing architecture, currently a traditional computer such as the Intel's Pentium 4 cannot execute the model efficiently [15]. The aircraft also requires a small, light, low power implementation. As a result, we need other hardware platforms to accelerate the performance. The system's computational performance is not only determined by the

hardware platform, but also by the system parameters. These parameters include the image size, the number of active nodes, k (via the k-WTA), and the size of the feature vectors. As a part of the work reported here, we implemented the Palm associative network on a Xilinx FPGA with dedicated SRAM. Since the Palm network is memory-access intensive, we were investigating whether the FPGA could be a reasonable alternative to the PC platform when the memory bandwidth of the FPGA board is high enough to satisfy the requirements of the application.

4.2.1.1. System description

Since a number of imaging applications have been implemented on FPGAs, we can rely on previous efforts to understand the cost-performance trade-offs of implementing the feature extraction. Consequently, we have ported only the Palm associative network to the FPGA board. All other algorithms of the EVS system are performed by a host PC. As shown in Figure 4-2, the hardware platform for the EVS system consists of a host PC and an FPGA board with dedicated SRAM. The PC is a DELL Dimension 8100, with Pentium 4 1.8 GHz CPU, Intel 850 chipset, 1024 MB RDRAM memory, 8 KB Data / 12 KB Instruction L1 cache, 256 KB L2 cache, and a 400 MHz front side bus. The EVS system is implemented in MATLAB® 6.5. The operating system was Windows XP Professional. The FPGA is a Xilinx XC2S200 -6. The FPGA development board is the Digilab D2 from Digilent Inc. The external SRAM consists of two 512K Bytes of 15ns SRAMs from Cypress (CY7C1049CV 33-15VC).

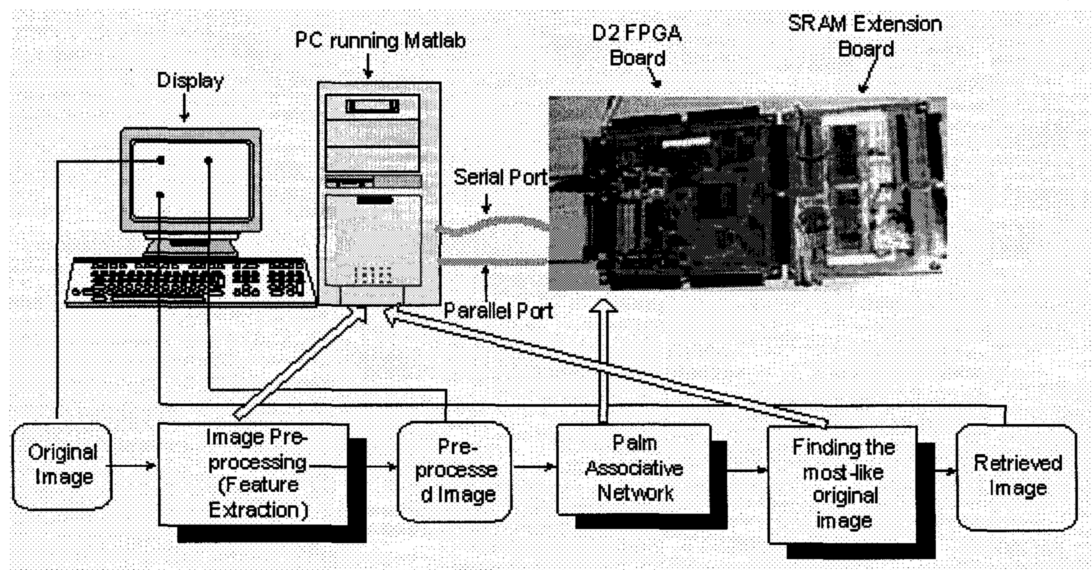


Figure 4-2: System organization and data flow for the EVS system with the WPNAM algorithm implemented on the D2 FPGA board.

4.2.1.2. FPGA implementation details

We added Gaussian noise to the original, clean, video images. The size of each frame of the video is 128×128 pixels with 8 bits of grayscale resolution per pixel. These noisy images are used to test the system. The PC begins pre-processing each frame of the test video by extracting the features and creating a sparse vector of the features. The PC then transmits this feature vector to the FPGA, which implements the associative memory algorithm. The first step for the FPGA is to perform an inner-product with the weight matrix stored in the SRAM, creating the result sum vector. A vector-vector inner product is performed on the input vector with each row of the weight matrix, generating one element of the result sum vector at a time.

The resulting sum vector is stored in the FPGA with k 24-bit registers, where k is the number of active nodes in the output vector. When the $(k+1)^{th}$ node's sum arrives, it is

compared with the smallest value, if it is smaller than or equal to that value, it is discarded. Otherwise, it is compared with the remaining values until a node is found whose value is greater than the value of the $(k+1)^{th}$ node (the node output that is currently being evaluated). Then the FPGA will insert this new value (and its index) into the list of active nodes. This merging process implements the Palm *k*-WTA function. After the *k* largest sums for a single vector-matrix inner product have been identified, the FPGA sends the sparse result vector (where the active nodes of the result vector are represented by the row-indices in the *k* 24-bit registers) back to the PC.

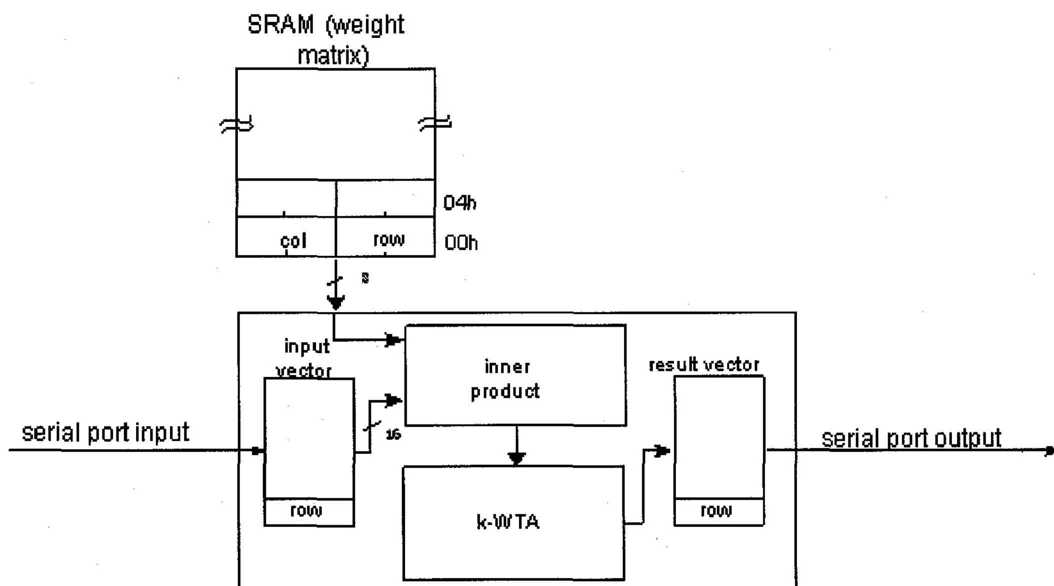


Figure 4-3: FPGA functional blocks - D2 FPGA board implementation.

Figure 4-3 shows the FPGA functional blocks and the weight matrix organization in the SRAM. The SRAM stores the weight matrix with sparse representation with the

row and column indices of the active weight bits. The weight matrix is stored in the SRAM with the data structure as in Figure 4-4. For one specific weight bit, we need four reads to get its row and column indices in the WPNAM. The FPGA stores the input vector in the “input vector” unit. The “input vector” unit contains the non-zero row information of the input vector. The FPGA reads in weight matrix with the same row index, searches the corresponding column numbers inside the “input vector”, then accumulates the value and writes to the “result vector” unit. This operation is done in the “inner-product” unit. The “*k*-WTA” unit will sort and insert the incoming result from the “inner product” unit to the “result vector” unit. The data structure of the “result vector” unit is shown in Figure 4-5.

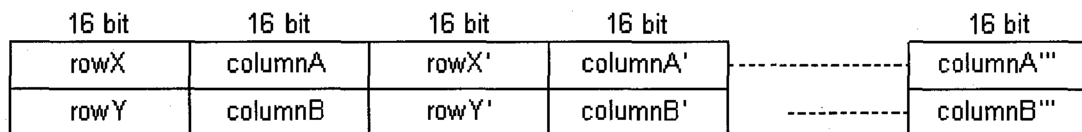


Figure 4-4: Weight matrix data structure.

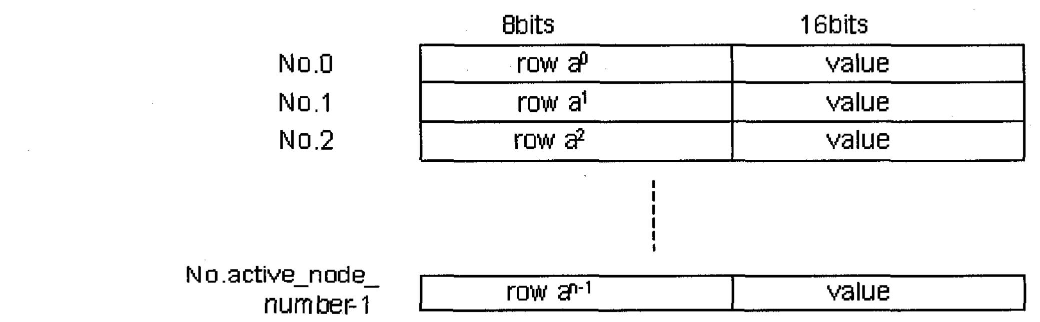


Figure 4-5: The data structure in the “result vector” unit.

4.2.2. SIMULATION RESULTS

In the preliminary system described in this dissertation, we used a database of 86 images. The average number of iterations required to obtain a stable output was about three. The accuracy of correct retrieval using this final stable output is 80% for the noise level as shown in Figure 4-1. On the PC side, the feature extraction and post-processing was implemented in MATLAB®. Figure 4-1 also shows a snapshot of those images in different stages during simulation. It can be seen that even though the input image is highly corrupted by Gaussian noise, the feature image obtained after the pre-processing step still captures fairly well the edges of the runway. After being processed by the associative memory, a clean database image is retrieved.

The accuracy of correct retrieval is not 100% as there is interference among the training vectors of images that are very close to one another in feature vector space. This happens with images that represent a fraction of a second difference in time during approach. Although these "nearly identical" images would appear as a mild flicker to the pilot, we are looking at ways to separate the images more in the feature vector space, such as by adding some smoothed temporal difference information to the feature vectors.

Table 4-1: Performance comparison of the WPNAM implementations on PC and D2 FPGA board.
"Time" denotes the time of 24 frames of video getting through the WPNAM network.

	Time (sec)	Power dissipation (Watt)
WPNAM on Pentium 4	0.43	64
WPNAM on D2	0.81	1.25

From Table 4-1, we can see that the computation time for the associative memory on the FPGA is twice as compared on the PC. However, the power dissipation of Pentium 4 is 51 times greater than that of the FPGA. The reason why the FPGA version is slower than the PC version is that we have not yet optimized the FPGA implementation with techniques such as pipelining and the use of a full binary matrix. Furthermore, the FPGA XC2S200 that we used was a very low-end but easily available FPGA at that time. It operates at a much lower frequency (50MHz) than the recent FPGA chips. And, the time penalty by the MATLAB® RS-232 serial port interface also detracts significantly from the performance of Palm network execution by the FPGA. However, even with these limitations, the FPGA still achieves a comparable performance to the Pentium 4. By exploiting parallelism in the FPGA with pipelining, preliminary analyses show that an FPGA based EVS system will operate within the real time constraints of this application [41, 43].

For example, ignoring the time penalty caused by the MATLAB® RS-232 serial port function, and replacing the SRAM with a faster SDRAM (at the mild cost compensation for power and silicon), which has 1.6 GB/s memory bandwidth, we found that the FPGA version has the potential to be at least twice as fast, as measured by node update rate, as the PC version. Figure 4-6 shows the analytic performance comparison of the Pentium 4 and a more highly optimized FPGA implementation [15].

In addition to matching desktop performance, the FPGA has a significant power usage advantage. Our FPGA implementation of the Palm network shows the feasibility of implementing associative networks on FPGAs, and provides important data on how

best to implement our application in an FPGA. Due to the increased parallelism, we expect the V1 implementation to be even more efficient.

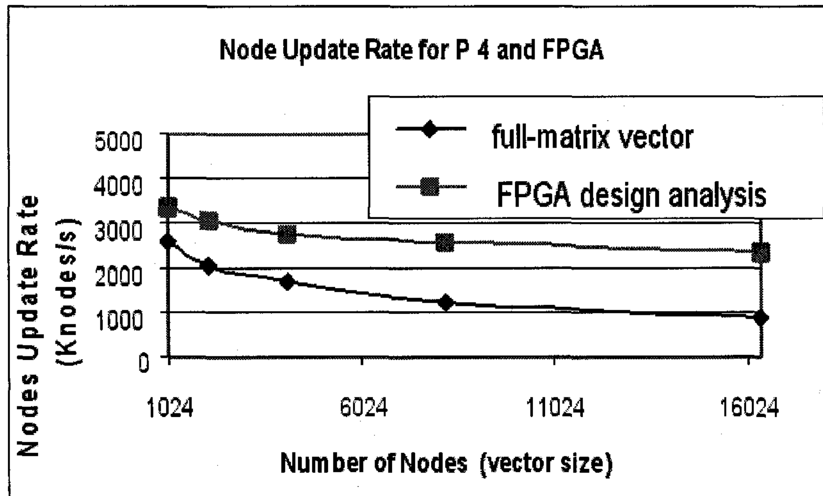


Figure 4-6: Node update rate (node outputs computed per second) for P4 and FPGA versus the vector dimension, *n*.
This is an analytic result of what is possible with higher FPGA memory and FPGA/PC IO bandwidth. The square-dotted curve is the node update rate for the FPGA implementation. The diamond-dotted curve is the node update rate for the P4 implementation with the weight matrix in full binary representation (for each weight bit, it is either 1 or 0).

4.3. IMPLEMENTATION ANALYSIS WITH THE RELOGIX FPGA BOARD

The D2 FPGA implementation of WPNAM network not only shows the feasibility of implementing the HMM model on an FPGA, but also provides important clues on how to improve FPGA architecture to execute the WPNAM algorithm for the EVS system more efficiently. For this reason it is useful for us to study an FPGA implementation in a system that has more highly optimized off-chip memory bandwidth. For this analysis, we propose an optimal FPGA board, called the Relogix board, to analyze the

performance of implementing the WPNAM model on a high performance FPGA platform. The Relogix board is a memory-intensive reconfigurable board with dedicated SDRAM. Each board has 1.6 GB/sec of memory bandwidth available for memory-intensive applications. The architecture allows multiple boards to be connected together to create larger, parallel configurations. Each accelerator is accessed by application software running on the host via an IDE/FireWire bridge. The objective of this board is to take advantage of inexpensive PC memory, FPGAs, and high speed interfaces to minimize the performance/price ratio in a reconfigurable accelerator. Figure 4-7 shows the basic board layout.

FPGAs can provide significant performance improvement for highly data parallel, lower precision applications. However, some of the performance is compromised when access external storage is required. Given the state of the art of FPGAs and their on-chip memory capacities, emulating AMs will require such external memory. The Relogix exercise[5] has been done to determine what performance is possible when the latest in DRAM memories and interfaces are paired with state of the art FPGAs.

4.3.1. FPGA System Architecture

The PQ208 pinout form factor accepts any size Xilinx Spartan-IIE chip. The Spartan-IIE, XC2S50E, 100E, 150E, 200E, or 300E, may be installed at assembly. All Spartan-

[5] "Relogix" comes from the name of a small company that intended to produce this board for the EVS among other things, but then lost its funding. Also, it is important to remember that the numbers and configurations used here were the latest in state of the art when this work was done, but now, as this dissertation is being written they are already out of date.

IIE chips have the same pinout, 142 general-purpose I/Os, and four global clock inputs, differing only in amount of logic and memory cells. By using smaller FPGA, we can stay in a QFP for the early boards, since the QFP package reduces costs. Because our algorithms make such extensive use of the SDRAM, there is a point of diminishing returns as one goes to larger FPGA.

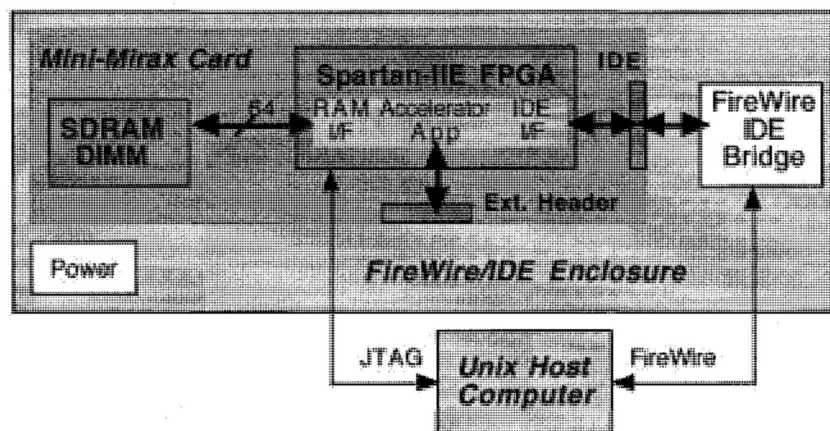


Figure 4-7 The basic Relogix FPGA board.

Connected to each FPGA is a single DIMM socket, which supports a 10 nsec DDR 64-bit pathway (two words are delivered in every 10 nsec clock cycle) into the FPGA. With today's memory technology this allows up to 512 MB (at the time this work was investigated) of memory to be directly connected to each FPGA. Finally there is an IDE to FireWire chip that creates the external FireWire interface for the board. An external header uses 20 uncommitted FPGA I/Os that can be connected to other Relogix boards or external devices such as sensors. A JTAG daisy chain header connects the FPGA to a Xilinx interface cable for programming with Xilinx software.

Many Relogix boards can be used in one JTAG loop. One or more Relogix boards are installed in one or more FireWire/IDE enclosures, which provide an IDE to FireWire bridge, power and cooling, and a connection to a Linux, OS-X or Solaris host.

The system will obviously be most cost-effective with larger numbers of parallel, low-precision operations, a configuration where typically there will be more computation per bit fetched from the SDRAM. This is also a perfect match to our WPNAM implementations. The board will track the latest memory density (commercial SDRAM) and bandwidth, following Moore's Law at the same rate as a general purpose microprocessor.

We do not yet have a functioning board, so we cannot provide exact performance measurements. We present an analysis of the expected performance of the Relogix board on the same WPNAM algorithm used for the PC. The resulting information has never the less been very useful in helping us form our AM implementation architecture methodology.

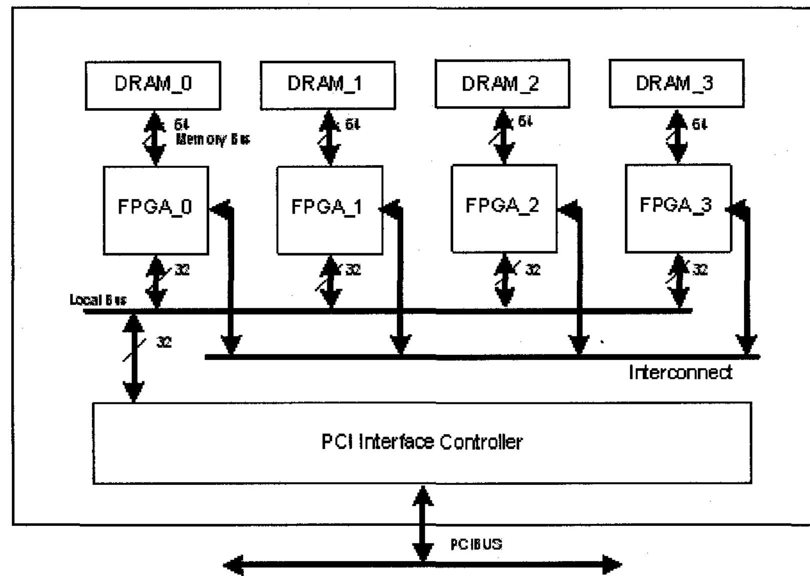


Figure 4-8: FPGA board components block diagram. For the performance analysis, the FPGA_x is Xilinx XC2V1000–5, the DRAM_x is a 64-bit 133 MHz DDR SDRAM DIMM.

The primary reason for building this board is that we want to be able to place the SDRAM, in the form of a DIMM immediately next to a modest sized FPGA. The goal is to leverage the maximum bandwidth of the largest commercially available memory. Because of the availability of inexpensive FPGAs and high capacity memory, this should provide a significant performance/price ratio.

The DRAM stores the weights. The weights are distributed evenly in the 4 DRAMs as shown below. The weight vectors are stored as 32 single weight bits per word. The input vector is assumed to be stored within the FPGA, also in binary, full-matrix form.

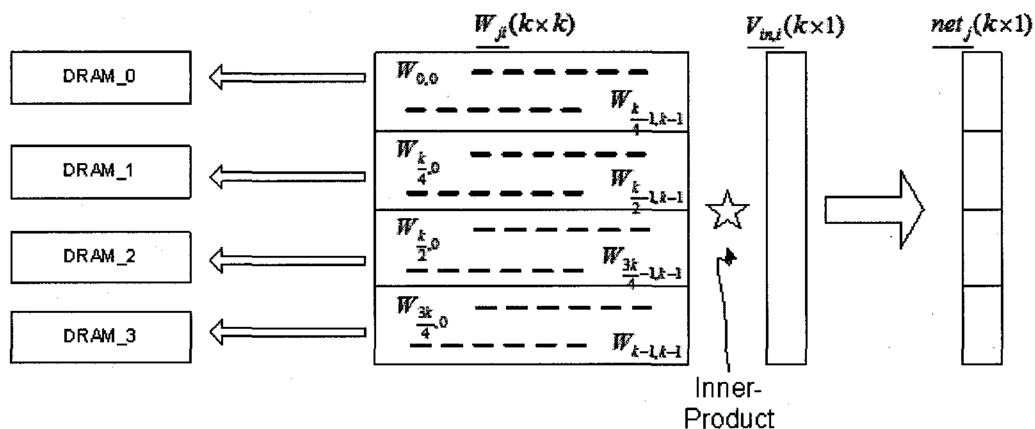


Figure 4-9: Weights distribution in the DRAMs.

The implementation of the WPNAM model in an FPGA is quite similar to that of the PC and PC cluster, in the sense that the weight values are stored in the external SDRAM with the inner-product and *k*-WTA performed by the FPGA. Figure 4-10 shows the block diagram of the Relogix FPGA. The "IDE Interface" connects the Relogix FPGA with the host through the "IDE Bus". The host sends the test vector to the Relogix FPGA through the "IDE Bus". The "Memory Bus" connects the "SDRAM Memory Bus Interface" in the Relogix FPGA and the external SDRAMs on the Relogix board. The "Interconnect" connects different Relogix boards. The "Inner-Product Unit" executes the inner-product operation of the WPNAM model. The Relogix board uses the column-wise inner-product method and stores the weight matrix in the full binary mode. However, the D2 board only stores the existing weight bits' row and column indices. The reason we used sparse weight representations (row and column indices) in the D2 board was because the huge amount of memory requirement (256 Mbits) was much more than D2 board's external SRAM (4 Mbits)

can provide. The "*k*-WTA Unit" sorts the inner-product result vector, keeps the intermediate values in the "RAM" unit, generates the result vector indices, and sends them back to the host through the "IDE Interface" unit. Because the Relogix board implementation is simple, and there are no complex entities such as multi-level caches, the results presented in the next section are a reasonably accurate representation of the performance we expect from the actual board.

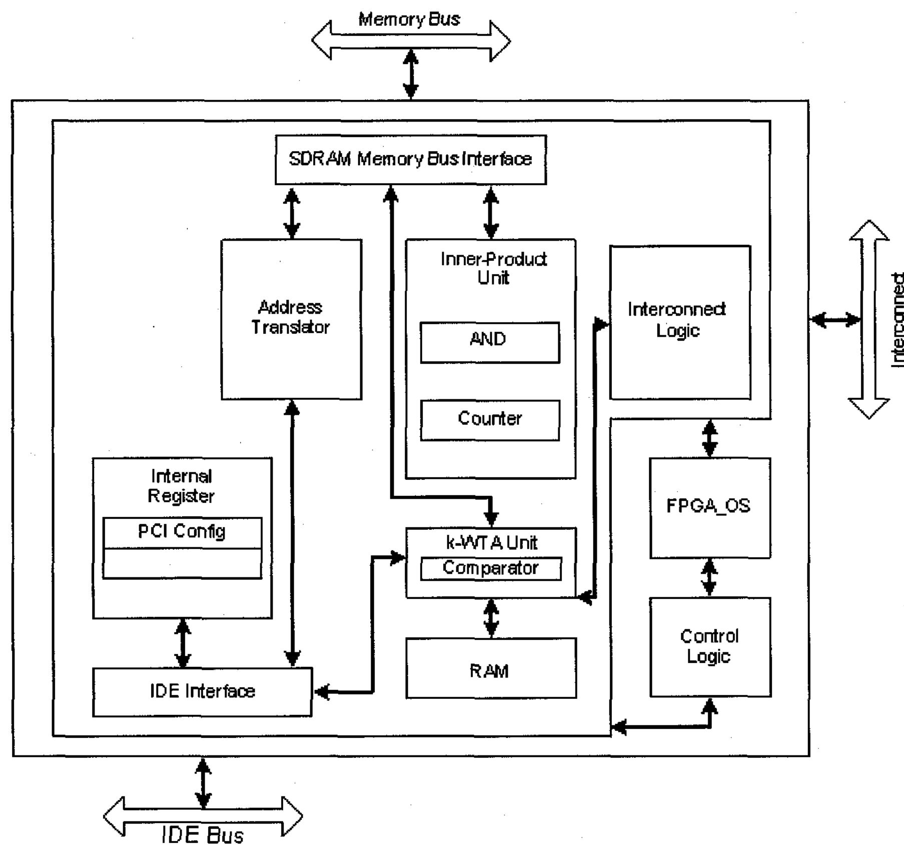


Figure 4-10: Relogix FPGA functional block diagram.

Inner-product operation: The input vector $V_{in,i}$'s addresses (row number and offset) are transferred to the Local Bus Interface. The address information is translated to the

corresponding weight matrix address by the Address Translator. The DRAM outputs the data requested by the FPGA through the DRAM Memory Bus Interface. The input vector $V_{\text{in,i}}$ and its corresponding weight will do the AND_SUM operation in the Inner-Product Unit. It also sends back the node value net_j to the local memory through the DRAM Memory Bus Interface.

***k*-WTA operation:** After executing the inner-product for the input vector $V_{\text{in,i}}$ and W_{ji}, the weights stored in the local memory, each FPGA performs the *k*-WTA for its nodes. It is assumed that the result vector remains inside the FPGA. The *k*-WTA Unit sorts the local node values and finds the *k* nodes with the largest values. These nodes retain their values and all other nodes are changed to 0s.

After the local *k*-WTA is complete, the other three FPGAs will transfer their *k* winning nodes to the master FPGA to do another cycle of *k*-WTA. During this transfer, since *k* is considerably small, the vectors are converted to a sparse representation. After doing one more *k*-WTA sort, the master FPGA transfers the output vector $V_{out,j}$ (address and data) to the CPU through the PCI Bus.

Local bus: The input vector transferred from the CPU to each of the FPGAs includes the row number and offset of $V_{\text{in,i}}$. The output vector transferred from the master FPGA_0 to the CPU also includes the row number and offset of $V_{out,j}$.

Interconnect: It includes the control logic and data/address bus. The three slave FPGAs need to transfer their k winner nodes to the master FPGA to do another k-WTA operation.

PCI bus: The host PC uses PCI bus to transfer input vectors to the FPGA board and receive the output vectors back from the FPGA board.

There are a number of assumptions about the WPNAM implementation on the Relogix board:

- A single FPGA/SDRAM board is used for analysis, even though the Relogix system supports multiple FPGA configurations. Likewise, it is likely that the communication overhead for connecting multiple boards, as a percentage of the total computation time, will be smaller than with multiple PCs in a cluster.
- Only a full binary weight matrix implementation is assumed where each bit is a binary weight. The weight matrix is stored by columns. One 64-bit word of a column corresponding to the non-zero element in the test vector is fetched from the SDRAM at a time. The fetching could be sequential so that it can save the latency time of the SDRAM, and be close to the maximum memory bandwidth.
- The input test vectors are sparsely encoded and the time transfer of data to and from the PC host is ignored. Each vector is assumed to have single bit precision and consists of a list of indices of the weight bits in the matrix row.

When a test vector is processed, this list of indices is brought from the PC into the FPGA.

- The inner-product operation is column-wise inner-product. The *k*-WTA is computed in a batch mode. That is, after all inner-product elements are generated they are sorted to get the *k* largest elements.

- The final output vector is sparse and its indices are written to the PC host thus completing the computation of a single vector. This time is also ignored.

4.3.2. Performance Analysis

4.3.2.1. Inner-product

Computation in the FPGA: The Counter counts how many bits in the AND operation's result are 1. The results are accumulated to get the final inner-product operation result for a node. The Counter has the potential of being the bottleneck in the inner-product operation. But bits sums can be accumulated in a parallel pipelined manner for fast execution. The FPGA can work at up to 400MHz.

Memory bus bandwidth: The DDR memory interface in the Xilinx Virtex II has 2.1GB/s bandwidth with 133MHz system frequency, 64-bit data width at both clock edges. The average memory access time for 8 Bytes is 3.8ns. To calculate the time for memory accessing, we use the following equation:

Time for memory accessing (sec) = $(\text{vector_size} / 4 \times \text{vector_size}) / 64 \times 3.8 / 10^9$

PCI bus bandwidth: The PCI specification v2.2 implementation enables burst transfers of up to 132 MB/s.

Time for feeding test vectors (sec) = number_test_vectors × active_nodes × 2 / 8 / 132M

Table 4-2 shows the time for the inner-product operation. From this table, we notice the bottleneck is at the time for memory accessing. And the time for memory access increases exponentially with the vector size.

Table 4-2: The time for the inner-product operation for the full-weight matrix representation. The weights are obtained from the SDRAM, the test vector is stored inside the FPGA.

	Time for memory accessing (sec)	Time for feeding test vectors (sec)	Time for inner-product (sec)
Conf1	1.6×10^{-5}	1.9×10^{-6}	1.8×10^{-5}
Conf2	6.2×10^{-5}	2.1×10^{-6}	6.4×10^{-5}
Conf3	2.5×10^{-4}	2.3×10^{-6}	2.5×10^{-4}
Conf4	1.0×10^{-3}	2.5×10^{-6}	1.0×10^{-3}
Conf5	4.0×10^{-2}	2.7×10^{-6}	4.0×10^{-2}

4.3.2.2. *k*-WTA

Time for sorting in the FPGAs: To determine the *k*-largest values from the inner-product results, we have to do a sort. As discussed above, this operation is performed twice, once by each FPGA over the inner-product results, then again by the master FPGA to get the final result. Currently the sorting process is implemented with a single comparator, we are exploring faster parallel sort mechanisms, but do not have results for those at this time.

For each clock, the 'Comparator' unit compares two values at a time. If the first one is smaller than the second one, then the 'Comparator' unit will exchange the two values, and write back to the RAM. For the worst case, the inner-product results are sorted reversed. If there are N nodes (vector_size) in the results, we need to access the node's data (including value and address offset) M times in the master FPGA (here we calculate the average value):

$$M = \frac{1}{2}\left(\sum_{1}^{\frac{N}{4}-1} i + \sum_{1}^{4k-1} i\right) = \frac{N(N-4)}{64} + k(4k-1)$$

Each time, the FPGA will use one clock cycle to do a 'read', 'compare', and 'write back' by the means of pipelining.

Time for sorting (sec) $= M \times 2.5/10^9$

Time for transferring data via the local interconnect: After the three slave FPGAs have executed the sorting algorithm on their local nodes, they will transfer the results to the main FPGA through the interconnect. The interconnect operates at 400MHz.

Time for data transferring on interconnect (sec) = active nodes × 3/4 × 2 / 400M

Time for moving the data from the FPGA to host PC: This is equal to the time for feeding test vectors.

Table 4-3 shows the time for k-WTA and the total time for the inner-product and k-WTA. Here we consider the three steps: sorting, data transferring on the local interconnect, and then moving the data from the FPGA back to the CPU as sequential

processes. The inner-product and *k*-WTA processes are also sequential. From the table, we can see, that the sorting in the FPGA occupied most of the time for *k*-WTA. And the time for *k*-WTA is a little more than the time for inner-product.

Table 4-3: Time for *k*-WTA and the total time for the inner-product and *k*-WTA.

	Time for sorting in the FPGA (sec)	Time for data transferring on interconnect (sec)	Time for putting back data from FPGA to CPU (sec)	Time for *k*-WTA (sec)	Total time for inner-product and *k*-WTA (sec)
Conf1	$4.2x10^{-5}$	$3.8x10^{-8}$	$1.9x10^{-6}$	$4.4x10^{-5}$	$6.2x10^{-5}$
Conf2	$1.6x10^{-4}$	$4.1x10^{-8}$	$2.1x10^{-6}$	$1.6x10^{-4}$	$2.2x10^{-4}$
Conf3	$6.6x10^{-4}$	$4.5x10^{-8}$	$2.3x10^{-6}$	$6.6x10^{-4}$	$9.1x10^{-4}$
Conf4	$2.6x10^{-3}$	$4.9x10^{-8}$	$2.5x10^{-6}$	$2.6x10^{-3}$	$3.6x10^{-3}$
Conf5	$1.0x10^{-2}$	$5.3x10^{-8}$	$2.7x10^{-6}$	$1.0x10^{-2}$	$1.4x10^{-2}$

4.3.3. PERFORMANCE COMPARISON BETWEEN THE P4 AND THE RELOGIX FPGA BOARD

Figure 4-11 shows that the Relogix FPGA implementation has advantages over the P4 implementation. That is mostly from the help of the parallel and hardware-specific implementation of the FPGA. Although the performance of the Relogix FPGA implementation is better than the P4 implementation, as the nodes size increases, the performance of Relogix FPGA implementation drops, which is mostly due to the time required to sort the result vectors. If we could use multiple 1-WTA regions (or hypercolumn AM from Lansner [67]), the time spent on sorting can be greatly reduced, although this hypercolumn AM also changes the network behavior.

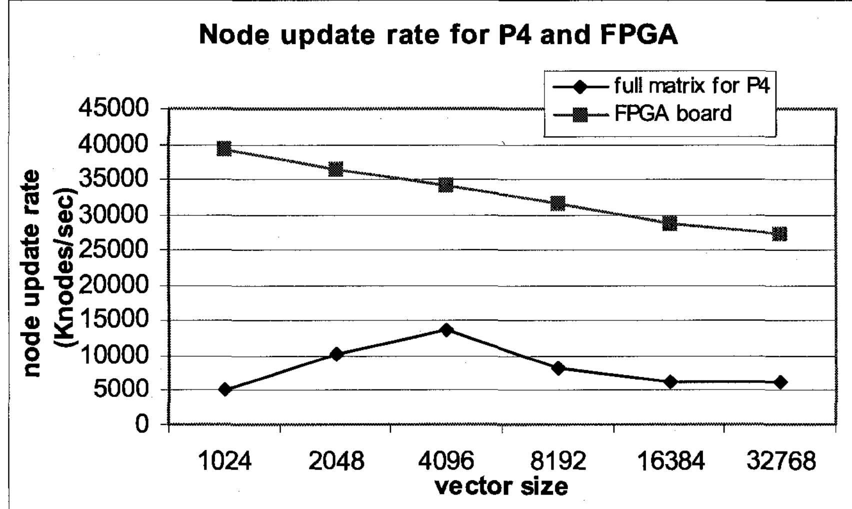


Figure 4-11: Node update rate for P4 and Relogix FPGA.
The inner-product operation is in the column-wise manner. The x-axis is the node size. The y-axis is the node update rate (Knodes/sec). The square-dotted curve is the Relogix FPGA implementation. The diamond-dotted curve is the simulation result from the P4 for full-matrix vector.

As we suspected, the memory bandwidth is the primary limitation to higher performance for the Pentium 4 implementation, where inner-product requires most of the time in the retrieval phase of the WPNAM network. For the Relogix FPGA, the *k*-WTA algorithm is the major performance limitation, if the *k*-WTA were executed at a higher speed, then like the Pentium 4, the memory bandwidth would also limit the performance. Therefore, increasing memory bandwidth can improve the performance of the WPNAM network implementation on P4 and inner-product algorithm on the Relogix FPGA implementation. Increasing the FPGA performance can improve the performance of *k*-WTA implementation on the FPGA.

4.4. CONCLUSION

Although still preliminary, we believe that the results given here indicate that an FPGA acceleration card based on the tight integration of an FPGA chip and commercial SDRAM creates an effective association network hardware emulation system with a very competitive performance/price. Although for this chapter we have not extended our results to include an analysis of multiple FPGA systems, we believe that such systems will have larger incremental performance than PC clusters built from commercial boxes, since we believe that memory bandwidth and *k*-WTA scaling will improve performance faster than inter-FPGA communication requirements will degrade it.

From the study of implementing the WPNAM models on PC and FPGA, we could see for AM algorithms with the SIMD computational architecture, a traditional desktop PC, in spite of high clock rates, caching, and deep pipelines, does not provide the best performance/price ratio since it does not leverage the natural data parallelism in the AM model. The PC cluster could not improve the performance/price ratio due to the same reason as the PCs. The FPGA implementation has a better performance/price ratio than the desktop PC, since FPGAs can exploit the parallelism of the hardware to match the parallel computational properties of the WPNAM model. The two main operations in the WPNAM, the inner-product and the *k*-WTA, are quite common in HDM models. Any models with inner-product and *k*-WTA could use the FPGA implementation as the hardware platform with the best performance/price ratio among the choices of desktop PC, PC cluster, and FPGA. However, the memory capacity and

the memory bandwidth constraints do limit the performance of the biologically-inspired computation models such as the WPNAM model on FPGAs. And FPGAs are resource-limited for more complex AM models. We have to do trade-offs between the FPGA implementation and the precision of the more biologically-inspired models.

This methodology for investigating the hardware platform with the best performance/price ratio for one AM model could be used to determine the performance of other HDM models and other hardware platforms. With the implementation and simulation of more AM models on different hardware platforms, we can decompose the AM models into several basic operations, such as the inner-product and *k*-WTA. Investigating the major operations on different hardware platforms can help give us an idea of how to achieve the best performance/price ratio. We use the same methodology in Chapters 5 and 6 to compare the performance/price ratio of AM implementations on full-custom VLSI and nanoelectronic circuits (CMOL).

5. DESIGN WITH SPECIAL-PURPOSE ARCHITECTURES (CMOS)

5.1. INTRODUCTION

For certain compute intensive applications, ASIC (application specific integrate circuit) or even full-custom CMOS VLSI designs are used to accelerate inner-loop calculations that dominate the time of the algorithm or application. From a recent paper by Shaw *et al.* [128], they used special-purpose ASIC to accelerate simulations of molecular dynamics for computational biology. With 512 identical parallel processing nodes for the molecular dynamics computations, the parallel machine, Anton [128], achieves about three orders of magnitude improvement over the current software simulations, which shows that the special-purpose CMOS design could achieve much higher performance than the PC.

This motives us to investigate full custom CMOS implementations of the non-spiking and spiking AM models. This chapter presents a methodology and an analysis for designing CMOS architectures for the non-spiking and spiking AM models. The CMOS technology includes all digital and mixed-signal (digital and analog) CMOS circuits. In the next chapter, we will present the digital and mixed-signal CMOL architectures for non-spiking and spiking AM models, and compare the performance/price ratios for CMOS and CMOL implementations.

5.2. IMPLEMENTATION WITH DIGITAL CMOS

5.2.1. NON-SPIKING AM MODEL

For the non-spiking model analysis, we assumed four basic configurations of CMOS and CMOL implementations: all digital CMOS, mixed-signal CMOS, all digital CMOL, and mixed-signal CMOL. The primary computations in the column-processor (for each WPNAM like associative memory model) are the inner-product of input vector and weight matrix, and *k*-WTA. Figure 5-1 shows the configuration for the four basic designs. However, we will introduce CMOL and explain CMOL configurations for the non-spiking and spiking AM models in the next chapter (Chapter 6). The result analysis for CMOS implementations (from this chapter) will also be given at the end of Chapter 6, together with the CMOL implementations.

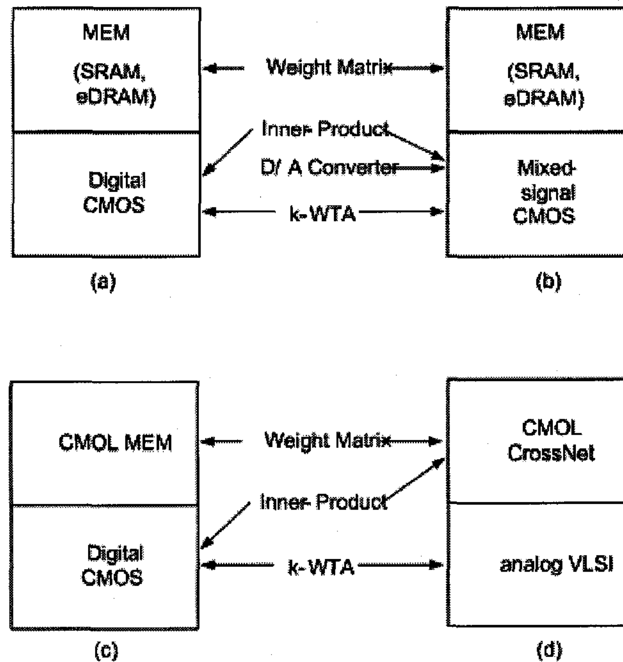


Figure 5-1: The functional partitioning of the four configurations for CMOS and CMOL implementations.
(a) Digital CMOS design; (b) mixed-signal CMOS design; (c) digital CMOL design; and (d) mixed-signal CMOL design. The different computation tasks are partitioned onto different hardware.

As illustrated in Figure 5-1 (a), the weight matrix is stored in CMOS memory (MEM), which could be implemented with SDRAM or embedded-DRAM (eDRAM [129]). The inner-product and *k*-WTA computations are performed by arithmetic logic in the digital CMOS platform. Because of the sparse activation of input vectors (on the order of $O(\log_2 N)$, where N is the network size), we only retrieve weight columns whose column indices correspond to those of the active nodes, and sum them. Thus, this column-wise inner-product saves time and power over the traditional row-wise inner-product (comparing $O(N\log_2 N)$ additions to $O(N^2)$ additions [42]).

Each column processor reads in the corresponding column weight bits from memory (SDRAM or eDRAM), adds them together (using a column-wise inner-product as discussed in Chapter 3 and [41, 42]), and sorts the inner-product results for the *k*-WTA. The output is recursively fed back to the input for auto-association. There is significant parallelism available, since each neuron, of which there are thousands in each column can be computed in parallel. The column neuron size is 16,384, and the weight bits are 4, as shown in Table 2-2. In both the digital CMOS and digital CMOL implementations of the non-spiking AM model, we chose to use 256 parallel computational units (or PNs) in each column processor (CP). In this way, each PN multiplexes 64 neurons' matrix-vector inner-product operations for the 16,384-neuron AM network. There is a global *k*-WTA operation in the CP without any multiplexing. The reason we chose to use 256 parallel PNs is because our preliminary study showed the best balance between the area of PNs and the area of memory to store the AM network weights under this configuration. In addition, it is hard for us to show the performance/price ratios for different hardware architectures in a single table, as shown in Table 6-4. In this approach we need only as many parallel memories as we have computational units, though the total amount of memory remains the same. We borrowed the hardware circuits from other groups (see Section 6.4). The details of those circuits and how we scaled the circuits' power, speed, and area to 22 nm in this work are addressed in Section 6.4.

5.2.2. SPIKING AM MODEL

The digital CMOS can also implement the spiking AM model, equations (2.5) and (2.6). Now, we present how to implement the spiking AM model in an all-digital system. The same architecture also applies to all-digital CMOL architecture for the same spiking AM model. When emulating the spiking AM models, the hardware is assumed to operate in real time [130]. Usually, an analog-circuit system has a dedicated circuit for each computation. The real time requirement sets constraints on each analog circuit. This in turn determines the signal processing rate for the analog circuits, and the power consumption in terms of response time or spiking rate. For digital circuits, computational resources are generally multiplexed. Therefore, there can be jitter noise, which needs to be minimized. One potential disadvantage of virtualization (multiplexing computational hardware) is that the more resource sharing there is, the more unpredictable processing time is, and the more jitter noise may be added to the signals. In digital systems, it is possible to keep a virtual system clock, which is updated as needed that would eliminate jitter noise. However, it adds significant complexity to the system and is not assumed here.

For the spiking model analysis, we have the same basic configurations we saw in the non-spiking case. For each design, because the computations and operations of the non-spiking and spiking AM models are different, the corresponding architectures, complexity, and underlying circuit components are also different. A few basic components, however, could be identical. Furthermore, because of the sparse

activation of the spiking AM, it is possible to leverage virtualization to build more efficient digital CMOS and CMOL designs.

In the spiking all digital, all CMOS design, we use a PN to emulate some part of the network. The virtualization (degree of multiplexing) chosen depends on the specific dynamic characteristics of the model being emulated. The column processor, as shown in Figure 5-2, consists of single or multiple PNs that perform the calculations, and a memory to store the weight values. In the HDM each column consists of some number of neurons, typically several thousand, which are fairly tightly connected with each other.

When implementing such a computation in a set of processors, the sparse activation of input spikes motivates the use of a sender-oriented multiplexed communication method to improve computational efficiency [131]. That is, the PN reads the sparse presynaptic events from the input neurons[6] – senders, computes the weighted postsynaptic potentials (2.6) for the connected output neurons according to the connection list and stored weights, and updates their membrane potentials (2.5).

Figure 5-2 shows the block diagram of each PN, with weight memory in the column processor system. Each PN time multiplexes the computations of one or more neurons.

---

[6] Because of the simple operations in the non-spiking AM model, we used node to denote the neuron in the AM network in the previous chapters. However, due to the more complex operations in the spiking AM models, we started to use neuron to denote each neuron in the spiking AM network. Also, we used processing node (PN) to denote the physical computational unit inside the physical column processor, which is used to implement the non-spiking and spiking AM algorithms.

For example, if a PN multiplexes four neurons in a 32 neuron network, the total system needs eight PNs to run in parallel. We call it a mux-4 PN system.

There are eight major operations that are performed by a spike PN:

1) *Read SE*: The column processor system has a dispenser to distribute the presynaptic events from the intra-column spike events or the AER-based inter-column communication channel to each PN, and put those events' indices and a countdown time into the PSEM (PreSynaptic Events Memory), shown in Figure 5-3. The presynaptic event is the spike from the incoming axon, as illustrated in Figure 2-8. The PN reads the presynaptic events from the PSEM and captures the event's timing information. This time is used to fetch the postsynaptic potential (PSP) from the PSP-LUT (look-up table). When the countdown time goes to zero, this event no longer affects the computation and the record can be invalidated. The PSEM could be implemented with an SRAM. The PSEM has *a* records of synaptic events, with a record width of $\log_2(index) + \log_2(time\ resolution)$.

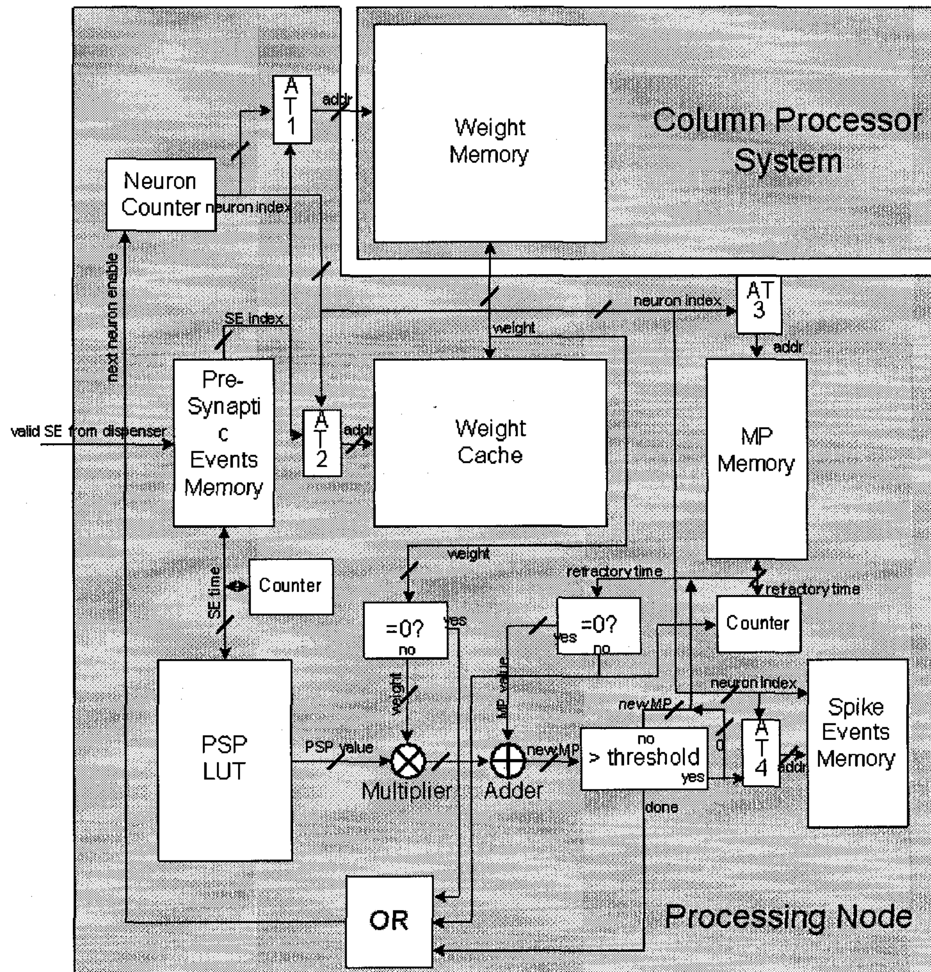


Figure 5-2: CMOS column processor system and functional blocks in a single PN for the spike-timing-dependent AM model.

Each column processor system has one weight memory for all PNs or several weight memories distributed for many PNs.

The PSP-LUT stores the postsynaptic potentials in terms of elapsed time. We could also calculate the PSP value according to (2.6). However, such a computation requires at least two dividers and two exponential arithmetic units, which consume either time or silicon area for the "Multiplier" and "Adder" in the PN as in Figure 5-2. If the look-up table has a small number of entries, it can be faster. A possible circuit for the LUT is Content-Addressable Memory (CAM) with SRAM [132].

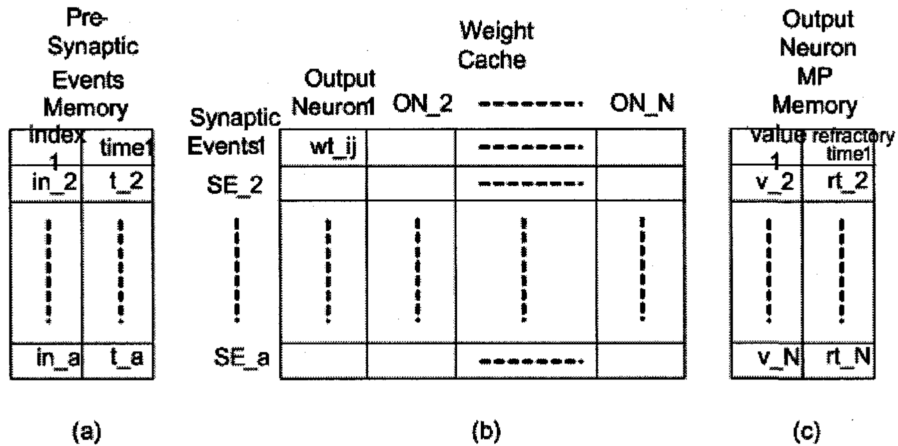


Figure 5-3: Memory architectures inside CP.
(a) The presynaptic events memory (PSEM) stores each valid event's index and time offset. (b) The weight cache stores the weights with a consecutive arrangement of the synaptic events index and the output neuron index as the row and column addresses respectively. (c) Output neuron membrane potential memory stores each output neuron's somatic membrane potential and remained refractory time.

2) *Read the weight values from weight memory*: The weight memory stores the weights with size of N^2, where N is the network size. Because this is generally the largest component of the column processor, we have assumed eDRAM technology (we assume that the eDRAM processing does not add considerable cost to the chip [129], so that it does not significantly impact cost). When the PN receives a new synaptic event, it will read the corresponding column weight data from the weight memory into the weight cache. If the event is not new or the weight information is already inside the weight cache, the PN will skip this stage.

3) *Read the weight from the weight cache*: The weight cache is implemented in SRAM and has lower latency and higher bandwidth than the weight memory. The weight cache stores at most the same number of record rows as the number of valid (i.e., active) synaptic events. The number of valid synaptic events is roughly $\log_2 N$, which reduces the capacity requirement of the weight cache as compared to the weight memory. Because of this sparse activation and the elapsed time of postsynaptic events in the PN, we can store the weight in the weight cache for the duration of the synaptic event guaranteeing that the weight is in the weight cache while the event is active, except for the first cycle of a new synaptic event. The weight cache block diagram is illustrated in Figure 5-3 (b). The size of the weight cache is $(\log_2 N) \cdot M \cdot b_w$, where b_w is the bit width of each weight. Since not all connections exist, the weight matrix could be sparse. We could store only the non-zero weights into the weight cache when $p((\log_2 N) + \log_2(M) + b_w) < b_w$, where p is the probability of a non-zero weight. A disadvantage of this sparse representation is that the non-zero weights are stored as a list, and we would need to traverse the entries in the weight cache to fetch a weight from a random request. Though not assumed here, it is possible to use CAM to store the non-zero weights.

In order to leverage the sparse connectivity, for the full representation of the weight cache (i.e., store all zero and non-zero weights according to their sequential addresses) we could read multiple weights at once instead of reading a single weight during a clock cycle, and Boolean OR them to see if the result is zero. If so, then there is no connection between those neurons and the driving synaptic event. If this value is not

zero, then we must test each connection sequentially. The multiple-weight read only works for PNs with multiplexed neurons. For non-multiplexing PNs, this multiple (i.e., more than one weight) read option is not possible.

4) *Multiply weight and PSP*: This operation uses the "Multiplier" unit, the inputs are the weight and PSP values, and the output is a weighted PSP. This assumes multi-bit weight values, since the PN does not need a multiplier for single-bit weight representations.

5) *Update neuron's somatic membrane potential (MP)*: The PN first checks if the neuron is still in the refractory period by examining whether the record in the MP memory is zero. If it is not zero, then the PN will ignore the new weighted PSP input and decrease the neuron's refractory time by a single time unit. Otherwise, the PN adds the new weighted PSP value with the neuron's last saved MP value. The structure of the MP is shown in Figure 5-3 (c).

6) *Compare the MP with threshold*: If a new MP is generated, the PN will compare the new MP with a stored threshold θ via the "> threshold" unit, which will enable a "yes" signal when the new MP $\geq \theta$, and a "no" signal otherwise.

7) *Write back MP if needed*: When there is a new MP value or new refractory time (from the Counter unit) available, the PN will write the update value into the MP memory.

8) *Write to spike event memory*: When the "> threshold" unit outputs a "yes" signal, the PN will write the neuron's index into the "Spike Events Memory", which either

goes to the column processor's dispenser directly, or to other chips via an AER transmitter.

These eight stages listed above are assumed to be pipelined reasonably well to improve the PN's performance and reduce the possibility of idle hardware. The overall performance is determined by the slowest pipe state of these eight stages. When the weight read from the weight cache is zero, the pipe stages following it will be in the idle state, which lowers the PN's computational efficiency, though it does improve the power efficiency.

In Figure 5-2, the AT units are address translators. The PN stores the weights and membrane potentials consecutively, since there is a known relation between the memory address and the stored items. Consequently, the address translators can use the current synaptic event index and the neuron index to encode the address. This simplified encoding allows us to ignore the speed and silicon area requirements for the address translators. The OR is a Boolean operator that generates a "next neuron enable" signal to the Neuron Counter unit to increment the current neuron index to the next neuron.

In the digital circuit design, the PSEM stores the presynaptic events. The size of this PSEM affects the maximum waiting time for the computation of each event. Assume there are three clock times: T_{ch}, T_{clk_sys}, and T_{clk_pn} for channel speed (intra-column communication channel or inter-column AER communication channel), column processor system clock, and PN clock, respectively. We assume synaptic events are

independent, identically distributed, and are generated as a Poisson approximation: $P(k,\lambda) = \lambda^k e^{-\lambda}/k!$, where λ is the expected spiking (or firing) rate in the channel to the column processor. As Boahen summarized [100], the average waiting time is

$$\overline{m} = \lambda / 2(1-\lambda), \tag{5.1}$$

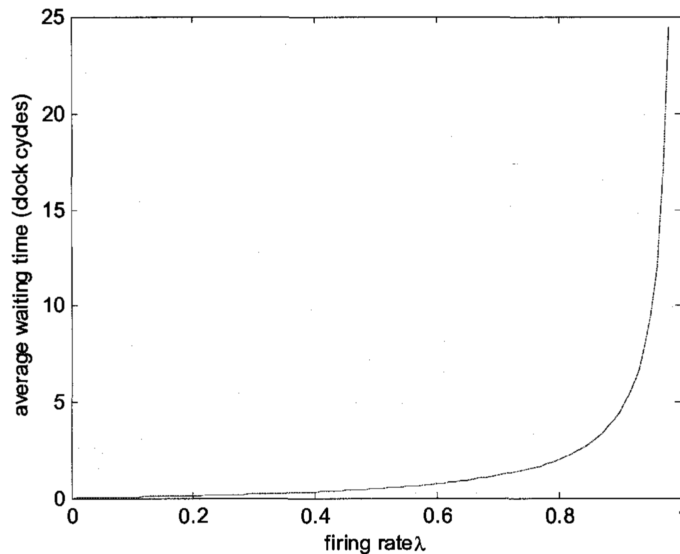


Figure 5-4: Average waiting time in terms of firing rate λ, according to (5.1).

Figure 5-4 shows the average waiting time (with unit of T_{ch}) in terms of spiking rate λ. In our system, assume there are a entries in the PSEM in each PN, and each postsynaptic event spreads over S number of T_{clk_sys} cycles, the maximum average waiting time should be aT_{ch}. That is to say, if the average waiting time is aT_{ch}, each

spike has to wait a channel cycles. We assume $\overline{m} = a = \log_2 N$, where N is the network size, the maximum firing rate is $\lambda = 2\overline{m}/(2\overline{m}+1)$, $(0 \le \lambda < 1)$. The maximum spiking rate of the PN is then given by λ / T_{ch}, where $T_{ch} = ST_{clk_sys}$. This means the maximum spiking rate is only a fraction of the channel speed.

For example, in our performance estimate, for a typical network size of $N = 16{,}384$, if the average waiting time is $\overline{m} = 14$ ($= \log_2 N$), then the maximum spiking rate $\lambda = 28/29 \approx 0.97$, which means the maximum spiking rate each PN can achieve is about 97% of the maximum channel speed $1/T_{ch}$. We also define the column processor's clock as $T_{clk_sys} = nT_{norm}T_{clk_pn}$, where n is the number of multiplexed neurons per PN, T_{norm} is the PN's synaptic potential calculation time normalized to the full connection calculation time, which will be explained in the next paragraph. If the PN's clock cycle time $T_{clk_pn} = 0.2$ ns (i.e., 5 GHz), the postsynaptic event's spread time $ST_{clk_sys} = 1000T_{clk_sys}$, and $n = 32$, then $T_{clk_sys} = 6.4T_{norm}$ ns, so the channel spiking rate is $\lambda/T_{ch} = 0.97/(1000 \times 6.4T_{norm}$ ns$) \approx 1.5 \times 10^5/T_{norm}$ Hz. For example, in Figure 5-5, with 0.001 connectivity, mux-32 PN, $T_{norm} = 0.06$, the column processor's final maximum input spiking rate is $1.5 \times 10^5/0.06 \approx 2.5 \times 10^6$ (spikes/s).

Because of the sparse activation and sparse connectivity, there is opportunity to multiplex the computational hardware without impacting execution time. In our current association memory models 0.1 (10%) connectivity is typical. However as the columns scale and are interconnected into a large AM array, it is less clear how sparse the local, intra-column connectivity will be. For the sake of our analysis, we start with

0.1, and then go down to very sparse connectivity to demonstrate the effectiveness of virtualization[7]. The inefficiency of not multiplexing results in idle silicon area, and puts a highly parallel digital system's performance/price far behind a coarser-grained PN system. As explained in the "*Read weight from weight cache*" paragraph, a multiple-weight read coupled with multiple neurons per PN design can save time compared to single-weight read or a non-virtual design. We use the term *normalized weight read time* to indicate the time for a multiple-weight read, divided by the time for a single-weight read.

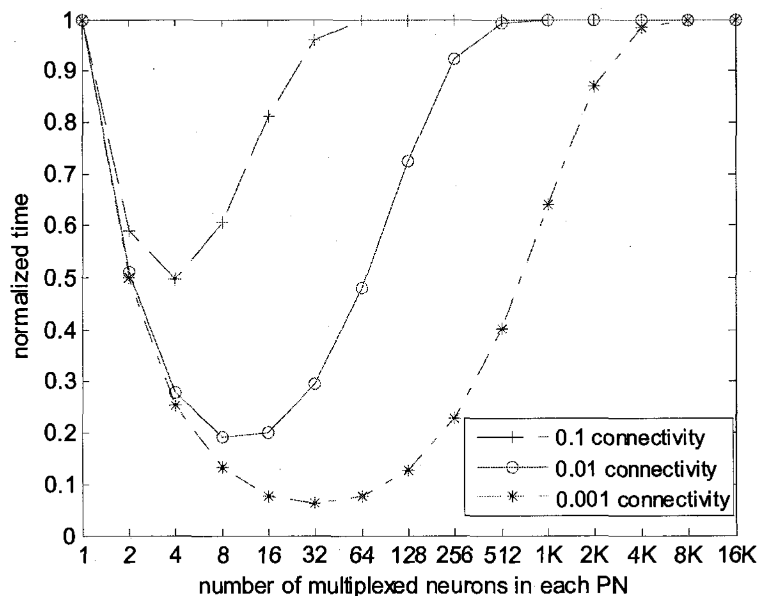


Figure 5-5: Normalized (weight read) time of multiple-weight read with the network size of 16,384.
The horizontal axis is the number of multiplexed neurons per PN with the same

[7] In fact, most AMs tend to fail at much less than about 10% connectivity. The HDM models do not depend extensively on intra-column sparseness, but rather on inter-column sparseness.

number of weights read; vertical axis gives the normalized weight read time with the longest time (the single-weight read). The three curves represent three different probabilities of memory connectivity. As shown in this figure, for 0.1 connectivity, the 4-weight read has the optimal normalized time of 0.5; for 0.01 connectivity, the 8-weight read with 0.2 normalized time; and for 0.001 connectivity, the 32-weight read with 0.06 normalized time.

Normalized time then is $T_{\text{norm}} = t_s/t_1$, where t_s is the time for reading s connections in a T_{clk_pn} cycle, and t_1 is the time for reading one connection in each PN cycle. That is, $t_s = \frac{M}{s}[(1-P_s^0)s + P_s^0]T_{\text{clk_pn}}$ and $t_1 = MT_{\text{clk_pn}}$, where P_s^0 is the probability that s consecutive connections are all zero, and M is the number of multiplexed neurons per PN. If λ' is the weight connectivity, then the average probability of s consecutive non-zero weights is $\lambda = s\lambda'$. According to queuing theory [133], with Poisson arrival and service times, we know that $P_s^0 = e^{-\lambda}$. Thus, we have the normalized time $t_s/t_1 = [(1-e^{-\lambda})s + e^{-\lambda}]/s$. Figure 5-5 shows the normalized memory reading time with three different levels of connectivity, for a network size of 16,384 neurons.

5.3. IMPLEMENTATION WITH MIXED-SIGNAL CMOS

5.3.1. Non-Spiking Mixed-signal CMOS Design

As illustrated in Figure 5-1 (b), in this option, because the inner-product operation does not scale with the network size (i.e., number of neurons), the weight matrix is still stored in CMOS memory and the inner-product is computed digitally. We could also implement the inner-product in mixed signal circuits, using a capacitor (requiring regular refresh) or floating gate transistors to store non-binary weights. This idea has

appeared in a number of neural-network chips over the years, one of the most well known was Intel's ETANN chip [15]. However, the floating-gate transistor implementation of the network connections with the analog inner-product operations was not cost-effective for a number of reasons, though the primary reason is that it speeds up only a small part of a larger, more complex algorithm, such as AM algorithm.

The digital inner-product unit realizes the circuit with complexity of $O(N\log_2 N)$, while the analog inner-product approaches $O(N^2)$ complexity with finer-grained PNs (fewer neurons multiplexed per PN). With the help of time-multiplexed digital inner-product circuits, we can use an analog k-WTA with the same $O(N)$ complexity. The k-WTA analog circuits use analog currents to generate the k highest voltages according to the k largest currents [134]. The column processor then converts those k highest-voltages to the addresses of the output neurons. Figure 5-6 shows a simple k-WTA analog circuit with $O(N)$ complexity, where the k largest injection currents drive the k outputs high, and the remaining outputs low.

However, the k-WTA is implemented in analog CMOS, so we need N parallel D/A converters to convert the digital signals from the inner-product results to analog inputs of the k-WTA circuit [134, 135].

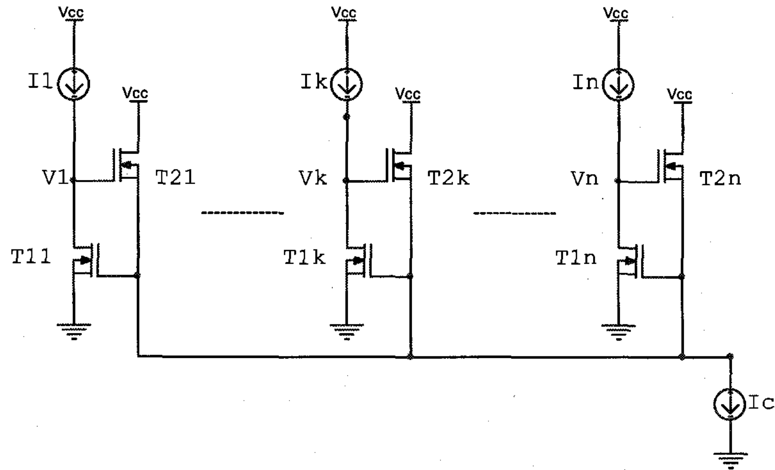


Figure 5-6: Schematic view of the *k*-WTA circuit.
(Adapted from [134].)

5.3.2. Spiking Mixed-signal CMOS design

The spike based mixed-signal CMOS design is not as simple as the non-spiking mixed-signal CMOS design, which time-multiplexes the inner-product operations in the digital domain. Furthermore, the analog *k*-WTA circuits replace the time-consuming and silicon-consuming digital *k*-WTA circuits. For the spiking models, it would not make sense to use multiplexed digital circuits for the weighted PSP computations and analog circuits for the I&F neuron model. This is because of the real-time requirement and the continuous operation of the analog circuits. Even if we did use these analog circuits, they could only replace the "Adder" and "> threshold" units in the digital counterparts, which are fairly simple and already fast. The PN also needs a D/A converter for each I&F neuron. Thus, the mixed-signal CMOS approach would not improve the performance/price by much, and it is not included in the

performance/price comparisons in Chapter 6, where we compare the CMOS and CMOL implementations' performance/price together.

6. DESIGN WITH CMOL

6.1. INTRODUCTION

There are a number of challenges facing the semiconductor industry, and, in fact, computer engineering as a whole. For metal-oxide-semiconductor field-effect transistors (MOSFET), the gate voltage threshold sensitivity to gate length grows exponentially as gates and gate oxide shrink, this is especially true for gate lengths below 10 nm [1, 136, 137], although less predictable MOSFET continue to operate at very small dimensions. However, as we approach 22 nm, it is becoming increasingly difficult to provide sufficiently accurate lithography in state of the art manufacturing processes. And it is not clear how much farther current approaches will take us below 22nm [108].

Other challenges include parameter variation, design complexity, and severe power density constraints. Nanoelectronic circuits have been touted as the next step for Moore's law, yet these circuits aggravate most existing problems and then create some of its own, such as a radical increase in levels of faults and defects. Borkar [138] indicated that currently there is no emerging nanoelectronics candidate that promises to replace CMOS in the next ten to fifteen years. Chau *et al.* [139] proposed four metrics for benchmarking nanoelectronics, and showed a promising future for nanoelectronics although their further performance and scalability need to be demonstrated.

In recent years, research in nanoelectronics has made tremendous progress, with advances in novel nanodevices [6], nano-circuits [140, 141], nano-crossbar arrays [32, 34, 35], manufacturing technologies such as nanoimprint lithography [8, 9], and CMOS/nano co-design architectures [1, 44, 142, 143] and their applications [107, 144, 145]. Although a two-terminal nanowire crossbar array does not have the functionality of FET-based circuits, it has the potential for incredible density and low fabrication costs [1]. In addition, unlike spintronics and other proposed nanoelectronic devices that use quantum mechanical state to compute [146], crossbar arrays use a charge accumulation model that is more compatible with existing CMOS circuitry.

Rückert *et al.* [103, 104, 147] have demonstrated digital and mixed-signal circuit designs for non-spiking and spiking neural associative memories. They did not fully explore time-multiplexing in their physical designs. Also, there is no universal benchmark to evaluate different hardware designs with different neural computational models. We believe that the unique combination of hybrid CMOS / nanogrids and the currently available biologically inspired models has the potential for creating exciting new computational capabilities. In our research we are taking the first few tentative steps in architecting such structures. Consequently, the goal of this chapter is to investigate the possible architecture and performance/price ratios in implementing cortical models taken from computational neuroscience with molecular grid based nanoelectronics [1].

In this chapter we first introduce some promising nanoscale device technologies, especially CMOL, and the discuss how to model the performance and price of CMOL

when used to implement HDM models. We then focus on specific CMOL implementations of the non-spiking and spiking AM models, and compare the performance/price of the CMOL results with results discussed earlier for the PC, FPGA, and CMOS.

6.1.1. Introduction of Nanoscale Devices

There are many kinds of nanoscale devices used for computation. They include, but are not limited to, single electron transistors, resonant tunneling diodes, quantum-dot cellular automata, and crossbar arrays. Because of the simplicity and near term feasibility of nanoscale crossbar arrays, we have chosen to include these structures in our analysis of promising architectures for implementing HDMs. The underlying materials for the crossbar array are single-wall carbon nano-tube, or silicon nanowires. Because single-wall carbon nano-tubes exhibit inconsistent electrical properties due to varying helicity [30], most nanogrid research assumes silicon nanowires.

For the convenience we use the following abbreviations:

- CB-FET: CrossBar Field Effect Transistor
- CB-D/R: CrossBar Diode/Resistor
- CMOL: hybrid CMOS / nanoelectronic circuits
- CB: CrossBar

The CB structure could be implemented with carbon nano-tube, silicon nanowire and metal nanowires. Different (chemical) treatment for the molecules sandwiched in between two perpendicular crossbars, or different chemical treatment for the cross points of the crossed bars, or different oxidation and doping for the silicon nanowires, the junction of the CB would yield different electrical components, such as Field-Effect Transistor (FET), diode, or resistor. In this work, we use nanogrid extensively. We define nanogrid cross-nanowire-based grid (with nanowire diameter and pitch size < 10 nm).

6.1.1.1. CB-FET

Lieber's research mainly focuses on doped silicon nanowires. In 2001, Lieber *et al.* proposed an implementation of logic gates with nanowire building blocks [29]. They reported assembly of p-type silicon (p-Si) and n-type gallium nitride (n-GaN) NWs to form crossed nanoscale p-n junctions and junction arrays in which the electronic properties and function are controlled in a predictable manner to provide both diode and FET elements in high yield. In 2003, Lieber *et al.* proposed another way to build the FET devices with specific treatment (aqueous or ethanol solution of tetraethylammonium chloride) on the desired cross points of NWs [30].

In 2003, Kuekes *et al.* patented a nanodevice structure built with crossed nanowires or nano-tubes [31], which demonstrated bipolar or field effect transistors characteristics. In 2004, Snider *et al.* proposed a CMOS-like logic capability using crossbar field effect transistors built with nanowires [32].

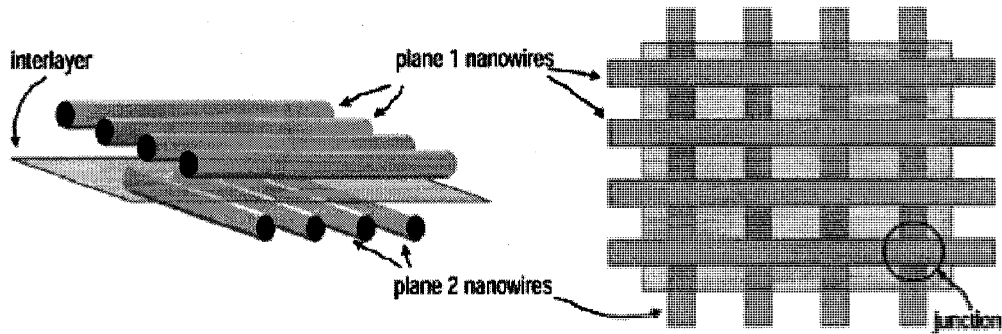


Figure 6-1: Two different views of a nanoscale crossbar.
(Adapted from [32].)

Figure 6-1 shows that the circuit consists of two perpendicular aligned arrays of nanowires. One layer of nanowires is metallic. The other layer is semiconductive. The doped silicon nanowire forms the source and drain, and the metal wire forms the gate by modulation the doping on the doped silicon nanowire at the point where the wires cross[8], between the source and drain, to form a field effect transistor. Junctions may be independently configured to behave as electronic devices. A chemical "interlayer" between the two planes of parallel nanowires, and the nanowire composition determines the type of devices that may be configured. The channel will be formed in the semiconductor nanowire, within a small region around the junction. However, for a semiconductor nanowire with typical doping levels of 10^{18} atoms of boron or arsenic per cubic centimeter, there would be, on average, only 0.1 dopant atom in a 5 nm × 5 nm junction. "*As a result, FETs at those dimensions might not behave predictably, if they would even function at all* [32]."

[8] Doping exists on both sides of the doped nanowire around the cross point.

6.1.1.2. CB-D/R

Lieber *et al.* [33] also reported the assembly of functional nanoscale devices from indium phosphide nanowires, the electrical properties of which are controlled by selective doping. Gate-voltage-dependent transport measurements demonstrate that the nanowires can be predictably synthesized as either n- or p-type.

In 2003, Chen *et al.* at HP formed nanoscale molecular-switch crossbar circuits [34]. The proposed nanoscale circuits are based on configurable crossbar architecture to connect molecular switches in a two-dimensional grid, as shown in Figure 6-2 (a).

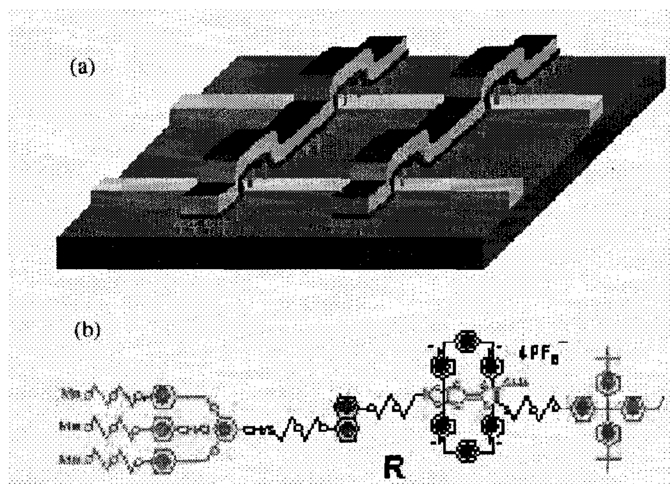


Figure 6-2: Nanoscale molecular-switch crossbar circuit from Chen [34].
(a) Schematic representation of the crossbar circuit structure. A monolayer of the rotaxane (green) is sandwiched between an array of Pt/Ti nanowires (gold, left–right) on the bottom and an array of Pt/Ti nanowires (gold, up–down) on the top. (b) Molecular structure of the bistable rotaxane R (adapted from [34]).

The electrical properties are discussed in [34]. For example, the initial resistance of a typical device measured at 0.2 V was 6.1×10^8 Ω. After sweeping the voltage bias

cycle from 0 to +5 V, the resistance subsequently measured at 0.2 V dropped to $4.3 \times 10^5\ \Omega$.

The basic element in the circuit is the Pt/rotaxane/Ti junction formed at each cross point, which acts as nonvolatile switches. And 64 such switches are connected to form an 8×8 crossbar circuit within a 1 um^2 area, as illustrated in Figure 6-3.

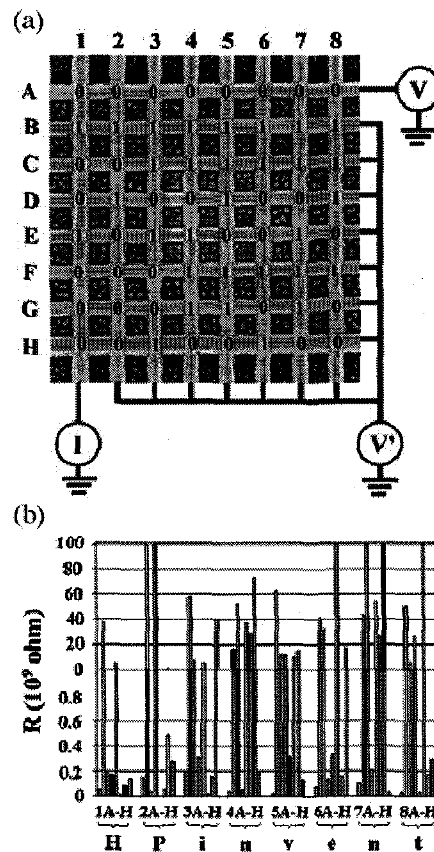


Figure 6-3: The crossbar as a 64-bit random access memory.
(a) The ideal 'write' and 'read' modes for the memory. To write a bit, V is increased in increments of 0.5 from 3.5 V until the bit is written or V reaches 7 V, while keeping V' = V/2. To read a bit, V = 0.5, and V' = 0. (b) Resistance at each cross point in the circuit after one particular set of bits was written into a defect-free crossbar (adapted from [34]).

The nanodevice mentioned above is like a bistable-switch latch. Standard semiconductor latch circuits use three-terminal transistors to achieve the switching illustrated in Figure 6-4.

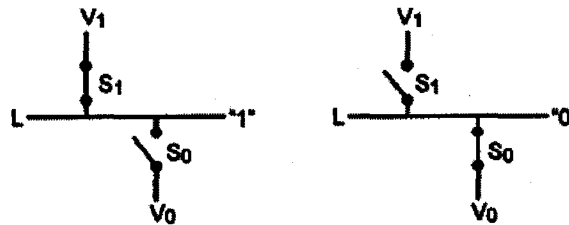


Figure 6-4: Schematic diagram illustrating the basic function of a latch used to transform a bidirectional switch (memory) into a voltage (logic) representation for representing logical data values.
The logic line L is connected to two control lines with voltages V_0 and V_1 representing logical 0 and logical 1, respectively. When switch S_1 is closed and S_0 is open, the line voltage is pulled up to V_1 and thus has a logical value of 1. When the switch S_1 is open and S_0 is closed, the line voltage is pulled down to V_0 and thus has a logical value of 0 (adapted from [35]).

6.1.1.3. CMOL

Likharev [1, 2] argued that the CMOS circuits can hardly be extended to a few nanometer region, because the sensitivity of parameters (e.g., the gate voltage threshold) of FET to fabrication grows exponentially. Also, for the single-electronic devices, the fabrication accuracy should be the order of 0.1 nm. This is not a realistic number for mass production. Consequently, Likharev proposed a nanoscale circuit model called CMOL, which originally was an abbreviation of Cmos/nanowire/MOLecular integrated circuits [1]. However, Likharev extend the concept of CMOL to more general hybrid CMOS / nanogrid circuits [2]. Likharev has

many publications presenting a systematic research of device, circuits, architectures and algorithm implementations with CMOL [1, 37, 144, 145, 148, 149].

CMOL is also based on the crossed nanowires. CMOL has three components, a nanowire grid, a set of pins connecting the nanowires to CMOS circuits – here is where the famous CMOL tilt [1] allows for alignment free grid structures, and a bistable resistance, which need not be a single molecule, at the nanowire crosspoints. A possible molecule acting as a switch or a 2-way diode is shown in Figure 6-5 (c). Under a certain threshold voltage, the device turns on (Figure 6-5 a).

Real progress has been made in implementation candidates for all those areas, however, the least well developed is the grid switch [2], though Likharev indicates some promising materials are being investigated [2]. Currently, there are no viable candidates for materials that demonstrate a bistable resistance and rectify current. Having resistive connections only allows for a certain amount of signal degradation in mixed signal designs and would seriously affect scaling.

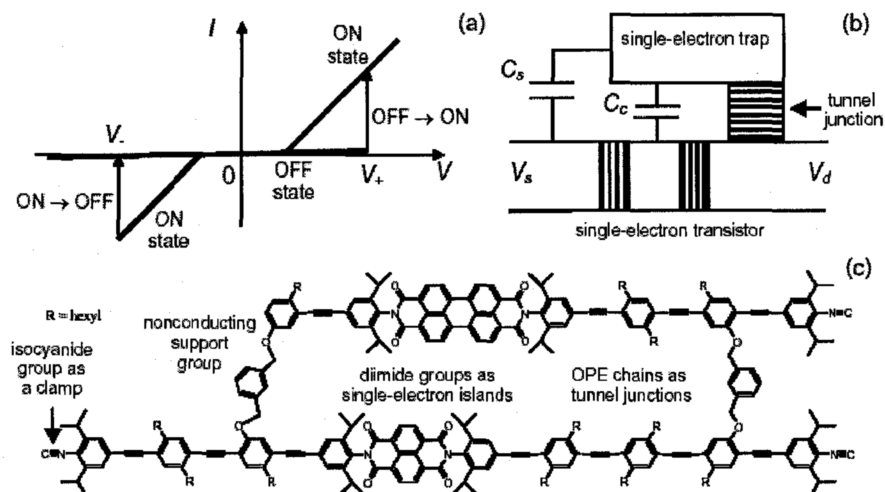


Figure 6-5: Two-terminal latching switch.
(a) *I-V* curve (schematically), (b) single-electron device schematics, and (c) a possible molecular implementation of the device (adapted from [1]).

From Figure 6-5 (a), we notice the *I-V* curve of the two-terminal bi-stable device. It can be implemented with a combination of two single-electron devices: a "transistor" and a "trap"[9], as shown in Figure 6-5 (b). If the applied drain-to-source voltage $V = V_d - V_s$ is low, the trap island in equilibrium has no extra electrons ($n = 0$), and its net electric charge $Q = -ne$ is zero. As a result, the transistor is in the virtually off state, and source and drain are essentially disconnected. If V is increased beyond a certain threshold value V_+, its electrostatic effect on the trap island potential (via capacitance C_s) leads to tunneling of an additional electron into the trap island: $n \rightarrow 1$. This change of trap charge affects, through the coupling capacitance C_c, the potential of the

[9] The problem with single electron traps is that they are very hard to create and usually work properly under extremely low temperature, otherwise the electrons do not stay put.

transistor island, and suppresses the Coulomb blockade threshold to a value well below V_+. As a result, the transistor, whose tunnel barrier should be thinner than that of the trap, is turned into on state in which the device connects the source and drain with a finite resistance R_{on}. If the applied voltage stays above V_+, this connected state is sustained indefinitely. However, if V remains low for a long time, the thermal fluctuations will eventually kick the trapped electron out, and the transistor will get closed, disconnecting the electrodes. This on → off switching may be forced to happen much faster by making the applied voltage V sufficiently negative, $V \approx V_-$.

Figure 6-5 (c) shows a possible molecular implementation of the device shown in Figure 6-5 (b). Here, two different diimide acceptor groups play the role of single-electron islands, while short oligo-ethynylenephenylene (OPE) chains are used as tunnel barriers. The chains are terminated by isocyanide-group "clams" ("alligator clips") that should enable self-assembly of the molecule across a gap between two metallic electrodes [1].

6.1.2. CIRCUITS

6.1.2.1. Memory

With the bistable-switch device mentioned in Section 6.1.1.2, Kuekes *et al.* proposed the crossbar latch circuit [35]. This device stored a logic value on a signal wire, enabling logic value restoration, and inversion. In combination with resistor/diode logic gates, these operations in principle enable universal computing for crossbar circuits, and potentially, integrated nanoscale electronics.

Likharev proposed an architecture of integration of nanoscale and microscale circuits [1, 145]. The architecture of CMOL memories is shown below:

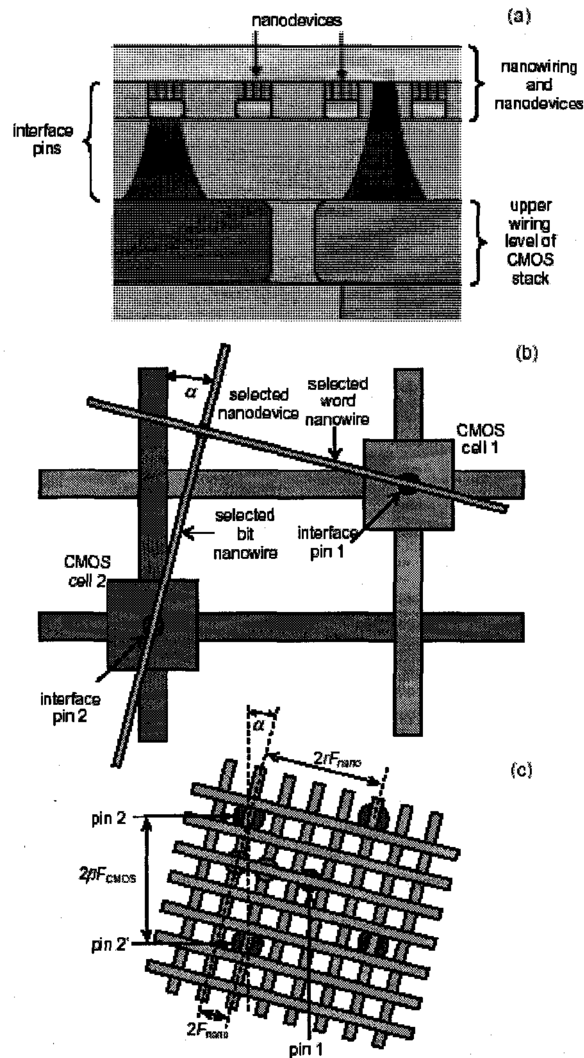


Figure 6-6: The generic CMOL circuit.

(a) a schematic side view; (b) a schematic top view showing the idea of addressing a particular nanodevice via a pair of CMOS cells and interface pins, and (c) a zoom-in top view on the circuit near several adjacent interface pins (adapted from [1]).

Figure 6-7 shows the top structure of the CMOL memory. It is essentially a matrix of L memory blocks, where each block is a rectangular array of $(n + a) \times (m + b)$ memory cells. Here a and b are the number of spare rows and columns, respectively, while $n \times m$ is the final block size after reconfiguration.

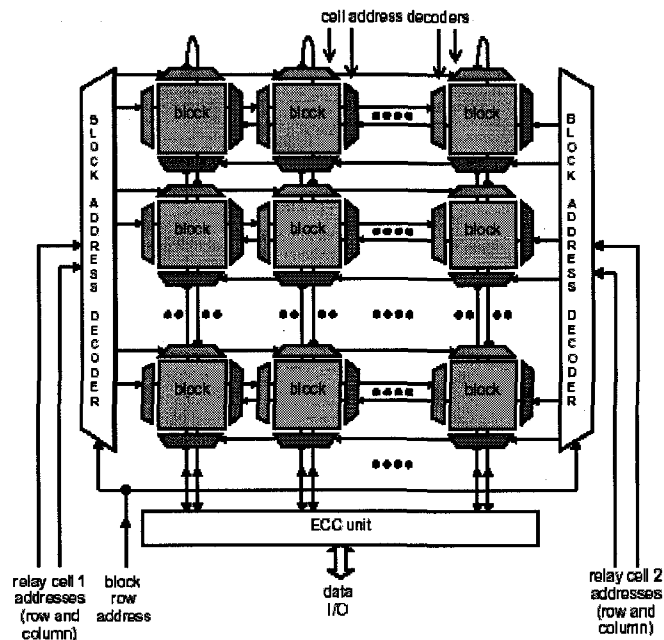


Figure 6-7: The system architecture of CMOL memory.
At each block, block address decoders allow sending the cell row and column addresses to a single row of blocks. The cell addresses are then processed by decoders for each block (adapted from [1]).

6.1.2.2. Logic Units

Snider *et al.* (from HP) had a very interesting idea about complementary circuits with CB-FET. The circuit is defect-tolerant. The simple structure is similar to a traditional CMOS structure, and can implement AND-OR-INVERT functions, which are sufficient for general computation. These arrays can be combined to create logic blocks capable of implementing sum-of-product functions, and larger computations, such as state machines, can be obtained by adding additional routing blocks. Figure

6-1 shows two different views of the nanoscale crossbar. Figure 6-8 and Figure 6-9 show the schematic of the CB-FET and the logic function implementation by the circuit.

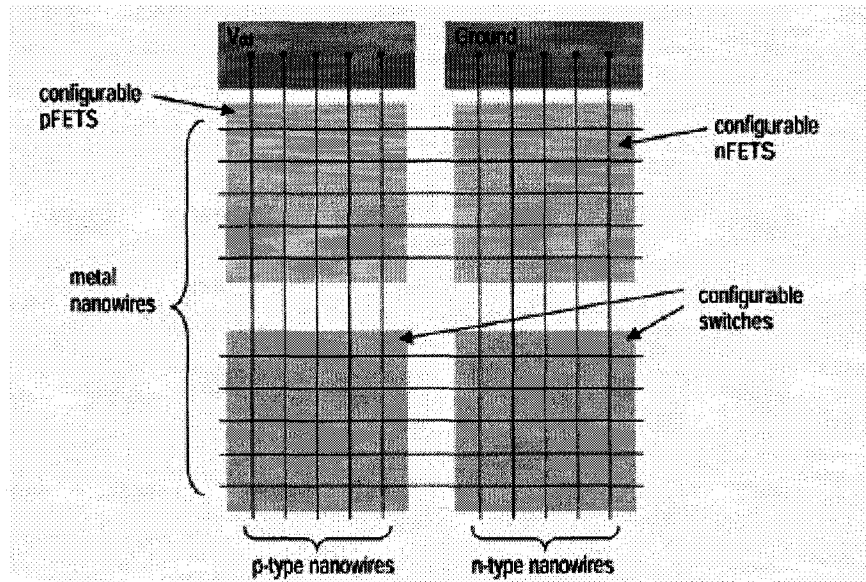


Figure 6-8: Logic blocks implemented with a complementary/symmetry array. Each junction in the pink quadrant may be independently configured to implement a p-FET, while each junction in the blue quadrant may be configured to implement an n-FET. The junctions in the two lower quadrants may be configured to perform signal routing (adapted from [32]).

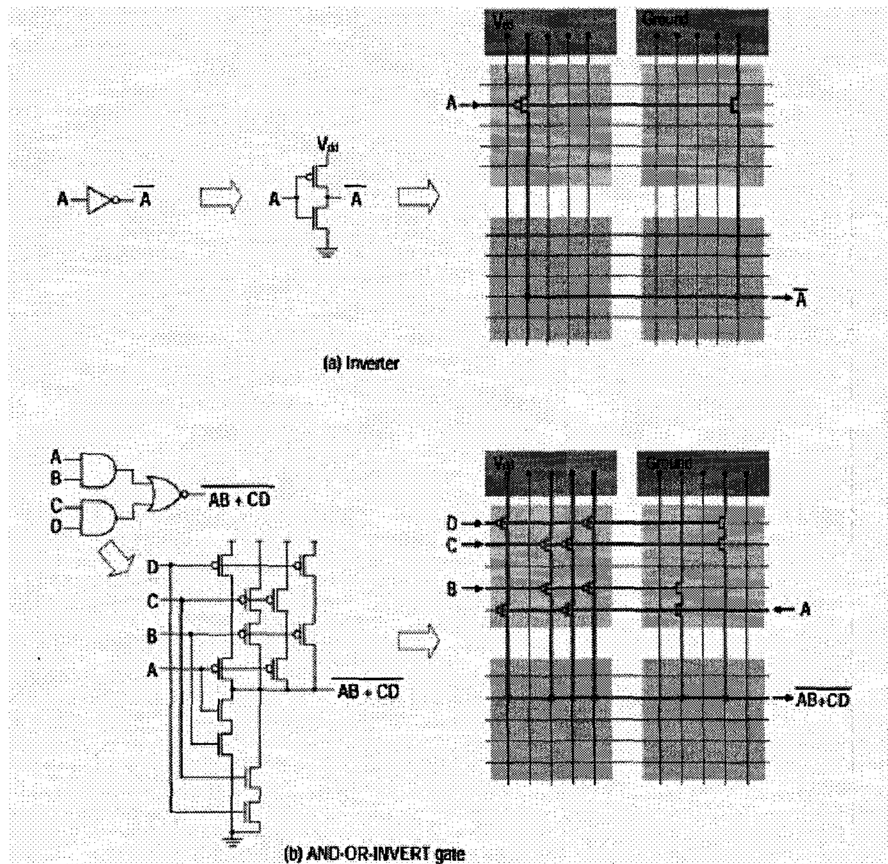


Figure 6-9: Implementing two logic functions by selectively configuring the nanodevice-based junctions.
(Adapted from [32].)

Strukov and Likharev [144] proposed a reconfigurable architecture to implement logics with CMOL, which is called CMOL FPGA, as illustrated in Figure 6-10.

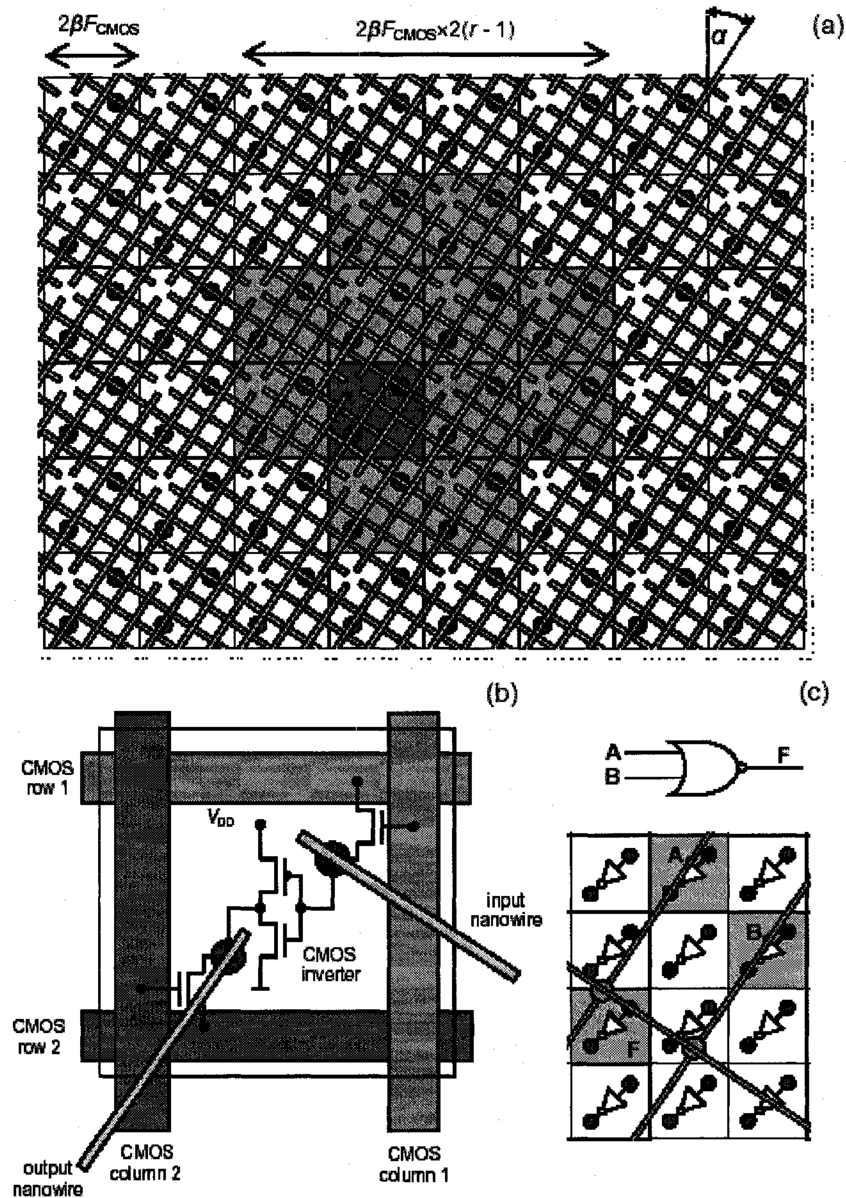


Figure 6-10: CMOL FPGA architecture.

(a) Generic CMOL, (b) a single CMOS cell, and (c) a NOR gate implementation. In panel (a) the cells painted light-gray may be connected to the input pin of a specific cell (dark-gray). Panel (b) shows only two nanowires (that contact the given cell), while panel (c) shows only three nanowires used to implement the NOR gate (adapted from [1]).

This approach to Boolean logic circuits based on CMOL is similar to the so-called cell-based FPGA [1]. In Figure 6-10, an elementary CMOS cell includes two pass

transistors and an inverter, and is connected to the nanowire/molecular subsystem via two pins. During the configuration process the inverters are turned off, and the pass transistors may be used for setting the binary state of each molecular device. Each pin of a CMOS cell can be connected through a nanowire-nanodevice-nanowire link to each of $M \equiv 2r^2 - 2r - 1$ other cells within a square-shaped "connectivity domain" around the pin. Figure 6-10 (c) shows how such a fabric may be configured for the implementation of a fan-in-of-two NOR gate, which sufficient to implement any logic function. Gates with larger fan-in and fan-out are clearly possible. Figure 6-11 shows a NOR-NOR network implementation of a Kogge-Stone adder.

Figure 6-11: Kogge-Stone adder and its NOR gate synthesis.
(a) The 32-bit Kogge-Stone adder and (b) its single (16th) bit slice implemented with NOR gates only (adapted from [1]).

Snider *et al.* [44] proposed a more conservative variation of CMOL, called FPNI (field programmable nanowire interconnect). Figure 6-12 illustrates the differences between CMOL and FPNI:

i) The architecture of FPNI is based on CMOL, with nanowire crossbars fabricated on top of CMOS.

ii) FPNI uses more conservative nanowire pitch data (about 9 nm) versus CMOL's 3 nm nanowire pitch.

iii) In FPNI the pins have greater clearance for contact with the nanowires.

iv) In FPNI, the CMOS logic for each cell is buffer based, not inverter based, as in CMOL. This simplifies the routing in the nanowire crossbars. For example, in Figure 6-13, a hypercell could include four three-input NAND/AND gates, one flip-flop and 26 buffers.

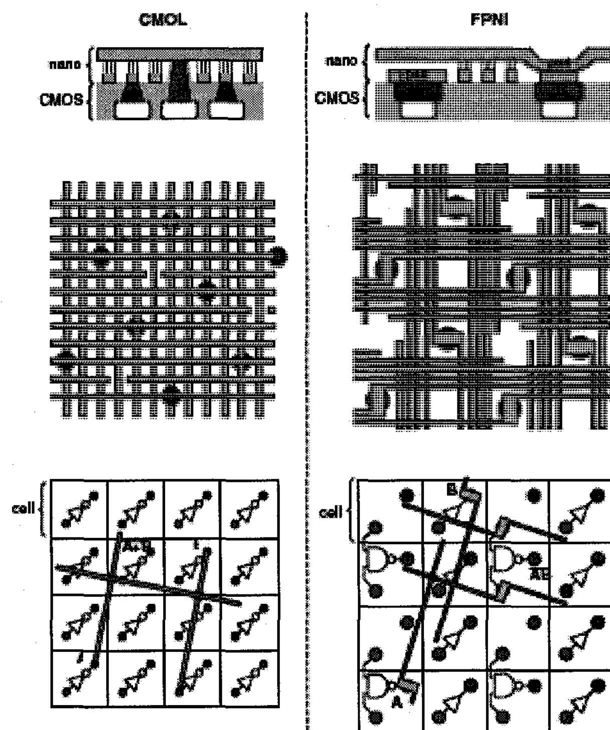


Figure 6-12: Schematic diagrams of hybrid circuits. The left is CMOL. The right is FPNI.
(Adapted from [44].)

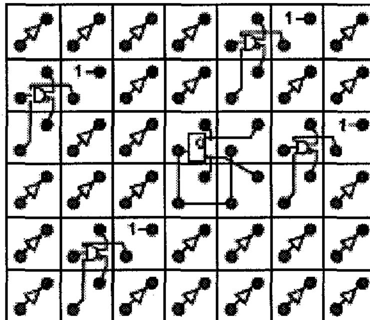


Figure 6-13: A hypercell consisting of four three-input NAND/AND gates, one flipflop and 26 buffers.
(Adapted from [44].)

6.1.3. Other Nanoarchitecture

6.1.3.1. Likharev's CMOL

Earlier we introduced Likharev's CMOL. It is not only a device or circuit innovation, but also a unique nanoarchitecture. It is also very promising for implementing the AM models assumed in this dissertation. In this section we introduce some other nanoarchitectures. Our objective is to show that there are many other possibilities for nanotechnology implementations.

6.1.3.2. Ziegler and Stan's CMOS/nano Co-design

Ziegler and Stan proposed a co-design of CMOS and nanoscale devices [143, 150]. They use the crossbar technology proposed by HP and UCLA [32, 35, 151]. This crossbar technology is composed of arrays of crossed nanowires with bistable nanoscale-switches sandwiched between the cross points of the nanowires. The upper-left portion of Figure 6-14 shows a simplified diagram of such a crossbar. Molecules are present at each junction, forming a two-terminal device that can be electrically

configured to behave as a low resistance or a high resistance diode. These molecules, such as rotaxanes or cantenanes, create a programmable computing fabric that can be used for memories and logic arrays.

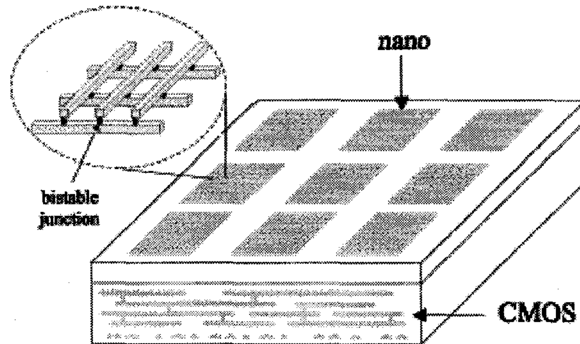


Figure 6-14: A design paradigm involving nanoelectronics on a CMOS IC. (Adapted from [143].)

The general concept of a mixed CMOS/nano circuit is to divide the functionality between conventional CMOS and nanoelectronic technology. Ziegler and Stan [143] favor a PLA (Programmable Logic Array) implementation over LUT (Look Up Table) implementation in terms of array size. Equations (6.1) and (6.2) govern the size of a diode-based crossbar, for LUT and PLA structures. In the equations, N is the number of literals in all the functions implemented on the crossbar, f is the number of functions, and c is the number of two-level minimized cubes in all the functions. Equation (6.3) shows the area overhead for a LUT structure versus a PLA structure. Thus, the optimality of a PLA representation increases as more functions with overlapping product terms are allocated to a single crossbar.

$$LUT_{area} = 2^N (2N + f) P_{wire}^{\ 2} \tag{6.1}$$

$$PLA_{area} = c(2N + f)P_{wire}^{\ 2} \tag{6.2}$$

$$PLAsavings = \frac{2N}{c} \tag{6.3}$$

Interface cost: there will be some overhead incurred when a signal switches media. Furthermore, the lack of signal gain in some crossbar technologies mandates that the computation must leave the crossbar periodically for restoring signal integrity.

6.1.4. Performance Modeling of Nano-structures

We focus on the performance and price (cost) modeling for the CMOL circuits in this section. The performance refers to speed (propagation time delay). The price of the circuit includes silicon (nanodevice) area and power consumption. A simplified performance and price model for our CMOL nanogrid is also given in this section.

6.1.4.1. CMOL crossbar arrays

For CMOL crossbar arrays, the nanodevices exist at the cross points, or junctions. The nanowire (NW) acts as interconnect. When we model the physical structures, we need to clarify the following:

a) Nanowire can act as interconnect. The NWs come in different materials, diameter, length, and conductivity (resistivity).

b) The cross points have different electrical properties due to built with different materials.

Based on those considerations, Figure 6-15 shows the schematic to a generic crossbar array.

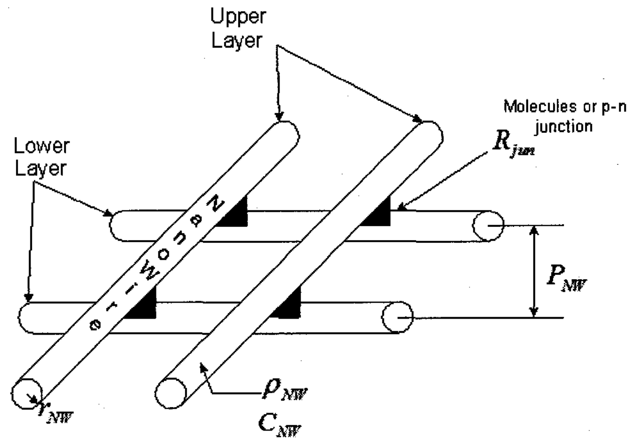


Figure 6-15: Schematic of crossbar arrays.

In Figure 6-15, there are two layers of nanowires. R_{jun} is the resistance of the junction between the two perpendicular nanowires. For CB-D/R, R_{jun} has the value of $R_{jun}(ON)$ and $R_{jun}(OFF)$ for closed and open junction resistance respectively (current can only flow in one direction for diode-like nano-switch, or both directions for resistor-like nano-switch); For CB-FET, R_{jun} has a large value and could be considered infinity; P_{NW} is the pitch between two parallel neighbor nanowires; r_{NW} is the radius of the nanowire (without coating); ρ_{NW} is the nanowire resistivity; C_{NW} is the nanowire capacitance.

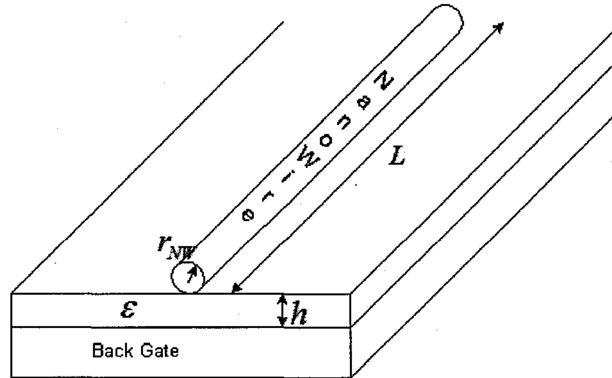


Figure 6-16: Calculation of nanowire capacitance with back gate.

In Figure 6-16, ε is the insulator permittivity; h is the height of the insulator; and L is the length of the nanowire for calculating the capacitance, which is Equation (6.4).

$$C_{NW} = 2\pi\varepsilon \frac{L}{\ln(2h/r_{NW})}, \tag{6.4}$$

The CMOL nanodevices we are using do not have back gates. The nanowire capacitance only exists around the crosspoints. Thus the nanowire capacitance for each crosspoint is modified to Equation (6.5).

$$C_{NW} = 2\pi\varepsilon \frac{2r_{NW}}{\ln(2h/r_{NW})}, \tag{6.5}$$

where h is the height of insulator between two nanowire layers.

If the nanowire is fabricated using nanoimprint lithography [8], the shape of the nanowire is more like the one depicted in Figure 6-17. Equation (6.5) only models one NW. We need to model the NW arrays, because it is more appropriate to fabricate the NW with nanoimprint [8] other than bottom-up fabrication.

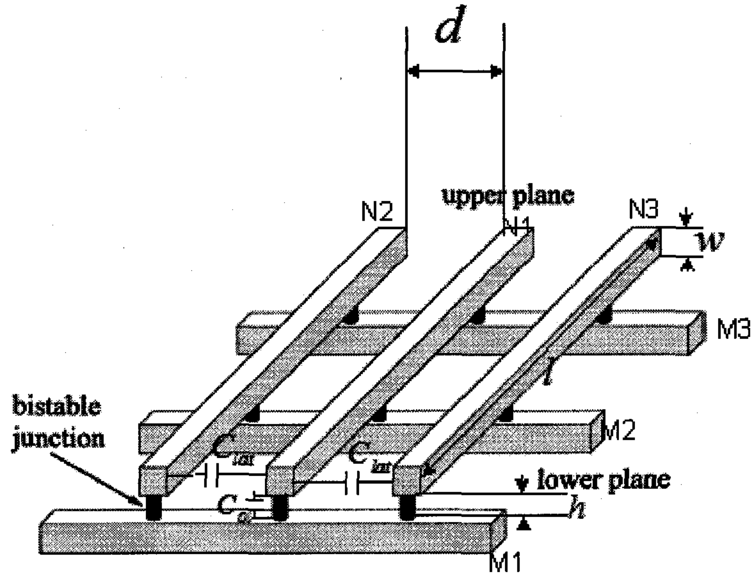


Figure 6-17: Nanowire arrays with square-like cross-section NWs.
C_{lat} is the lateral capacitance; C_{ol} is the overlap capacitance; l is the length of N1; w is the width of cross-section of N1; h is the distance between the two NW layers.

Figure 6-17 shows that the capacitance for N1 includes the lateral capacitance between N1 and N2, N1 and N3 (if we only count the nearest nanowire influence on N1 in the upper plane), and the overlap capacitance between N1 and M1, N1 and M2, …, and N1 and M*m*, if there are m nanowires in the lower plane overlapped with N1. The total capacitance C_{NW} is given by

$$C_{NW} = 2C_{lat} + mC_{ol} , \qquad (6.6)$$

where C_{lat} is the lateral capacitance, and C_{ol} is the overlap capacitance.

If we assume all the NWs are connected to ground except N1, then we can compute capacitance as follows. Although in real circuits, the other NWs could not be in the

same contact, we still can use this assumption to calculate the capacitance in the worst case.

$$C_{lat} = \kappa\varepsilon_0 \frac{lw}{d} , \tag{6.7}$$

$$C_{ol} = \kappa'\varepsilon_0 \frac{w^2}{h} , \tag{6.8}$$

where κ is the dielectric constant for material between the NWs in the upper layer; κ' is the dielectric constant for material at the intersection (crosspoint) of the NWs.

6.1.4.2. CMOL FPGA

For a CMOL FPGA, as illustrated in Figure 6-18, we notice that the CMOS signals and nanoscale signals interleave with each other very tightly. The signals coming from CMOS go to input nanowires via CMOS inverters, which restore the signal to the tail voltage; the output nanowires signals are input to a CMOS cell, where the signal is restored to full voltage.

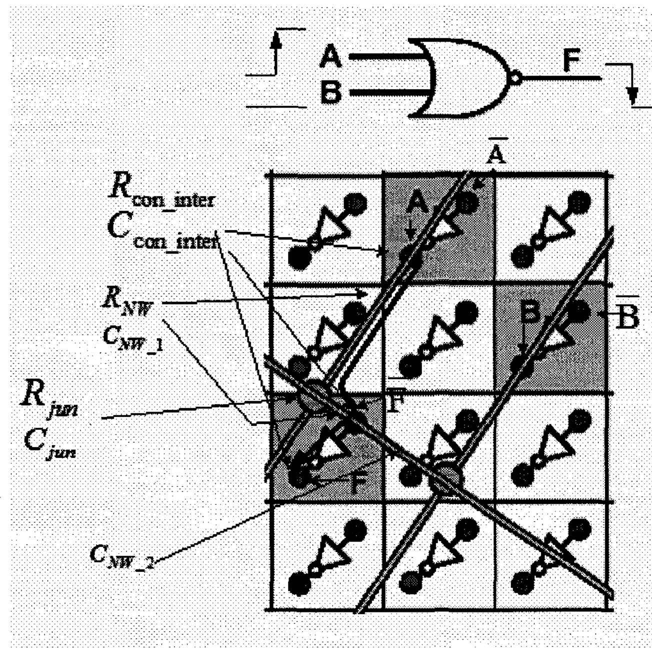


Figure 6-18: Schematic of NOR gate with CMOL FPGA.

In Figure 6-18, when A is enabled (high) and B is low, the current will flow along the bold black line from A to F. C_{NW_1} is the capacitance along the input nanowire and C_{NW_2} is the capacitance along the output nanowire.

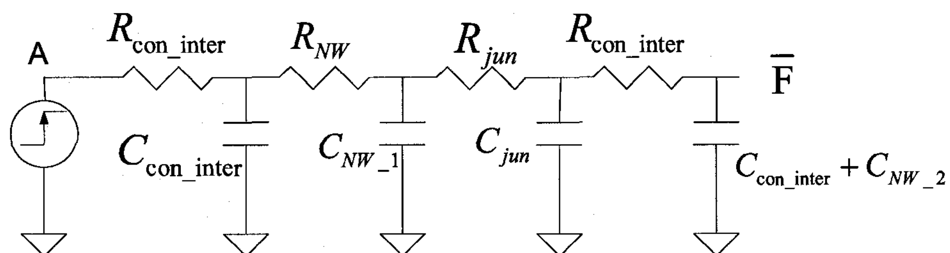


Figure 6-19: Equivalent circuit of transmission line for the circuit in Figure 6-18.

The time constant (propagation delay) for the RC (resistor-capacitor) interconnect tree is given by (6.9), which is based on the classical Elmore delay model [50].

$$t_{pd} = \sum_{i=1}^{N}(\sum_{j=1}^{i} R_j)\cdot C_i \tag{6.9}$$

According to Elmore delay model in (6.9), the transient time delay (high to low) for the equivalent circuit in Figure 6-18 is given by:

$$\begin{aligned} t_{phl}' &= R_{\text{con_inter}}C_{\text{con_inter}} + (R_{\text{con_inter}} + R_{\text{NW}})C_{\text{NW_1}} \\ &\quad + (R_{\text{con_inter}} + R_{\text{NW}} + R_{\text{jun}})C_{\text{jun}} + (R_{\text{con_inter}} + R_{\text{NW}} + R_{\text{jun}} + R_{\text{con_inter}})(C_{\text{con_inter}} + C_{\text{NW_1}}) \\ &= (3R_{\text{con_inter}} + R_{NW} + R_{jun})C_{\text{con_inter}} \\ &\quad + (3R_{\text{con_inter}} + 2R_{NW} + R_{jun})C_{NW} + (R_{\text{con_inter}} + R_{NW} + R_{jun})C_{jun} \end{aligned} \tag{6.10}$$

Equation (6.10) does not include the time delay from $\overline{\text{F}}$ to F. We assume $C_{NW_1}=C_{NW_2}=C_{NW}$. In order to add the time delay for the CMOS inverter $t_{CMOS_inverter}$, the total propagation time delay t_{phl} (high to low) is expressed as below:

$$t_{phl} = t_{phl}' + t_{CMOS_inverter} \tag{6.11}$$

Because the circuit in Figure 6-18 has the same current flow route for output F going from low to high as the current flow route for output F going from high to low, the propagation time delay t_{plh} (low to high) is the same as t_{phl} (high to low).

$$\therefore t_p = t_{phl} = t_{plh} \tag{6.12}$$

6.1.4.3. CMOL Memories

For CMOL memories, from Figure 6-20, we can analyze the transient time delay with the similar schemes with thouse in the previous section.

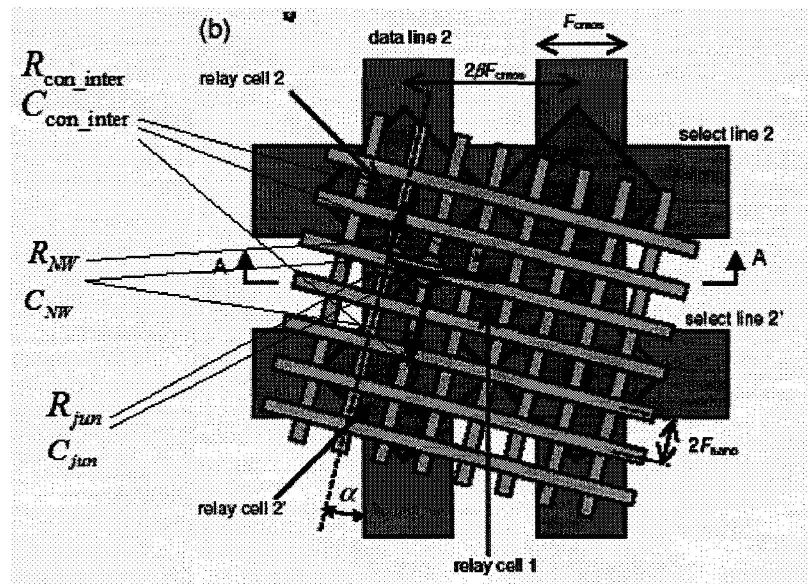


Figure 6-20: Schematic of current flow and R, C parameters for CMOL memories.

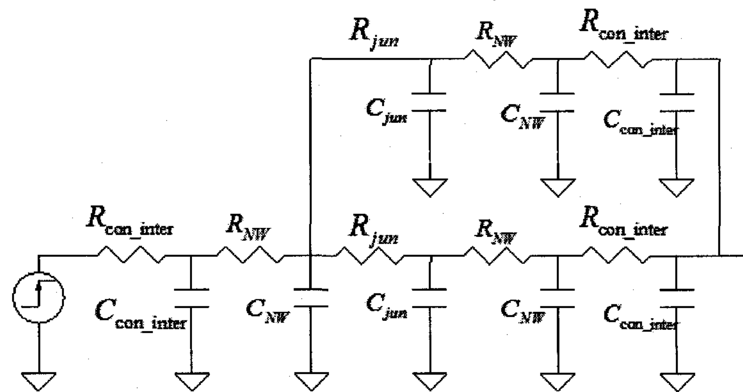


Figure 6-21: Equivalent circuit of Figure 6-20.

We assume the two parallel parts in Figure 6-21 are symmetric for simplicity, then the equivalent circuit in Figure 6-21 could be re-drawn in Figure 6-22.

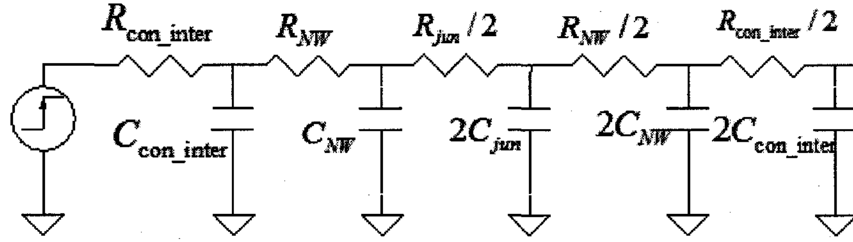


Figure 6-22: Simplified equivalent circuit of Figure 6-21.

According to the Elmore delay model (6.9), we can get the delay time constant of Figure 6-22 as below:

$$
\begin{aligned}
t_p &= R_{\text{con_inter}}C_{\text{con_inter}} + (R_{\text{con_inter}} + R_{NW})C_{NW} + (R_{\text{con_inter}} + R_{NW} + R_{jun}/2)2C_{jun} \\
&+ (R_{\text{con_inter}} + R_{NW} + R_{jun}/2 + R_{NW}/2)2C_{NW} \\
&+ (R_{\text{con_inter}} + R_{NW} + R_{jun}/2 + R_{NW}/2 + R_{\text{con_inter}}/2)2C_{\text{con_inter}} \\
&= (2R_{\text{con_inter}} + 1.5R_{NW} + R_{jun}/2)2C_{\text{con_inter}} + (R_{\text{con_inter}} + R_{NW} + R_{jun}/2)2C_{jun} \\
&\qquad + (1.5R_{\text{con_inter}} + 2R_{NW} + R_{jun}/2)2C_{NW} \qquad (6.13)
\end{aligned}
$$

6.1.4.4. Simplified CMOL-based nanogrid performance modeling

For the nanogrid model used in this analysis, we use CMOL, a hybrid CMOS / nanoscale circuit architecture developed by Likharev *et al.* [1, 2]. Although nanoelectronics allows much denser circuits, it does have a number of limitations. Perhaps the biggest limitation is that it is a faulty computation platform. In CMOL circuits, static defects (permanent defects) and transient faults are possible in the nanodevices, the nanowires, and the CMOS-to-nanowire contacts. Strukov and

Likharev [145] demonstrated two methods of fault tolerance for CMOL memory. For associative algorithms, Rückert *et al.* [152] showed that the stuck-at-0 connection errors have a greater impact on network performance than the stuck-at-1 connection errors. Sommer *et al.* [153] used iterative retrieval by probabilistic inference to improve the network's information capacity in the presence of weight matrix errors. Zhu [72] also demonstrated detailed fault tolerance of WPNAM model, with static and dynamic changes (faults added) to the weight (connection) matrix. We [154] simulated a WPNAM-like network with static and dynamic faults added to the input vectors and weight matrix, and found the network favors weight matrix (connection) faults than input faults. The fundamental fault tolerance of our target algorithms, coupled with Strukov and Likharev's results [145], leads us to believe that any additional overhead for tolerating a reasonable level of faults will be minimal (5-10%) and so it is not factored into this analysis.

The nanodevice in CMOL is a binary "latching switch" based on molecules with two metastable internal states. Figure 6-23 shows the schematic *I-V* curve of this two-terminal nanodevice. Qualitatively, if the drain-to-source voltage is low during programming, the nanodevice will be in the "off" state with a high resistance (R_{off}); if the applied voltage is greater than the threshold voltage (V_t), the nanodevice will be in the "on" state with a lower resistance (R_{on}).

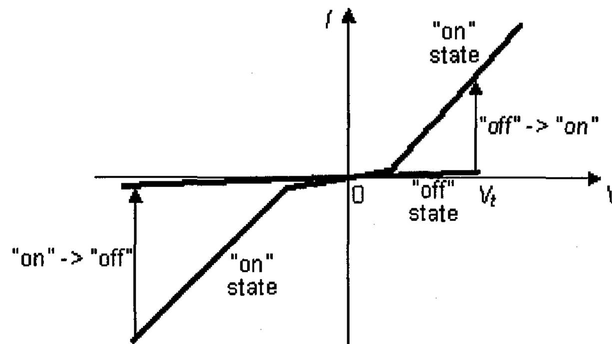


Figure 6-23: Schematic *I-V* curve of a two-terminal nanodevice [10]. (Adapted from [1].)

In this analysis we develop the performance/price analysis of various CMOL configurations when emulating an auto-associative memory model. The components that affect the performance of the circuit include the nanodevice itself, the nanowire, and the pin-to-nanowire contact (pins interface CMOS and nanowire, Figure 3(a) in [1]), as shown in Figure 6-24. In CMOL, we assume that each latching switch is implemented as a parallel connection of D single-electron devices. The molecular capacitance is typically negligible in comparison with the capacitance between the wires. What is changing is D. Theoretically D increases with the half pitch of nanowire F_{nano}, however, it is highly related to manufacturing precision. If we assume the scaling F_{nano} is t = 8nm/F_{nano}, then the scaling of D is $1/t^2$ (i.e., $D/18 = 1/t^2$); and

[10] This *I-V* curve also necessarily represent the best material at the NW intersections in HP's memristor, the fourth passive component type after resistors, capacitors and inductors, which was first postulated by Chua (L. Chua, "Memristor - The missing circuit element," *IEEE Transactions on Circuits Theory*, vol. 18, pp. 507-519, 1988. And URL: http://www.eetimes.com/showArticle.jhtml?articleID=207403521&printable=true).

the scaling of R_{on}/D is t^2. For nanowire capacitance and resistance, refer to Fig. 13 and Eq. (5) in [144]. Size issues also need to be considered because of very high resistance of the nanowire. We assume the pin-to-nanowire contact is ohmic. The contact resistance is $R_{con} = \rho / F^2_{nano}$, where ρ is about 10^{-8} $\Omega\,\text{cm}^2$ with $N_D \approx 3\times10^{20}$ cm^{-3} doping.

Figure 6-24 shows a signal current flowing through a nanowire crossbar. With values for the resistance and capacitance of the basic CMOL components, according to the classic Elmore delay model [50] we estimate the time delay from the input pin to the output pin through the nanowires and nanodevices as follows:

$$\tau = (2R_{con} + 1.5R_{wire} + R_{on}/D)C_{wire}, \tag{6.14}$$

where R_{con} is the pin-to-nanowire contact resistance; R_{wire} is the nanowire resistance; and C_{wire} is the nanowire capacitance.

For CMOL crossbar arrays, the static power consumption includes both the working power and the leakage power. A working "on" power is due to the "on" nanodevices, and is given by

$$P_{on} \equiv \alpha\beta NMV^2 / (2R_{con} + R_{wire} + R_{on} / D), \tag{6.15}$$

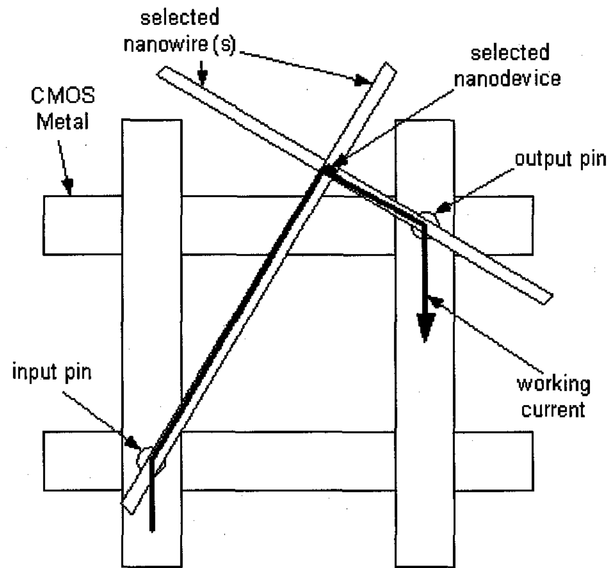


Figure 6-24: Current (the arrowed line) flows from the input pin via an input nanowire through the nanodevice and output nanowire to the output pin.

where α is the average probability that the driving voltage to the input nanowire is high (voltage on the nanodevice is over V_t); β is the probability that the nanodevices are "on"; and N and M are the horizontal and vertical nanowire counts, respectively.

Due to the current leakage through the "off" nanodevices, the leakage power is given by

$$P_{leak} \equiv \alpha(1-\beta)NMV^2/(2R_{con}+R_{wire}+R_{off}/D). \qquad (6.16)$$

If we know the average current $\overline{I_b}$ for each output nanowire or each bundle (a group) of output nanowires (Figure 6-27 (b)), the average power that CMOL nanogrids dissipate is given by

$$\overline{P} \equiv N_b V \overline{I_b}\,, \tag{6.17}$$

where N_b is the number of output nanowires or the number of bundles of output nanowires depending on applications. The dynamic power due to the dynamic charging of the nanowires is

$$P_{dyn} \equiv \gamma (N+M) C_{wire} V^2 / \tau\,, \tag{6.18}$$

where γ is the average probability that the nanowires are charged during the cycle time τ. The area for a CMOL crossbar array is

$$A \equiv 4NMF^2{}_{nano}\,. \tag{6.19}$$

6.2. IMPLEMENTATION WITH DIGITAL CMOL

As of the CMOS implementations, as similar design space is assumed for the CMOL technology in Figure 5-1. It does not matter whether non-spiking or spiking AM models are used for mixed-signal CMOL implementations, since mixed-signal CMOL implementations can leverage the CMOL nanogrids to implement the weight matrix and inner-product, and do not need the D/A conversion.

6.2.1. NON-SPIKING AM MODEL

Here, CMOL is used only as very dense (and somewhat slow and unreliable) memory to replace the CMOS weight memory of the all digital CMOS design. The inner-product and *k*-WTA computations are still in digital CMOS and have the same circuits

as those in the digital CMOS design. The CMOL memory performance/price modeling is referred to Section 6.1.4.3 and Section 6.1.4.4. The performance/price analysis of the CMOS part is referred to Section 5.2.1 and Section 6.4.

6.2.2. Spiking AM Model

This design is similar to the spiking all digital CMOS implementation, except that CMOL memory is used to hold the weight values as compared to using eDRAM in the spiking digital CMOS design. The CMOL memory performance/price modeling is referred to Section 6.1.4.3 and Section 6.1.4.4. For the performance/price analysis of the CMOS part is referred to Section 5.2.2 and Section 6.4.

6.3. IMPLEMENTATION WITH MIXED-SIGNAL CMOL

6.3.1. Non-spiking AM Model

In this configuration, we use CMOL CrossNets [46, 107] to represent the network connections (i.e., the weight matrix). Our usage is a variation of neuromorphic CMOL CrossNets [46, 107], with somewhat different CMOS cells and network topology. Due to the use of CMOL nanowires to represent the network connections, we refer to this configuration as *CMOL nanogrids*. Where CMOS circuits drive the input nanowires, the output nanowires connect to the inputs of the analog (CMOS) k-WTA circuits.

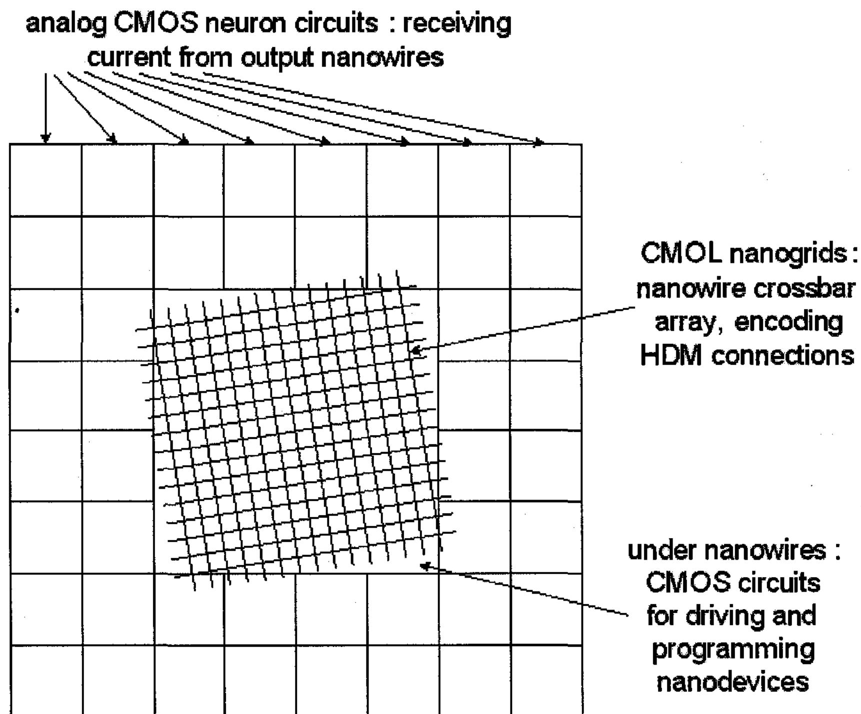


Figure 6-25: A structural view of the mixed-signal CMOL design.
The denser crossbar array in the center is a CMOL nanogrids (nanowire crossbar arrays). Beneath the CMOL nanogrids are the CMOS drive and programming circuits for the nanodevices. The larger square blocks are analog CMOS circuits for each output neuron.

Figure 6-25 shows the structure of the mixed-signal CMOL design – CMOL nanogrids. In this figure, the CMOL nanogrids sit in the center of the layout. The nanogrids are fabricated on top of the CMOS circuits, which are used for driving, programming, and reading the outputs of the nanodevices. The nanowires connect to the CMOS using the CMOL self-aligning architecture. Each input block of the analog *k*-WTA circuit represents a competing neuron. Because the analog circuits are assumed to only scale to 250 nm (due to the transistor sizing of analog circuits), instead of to 22 nm, the area for each neuron is about 12.5 um^2, which is much larger

than nanowire cells ($4F^2_{nano}$). An important characteristic of CMOL is that the CMOS circuits need $2N$ pins to connect to the $2N$ nanowires within the $4N^2F^2_{nano}$ area. This requires that $F_{\text{cmos}}/F_{\text{nano}} \leq \sqrt{N}$, where F_{cmos} is the half pitch of CMOS.

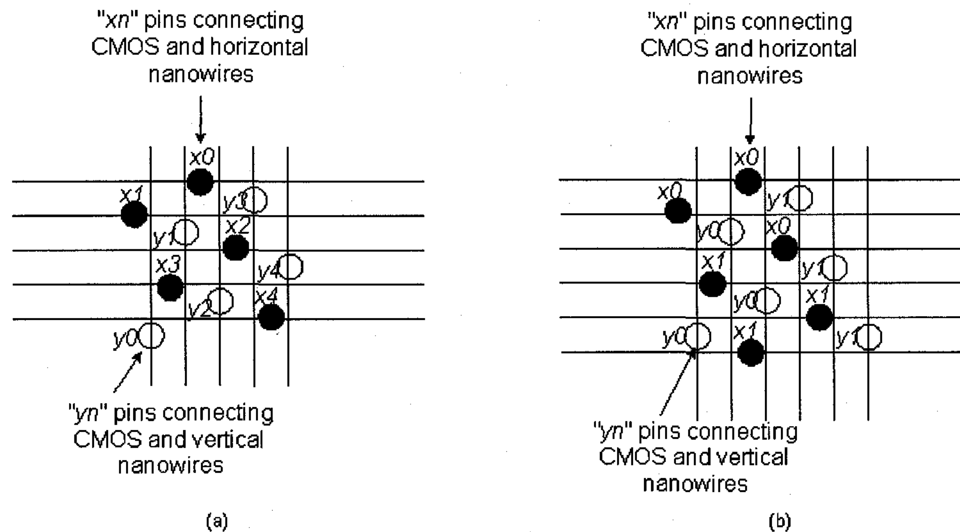


Figure 6-26: CMOL nanogrid interface between nanowires and CMOS.
(a) Single-bit CMOL nanogrid and pin connection diagram, where the "*x<n>*" are the driving pins from CMOS to nanowires, and "*y<n>*" are the pins connecting output nanowires and analog CMOS neuron circuits. (b) Multi-bit CMOL nanogrids and pin connection diagram. Each driving signal "*x<n>*" and output signal "*y<n>*" connects three nanowires in this diagram. The dark circles represent the pins connecting CMOS signals and horizontal nanowires. The hollow circles represent the pins for the vertical nanowires.

Figure 6-26 shows a schematic diagram of the CMOL nanogrids of Figure 6-25, only the layout of pins and nanowires is displayed. The dark circles represent pins connecting horizontal nanowires, which are the inputs to the nanogrid, to the top level of metal of the underlying CMOS. The hollow circles represent pins connecting vertical nanowires, which are the outputs from the nanogrid. In Figure 6-26 and Figure

6-27, *x0*, *x1*, ···, represent input nanowires; and *y0*, *y1*, ···, represent output nanowires. Figure 6-26 does not show the inter-wire molecular connections. Figure 6-27 is a schematic that includes these inter-grid devices. The small black dots at the cross-points of the nanowires are “on” nanodevices. The “off” nanodevices are not shown in the diagram. The positions of the “on” nanodevices are used to illustrate the current flow.

During the operation for single-bit-weight computation, input active nodes (CMOS circuits to emulate the input neurons) pull their nanowires to the input active voltage “high”; all output nodes (CMOS circuits to emulate the output neurons) pull their nanowires to voltage “low”. If there is a connection between the input neuron and the output neuron (i.e., the synapse value is “1”), which means that the nanodevice is in the “on” state, an “on” current will flow through the connection from the input neuron to the output neuron. The currents from different input neurons will be summed together to form a single output. As illustrated in Figure 6-27 (a), nanowire y0 sums three units of current.

Although auto-associative models (input and output vectors are same) work quite well with binary weights, there are situations, such as when we are doing dynamic adaptation, where we would like a few bits of precision so that we can do incremental learning. Because the nanodevices at the wire cross point can only take two states, we need $O(2^N)$ nanodevices to represent an N-bit weight. For example, if the weight has three bits, we need at least eight nanodevices to represent all values. This is illustrated in Figure 6-27 (b), where each input neuron and output neuron connects to three

nanowires. For a pair of input and output neurons, they have nine nanodevices to connect their nanowires. These nanodevices can then be programmed to represent the different values.

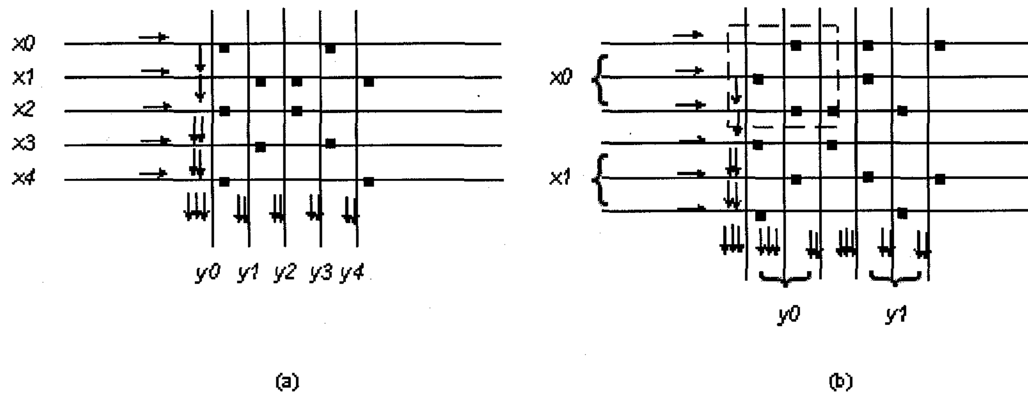


Figure 6-27: CMOL nanogrid weight bits.
(a) **Single-bit** CMOL nanogrids schematic diagram. (b) **Multi-bit** CMOL nanogrids schematic diagram. Here, for example, each input signal and each output signal connects a bundle of three nanowires, which can satisfy a 3-bit precision requirement.

There are other ways to implement multi-bit weights. Figure 6-28 shows a one-dimensional implementation of multi-bit weight. If the number of bits is N, then the area (complexity) of this method is $O(2^N)$. Figure 6-29 shows another way to implement the multi-bit weight in a CMOL nanogrid and with tighter a tighter CMOS connection, where, different nanowires output to different CMOS amplifiers with different gains. Although, this method seems to save some area in the CMOL nanogrid, it loses its area advantage to other methods because of the large area consumption of the CMOS amplifiers.

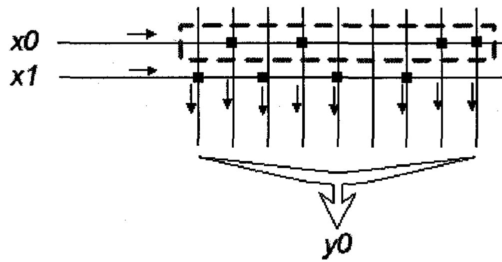


Figure 6-28: 1-D asymmetric multi-bit weight nanogrid implementation.

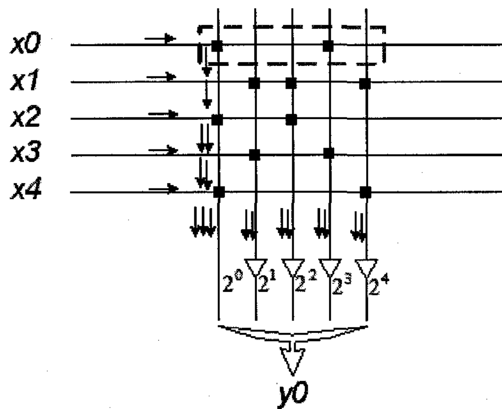


Figure 6-29: CMOS weighted multi-bit implementation, with complexity of *O*(*N*).

As mentioned by Türel *et al.* [107], Figure 6-30 shows one way to program multi-bit CMOL nanogrids. When programming the nanodevices, voltage differences A and B are added to the metallic resistors connecting to the horizontal nanowires and vertical nanowires, respectively. As shown in the picture, the slope angle of the "boundary" is $\alpha = \arctan A/B$, ($A, B \geq 0$, A and B are not both integers). The boundary is located at the point where the voltage is equal to the threshold voltage V_t. However, in order to be able to program each of the N^2 nanodevices, the boundary should avoid crossing

two or more nanodevices simultaneously. Thus, we have the constraint that A and B are not both integers at the same time.

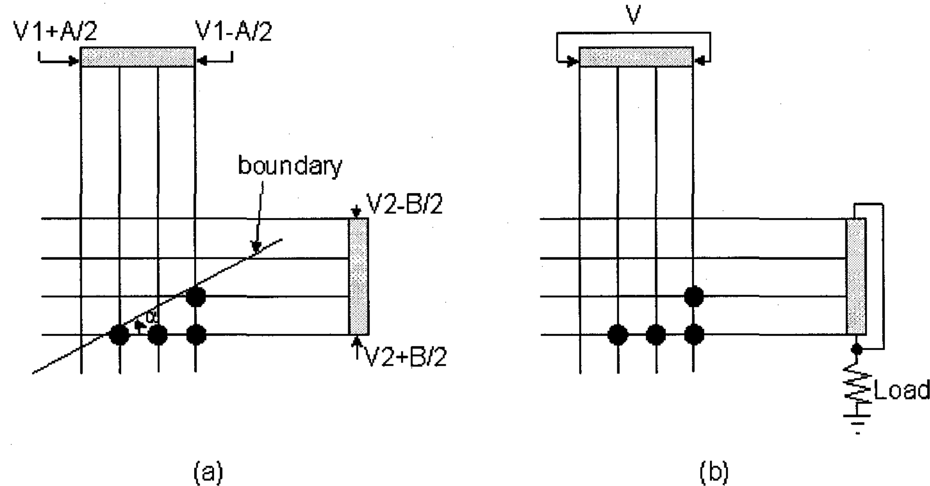


Figure 6-30: (a) Programming nanodevices with multi-bits. (b) Operation of CMOL nanogrids with multi-bits.
(Adapted from [107].)

One big advantage of CMOL nanogrids as they are used here is that they do not require the line encoding and decoding circuits of a memory. They not only provide memories for the synapses, but also implement the inner-product computations naturally. Furthermore, the CMOL nanogrids convert the digital data (voltages) to analog data (currents). This saves space for the D/A converters required in the mixed-signal CMOS design, and is why we only need to perform one computation (i.e., *k*-WTA) inside CMOS.

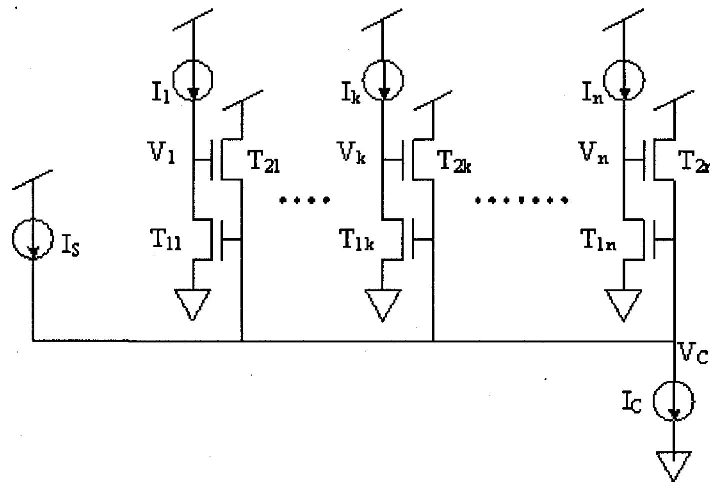


Figure 6-31: Mead *k*-WTA CMOS circuit.
(Adapted from [134].)

Figure 6-31 shows the Mead *k*-WTA circuit [134]. The circuit can adjust I_S adaptively to make sure V_1, V_2, …, V_k are high, others are almost zero, with input condition $I_1 > I_2 > \ldots > I_N$.

6.3.2. Spiking AM Model

Like the non-spiking mixed-signal CMOL design, we use CMOL nanogrids (Figure 6-25) to represent the network connections (i.e., the weight matrix). Pulses (current spikes) from the CMOS circuitry drive the CMOL output nanowires, which connect to the inputs of the analog I&F neuron circuits, which, as discussed earlier, is a variant of spiking models that is particularly well suited to the CMOL mixed signal design. Indiveri's [90] circuit implements the leaky I&F neuron, which could control the output firing rate by changing the bias voltage. Figure 6-32 shows the schematic view of this analog I&F neuron circuit.

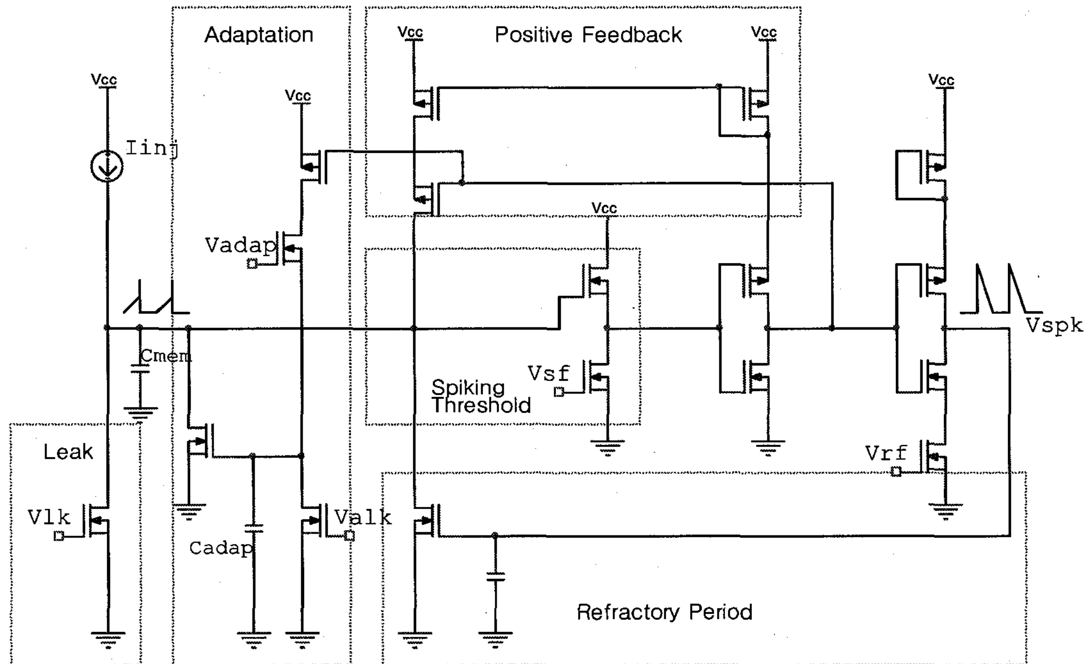


Figure 6-32: Schematic view of an analog integrate-and-fire neuron circuit. (Adapted from [90].)

Each CMOL nanogrid output nanowire connects to the input of the I&F neuron circuit, i.e., the I_{inj} in Figure 6-32. The current from the CMOL nanogrids output nanowire charges the capacitor C_{mem}, which represents the neurons somatic voltage potential. When the capacitor's voltage reaches the threshold, the circuit will generate an output spike, which will discharge the C_{mem}. As with real neurons, this circuit will generate spikes at a fixed rate if we have a continuous injection current.

6.4. PERFORMANCE/PRICE COMPARISONS BETWEEN CMOS AND CMOL

For non-spiking implementations, the components used by each of the four designs are shown in Table 6-1, in which a "Y" indicates where in the target system the component is used.

Table 6-1: Components for different systems of non-spiking AM model.

Circuit Component	Digital CMOS	Mixed-signal CMOS	Digital CMOL	Mixed-signal CMOL
SRAM/eDRAM	Y	Y		
Digital inner-product	Y	Y	Y	
Digital *k*-WTA	Y		Y	
D/A converters		Y		
Analog *k*-WTA		Y		Y
CMOL memory			Y	
CMOL CrossNet				Y

These designs are evaluated according their performance/price ratios, where performance is measured by speed (Connections Per Second, CPS, for the non-spiking model, or maximum input spiking rate for the spiking model). CPS is a traditional performance measure when emulating neural networks. Unfortunately, it is not as precise with the incremental, spike based models presented here, but the maximum spike processing rate still gives a reasonably good predictor of hardware performance. Price is measured by silicon area and power (regarding the total chip size of 858 mm^2, which is the maximum reticle field size expected at the 22 nm ITRS node [108]).

Table 6-3 lists the equations used to estimate the performance/price for each component in Table 6-1 and Table 6-2. For the CMOL circuit performance/price estimates, we refer to Section 6.1.4.4, and estimate the typical design density for a

number of circuits using examples from the literature: the digital k-WTA [155], the D/A converter [135], the CAM [132], the multiplier [156], and the adder [132]-p.678. We then scale these circuits to our hypothetical 22 nm technology according to the ITRS projections [108], using the first-order constant field scaling principle [132], where S is the scaling factor. We know that current scales as $1/S$, resistance as 1, gate capacitance as $1/S$, gate delay as $1/S$, frequency as S, chip area as $1/S^2$, and dynamic power dissipation as $1/S^2$. Analog circuits do not scale at the same pace as digital circuits, so we conservatively scaled the analog circuits to 250 nm.

Table 6-3 shows the area, power, and time delay scaling estimates for the various components. Our performance/price estimates cover a range of parallelism ("virtualization" as defined earlier), from a single PN for each neuron, to having a single PN multiplex all the neurons in the column. The estimates also explore variations in model parameters, such as network size, weight data precision, and sparseness of connections.

Table 6-2: Components for different systems of spiking AM model.

Circuit Component	Digital CMOS	Digital CMOL	Mixed-signal CMOL
eDRAM	Y		
SRAM	Y	Y	
CAM (PSP LUT)	Y	Y	
Multiplier	Y	Y	
Adder	Y	Y	
Analog I&F Neuron			Y
CMOL memory		Y	
CMOL CrossNet			Y

Table 6-3: Circuit performance/price scaling.

Circuit Component	Area (mm^2)	Power (W)	Time delay (ns)
SRAM	$(6N/2.85)\times10^{-7}$ (N total bits)	0.64 Area	0.025
eDRAM	$(2N/24.7)\times10^{-7}$ (N total bits)	0.64 Area	0.1
Digital adder	$a_t^{2 N\log_2 N / 32\log_2 32} 1.2\times10^{-2}$ (a_t=22/180, N-bits adder)	0.04 Area	$a_t^{\log_2 N / \log_2 32} 0.75$
Digital k-WTA	$0.23a_t^2 a_b a_f$ (a_t=22/350, a_b=n/8, a_f=m/8, m n-bit k-WTA)	0.04 Area	$15a_t a_f k$ (k winner count)
D/A converter	1.8 a_t, (a_t=22/80)	0.08 Area	2.2
Analog k-WTA	$1.25N\times10^{-5}$ (N size network)	$2.5N\times10^{-8}$	900
Digital multiplier	1.2×10^{-4} (16-bits × 16-bits)	1.7×10^{-5}	0.2
Digital CAM (PSP LUT)	$(9\log_2 k+6m)k\times10^{-7}/2.85$, where m the output data width, k record count	0.32 Area	0.05
Analog IF neuron	1.7×10^{-4}	5.7×10^{-3} 8.6×10^{-3} 2×10^{-2}, at 57, 140, 482Hz respectively	Average neuron spiking rate: 57Hz 140Hz 482Hz

6.5. RESULTS AND DISCUSSION

The resulting performance/price estimates are presented in two parts. Table 6-4 shows the performance/price comparisons of various architectures for the non-spiking model, while Table 6-5 shows the performance/price comparisons of various architectures for the spiking model.

In Table 6-4, the estimates are based on a model size (for a single column) of 16,384 neurons, with 4-bit weight resolution, 256 PNs per column processor, and eDRAM technology for the CMOS designs. The total chip size is 858 mm^2.

Table 6-4 shows that the CMOL designs have lower power consumption (by one to two orders of magnitude) than the CMOS designs, due to greatly reduced charging

power. Because the digital *k*-WTA circuit is at least ten times slower and ten times more costly in area than its analog counterpart, the CPS performance of mixed-signal CMOS and CMOL designs have roughly two orders of magnitude advantage over their digital counterparts.

We also estimated the performance/price with different algorithm parameters, for example with a network size of 1,024, and single-bit weights; the relative performance/price comparisons above are still valid [49]. We also find the update rate for CMOL mixed-signal design has seven orders of magnitude advantage over a scaled microprocessor (about 20 M nodes/sec for microprocessor when scaled down to 22 nm technology [41]). For the CMOL mixed-signal design, we can implement more than 1700 column processors (each with 16K-neuron network size) onto a single chip (if we neglect the silicon area for the inter-column communications.) *Such a capability cannot be approached by the full-custom CMOS designs, let alone by a single microprocessor or a single FPGA (expecting seven orders of magnitude of CPS improvement from the mixed-signal CMOL implementation over the PC implementation).*

Table 6-4: Performance/price comparison for the non-spiking AM model.

Design	# Column Processors	Power (W)	CPS (10^{12}connections/s)
Digital CMOS	123	330	0.5
Mixed-signal CMOS	195	507	41.5
Digital CMOL	226	27	1.0
Mixed-signal CMOL	1,716	1.4	365.3

For the spiking CMOL and CMOS designs, we compared the input spiking rate (i.e., the maximum input spiking rate that the chip can process), power, and the number of column processors on a chip based on digital CMOS, digital CMOL, and mixed-signal CMOL designs. The performance/price here means the spiking rate of a chip size of 858 mm^2.

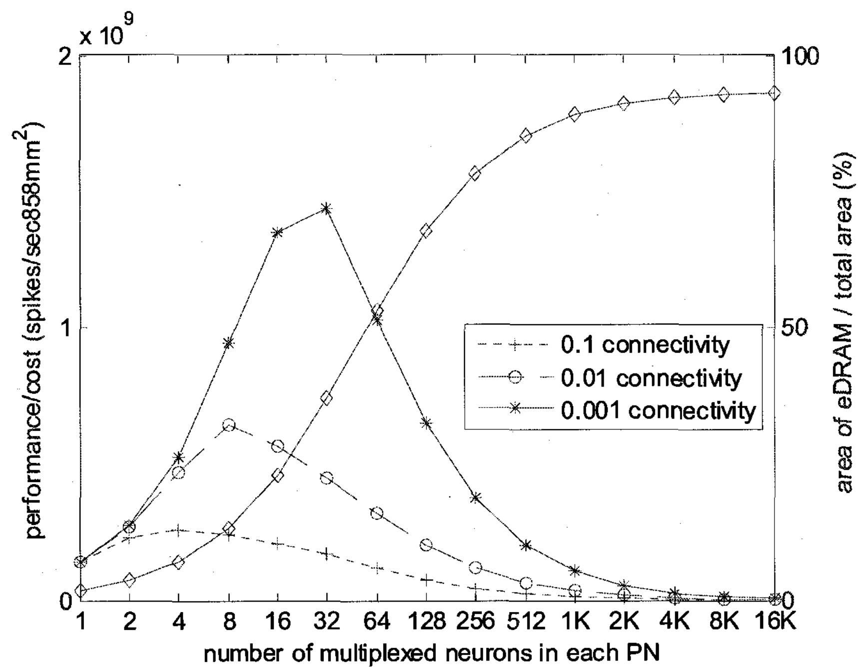


Figure 6-33: A log-log plot of the input spiking rate of the digital CMOS design for an 858 mm^2 chip with three different levels of connectivity.
The "diamond" marked curve shows the percentage of the area that is consumed by the eDRAM.

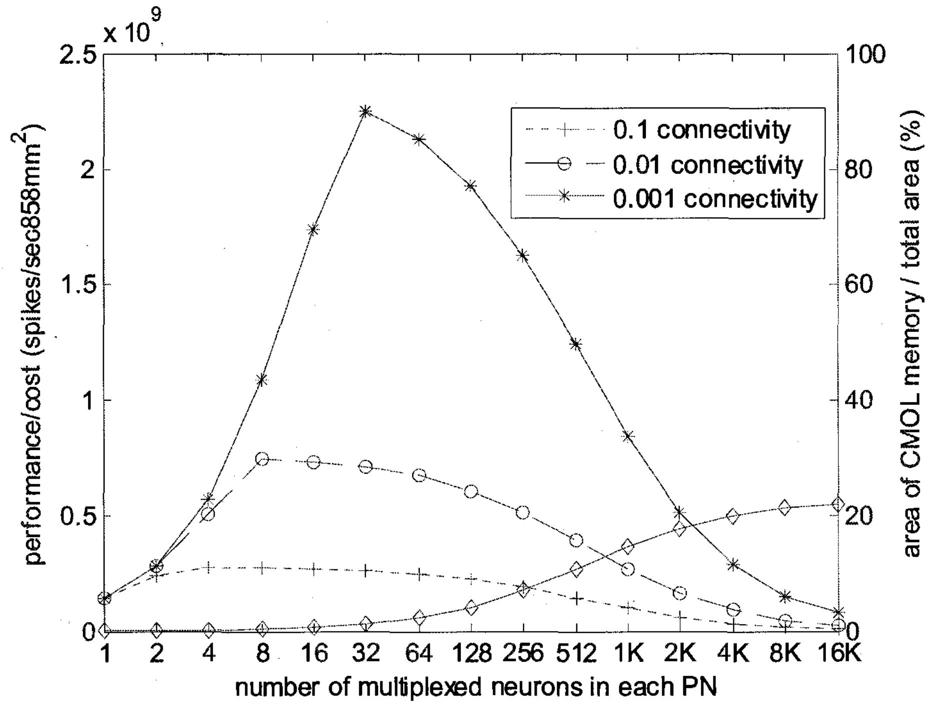


Figure 6-34: A log-log plot of the input spiking rate of the digital CMOL design for an 858mm² chip with three scenarios of connectivity.
The "diamond" marked curve shows the area percentage of the CMOL memory.

Figure 6-33 and Figure 6-34 show the input spiking rates per chip for digital CMOS and digital CMOL, respectively. With less connectivity, the PN should be able to multiplex more neurons (total connections tends to be a more important indicator than number of neurons), and the whole chip can process a higher input spiking rate. For example, in Figure 6-33, for 0.1 connectivity, the highest input spiking rate occurs when four neurons are multiplexed by each PN. For 0.01 connectivity, the highest input spiking rate occurs when 32 neurons are multiplexed by each PN. With more multiplexed neurons in each PN, the weight memory (eDRAM and CMOL memory)

occupies a greater proportion of the chip area as fewer PNs are needed. This is an issue in the CMOS design when the eDRAM area approaches 90% of the whole chip with maximum neuron multiplexing (all neurons being emulated by one PN). CMOL memory is slower than eDRAM, but occupies much less silicon area. Figure 6-34 shows the improved performance/price of a digital CMOL design over the digital CMOS design (about 50% improvement).

Table 6-5 shows the performance/price results of the spiking AM models for the digital CMOS and mixed-signal CMOL designs, assuming the same benchmark input spiking rate for both designs. The benchmark input spiking rate is the maximum input spiking rate the digital CMOS can process under the three different connectivity levels used in Figure 6-33. Although the power consumption of the mixed-signal CMOL design increases with the input spiking rate, it has at least two orders of magnitude of advantage (in terms of the *Power* × *Input Spiking Rate*) over the digital CMOS design under the same network conditions. On the other hand, we can also see a much narrower performance/price gap between the digital CMOS and mixed-signal CMOL implementations for the spiking model than for the non-spiking model. This is due primarily to hardware virtualization.

The dynamic power dissipated by the CMOL memory in the nanowire/nanodevice crossbars is defined by (6.18). If the horizontal and vertical nanowires are $N = M = 10^3$, the connectivity $b = 0.1$, nanogrid half pitch $F_{\text{nano}} = 3$ nm, applied voltage $V = 1$ V, in order to satisfy the power density < 0.64 W/mm^2, we have the constraint of $2R_{con} + 1.5R_{wire} + R_{on}/D > 8.6\,M\Omega$. If we increase the nanogrid size by 1000 times,

that is, $N = M = 10^6$, the constraint will be $2R_{con} + 1.5R_{wire} + R_{on}/D > 8.6\,k\Omega$. These are reasonable constraints. On the other hand, the time delay defined by (6.14) degrades when the nanowire length increases. This means that when the CMOL nanogrids footprint increases, the dynamic power density decreases, while the time delay increases.

Table 6-5: Performance/price comparison for spiking AM model.

Design	Connectivity	# Column Processors	Power (W)	Input Spiking Rate / Chip (10^9spikes/sec·858mm^2)
Digital CMOS	0.1	15	481	0.26
MS* CMOL	0.1	276	1.8	0.26
Digital CMOS	0.01	28	482	0.65
MS CMOL	0.01	276	2.7	0.65
Digital CMOS	0.001	780	490	2.2
MS CMOL	0.001[11]	276	6.2	2.2

* MS stands for Mixed-Signal

Digital CMOS circuits need D/A converters to interface with analog CMOS circuits, which are expensive in both area and power. The mixed-signal CMOL design does not require converters. Currents from CMOL nanogrids can feed directly into analog circuits, such as the *k*-WTA (illustrated in Figure 6-31) and the I&F neuron (see Figure 6-32). The average injection current determines the analog circuit's dynamic response. For example, the I&F circuit requires at least 10 *p*A of injection current to spike at 10 Hz. The nanowire connecting the CMOL to the input node of the I&F neuron circuit can provide such current. The CMOL power density is $\frac{NVI_{inj}}{2^n N^2 4F^2_{nano}} < 0.64W/\text{mm}^2$, which leads to the constraint $2^n N > 0.43$, where *n* is the

[11] Too less connectivity in the WPNAM model can cause remarkably smaller capacity.

weight bits, V = 1 V, F_{nano} = 3 nm, and I_{inj} = 10 pA. CMOL nanogrids can easily satisfy this constraint. However, if there is sparse connectivity, the power density of the hot spots (i.e., where the "on" nanodevices are located) is $\frac{NVI_{inj}}{\varsigma 2^n N^2 4F^2_{nano}} < 0.64W/\text{mm}^2$, where ς is the connectivity. This gives the constraint of $2^n N > 430$ with $\varsigma = 0.001$.

Another nanodevice average power density constraint derived from CMOL nanogrids operation is given by $\overline{Power}/Area = DT_d V^2/R_{on}F^2_{nano} < 0.64\,\text{W}/\text{mm}^2$, where T_d is the duty cycle defined as T_d = (spike width) × (average spike rate per neuron). For $T_d = 0.01$, $R_{on}/D > 4.3\times10^8\Omega$, which might be possible for single electron molecules [157]. However, R_{on} should not be too high, otherwise it will degrade the dynamic response of the CMOL CrossNets, given (6.14).

The possibilities created by hybrid CMOS / nanogrid electronics are very exciting, especially in the area of neural model emulation. An important conclusion of the architectural trade-offs presented here is the value of leveraging sparse activation and connectivity to multiplex scarce resources. We have demonstrated that, because of the sparse activation and sparse connectivity of our models, a simple time-multiplexing scheme for digital CMOS can achieve comparable throughput as a mixed-signal CMOL configuration while using the same silicon area (Table 6-5), although this approach does consume more power (but all digital CMOS is much easier to debug, manufacture, and test).

Furthermore, when we begin to add dynamic learning to our architecture, fine-grained, dedicated per synapse hardware that incorporates learning will be expensive and under-utilized. Hardware virtualization will improve the efficiency when emulating the dynamic adaptation.

7. SUMMARY AND FUTURE WORK

7.1. SUMMARY AND CONCLUSIONS

In this dissertation, we motivated why we are interested in HDM models, that it has significant potential as a building block for intelligent systems. It is inspired by and is based on high level models of cerebral cortex. We also believe that it strikes a reasonable compromise between implementing neuron level details and model abstraction. In this work, we used associative memory, in particular the models developed by Wilshaw and Palm to approximate a Bayesian Memory which is the assumed high level functionality of a single node in an HDM. We used Wilshaw/Palm model (WPNAM) for the non-spiking model and Gerstner's spiking neuron model for the spiking associative memory model.

When conducting this research, we used the methodology in Figure 2-17 to investigate the hardware architectures' performance/price ratios for the target algorithms. This methodology is very important for exploring hardware architectures due to the following reasons:

- For any compute intensive models, we need to understand the critical operations in the models, and whether they can be implemented in a parallel fashion. And while doing that we need to keep Amdahl's law in mind. Steps (1) and (2) in our methodology (Figure 2-17) address these concerns.

- The methodology creates a formalism for studying the entire virtualization spectrum, Figure 2-10, in search of the sweet spot. This is an important contribution of this work, since the neural network field has traditionally gone from standard processors to maximum, usually mixed signal, parallelism, never stopping to look at intermediate points on the spectrum.
- Although we made assumptions on the parameters of the algorithms and based the subsequent analyses on those versions of the algorithms, it would not be difficult to take the methodology and the various steps taken here and redo the various analyses with very different parameters.

In addition to a wider range of WPNAM networks, many of these ideas can be used in exploring a broader set of algorithms. It will also continue to serve as an important tool for our group to conduct further study of hardware architectures for biologically-inspired computations.

With this tool (methodology) in hand, we did the performance/price comparisons for hardware implementations of AM algorithms. For the non-spiking AM algorithm, or WPNAM, for the CMOL mixed-signal design, we can implement more than 1700 column processors onto a single chip (if we neglect the silicon area for the inter-column communications.) That is about 3×10^{14} connections per second. These are densities and speeds that are approaching biology. The mixed-signal CMOS design is dwarfed by mixed-signal CMOL's performance; with 10× less CPS and 100× more power consumption. Furthermore, the mixed-signal CMOL implementation is

10,000,000× faster than the PC's in CPS. This makes a strong case for investing in CMOL technology, if for no other reason than to move intelligent computing to the next level.

For spiking AM models, the mixed-signal CMOL implementation is also the winner in the race with digital CMOS implementation. Although the power consumption of the mixed-signal CMOL design increases with the input spiking rate, it shows at least two orders of magnitude of advantage (in the measure of *Power* × *Input Spiking Rate*) over the digital CMOS design under the same network conditions. However, we also notice a much narrower performance/price gap between digital CMOS and mixed-signal CMOL implementations for the spiking model than for the non-spiking model. Digital CMOS designs leverage sparse connectivity and activation ("virtualization") to balance power consumption and performance speedup. As discussed in Section 5.2, we explored virtualization by multiplexing PN resources for different number of neurons. With the analysis and equations in Section 5.2, we demonstrated a method for calculating the virtualization performance gain against a measure of normalized time spent. Although we did not provide sweet spots of how much virtualization we need for different implementations in a variety of applications, we did show certain levels of virtualization for the specific digital CMOS / CMOL cases for the spiking AM algorithms in Figure 6-33 and Figure 6-34.

Recall the discussion on front end and back end operations in Section 2.4.2, when we move to emulate the more "intelligent" or ISP-based HDM algorithms; we need a huge amount of memory storage for the diffuse data from the front end sensors. For

example, to implement 226 column processors with the digital CMOL architecture, as shown in Table 6-4, we need to store one trillion connections in the hardware. Because each connection needs 4 bits to represent the weight, we need to store totally 4×10^{12} bits in a single chip. Nanoelectronic architectures provide a huge density advantage when implementing such a large amount of data in the hardware. This is proved in Figure 6-33 and Figure 6-34, where for the digital CMOS architecture more than 90% of the silicon area is devoted to memory to store the network connections (weights). However, this same number for the digital CMOL architecture is only about 25%, which is a better balanced use of chip real estate.

Also, when we look at the mixed-signal CMOL's performance/price results for the non-spiking AM algorithm in Table 6-4 and neocortex data from the human brain in Table 2-1, it turns out that we only need 711 such mixed-signal CMOL chips (each one is 858 mm^2 in area) to achieve the 2.0×10^{10} neuron count in human neocortex. However, with that many CMOL chips, we can update each neuron's state in 1 μs. This is 10,000x faster than human neocortex, because Lansner *et al.* [158] stated that in human neocortex, each neuron updates their state in 10 ms. Such a speed advantage could also be traded-off for lower power utilization.

We do not mean to imply that 711 chips would recreate the functionality of the human brain. Our cortical models are a significant simplification of real biology and there are large parts of the brain where we do not even have such high level models. However, this studying is promising in that CMOS / nanogrid hardware is getting into the closer all the time to "Reverse-Engineer the Brain" – one of the fourteen Grand Challenges

for Engineering determined by the National Academy of Engineering (http://www.engineeringchallenges.org).

CMOL not only provides a high-density storage medium for the network connections, but also integrates the multiply-accumulate-based matrix-vector inner-product operation into its architecture. This natural mapping of biologically-inspired computational models' most common operation (inner-product) onto hardware architectures could be useful for a broad range of bio-inspired and more traditional computational models. And would be of interest to researchers from areas such as neuroscience, computer architecture, semiconductor, and applied physics.

Although the CMOL nanoarchitecture has its own speed disadvantage (see the end of Section 6.1.4.4) and power density issues (see the end of Section 6.5), utilizing nanoarchitecture's circuit density to emulate the high-level HDM algorithms, especially those with sparse input activity and sparse connectivity, was shown to be of significant value. On the other hand, nanoarchitecture, due to the nature of its fault-prone nanodevices, benefits from asynchronous, low precision, massively parallel, fault-tolerant HDM algorithms.

7.2. FUTURE WORK

We believe that this work is a useful first step in architecting and implementing hardware for executing algorithms inspired by neuroscience. Consequently there are many directions where we can go from here. These include:

- Dynamic learning embedded into the hardware. To fully emulate neocortex, we need to integrate the adaptation of the connection efficacy along with the network's updating or memory retrieval. Only then, we can compare the performance/price ratios from different hardware implementations with the fully functional biology-like ability and intelligence. This is probably the most important short-term objective.

- Multi-core implementation's performance/price ratio for AM. Although we have included the PC cluster implementation in this dissertation, the latest developments from the computer industry, especially multi-core processor chips with very high memory bandwidth need to be benchmarked and integrated into our results. The most flexibility of the multi-core microprocessor is a very important thing that we should not neglect in this performance/price war among different hardware candidates. This statement is also true for graphics processors, GPUs.

- Heterogeneous CMP (chip multiprocessor) architectures. An important variation on multi-core CMP architectures concerns moving to heterogeneous

cores for accelerator cores that incorporate many of the ideas presented in this dissertation.

- More complex associative memory models (such as the Bayesian memory model in Zaveri *et al.*'s work [40]) are needed, since the models here, although an important first step, do not implement all the functionality required of modular cortical models.

8. REFERENCES

[1] K. K. Likharev and D. V. Strukov, "CMOL: devices, circuits, and architectures," in *Introduction to Molecular Electronics*, G. Cuniberti and et al., Eds. Berlin: Springer, 2005, pp. 447-478.

[2] K. Likharev, "CMOL technology: devices, circuits, architectures, and possible applications," Stony Brook University, Stony Brook, NY 2008, http://pavel.physics.sunysb.edu/~likharev/personal/CMOL08.pdf.

[3] Intel, "60 Years of the transistor: 1947-2007," 2008, http://www.intel.com/technology/timeline.pdf.

[4] D. Hammerstrom, "A survey of bio-inspired and other alternative architectures," in *Nanotechnology Information Technology - II*, vol. 2. Weinheim, Germany: Wiley-VCH Verlag GmbH & Co., 2008, pp. 251-285.

[5] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," *Technology Intel Magazine*, pp. 1-10, 2005.

[6] J. Xiang, W. Lu, Y. Hu, Y. Wu, H. Yan, and C. M. Lieber, "Ge/Si nanowire heterostructures as high-performance field-effect transistors," *Nature*, vol. 441, pp. 489-493, 2006.

[7] A. B. Greytak, L. Lauhon, M. S. Gudiksen, and C. Lieber, "Growth and transport properties of complementary germanium nanowire field-effect transistors," *Appl Phys. Lett.*, vol. 84, pp. 4176-4178, 2004.

[8] S. Zankovych, T. Hoffmann, J. Seekamp, J.-U. Bruch, and C. M. S. Torres, "Nanoimprint lithography: challenges and prospects," *Nanotechnology*, vol. 12, pp. 91-95, 2001.

[9] D. J. Resnick, W. J. Dauksher, D. Mancini, K. J. Nordquist, T. C. Bailey, S. Johnson, N. Stacey, J. G. Ekerdt, C. G. Willson, and S. V. Sreenivasan, "Imprint lithography for integrated circuit fabrication," *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures*, vol. 21, pp. 2624, 2003.

[10] R. I. Bahar, D. Hammerstrom, J. Harlow, W. H. Joyner Jr., C. Lau, D. Marculescu, A. Orailoglu, and M. Pedram, "Architectures for silicon nanoelectronics and beyond," *Computer*, vol. 40, pp. 25-33, 2007.

[11] E. Rechtin, "The art of systems architecting," *IEEE Spectrum*, vol. 29, pp. 66-69, 1992.

[12] J. Hawkins and D. George, "Hierarchical temporal memory - concepts, theory and terminology," Numenta Inc. 2006, http://www.numenta.com/Numenta_HTM_Concepts.pdf.

[13] M. Djurfeldt, M. Lundqvist, C. Johansson, M. Rehn, Ö. Ekeberg, and A. Lansner, "Brain-scale simulation of the neocortex on the IBM Blue Gene/L supercomputer," *IBM J. Res. & Dev.*, vol. 52, pp. 31-41, 2008.

[14] R. Hecht-Nielsen, "Tutorial: Cortronic Neural Networks," presented at International Joint Conference on Neural Networks, Washington, DC, 1999.

[15] M. Holler, S. Tam, H. Castro, and R. Benson, "An electrically trainable artificial neural network (ETANN) with 10240 "floating gate" synapses," presented at International Joint Conference on Neural Networks, pp. 191-196, 1989.

[16] A. Lansner and others, "Detailed Simulation of Large Scale Neural Networks," in *Computational Neuroscience: Trends in Research 1997*, J. M. Bower, Ed. Boston, MA: Plenum Press, 1997, pp. 931-935.

[17] G. Palm, F. Schwenker, F. T. Sommer, and A. Strey, "Neural associative memories," in *Associative Processing and Processors*. Los Alamitos, CA.: IEEE Computer Society, 1997, pp. 284-306.

[18] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins, "Non-holographic associative memory," *Nature*, vol. 222, pp. 960-962, 1969.

[19] D. Willshaw and B. Graham, "Improving Recall From An Associative Memory," *Biological Cybernetics*, vol. 72, pp. 337-346, 1995.

[20] D. Willshaw, "Marr's theory of the neocortex as a self-organizing neural network," *Neural Computation*, vol. 9, pp. 911-936, 1997.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Palo Alto, CA: Morgan Kaufmann, 2002.

[22] C. Mead, *Analog VLSI and Neural Systems*. Reading, Massachusetts: Addison-Wesley, 1989.

[23] R. Douglas, "Fifteen years of Neuromorphic Engineering: progress, problems, and prospects," in *Brain Inspired Cognitive Systems - BICS2004*. Scotland, UK: University of Stirling, 2006.

[24] J. N. H. Heemskerk, J. Hoekstra, J. J. J. Murre, L. H. J. K. Kamna, and P. T. W. Hudson, "The BSP400: A Modular Neurocomputer," *Mircroprocessors and Microsystems*, vol. 18, pp. 67-78, 1994.

[25] H. Speckman, P. Thole, and W. Rosentiel, "COKOS: A Coprocessor for Kohonen's Selforganizing Map," presented at Proceedings of the ICANN-93, Amsterdam, pp. 1040-1045, 1993.

[26] D. Hammerstrom, "Techniques for the Efficient Execution of Sparse Matrix Neural Networks Algorithms on SIMD Machines," Adaptive Solutions, Inc., Beaverton, OR July 1992 1992.

[27] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Brüls, M. Weßeling, E. Sicheneder, R. Männer, J. Gläß, and A. Wurz, "Multiprocessor and Memory Architecture of the Neurocomputer SYNAPSE-1," presented at World Congress on Neural Networks, Portland, OR, pp. 775-778, 1993.

[28] M. Huch, W. Pöchmueller, and M. Glesner, "BACCHUS: a VLSI architecture for a large binary associative memory," presented at Proceedings of the International Neural Network Conference, Paris, 1990.

[29] Y. Huang, X. Duan, L. J. Lauhon, K. Kyoung-Ha, and C. M. Lieber, "Logic gates and computation from assembled nanowire building blocks," *Science*, vol. 294, pp. 1313-1317, 2001.

[30] Z. Zhong, D. Wang, Y. Cui, M. W. Bockrath, and C. M. Lieber, "Nanowire crossbar arrays as address decoders for integrated nanosystems," *Science*, vol. 302, pp. 1377-1379, 2003.

[31] P. J. Kuekes and R. S. Williams, "Molecular wire transistor (MWT)," in *United States Patent*, vol. US 6,559,468 B1. United States: Hewlett-Packard Development Company LP, Houston, TX (US), 2003.

[32] G. S. Snider, P. J. Kuekes, and R. S. Williams, "CMOS-like logic in defective, nanoscale crossbars," *Nanotechnology*, vol. 15, pp. 881-891, 2004.

[33] X. Duan, Y. Huang, J. Wang, and C. M. Lieber, "Indium phosphide nanowires as building blocks for nanoscale electronic and optoelectronic devices," *Nature*, vol. 409, pp. 66-69, 2001.

[34] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, "Nanoscale molecular-switch crossbar circuits," *Nanotechnology*, vol. 14, pp. 462-468, 2003.

[35] P. J. Kuekes, D. R. Stewart, and R. S. Williams, "The crossbar latch: logic value storage, restoration, and inversion in crossbar circuits," *Journal of Applied Physics*, vol. 97, pp. 034301-1-5, 2005.

[36] K. Likharev, "Hybrid Semiconductor-Molecular Nanoelectronics," *Industrial Physicist*, vol. 9, pp. 20-23, 2003.

[37] K. K. Likharev, "CMOL: A New Concept for Nanoelectronics," presented at 12th Int. Symp. "Nanostructures: Physics and Technology", St Petersburg, Russia, 2004.

[38] G. F. Cerofolini, "Realistic limits to computation I. Physical limits," *Applied Physics A*, vol. 86, pp. 23-29(7), 2007.

[39] W. Gerstner, "Spiking Neurons," in *Pulsed Neural Networks*, W. Maass and C. M. Bishop, Eds. Cambridge, MA: MIT Press, 1998, pp. 3-53.

[40] M. S. Zaveri and D. Hammerstrom, "Nano/CMOS implementations of cortical model based on Bayesian memory - a generic architecture assessment methodology," *IEEE Transactions on Nanotechnology*, vol. (submission number: TNANO-00261-2008), 2008.

[41] C. Gao and D. Hammerstrom, "Platform Performance Comparison of PALM Network on Pentium 4 and FPGA," presented at International Joint Conference on Neural Networks, Portland, Oregon, pp. 995-1000, 2003.

[42] D. Hammerstrom, C. Gao, S. Zhu, and M. Butts, "FPGA implementation of very large associative memories - scaling issues," in *FPGA Implementations of Neural Networks*, A. Omondi, Ed. Boston: Kluwer Academic Publishers, 2003.

[43] C. H. Luk, C. Gao, D. Hammerstrom, M. Pavel, and D. Kerr, "Biologically inspired enhanced vision system (EVS) for aircraft landing guidance," presented at International Joint Conference on Neural Networks, Budapest HUNGARY, pp. 1751-1756, 2004.

[44] G. Snider and R. Williams, "Nano/CMOS architectures using a field-programmable nanowire interconnect," *Nanotechnology*, vol. 18, pp. 1-11, 2007.

[45] M. LaPedus, "HP claims FPGA breakthrough," in *EE Times (online)*, vol. 2007: EE Times, 2007, http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=196901012.

[46] Ö. Türel and K. Likharev, "CrossNets: possible neuromorphic networks based on nanoscale components," *Int. J. of Circ. Theor. Appl.*, vol. 31, pp. 37-53, 2003.

[47] C. Gao and D. Hammerstrom, "Cortical models onto CMOL and CMOS - architectures and performance/price," *IEEE Tran. on Circuits and Systems - I: Regular Papers - Special Issue on Nanoelectronic Circuits and Nanoarchitectures*, vol. 54, pp. 2502-2515, 2007.

[48] C. Gao, M. S. Zaveri, and D. Hammerstrom, "CMOS/CMOL architectures for spiking cortical column," presented at IEEE International Joint Conference on Neural Networks (IJCNN 08), Hong Kong, pp. 2441-2448, 2008.

[49] C. Gao and D. Hammerstrom, "CMOL based cortical models," in *(accepted for publication) Emerging Brain-Inspired Nano-Architectures*, V. Beiu and U. Rückert, Eds., 2006.

[50] W. C. Elmore, "The transient response of damped linear networks," *Journal of Applied Physics*, vol. 19, pp. 55-63, 1948.

[51] R. Ananthanarayanan and D. S. Modha, "Anatomy of a cortical simulator," presented at ACM/IEEE Conf. on High Performance Networking and Computing: Supercomputing, Reno, NV, 2007.

[52] V. Beiu, "Grand challenges of nanoelectronics and possible architectural solutions: what do Shannon, von Neumann, Kolmogorov, and Feynman have to do with Moore," presented at the 37th International Symposium on Multiple-Valued Logic (ISMVL '07), Oslo, Norway, 2007.

[53] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*: Kluwer Academic Press / Springer, 1992.

[54] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509-517, 1975.

[55] A. Andoni and P. Indyk, "Near-optimal Hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, pp. 117-122, 2008.

[56] G. Palm, "On associative memory," *Biological Cybernetics*, vol. 36, pp. 19-31, 1980.

[57] J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of National Academy of Science*, vol. 79, 1982.

[58] G. Zeng and D. Hammerstrom, "Distributed Associative Neural Network Model Approximates Bayesian Inference," ECE Department, OGI School of Science and Engineering, OHSU, Beaverton, OR 2002.

[59] V. Braitenberg and A. Schüz, *Cortex: Statistics and Geometry of Neuronal Connectivity*: Springer-Verlag, 1998.

[60] C. Johansson and A. Lansner, "Towards cortex sized attractor ANN," in *LNCS -- Biologically Inspired Approaches to Advanced Information Technology*, vol. 3141. Berlin, Germany: Springer Verlag, 2004, pp. 63-79.

[61] D. O. Hebb, *The Organization of Behavior*. New York: Wiley, 1949.

[62] V. Mountcastle, *Perceptual Neuroscience - The Cerebral Cortex*. Cambridge, MA: Harvard University Press, 1998.

[63] D. O'Kane and A. Treves, "Why the Simplest Notion of Neocortex as an Auto-associative Memory Would Not Work," *Network*, vol. 3, pp. 379-384, 1992.

[64] R. Hecht-Nielsen, "A theory of thalamocortex," in *Computational Models for Neuroscience – Human Cortical Information Processing*, R. Hecht-Nielsen and T. McKenna, Eds.: Springer, 2003.

[65] J. A. Anderson, "Programming Considerations for a Brain-Like Computer," Dept. of Cognitive and Linguistic Sciences, Brown University, Providence, RI 02912 June 14 2005 2005, www.cog.brown.edu/Research/ErsatzBrainGroup/index.html

[66] C. Johansson and A. Lansner, "Towards cortex sized artificial nervous systems," presented at Knowledge-Based Intelligent Information and Engineering Systems KES'04, Wellington, New Zealand, pp. 959-966, 2004.

[67] C. Johansson, M. Rehn, and A. Lansner, "Attractor neural networks with patchy connectivity," *Neurocomputing*, vol. 69, pp. 627-633, 2006.

[68] C. Fulvi Mari, "Extremely Dilute Modular Neuronal Networks: Neocortical Memory Retrieval Dynamics," *Journal of Computational Neuroscience*, vol. 17, pp. 57-79, 2004.

[69] R. Granger, "Brain circuit implementation: high-precision computation from low-precision components," in *Replacement Parts for the Brain*, T. Berger and D. Glanzman, Eds.: MIT Press, 2005, pp. 277-294.

[70] D. George and J. Hawkins, "A hierarchical Bayesian model of invariant pattern recognition in the visual cortex," presented at IJCNN '05, pp. 1812-1817 vol. 3, 2005.

[71] R. Hecht-Nielsen, "The Mechanism of Thought," presented at IJCNN '06, pp. 419-426, 2006.

[72] S. Zhu, "Associative memory as a Bayesian building block," Ph.D. dissertation, OGI School of Science and Engineering, Oregon Health and Science University, Beaverton, Oregon 2008.

[73] B. Kosko, "Bidirectional associative memories," *IEEE Transactions on Systems, Man, Cybernetics*, vol. 18, pp. 49-60, 1988.

[74] T. Dean, "Learning invariant features using inertial priors," *Annals of Mathematics and Artificial Intelligence*, vol. 47, pp. 223-250, 2006.

[75] T. S. Lee and D. Mumford, "Hierarchical bayesian inference in the visual cortex," *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 20, pp. 1434-1448, 2003.

[76] F. V. Jensen, *Bayesian Networks and Decision Diagrams*: Springer, 2001.

[77] J. Pearl and S. Russell, "Bayesian Networks," in *Handbook of Brain Theory and Neural Networks*, M. Arbib, Ed. Cambridge MA: MIT Press, 2001.

[78] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1988.

[79] Numenta, "Introduction to Numenta technology," 2007, http://www.numenta.com/about-numenta/numenta-technology.php.

[80] S. Zhu and D. Hammerstrom, "Associative Memory and Bayesian Classification," *Submitted for Publication*, 2007.

[81] S. Haykin and B. Kosko, "Intelligent Signal Processing," in *Proceedings of the IEEE*, vol. 86, 11 ed: IEEE Press, 1998.

[82] G. Palm and F. T. Sommer, "Information capacity in recurrent McCulloch-Pitts networks with sparsely coded memory states," *Network*, vol. 3, pp. 177-186, 1992.

[83] D. Willshaw and B. Graham, "Improving Recall from an Associative Memroy," *Biological Cybernetics*, vol. 72, pp. 337-346, 1995.

[84] S. Amari, "Neural Theory of Association and Concept Formation," *Biol. Cybern.*, vol. 26, pp. 175-185, 1977.

[85] J. D. Lynch and D. Hammerstrom, "Triplex micropipelines: A fault-tolerant clockless processing architecture," *submitted to IEEE Transactions on VLSI Systems*, 2008.

[86] A. Knoblauch and G. Palm, "Pattern separation and synchronization in spiking associative memories," *Neural Networks*, vol. 14, pp. 763-780, 2001.

[87] A. Knoblauch and G. Palm, "Spiking associative memory and scene segmentation by synchronization of cortical activity," *Lecture Notes in Computer Science*, vol. 2036, pp. 407-411, 2001.

[88] R. E. Suri, "A computational framework for cortical learning," *Biol. Cybern.*, vol. 90, pp. 400-409, 2004.

[89] G. Indiveri, "A low-power adaptive integrate-and-fire neuron circuit," pp. IV-820-IV-823 vol.4, 2003.

[90] G. Indiveri, E. Chicca, and R. Douglas, "A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *IEEE Transactions on Neural Networks*, vol. 17, pp. 211-221, 2006.

[91] R. P. N. Rao, "Hierarchical Bayesian inference in networks of spiking neurons," *NIPS'04*, vol. 17, 2005.

[92] S. Song, K. D. Miller, and L. F. Abbott, "Competitive hebbian learning through spike-timing-dependent synaptic plasticity," *Nature Neurosience*, vol. 3, pp. 919-926, 2000.

[93] J. Bailey, "A VLSI Interconnect Strategy for Biologically Inspired Artificial Neural Networks," Ph.D. dissertation, Department of Computer Science/Engineering, Oregon Graduate Institute, Beaverton, OR 1993.

[94] J. Bailey and D. Hammerstrom, "Why VLSI Implementations of Associative VLCNs Require Connection Multiplexing," *1988 International Conference on Neural Network*, pp. 173-180, 1988.

[95] R. Figueiredo, P. A. Dinda, and J. Fortes, "Guest Editors' Introduction: Resource Virtualization Renaissance," *Computer*, vol. 38, pp. 28-31, 2005.

[96] H. C. Card, D. K. McNeill, and C. R. Schneider, "Analog VLSI circuits for competitive learning networks," *Analog Integrated Circuits and Signal Processing*, vol. 15, pp. 291-314, 1998.

[97] U. Rückert, "ULSI architectures for artificial neural networks," *Micro, IEEE*, vol. 22, pp. 10-19, 2002.

[98] T. Schoenauer, S. Atasoy, M. Nasser, and H. Klar, "NeuroPipe-chip: a digital neuro-processor for spiking neural networks," *IEEE Trans. on Neural Networks*, vol. 13, pp. 205-213, 2002.

[99] A. Bofill-i-Petit and A. F. Murray, "Synchrony detection by analogue VLSI neurons with bimodal STDP synapses," presented at NIPS, 2003.

[100] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing*, vol. 47, pp. 416-434, 2000.

[101] M. Porrmann, U. Witkowski, H. Kalte, and U. Rückert, "Implementation of artificial neural networks on a reconfigurable hardware accelerator," presented at Euromicro Workshop on Parallel Distributed and Network-based Processing (PDP2002), Gran Canaria Island, Spain, 2002.

[102] U. Rückert, A. Funke, and C. Pintaske, "Accelerator-board for Neural Associative Memories," *Neurocomputing*, vol. 5, pp. 39-49, 1993.

[103] U. Rückert, "An associative memory with neural architecture and its VLSI implementation," presented at HICSS-24, Koloa, Hawaii, 1990.

[104] U. Rückert, "VLSI design of an associative memory based on distributed storage of information," in *VLSI Design of Neural Networks*, U. Ramacher and U. Rückert, Eds. Boston: Kluwer Academic Publishers, 1991, pp. 153-168.

[105] D. Hammerstrom, "A Digital VLSI Architecture for Real-World Applications," in *An Introduction to Neural and Electronic Networks*, S. F. Zornetzer, J. L. Davis, C. Lau, and T. McKenna, Eds., Second ed. San Diego, CA: Academic Press, 1995, pp. 335-358.

[106] T. Sejnowski and C. Rosenberg, "NetTalk: A parallel network that learns to read aloud.," The John Hopkins Univeristy Electrical Engineering and Computer Science Department JHU/EECS-86/01, 1986.

[107] Ö. Türel, J. H. Lee, X. Ma, and K. K. Likharev, "Architectures for nanoelectronic implementation of artificial neural networks: new results," *Neurocomputing*, vol. 64, pp. 271-283, 2005.

[108] ITRS, "International Technology Roadmap for Semiconductors 2005 edition - Executive Summary," 2005, http://www.itrs.net/Links/2005ITRS/ExecSum2005.pdf.

[109] J. M. J. Murre, "Neurosimulators," in *Handbook of brain research and neural networks*, M. Arbib, Ed.: MIT Press, 1995.

[110] H. Markram, "The Blue Brain Project," *Nature Reviews Neuroscience*, vol. 7, pp. 153-160, 2006.

[111] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI - Portable Parallel Programming with the Message Passing Interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.

[112] S. Zhu and D. Hammerstrom, "Simulation of associative neural networks," presented at Proc. of the ICONIP, Singapore, pp. 1639-1643, 2002.

[113] S. Pissarnezky, *Sparse Matrix Technology*. New York: Academic Press, 1984.

[114] C. Reschke, T. Sterling, D. Ridge, D. Savarese, D. Becker, and P. Merkey, "A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation," presented at High Performance and Distributed Computing, 1996.

[115] T. Sterling, *Beowulf cluster computing with Linux*: MIT Press, 2002.

[116] T. K. Priya and K. Sridharan, "An efficient algorithm to construct reduced visibility graph and its FPGA implementation," presented at 17th International Conference on VLSI Design, pp. 1057-1062, 2004.

[117] B. Christos-Savvas, Y. K. C. Peter, and Z. Li, "FPGA-Accelerated Pre-Attentive Segmentation in Primary Visual Cortex," presented at International Conference on Field Programmable Logic and Applications (FPL'06), pp. 1-6, 2006.

[118] G. Marcus and J. A. Nolazco-Flores, "An FPGA-based coprocessor for the SPHINX speech recognition system: early experiences," presented at International Conference on Recofigurable Computing and FPGAs (ReConFig 2005), pp. 4 pp., 2005.

[119] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving High Performance with FPGA-Based Computing," *Computer*, vol. 40, pp. 50-57, 2007.

[120] M. B. I. Reaz, S. Z. Islam, M. A. M. Ali, and M. S. Sulaiman, "FPGA realization of backpropagation for stock market prediction," pp. 960-964 vol.2, 2002.

[121] A. J. N. Batista, D. Alves, N. Cruz, J. Sousa, C. A. F. Varandas, E. Joffrin, R. Felton, J. Farthing, and J. E. Contributors, "An FPGA-based multi-rate interpolator with real-time rate change for a JET test-bench system," *Nuclear Science, IEEE Transactions on*, vol. 53, pp. 756-760, 2006.

[122] D. Kerr, D. Hammerstrom, and M. Pavel, "Real Time Sensor Image Fusion For Enhanced Vision Systems," Air Force Phase II SBIR 2003.

[123] J. R. Kerr, C. H. Luk, D. Hammerstrom, and M. Pavel, "Advanced integrated enhanced vision systems," presented at SPIE Aerosense - Specific Conference (no. 5081): Enhanced and Synthetic Vision, Orlando, Florida, 2003.

[124] B. Olshausen and D. J. Field, "Sparse coding with an overcomplete basis set: a strategy employed by V1," *Vision Research*, vol. 37, pp. 3311-3325, 2001.

[125] P. O. Hoyer and A. Hyvarinen, "A multi-layer sparse coding network learns contour coding form natural images," *Vision Research*, vol. 42, pp. 1593-1605, 2001.

[126] D. George and J. Hawkins, "Invariant pattern recognition using Bayesian inference on hierarchical sequences," 2004.

[127] H. Wersing and E. Korner, "Learning optimized features for hierarchical models of invariant object recognition," *Neural Computation*, vol. 15, pp. 1559–1588, 2003.

[128] D. Shaw and et al., "Anton, a special-purpose machine for molecular dynamics simulation," presented at The 34th International Symposium on Computer Architecture (ISCA 2007), San Diego, CA, USA, 2007.

[129] S. S. Iyer, J. E. Barth. Jr, P. C. Parries, J. P. Norum, J. P. Rice, L. R. Logan, and D. Hoyniak, "Embedded DRAM: technology platform for the Blue Gene/L chip," *IBM J. Res. & Dev.*, vol. 49, pp. 333-350, 2005.

[130] A. Lansner and A. Holst, "A Higher Order Bayesian Neural Network with Spiking Units," *Int. J. Neural Systems*, vol. 7, pp. 115-128, 1996.

[131] M. Schäfer and G. Hartmann, "A flexible hardware architecture for online Hebbian learning in the sender-oriented PCNN-neurocomputer Spike 128 K," presented at Proc. MicroNeuro '99, pp. 316-323, 1999.

[132] N. Weste and D. Harris, *CMOS VLSI Design - A Circuits and Systems Perspective*, 3rd ed: Addison Wesley, 2004.

[133] L. Kleinrock, *Queueing Systems*. New York, NY: Wiley, 1976.

[134] J. Lazzaro, S. Rychkebusch, M. A. Mahowald, and C. A. Mead, "Winner-take-all networks of *O(N)* complexity," Computer Science Department, California Institute of Technology, Pasadena, CA CALTECH-CS-TR-21-88, 1989.

[135] S.-Y. Chin and C.-Y. Wu, "A 10-b 125-MHz CMOS digital-to-analog converter (DAC) with threshold-voltage compensated current sources," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 1374-1380, 1994.

[136] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," presented at DAC 2003, Anaheim, CA, USA, pp. 338-342, 2003.

[137] Q. Chen and J. D. Meindl, "Nanoscale metal-oxide-semiconductor field-effect transistors: scaling limits and opportunities," *Nanotechnology*, vol. 15, pp. S549-S555, 2004.

[138] S. Borkar, "Electronics beyond nano-scale CMOS," presented at DAC 2006, San Francisco, CA, U.S.A., pp. 807-808, 2006.

[139] R. Chau, S. Datta, M. Doczy, B. Doyle, B. Jin, J. Kavalieros, A. Majumdar, M. Metz, and M. Radosavljevic, "Benchmarking nanotechnology for high-performance and low-power logic transistor applications," *IEEE Transactions on Nanotechnology*, vol. 4, pp. 153-158, 2005.

[140] A. Bachtold, P. Hadley, T. Naknishi, and C. Dekker, "Logic circuits with carbon nanotube transistors," *Science*, vol. 294, pp. 1317-1320, 2001.

[141] R. S. Friedman, M. C. McAlpine, D. S. Ricketts, D. Ham, and C. M. Lieber, "High-speed integrated nanowire circuits," *Nature*, vol. 434, pp. 1085, 2005.

[142] A. DeHon, P. Lincoln, and J. E. Savage, "Stochastic assembly of sublithographic nanoscale interfaces," *IEEE Trans. on Nanotechnology*, vol. 2, pp. 165-174, 2003.

[143] M. M. Ziegler and M. R. Stan, "CMOS/nano co-design for crossbar-based molecular electronic systems," *IEEE Transactions on Nanotechnology*, vol. 2, pp. 217-230, 2003.

[144] D. B. Strukov and K. K. Likharev, "CMOL FPGA: a reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices," *Nanotechnology*, vol. 16, pp. 888-900, 2005.

[145] D. B. Strukov and K. K. Likharev, "Prospects for terabit-scale nanoelectronic memories," *Nanotechnology*, vol. 16, pp. 137-148, 2005.

[146] V. Cerletti, W. A. Coish, O. Gywat, and D. Loss, "Recipes for spin-based quantum computing," *Nanotechnology*, vol. 16, pp. R27-R49, 2005.

[147] A. Heittmann and U. Rückert, "Mixed mode VLSI implementation of a neural associative memory," presented at MicroNeuro '99, pp. 299-306, 1999.

[148] J. H. Lee, X. Ma, D. B. Strukov, and K. K. Likharev, "CMOL," presented at International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures, Palm Springs, CA, pp. 3.9-3.16, 2005.

[149] Ö. Türel, J. H. Lee, X. Ma, and K. K. Likharev, "Neuromorphic architectures for nanoelectronic circuits," *Int. J. of Circ. Theor. Appl.*, vol. 32, pp. 277-302, 2004.

[150] M. M. Ziegler and M. R. Stan, "A Case for CMOS/Nano Co-design," 2002.

[151] J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams, "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology," *Science*, vol. 280, pp. 1716-1721, 1998.

[152] U. Rückert and H. Surmann, "Tolerance of a binary associative memory toward stuck-at-faults," presented at Proceedings of the 1991 International Conference on Artificial Neural Networks (ICANN-91), Espoo, Finland, pp. 1195-1198, 1991.

[153] F. T. Sommer and P. Dayan, "Bayesian retrieval in associative memories with storage errors," *IEEE Trans. on Neural Networks*, vol. 9, pp. 705-713, 1998.

[154] C. Gao, "Fault tolerance for WPNAM-like associative memory," Dept. Electrical and Computer Engineering, Portland State University 2006.

[155] C. S. Lin, S. H. Ou, and B. D. Liu, "Design of k-WTA/sorting network using maskable WTA/MAX circuit," presented at International Symposium on VLSI Technology, Systems, and Applications, pp. 69-72, 2001.

[156] R. K. Kolagotla, H. R. Srinivas, and G. F. Burns, "VLSI implementation of a 200-MHz 16x16 left-to-right carry-free multiplier in 0.35 μm CMOS technology for next-generation DSPs," presented at Proc. IEEE 1997 Custom Integrated Circuits Conference, pp. 469-472, 1997.

[157] J. C. Ellenbogen and J. C. Love, "Architectures for molecular electronic computers: 1. Logic structures and an adder designed from molecular electronic diodes," *Proceedings of the IEEE*, vol. 88, pp. 386-426, 2000.

[158] C. Johansson and A. Lansner, "Towards cortex sized artificial neural systems," *Neural Networks*, vol. 20, pp. 48-61, 2007.