Hardware Architecture for a Bi-directional Protein Processor Associative Memory

Omer Qadir

PhD Dissertation

University of York

Intelligent Systems Group, Department of Electronics

October 2011.

Abstract

The evolution of Artificial Intelligence has passed through many phases over the years, going from rigorous mathematical grounding to more intuitive bio-inspired approaches. However, to date, it has failed to pass the Turing test. A popular school of thought is that stagnation in the 1970s and 1980s was primarily due to insufficient hardware resources. However, if this had been the only reason, recent history should have seen AI advancing in leaps and bounds – something that is conspicuously absent. Despite the abundance of AI algorithms and machine learning techniques, the state of the art still fails to capture the rich analytical properties of biological beings or their robustness. Moreover, recent research in neuroscience points to a radically different approach to cognition, with distributed divergent connections rather than convergent ones. This leads one to question the entire approach that is prevalent in the discipline of AI today, so that a re-evaluation of the basic fabric of computation may be in order.

In practice, the traditional solution for solving difficult AI problems has always been to *throw* more hardware at it. Today, that means more parallel cores. Although there are a few parallel hardware architectures that are novel, most parallel architectures – and especially the successful ones – simply combine Von Neumann style processors to make a multi-processor environment. The drawbacks of the Von Neumann architecture are widely published in literature. Regardless, even though the novel architectures may not implement non-Von-Neumann style cores, computation is still based on arithmetic and logic units (ALU). The aim of this research is to explore the possibility of whether an alternative hardware architecture inspired from the biological world, and entirely different from traditional processing, may be better suited for implementing intelligent behaviour while also exhibiting robustness.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I had the great fortune of having three supervisors and I must start by thanking them for all their help and guidance. Andy Tyrrell for the really quick, lightning fast turn-around on anything I sent him for a review, despite the fact that it was often late on a Friday evening. Gianluca Tempesti for the attention to detail and for always poking holes in my arguments – helping me make them stronger. Jon Timmis for always picking out that minute detail missing in my hypothesis, and of course for the awesome culinary treats, though Andy's Tiramisu certainly deserves a mention at this point.

I must also thank Jerry (Yang Liu) for all the encouragement and the long discussions, which helped me crystallise my hypothesis. The endless discussions on the etymology of words and names, and the long philosophical arguments about the mixing of cultures might have appeared to be distractions to many, but for me they were quite enjoyable.

In addition, I must mention all the wonderful friends I have made in the department over the years. Tüze Kuyucu for introducing me to climbing and convincing me that its better than football. This is going to stay with me for life now dude – thanks! Antonio Gomez Zamorano, for all the jokes and the (often dark) humour, not to mention all the profanity. I can now feel confident that I won't feel lost in Spain. My football buddies: Piti, Piero, Lachie, Bo, Luis, James, the other James, and the other-other James. My ankles are stronger for all the injuries they've received over the years. The Mexican League, who wish to remain separate from the Spanish league, Luis and Alfonso. I now know not to think of americans as only people from the USA. Colin Bonney who took me climbing to all those places inaccessible without a car, and let me use his gear as well. Martin Trefzer for keeping me company in the *booming-voice* and the *loud-laughter* club.

Finally, but most importantly, I must thank my family for the constant support and encouragement. It goes without saying that I would not have any of the opportunities in life today if it wasn't for them. It is impossible to repay you guys or to thank you enough!

List of Publications

The work presented in the following chapters has been published at various venues, as listed below:

- Qadir, O.; Timmis, J.; Tempesti, G. & Tyrrell, A. The Protein Processor Associative Memory on a Robotic Hand-Eye Coordination Task. *In review for the 6th International ICST Conference on Bio-Inspired Models of Network, Information and Computing Systems (Bionetics2011), York, UK*, **2011**
- Qadir, O.; Liu, J.; Timmis, J.; Tempesti, G. & Tyrrell, A. Hardware architecture for a Bidirectional Hetero-Associative Protein Processing Associative Memory. *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2011), New Orleans, USA*, **2011**
- Qadir, O.; Liu, J.; Timmis, J.; Tempesti, G. & Tyrrell, A. From Bidirectional Associative Memory to a noise-tolerant, robust Self-Organising Associative Memory. *Artificial Intelligence*, **2011**, 175, 673–693
- Qadir, O.; Liu, J.; Timmis, J.; Tempesti, G. & Tyrrell, A. Principles of Protein Processing for a Self-Organising Associative Memory. *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2010), Barcelona, Spain*, **2010**
- Benatar, N.; Qadir, O.; Owen, J. & Baxter, P. P-Controller as an expert system for manouevering rudderless sail boats. *Proceedings of the 9th Annual Workshop on Computational Intelligence (UKCI 2009), Nottingham, UK*, **2009**

Chapter 1

Problem Description

Over the years, *computation* has evolved to mean many different things. Stepney et al. (2005) defines the classical view of computation and encourages the research community to stretch and break these classical paradigms; suggesting that the grand challenge for computation today is:

> "to journey through the gateway event obtained by breaking our current classical computational assumptions, and thereby develop a mature science of Non-Classical Computation."

This research is an attempt to explore possible alternate non-standard architectures for computation; in particular, computation for Artificial Intelligence (AI) applications. Section 1.1 gives a brief introduction to the problem of implementing AI algorithms on existing hardware architectures while Section 1.2 describes the motivation for designing fault tolerant hardware. Section 1.3 formally defines the hypothesis, summarises the solution and describes the structure of the rest of this document.

1.1 Intelligent Algorithms on Current Hardware Architectures

The Von Neumann architecture is characterized by a processing unit and a (logically) single, separate storage unit. The heart of the processing unit is the ALU and instructions are executed sequentially (Stallings, 2000). Most of the area is memory and only a few locations can be accessed at any given moment. Therefore, performance is limited by the bandwidth of data transfer between processor and memory – resulting in the *Von Neumann Bottleneck* (Backus, 1978) – something that is becoming increasingly apparent with the recent increase in processor speeds. Despite the effect of this bottleneck on the effective processing speed of a conventional processor, an individual processor today still operates much faster than basic individual biological elements, like neurons. Depending on the neuron, this can be anywhere between 2ms and 10ms (Thorpe et al., 2001, Vreeken, 2003) – resulting in neuron firing rates of between 100Hz to 500Hz. Nonetheless, Artificial Intelligence (AI) algorithms implemented on processors running at GHz speeds are still unable to match the human brain in many respects. Moreover, as shown by Hillis (1984), even a nano-second cycle time is not nearly fast enough in a serial Von Neumann architecture to solve million-scaled AI problems[1]. His solution was to move processing into memory, using a parallel architecture based on Cellular Automata, that he calls the Connection Machine. Although a detailed discussion of parallel architectures including the Connection Machines is presented in Chapter 2, it is evident at the outset that some level of parallelism is essential to solve the problems in this domain.

Stagnation in the field of Artificial Intelligence in the 1970s and 1980s is widely attributed to insufficient hardware resources (Abraham, 2005, Moravec, 1998). However, if this had been the only reason, the following decades should have provided significant progress in AI – which has not been the case. Despite the abundance of AI algorithms today, these algorithms are very specific in terms of their applicability. Although the algorithms that govern the operation of biological brains are also specific (for example, typically slow at number crunching) they find it trivial to perform tasks such as image processing and hetero-associative recall (Section 2.1.3.4), that are beyond the capability of even the most sophisticated machines. Hillis (1986) claims that the strength of biological intelligence seems to be the ability to solve problems that require manipulating poorly structured data. On the other hand, the strength of machines is the ability to perform repetitive deterministic tasks, typically involving large or long calculations. The serial nature of the classical computing machine coupled with the causal nature of the human psyche means that researchers tend to think of AI problems as search problems.

[1]AI problems that use millions of facts to come up with solutions.

A common analysis offered for this inability of computers to match the human brain (and consequently, the inability to pass the Turing test) is to estimate the computing power of the human brain and relate it to the computing power of the modern processor. Although there are a number of ways of doing this, one of the more widely accepted methods is based on the assumption that nerve volume is proportional to computation power (Moravec, 1998). Since the retina has been extensively studied and is well understood, it can be used as a starting point to estimate the number of instructions required to perform the same task (vision) on a state-of-the-art processor. With this estimate, and knowing the number of neurons in the retina, it is a trivial calculation to estimate the equivalent instructions count per second that is performed in each neuron. Furthermore, knowing the total number of neurons in the human brain (or a close approximation of it), the total equivalent MIPS (Millions of Instructions Per Second) can be estimated in terms of instructions executed by the processor. Moravec (1998) indicates that the retina seems to process ten images per second, each of 1×10^6 pixels. Moravec (1998) further claims that it takes robot vision programs 100 instructions to derive edge detection or motion detection from a video of comparable images; thereby estimating 1,000 MIPS as being equivalent to the retina. Furthermore, the human brain being 1×10^5 times bigger than the retina in terms of nerve volume, Moravec (1998) estimates the computing power of the human brain to be around 1×10^8 MIPS.

Although supercomputers today are closing in on this number, the expense of such machines is too prohibitive to dedicate them for a human brain implementation; and until such an implementation is actually done, there is no guarantee that the hypothesis about insufficient hardware resources holding back AI is correct. Moreover, a neglected assumption of this method of estimating the human brain's computing ability is that it is related to the hardware architecture used to implement the electronic equivalent of the retina. The fact that Moravec (1998) describes an edge detection operation as 100 instructions is quite suggestive of the type of computing architectures being considered. These architectures are composed of (possibly multiple) general-purpose processors, executing instructions sequentially which are fetched from memory. Using a different architecture might result in a much smaller MIPS count (or a larger one). In addition, although using general purpose processors provides ease and flexibility, it must be noted that nature's equivalent (the brain) does not use general purpose elements. Neurons perform only one task, namely firing in response to input. This is, in effect, hardwired, although it may be fine-tuned.

Also, note that neurons operate at a much lower frequency than general purpose processors and are still able to out-perform their digital counterparts. It is to be expected of course, that custom hardware out-performs general purpose hardware for the task it has been customised for. However, could it also be that using non-standard custom hardware might sufficiently ease the constraints on timing and spatial requirements, thereby making a solution possible in the current state of the art? In order to evaluate this, standard computation must first be defined. Johnson (2007), Stepney et al. (2005) define six properties of classical computation which can be summarised as follows.

Turing paradigm: A computer processes discrete states, with unlimited resources, and the choice of substrate for the implementation of the computer is immaterial to its processing.

Von Neumann paradigm: A computer is a single, serial processor and information is brought to the processor.

Output paradigm: Outputs are generated from a well-defined output channel and are the only thing of interest.

Algorithmic paradigm: Computers execute well-defined, non-deterministic processes, which are bounded in time and have well-defined inputs and outputs.

Refinement paradigm: Problems can be defined in exact specifications which do not change over time and which can then be used to implement the solution. Binary is desirable and emergence is undesirable.

Computer-as-an-artefact paradigm: The computer hardware is not composed of elements in the physical world that already perform the task and the hardware does not change over time; thereby implying that the hardware is general purpose enough to fit the problem.

Any attempted solution based on non-standard custom hardware must challenge at least one aspect of the above, though more than one would be preferable. This is because in order to escape the traditional computation paradigm, it might prove useful to perturb the existing solution by a large amount, rather than a small one.

1.2 Robust Intelligence on Current Hardware Architecture

A wide range of conventional engineering techniques exist for fault tolerance and error correction. A detailed discussion of such techniques is presented in chapter 3, however, it should be noted that these techniques are used in many critical systems such as manned missions to outer space (Iyer and Kalbarczyk, 2003, She and Samudrala, 2009, Storey, 1999); indicating that they do provide a certain level of fault tolerance. What is lacking however, is the kind of robustness that can be observed in biological entities (for instance the neuro-plasticity in the human brain) that can recover from extremely severe faults. Part of this ability in biological organisms is due to regeneration that stems from cellular reproduction. This is beyond the current abilities of electronic devices and in fact, it is unknown exactly *how* important cell division is to regeneration and plasticity (Brockes and Kumar, 2003). However, a major part of this ability for robustness can be attributed to other properties of the biological organisms. These include the inherent degeneracy of cellular functions, the totipotent[2] nature of the genome, the dynamic re-routing capabilities of connections and the fact that a single function is performed by a combination of a large number of cells so that no single cell becomes a critical element. These features are at direct odds with the traditional approach to hardware design. Hardware designers attempt to optimize the design and, in order to get maximum efficiency, prefer to remove duplicated hardware, to the extent of sometimes sacrificing functions that are not used frequently. Elements are designed to be unique and therefore each is crucial to the correct operation of the whole.

Existing solutions vary from hardware redundancy and sparing to information redundancy and timeouts, but mission critical environments tend to duplicate hardware. For instance, NASA space shuttles use 4 computers with majority voting (Iyer and Kalbarczyk, 2003). On the other hand, biological systems display degeneracy rather than simple duplication and redundancy. Degeneracy is usually defined as *the ability of two or more systems to perform the same function while being structurally different.* Therefore, degenerate systems tend to perform slightly different versions of the same task rather than being exact duplicates of each other. For example, although the B-Cell and the Dendritic Cell

[2]Full potential (Prodan et al., 2003)

both generate an immune response, they are not exact duplicates so that if there are events that escape one, they may be captured by the other (de Castro and Timmis, 2002, Greensmith et al., 2005). In addition, although there may be many B-Cells that make up an immune response, they are all actively performing a function that makes up a collective B-Cell immune response (Mendao et al., 2007) and no cell is spare in the sense of the traditional electronics "sparing" technique. If a few cells become faulty, there is a transient drop in performance but the operation is not dependent upon the cells being replaced for the operation to continue. Not only is this an added layer of protection, but it also means that *there is no portion of the system that is idle or spare or redundant* as are the duplicate copies in more traditional approaches like Triple Modular Redundancy (TMR)(Iyer and Kalbarczyk, 2003).

This work is part of an EPSRC-funded project called SABRE[3], under grant number FP/F06219211. SABRE is a collaboration between the University of York and the Bristol Robotics Laboratory (BRL) at the University of West of England in Bristol and aims to explore possible hardware architectures for fault tolerance in mission critical environments[4]. The architecture is developed as three levels (Figure 1.1), with the idea that each level provides a degree of fault tolerance and signals to upper or lower levels if faults cannot be handled at its own level. The lowest level is composed of *cells* and was

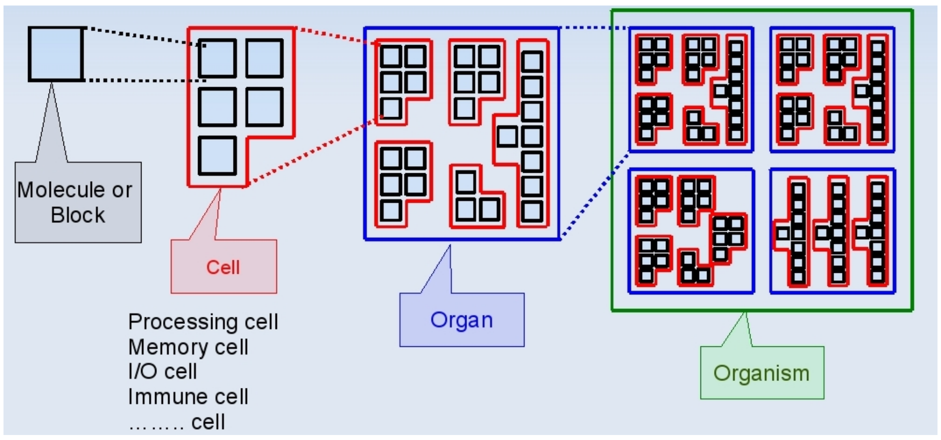


Figure 1.1: Levels in the SABRE architecture

designed at BRL, while the highest level is called the *organism* and was developed in York. *Cells* are composed of a number of basic building blocks, called *molecules*. Each cell has a particular function, such as processing data, storing memory, performing I/O and so

[3]Self-healing cellular Architectures for Biologically-inspired highly Reliable Electronic systems
[4]Situations where it is imperative that the system continue operation, even in the presence of errors.

on. An *organ* is composed of multiple cells and in turn makes up an *organism* which is expected to be capable of performing application level functions. The lowest level design is a reconfigurable fabric inspired by prokaryotic organisms and is called *Unitronics*. Refer to Samie et al. (2009a,b) for complete details about the architecture and the robustness it adds to the whole.

1.3 Initial Hypothesis

Although a more detailed review of the problem domain is presented in Chapters 2 and 3, the preceding sections outlined its major aspects. From this, the following initial hypothesis can be identified, which is later expanded upon in Chapter 4:

> " *A novel, non-standard computation method for a bidirectional, hetero-associative memory, implemented using a non-standard hardware architecture, composed of multiple (of the order of hundreds) parallel processing elements, can perform meaningful computation in the context of AI applications, and can perform better and in a more robust way than other traditional techniques based on traditional hardware.* "

In order to measure the success or failure of this hypothesis, some of the terms used need further definition (fully specified in Section 4.1). This is presented as part of the following implied objectives:

1. In order to be non-standard, the novel computation method must differ in at least one aspect from the definition of classical computation as described in Section 1.1. Details for this are presented in the final hypothesis in Chapter 4.
2. Since the architecture is non-standard, before testing for *meaningful* computation, the architecture must be tested for *any* computation. This can be tested by measuring the accuracy of memory recall of the training data itself after training.
3. Meaningful computation can be defined as the ability to generalise about the underlying dataset from the sample presented in the training dataset. Performance on previously unobserved data must, therefore be measured.

4. In order to test if the novel architecture is *better*, it must be compared with existing techniques implemented on existing architectures of comparable silicon real-estate: If *toy* datasets are used (for the comparison or in tests), the novel architecture must be shown to be at least as scalable as existing comparable techniques.

5. Improved robustness can be tested in a number of ways, at least one of which must be true:

    - Correct operation or execution must not hinge on any single element (of the parallel architecture) so that faults result in graceful degradation of performance.
    - Fault tolerance must be achieved through degeneracy rather than redundancy.

Note that these definitions are not very *exact* and more detailed ones are presented in Section 4.1 after the literature review.

1.3.1 A brief description of the novel solution

A complete list of the novel contributions of this thesis are presented in Section 8.1.1, however, a concise summary of the proposed novel solution is presented here which helps to put the literature review in context. A novel non-standard computation method for a bi-directional, hetero-associative memory has been designed and developed, that is an amalgamation of inspirations from the biological neural network and the biological genetic regulatory network. The memory is distributed over nodes operating in parallel and in a decentralised manner. In addition to designing the novel computation paradigm, a novel hardware architecture for implementing this memory has also been designed. The non-standard computation is achieved by extracting computation from memory operations rather than using arithmetic operations and therefore, nodes in the hardware architecture are not the standard Von Neumann processors of other parallel architectures. The architecture operates on abstract symbols and (unlike reconfigurable hardware solutions) does not require a configuration bit-stream. Furthermore, there is no distinct training phase, therefore, learning is *life-long* and so the architecture is able to adapt dynamically to new data without requiring reconfiguration (unlike reconfigurable hardware solutions which need to handle dynamic reconfiguration issues). Fault tolerance is ensured through degeneracy rather than redundancy such that no individual node is critical to the operation of the whole and faults result in graceful degradation.

This thesis is organised as shown in table 1.1

Chapter	**Description**
1	Presents an overview of the problem domain with an initial definition of the *hypothesis*.
2	Reviews existing hardware solutions and the effect of implementing AI algorithms on such hardware.
3	Reviews existing fault tolerance mechanisms
4	Presents the detailed definition of the *hypothesis* and describes a preliminary abstract view for the proposed solution.
5	Presents results from experiments performed on the proposed solution.
6	Describes an improved version of the proposed solution including details about the hardware implementation.
7	Presents results from experiments performed on the improved version.
8	Discusses possible improvements and concludes.

Table 1.1: Document organisation

Chapter 2

Artificial Intelligence and Hardware Architectures

Although AI is still in its infancy, it has progressed significantly from the days when a literature review of the field could be presented in one chapter. More than space, however is the consideration that a complete review is not relevant to the focus of this thesis. Similarly, a complete review of the state of the art in hardware architectures is not required. Instead, existing hardware architectures and AI techniques are divided into categories and a review of each category is presented, with the objective of then being able to discuss the implications of implementing these AI techniques on existing hardware architectures (categories). Section 2.1 describes the broad categories of current AI techniques while Section 2.2 reviews hardware architectures and Section 2.3 discusses the aforementioned implications of implementing AI techniques on the existing hardware architectures.

2.1 History and Methods of AI

Artificial Intelligence can be said to date back to the 1930s when Alan Turing proposed the Universal Turing Machine. Turing and Emil Post both independently proved that "determining the decidability of a mathematical proposition is equivalent to asking what sort of sequences of a finite number of symbols can be recognized by an abstract machine with a finite set of instructions" (Abraham, 2005). Around 1935, Alonzo Church et al. came up with a mathematical definition for intuitive computability (Li and Du, 2007), which effectively stated that:

> *"A function of positive integers is effectively calculable only if recursive".*

At about the same time, in 1936, Turing independently came up with the concept of calculability by a Logical Computing Machine (LCM), which was Turing's expression for the Turing machine (Li and Du, 2007):

> *"LCMs can do anything that could be described as 'rule of thumb' or 'purely mechanical' ".*

The Turing test is based on the principle that if an observer cannot differentiate between the behavior of a machine and a human, then the machine can be termed intelligent (Turing, 1950). Although there are many detractors of the Turing test, it is still the most accepted measure of deciding whether a machine (or system or algorithm) is artificially intelligent or not. Turing (1950) estimated that by the year 2000, machines with 10^9 bits of memory would have a 70 percent probability of passing the Turing test within 5 minutes.

There have been many other landmarks in the history of AI including the famous Dartmouth Symposium in 1956 and the first session of the International Joint Conference on AI in 1969. Over the course of its evolution, there have been many definitions for Artificial intelligence, primarily depending upon whether the person leans towards strong AI or weak AI. According to Coppin (2004, Chapter 1), two of the more popular definitions are:

Strong AI : "AI is the study of systems that act in a way that to any observer would appear to be intelligent"

Weak AI: "AI involves using methods based on the intelligent behavior of humans and other animals to solve complex problems"

The proponents of strong AI claim that with enough processing power and intelligence, machines can think and be conscious like humans. On the other hand, advocates of weak AI believe that intelligent behaviour can only be modeled and used by machines to solve complex problems.

The following sections discuss some of the major categories of AI techniques. This is not intended as a review of the entire domain of AI, but rather as a discussion of the areas relevant to the focus of this thesis as described in the problem description (Section 1.1). Section 2.1.1 briefly introduces the concept of learning. Section 2.1.2 discusses symbolic methods, while Section 2.1.3 talks about sub-symbolic AI.

2.1.1 Learning Techniques

In the classical approach, the task for AI is to generalise from training data. A small portion of the underlying dataset is presented to an algorithm for training, and the algorithm should be able to determine the relationship between inputs and outputs from this subset. The simplest method of doing this is *rote learning* where each piece of training data is stored along with its classification. This, of course, has the disadvantage that it does not generalise at all. Techniques that generalise can be classified into one of the following (Abraham, 2005):

Supervised learning: learn by generalising from pre-classified training data.

Unsupervised learning: learn without training by discovering statistical features in the input data that create clusters of patterns.

Reinforcement learning: learn by maximizing a numerical reward function from the environment through trial and error.

The semantic networks mentioned in Section 2.1.2 are an example of supervised learning while the self-organising maps mentioned in Section 2.1.3 use unsupervised learning.

The classical approach is to rely on mathematics and assume that if a function can be found that maps a large set of training data to its classification, then this function will also be able to map unseen data correctly. This is *inductive learning* (Michalski, 1983). A simple structured way to induce the function is to use *general-to-specific* ordering or *specific-to-general* ordering. A most-general hypothesis is of the form $h_g = <?, ?, \cdots, ?>$ where ? can take any value. A most-specific hypothesis is of the form $h_s = <\phi, \phi, \cdots, \phi>$ where ϕ cannot take a value[1]. In general-to-specific ordering, for each positive training example, ϕ is replaced with a more general value. The resulting hypothesis fits all the training data without having used any negative examples.

[1]Is a *null* element – or an empty set

Another structured method for inductive learning is to construct *decision trees* from training data. Leaf nodes in decision trees are boolean output values and intermediate nodes are questions, with intermediate links being answers to the questions. Quinlan (1986) presents the popular *Iterative Dichotomiser 3* (ID3) algorithm, which, as shown by Coppin (2004, Chapter 10), is still the best known decision tree induction algorithm as it finds the shortest possible decision tree by building top-down decision trees. Note however, that at the end of the learning phase, data is present in some data representation (for instance in a hypothesis or a decision tree). Upon presentation of a new test input, the data representation will need to be parsed to generate the output. If the test input falls outside the training dataset, decision trees or general hypotheses on their own cannot extrapolate from the training set.

An alternative to inductive learning is to use *instance-based learning.* Instead of trying to generalize from training data, these techniques store the data and use it every time a new piece of data is encountered – like the Hamming associative memory described in Section 2.1.3.4. Algorithms like *K-nearest neighbour* (Bremner et al., 2005) and *K-means clustering* (MacQueen, 1967) are also popular examples of instance-based learning. Nearest neighbour algorithms use a distance measure (usually Euclidean distance) for each new data point to calculate the distance from elements in the training dataset and points are classified as being the same as the one with lowest distance. This means, that unlike decision trees, extrapolation (or interpolation) can be used to *guess* the correct output in case of data that is outside the training set. The nearest-neighbour algorithm has spawned many variations over the years including *Shepard's method* (Shepard, 1968) and *AIRS*[2] (McEwan and Hart, 2009). Nearest neighbour algorithms are typically good at classifying noisy data, but can be misled by erroneously including irrelevant attributes in the training data. However, note that instance-based learning methods must store the entire dataset and upon presentation of new input, parse the entire dataset to find the correct (or closest) value. Implications of performing this search operation using traditional hardware architectures is discussed in Section 2.3.

As opposed to the nearest neighbour algorithms, clustering algorithms typically use unsupervised learning. They operate in an iterative fashion, assigning data to a class such that the distance between the data-point and the centre of the class is minimized. After

[2]Artificial Immune Recognition System

assigning the data, the centre of the class (average or mean) is recalculated (Faber, 1994). A major weakness with this approach is the method for determining the number of classes. Despite this, Dayan (1999) claims that unsupervised learning is much more likely to be the method used in the brain, since, unlike supervised or reinforcement learning techniques, unsupervised learning is not dependent upon target outputs or environment evaluation functions. Rather, it uses prior biases about aspects of the structure of the input that it considers to be important. It operates by generalizing from some *a priori* information about observed input patterns that are independent samples from an unknown probability distribution. Note however, that the level of complexity where this sort of unsupervised learning occurs (as discussed by Dayan (1999)) is much higher than the level at which individual neurons or even cortical networks of neurons – cognits (Fuster, 2006) – tend to operate. The unsupervised learning mechanism described by Dayan (1999) is at the level of human consciousness, which requires all the billions of neurons in the brain and trillions of connections. On the other hand, the state of the art only allows us to operate at the level of hundreds (or at best thousands) of neurons (Section 2.3.2).

Density estimation methods are a set of unsupervised learning algorithms that explicitly build statistical models of underlying causes (Dayan, 1999). One of the more popular examples of this is the *Bayesian belief network* which is based on Bayes theorem summarised in equation 2.1.

$$P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)} \tag{2.1}$$

$P(B|A)$ is known as the conditional probability of B (the probability of event B, given event A has occurred) and is described by Equation 2.2.

$$P(B|A) = \frac{P(B \wedge A)}{P(A)} \qquad \text{where} P(A) \neq 0 \tag{2.2}$$

The Bayesian belief network is an acyclic graph where nodes are pieces of evidence or hypotheses and arcs represent the dependence between nodes. Together with a list of probabilities this provides us with a set of hypotheses and the ways they interact. More details and the mathematical derivations can be found in Coppin (2004, Chapter 12) and Abraham (2005). Note however, that Bayesian belief networks are essentially graphs which need to be traversed to find the correct (or closest) result upon presentation of test data. Once again, this *search* has implications when implemented on traditional hardware architectures and these are discussed in Section 2.3.

2.1.2 Symbolic Methods

One of the older and more traditional philosophical bases of AI is *symbolism*, which is based on predicate calculus. This holds that cognition is symbol processing and that knowledge and reasoning can be represented in terms of discrete *symbols*, which are patterns that form the basic unit in describing cognitive and intellectual activities and can be distinguished from other symbols. Traditionally a symbolic system is expected to have 6 functions (Li and Du, 2007):

- Inputting Symbols
- Ouptutting Symbols
- Storing Symbols
- Duplicating Symbols
- Creating Symbols – through the discovery of relations
- Conditional migration – continuing action based on existing symbols

Note that data representation is critical to the operation of symbolic methods.

2.1.2.1 Semantic Networks

Semantic networks are one of the basic tools for representing knowledge and are used in many symbolic AI algorithms. Objects are represented as nodes in a network and directional links between nodes represent the relationship between the objects. For example, an "apple" may be a node in the network, as may be "fruit" and both nodes may be connected by a link labeled "is a". This can be extended so that the links themselves may be objects. Therefore, in essence, semantic networks are labelled, directed and (potentially) cyclic graphs; A concise description of directed graphs may be found in Jong et al. (2003). Hillis (1984) provides an introduction to *marker propagation* which allows the time for retrieval of information from semantic networks to be independent of the network size. Minsky (1975) introduced frame-based representations that can be used to expand semantic networks such that objects can inherit relationships and properties from a super-set. Note that, typically, semantic networks are *searched* for answers. Semantic trees are acyclic semantic networks which are largely used to assist the search of semantic networks by limiting the scope. Thus, if data is represented in semantic networks, it can be searched using semantic trees (more on tree-based search in Section 2.3.1.1). More information about semantic networks and frame based systems can be obtained from Hillis (1984, 1986), and Coppin (2004, Chapter 3).

2.1.2.2 Logic and Calculus

One of the major limitations of semantic networks is that they cannot represent negative relationships – e.g. "Mickey is not a mouse". This is solved by *First Order Predicate Logic* (FOPL) and rule based systems which are considered to have "greater representational adequacy than frame-based representations" (Coppin, 2004, Chapter 3). FOPL can be used to express properties about objects. In predicate logic, the information "I own a dog" is represented as *O(me, dog)* where O is the predicate and shows the relationship *owning*, instead of representing it as A[3] (as in propositional logic). This has implications on the kind of operators that may be applied; further details on this can be found in Coppin (2004, Chapter 7).

In FOPL, it is hard to express change or temporal relationships. Event Calculus adds temporal logic operators such as *henceforth*, *awaits*, *until*, *so far* and *precedes* to resolve this (Coppin, 2004, Chapter 17). Alternately, situation calculus is a form of predicate calculus that describes objects in one state or situation and then describes how the object will change for a given action. To illustrate, equation 2.3 describes a situation S_1 where the mouse and the cat are in the same room.

$$\exists x(\text{ In (Mouse}, x, S_1) \wedge \text{ In (Cat}, x, S_1)) \tag{2.3}$$

Then, $\text{Result (Move}_{i,j}, S_1) = S_2$ describes how the action *Move* will result in changing from situation S_1 to situation S_2. For more details about situation calculus and methods to extend it using various axioms, the reader is directed to Coppin (2004, Chapter 15). Note, however, that even though calculus adds some important features, it is a data representation and the actual method of extracting information is still a matter of *searching.*

[3]Where A represents the complete piece of information "I own a dog"

2.1.2.3 Expert Systems

Expert systems use a knowledge base, typically composed of *if-then* rules and created by experts to solve complex problems (Coppin, 2004, Chapter 9). The 1970s saw the advent of expert systems with *blackboard architectures* (Erman et al., 1980) and *production systems* (Forgy, 1979). These have been the basis for a variety of parallel hardware architectures for solving AI problems which are discussed in Section 2.2.2.

Blackboard architectures were invented by H. Penny Nii in the 1970s and are methods for structured knowledge representation. A central database is written by a number of disparate knowledge sources such as human experts and encyclopedias. The experts both write to the database and use it for solving a problem. Coppin (2004, Chapter 17) and Erman et al. (1980) provide more details and some examples of existing architectures, including the HEARSAY architecture. Corkill (1991) discusses the similarities and differences between blackboard architectures and other expert systems.

Production systems primarily consist of a set of rules, called *productions*, that determine behaviour. Like other rule based systems, productions consist of two parts, namely the precondition (IF) and the action (THEN). If the precondition is matched, the production is said to be *triggered*. If the corresponding action is executed, the production is said to have *fired*. However, the rule system needs to be able to resolve conflicts when more than one production is triggered. Neiman (1991) discusses the problems of conventional conflict resolution algorithms for parallel architectures and compares blackboard systems and production systems in order to apply principles of one to the other. Amaral and Ghosh (1994) present a parallel hardware architecture for production systems which acts as an associative memory (discussed further in Section 2.2.2). This is based on the popular *Rete* algorithm by Forgy (1979), which is a pattern matching algorithm for production systems. Rete, which means *net* in Latin, builds a network of nodes where each node, except the root node, corresponds to a portion of the antecedent (IF condition) of a rule. Therefore the path from root to leaf completely defines the conditional portion, and rule matching is equivalent to searching a tree. A latter version called the Rete II algorithm was developed in the 1980s, but is the intellectual property of Production System Technologies, and therefore closed to the public.

Classifier systems were created by Holland et al. (1986) and can be thought of as an extension of rule-based expert systems, in that the rules are optimised through evolution. This typically works better than standard expert systems since it is able to respond to unfamiliar situations better. A discussion of evolutionary algorithms is considered to be irrelevant to this thesis and so is not included. A comparison of the Pittsburgh Learning Classifier System and the Michigan Learning Classifier System can be found in Pipe and Carse (2007). Note, however, that expert systems also need to be searched for answers. Implications of this are discussed in Section 2.3.1.

2.1.3 Sub-Symbolic AI

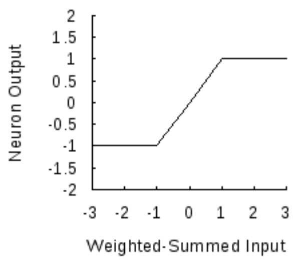
A newer, more bio-inspired paradigm that has been gaining more support recently is that of *connectionism* or the sub-symbolic approach. This holds that cognition is based on a network of nodes (or neurons) and information is stored in the weights or strength of connections between these nodes. Miller et al. (1990, cited in Li and Du (2007)) describes the three main characteristics of connectionism as follows:

1. distributed information storage and large-scale parallel processing,
2. ability to learn, self-adapt and self-organise,
3. fault tolerance.

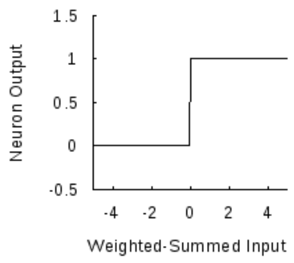
The following sections present a brief review of the connectionist methods and implications of this for AI applications on traditional hardware architectures is discussed in Section 2.3.

2.1.3.1 Origins - McCulloch and Pitts

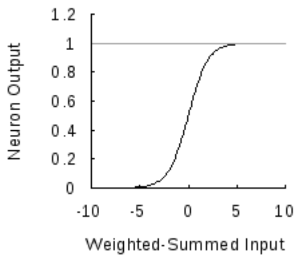
Biological neural networks have been inspiring researchers for many years. As early as 1943, McCulloch and Pitts successfully modelled the principles of neurons and created models that later came to be known by their names. This was the first generation of Artificial Neural Networks (ANNs). Neurons receive multiple inputs which are multiplied by their weights and summed. This summed value is then applied to an activation function. Some of the more common activation functions are shown in figure 2.1. This principle (of summation of weighted inputs) is the basis of all ANNs.



(a) Linear



(b) Step



(c) Sigmoid

Figure 2.1: Some activation functions: The linear saturated function is typical of the first generation neurons. The step function is used when binary neurons are desired. The sigmoid is the default for most applications and is recommended for classification problems.

Rosenblatt (1958) proposed the *perceptron* which is a landmark in the history of Artificial Neural Networks. Equation 2.4 describes its function, where X is the output and w_i are the weights of the connections from $x_0, x_1, \cdots x_N$ inputs.

$$X = \sum_{i=0}^{N} x_i w_i \tag{2.4}$$

A major difference between the McCulloch & Pitts neuron and the perceptron is that the perceptron includes a *bias* which is a constant term that does not depend on the inputs of the perceptron. Therefore, whereas in the perceptron i goes from 0 to N, in the McCulloch & Pitts neuron, it goes from 1 to N. The extra term $(x_0 w_0)$ in the perceptron is the bias. If $x_0 = 1$ and $w_0 = -t$, this defines the activation function as a step function with t as the threshold of the step function.

The fact that there is a learning algorithm used by the perceptron is also a major difference from the Pitts & McCulloch neuron. The perceptron algorithm first applies random initial weights to the inputs. Then, each item of the training data is presented and the weights are modified to move the current output closer to the desired output. This process is repeated (in *epochs*) until the output is as close to the desired output as is required.

2.1.3.2 Multi-Layer Feed Forward Networks

Perceptrons are single layer neural networks and can only model linearly separable functions. To model functions like the XOR, which are not linearly separable, multi-layer neural networks may be used. In *feed-forward* multilayer neural networks, signal flow is in one direction only – from inputs to outputs – and there is no feedback (Figure 2.2). The first layer is connected to the inputs and is called the *input layer*. This is followed by one or more *hidden layers*, which finally connect to an *output layer* that generates the outputs. Feed-forward networks can be described as being without memory since their outputs do not change after training, and the ANN acts like a mapping function. Radial Basis Function networks are feed-forward networks that use the radial basis function as their activation function (Liu, 2009). Radial basis functions are a regression model in statistics for probability density estimation.

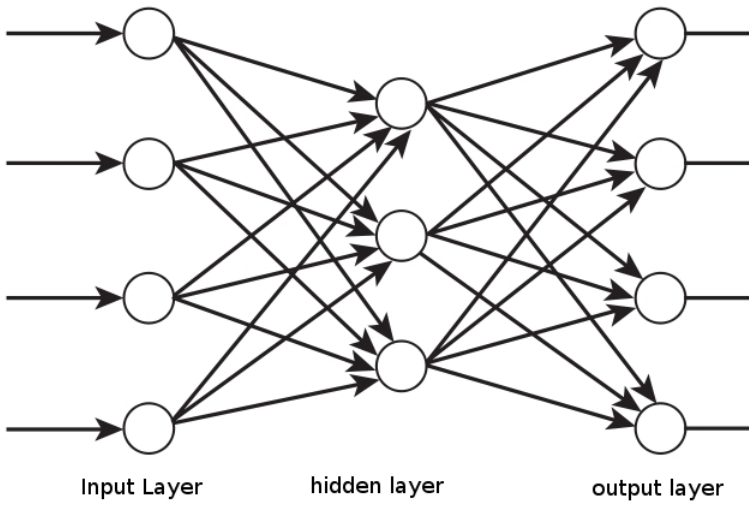


Figure 2.2: Multi-Layer feed forward artificial neural network. The image has been modified from Coppin (2004, Chapter 11).

Multi-layer networks form the second generation of ANNs with their continuous activation functions. Where the first generation ANNs are universal computation devices for digital computations, the second generation ANNs are universal for digital as well as analogue computations (Vreeken, 2003). Moreover, they are universal for digital computations with fewer neurons than the first generation of ANNs. One of the most popular and common method of training such networks is the *back propagation algorithm* and its variants, some of which are described in Tveter (2001), Liu (2009) and Almeida et al. (2008).

2.1.3.3 Recurrent Neural Networks

Multi-layer networks that have feedback connections are called recurrent neural network. In fact, recurrent networks can have arbitrary connections from any layer to any other layer and, as opposed to feed-forward networks, the dynamic properties of the network play an important part. Recurrent networks are attractor networks that feed inputs, including feedback, into the network until the outputs become stable. Therefore, they can be said to have memory since they use all previous inputs to generate outputs. However, if these networks are not designed properly, they can become unstable so that the outputs keep oscillating between attractors. One of the more famous recurrent neural networks is the *Hopfield network* (Hopfield, 1982) which was the cause for much of the interest in the field in the 1980s (Maji et al., 2002). John Hopfield proposed a single layer recurrent neural network with each node connected to all others and with the sign activation function defined in equation 2.5.

$$Sign(X) = \begin{cases} +1 & \text{for} X > 0 \\ -1 & \text{for} X < 0 \\ \text{no change} & \text{for} X = 0 \end{cases} \tag{2.5}$$

Details about the training method and mathematical properties of Hopfield networks can be found in Coppin (2004, Chapter 11).

Self Organising Maps (SOM) and Spiking Neural Networks (SNN) deserve a mention due to their popularity, but a review of these techniques is not considered central to the focus of the thesis because the objective of this review is to illustrate the kinds of operations that AI algorithms perform on existing hardware architectures. For a review of SOMs and SNNs, the reader may refer to Abraham (2005), Kasabov (2007), Liu (2009), Piuri et al. (1992), Ros et al. (2006), Stewart (2003), Vreeken (2003).

2.1.3.4 Associative Memories

Traditional memory stores data at a unique address and can *recall* the data upon presentation of the complete unique address. Auto-associative memories are capable of retrieving a piece of data upon presentation of only partial information from *that* piece of data. Hetero-associative memories, on the other hand can recall an associated piece of data from *one* category of input upon presentation of data from *another* category of inputs. Hopfield networks act as auto-associative memories since they are capable of remembering data by observing only a portion of the data. Biological neural networks, on the other hand, are hetero-associative memories since they can remember a completely different item to the one presented as input.

Bidirectional Associative Memories (BAM) (Kosko, 1988) are Artificial Neural Networks that have long been used for performing hetero-associative recall. They are a generalization of the Hopfield networks (Hopfield, 1982); and both (Hopfield networks and BAMs) have their beginnings in the Correlation Matrix Memories (Kohonen, 1972, cited in Acevedo-Mosqueda et al. (2006)). Hopfield networks can, therefore, be said to be BAMs with the following additional constraints:

- X inputs produce X outputs rather than Y outputs.
- The weight matrix is square.
- The neurons should not be connected to themselves or other neurons in their own layers.

Like other traditional neural networks, this 2-layer, non-linear, recurrent neural network starts by training on a defined training dataset and then moves on to the actual test set. The weighted-sum approach means that the capabilities can be analysed and proven mathematically. Furthermore, the original training algorithm does not require batch learning[4] and therefore is easily adaptable for real-time online learning. However, it results in sub-optimal utilization of capacity. Furthermore, as shown by Kosko (1988), the BAM can get confused when storing one-to-many or many-to-one relationships. In addition, if too many pairs are trained, the correlation matrix becomes unstable and the BAM is likely to forget all previously learnt pairs.

It is obviously desirable to maximise the storage capacity or at least to quantify it and many attempts have been made in this (latter) direction. Tanaka et al. (2000) performs statistical analysis and estimates the capacity of pairs to be retrievable (allowing a finite fraction of retrieval error) to be $0.1998N$ where N is the number of neurons in each layer of two layers. The fact that the capacity of a BAM is tied to the number of nodes is

[4]training set presented for learning during a distinct training phase

a drawback in terms of scalability. Wang and Vachtsevanos (1991) derive the storage capacity of a discrete BAM which is further verified by experimental results presented in Chapter 5. It shows that the storage capacity is much smaller than expected because of the *mis-learning behavior* where the BAM connection matrix confuses similar patterns because of the bipolar encoding scheme. Since bipolar encoding performs better on average than binary encoding, as shown by Kosko (1988), this mis-learning is unavoidable.

In essence, the problem is to maximise the number of patterns superimposed on one memory medium, namely, the weights of connections in a correlation matrix. Therefore, to guarantee recall, the training vectors *must* be orthogonal. One obvious solution is to re-encode non-orthogonal training data so that it becomes orthogonal as done by Simpson (1988, cited in Oh and Kothari (1994)). Of course this does imply that the complete training set is known in advance, an assumption that is not valid for online, real-time learning where training data is *encountered* sequentially and the complete size is not known in advance. Attempts to find the best training algorithm include Chen et al. (1997) which uses an exponential rule, Shen and Cruz (2005) which uses genetic algorithms, Zheng et al. (2005) which uses a descending gradient method, Eom et al. (2001) which uses linear programming techniques, among others. Ritter et al. (1999) is a non-iterative Morphological BAM while Wu and Pados (2000) is a non-iterative feed-forward BAM. Acevedo-Mosqueda et al. (2006) is yet another variation based on two binary meta-operators and called the Alpha-Beta BAM. Oh and Kothari (1994) present the Pseudo-Relaxation Learning Algorithm for BAM (PRLAB), which is an iterative learning algorithm for discrete BAMs that maximises the storage capacity of the BAM without the need for any preprocessing or re-encoding of training data. Furthermore, it guarantees perfect recall of all training pairs if it is possible to store them as stable states in the correlation matrix. All training pairs are examined cyclically in each iteration (called an *epoch*) and if an associative pair is not stored in a stable state, the weights are adjusted further. Where other algorithms suffer from parameter tuning issues (notably deciding the optimum step size to maximize learning for iterative algorithms), PRLAB is highly insensitive to parameter values and initial configurations. To guarantee training, the PRLAB needs to see *all* training pairs and adjust weights in multiple iterations. Sudo et al. (2009) adapt PRLAB for online incremental learning by sequentially providing it with associative pairs so that the algorithm observes only one pair. They show how this results in the BAM (and also Hopfield networks) forgetting previously learned data. However, this is an unfair comparison because the entire strength of PRLAB lies in the fact that it cycles through training pairs iteratively and makes changes in the weight matrix if the pair is not stored properly.

Hamming associative memories use Hamming distance to determine the correct element from the *fundamental set* that should be recalled. Although technically Hamming memories do not fall in the category of sub-symbolic methods, but rather are a form of instance-based learning (Section 2.1.1), they are discussed here along with other associative memories. As shown by Ikeda et al. (2001), Hamming memory has the following advantages over other associative memories:

- Huge capacity – exponential in the input dimension.
- Precise bounds on the error correction capabilities.
- No spurious memories – i.e. no data retrieved that was not part of the memory set.

It has the disadvantage, however, of slow retrieval speed and impractical hardware implementation, particularly since the data needs to be stored and searched every time a new input needs to be classified. Variations on the Hamming memory attempt to resolve these disadvantages. Grounded Hamming memories (Watta and Hassoun, 2001) output a ground state if the input does not match any element in the fundamental set with a small enough distance. Cellular Hamming memories (Watta and Hassoun, 2001) are based on Cellular Automata principles, and decoupled Hamming associative memories partition the input vector into non-overlapping local windows and perform the distance computation locally. However, this decoupling is dependent upon the data and whether it can be decoupled in the first place. For the typical case of image data, this decoupling does provide major improvements in hardware implementability. Two-level decoupled Hamming associative memories attempt to resolve the issue of spurious memory by placing a voter on top of the local decoupled memories. Although these advances go some way towards resolving the disadvantages of Hamming memories, the fact that nodes are based on general-purpose processors means that the local processing is serial and still a *search*. Implications of this are considered in Section 2.3.

As mentioned earlier, the previous sections are a summary of some of the more relevant areas of AI and are by no means a complete review of the domain. Areas excluded were considered irrelevant to the focus of this thesis or not significantly differently in terms of the kind of computation requirements that they place on the underlying hardware resources. Most notable among the missing are fuzzy logic (Coppin, 2004, Chapter 18), (Abraham, 2005, Pipe and Carse, 2007, Sahba et al., 2005, Stelzer et al., 2007, Wang and Mendel, 1992), swarm intelligence (Berger et al., 2006, Liu et al., 2008), (Coppin, 2004, Chapter 19), and artificial immune systems (de Abreu et al., 2006, de Abreu and Mostardinha, 2009, de Castro and Timmis, 2002, Elberfeld and Textor, 2009, Greensmith et al., 2005, 2007, Hart and Davoudani, 2009, Hofmeyr and Forrest, 1999, McEwan and Hart, 2009, Pagnoni and Visconti, 2005, Stibor et al., 2006, Timmis et al., 2008). The following sections present existing hardware architectures (Section 2.2) and then discuss the implications (Section 2.3) of using these architectures to implement the categories of AI algorithms described above.

2.2 Hardware Architectures

John Von Neumann is widely accepted as the father of the modern computer, since the principles he set out in his famous first draft in 1945 (Von Neumann, 1945)[5] form the basis of most computer architectures today. The Von Neumann architecture is characterized by a processing unit and a (logically) single, separate storage unit. Most of the area is memory and only a few locations can be accessed simultaneously. Instructions are fetched from memory and executed sequentially. The reason for this separation of memory and processing units is partly historical (Hillis, 1986): when the first computers were designed, the processors were made of fast, expensive switching components like vacuum tubes while memories were composed of slow and inexpensive components like delay lines and memory tubes. Therefore, it made sense to organise the architecture such that the expensive components – the processors – were kept as busy as possible. With the drop in relative cost for the processor components two major drawbacks have arisen. Firstly, since most of the area is memory, a large percentage of the architecture is idle for a large part of the time. Secondly, performance is limited by the bandwidth of data transfer between the processing unit and the memory unit(s). This is known as the *Von Neumann Bottleneck* (Backus, 1978). With the increase in processor speeds, this bottleneck is

[5]This is an exact copy of the original draft republished

becoming increasingly obvious. For a general overview of computer organisation and the Von Neumann architecture, the interested reader is referred to Stallings (2000).

Technically, today's processor architectures are based on a modified Harvard architecture (Wilkes, 1956). However, the difference between this and the Von Neumann architecture is only in allowing the processor to access two (or more) memory buses concurrently. Therefore, for the purposes of this review, the Harvard architecture may be considered as just another variation of the Von Neumann architecture. Accepting this, all general purpose computing architectures – from the ENIAC[6] and the EDVAC[7] to the modern Intel and AMD architectures – can be considered to be just variations of the Von Neumann architecture.

History has shown the Von Neumann architecture to be extremely successful, particularly for *number-crunching* and sequential applications. This is, in part, due to the causal nature of human intelligence, which attempts to solve all problems using a sequence of steps. However, processing in the natural world is inherently parallel and many problems need parallelism, while Von Neumann machines are sequential. Moreover, as shown by Hillis (1984), even a nano-second cycle time is not nearly fast enough in a serial Von Neumann architecture to solve million-scaled AI problems. The complexity of AI algorithms on sequential machines is dominated by iterative searches through a large concept space. Although Von Neumann style machines can be connected together to make parallel architectures, this introduces the problem of parallel programming. The following sections review some relevant architectures and attempt to identify broad categories.

2.2.1 Processor Arrays and Cellular Automata

John Von Neumann and Stanislaw Ulam invented the concept of Cellular Automata (CA) in the 1950s and proved that a CA was equivalent to a Turing Machine (Von Neumann, 1966). Their cellular automata were a collection of a number of *Finite State Automata*(FSA) arranged in a grid. A FSA is a device with finite potential states that processes inputs sequentially, using rules to determine the next state. In the original Von Neumann CA, each cell had a total of 29 possible states and the inputs were the states of the neighbours.

[6]Electronic Numerical Integrator And Computer
[7]Electronic Discrete Variable Automatic Computer

Typically, CA operate in discrete time steps and the state of a cell at time t is a function of the states of its neighbours (plus the cell itself) at time $t-1$. No cell is functionally unique and so all cells are homogeneous, although cells in different states may be performing different functions. In theory, no single cell should be critical to the correct behavior of the entire collection because of the large number of cells and the similarity in behaviour. If a single cell becomes faulty the rest of the cells may continue to operate correctly as long as the faulty cell is ignored by its neighbours. In practice however, CA have been shown to be quite fragile and brittle as neighbourhood size plays an important part in determining the influence of a faulty cell. Fault tolerance is discussed in greater detail in Chapter 3.

CA are not simply a software algorithm, but fit quite naturally into a hardware implementation of cells operating in parallel. As indicated by Ortega and Tyrrell (1999), from the perspective of fault tolerance, processor arrays can be considered as a special case of CA. Therefore, cellular automata principles can be extended to processor arrays like IBM's BlueGene/L supercomputer[8] using 1000 dual PowerPC 440 processors (Gara et al., 2005), or Intel's 80-tile tera-flop chip (Held et al., 2007, Vangal et al., 2008), or NASA's SGI Columbia supercomputer[9] with 500 Intel 1.5 GHz Itanium2 processors (Mavriplis et al., 2005), or even the university of Manchester's SpiNNaker which will have more than one million cores operating at 200MHz (Navaridas et al., 2009).

The classical CA is deterministic such that the next state can be unambiguously determined from the current state of the CA. Starting with various different initial configurations, the simple rules of Conway's *Game of Life* generate interesting and emergent behavior – most notably, the glider and the glider-gun (Gardner, 1970). The glider-gun and Langton's *q-loops* are prime examples of how CA can display reproductive behavior. CA variants called Probabilistic Cellular Automata (PCA) use a probability distribution for the transition. Other notable variants include the CA with Monte Carlo procedure and the CA using Ordinary Differential Equations in each cell (CA-ODE) to model average behavior of large numbers of particles. The method of Direct Simulation by Monte Carlo models reaction-diffusion equations while reactive Lattice Gas CA describe spatially extended dynamic systems and are based on modelling microscopic collisions between particles. For more details about CA, including Von Neumann's original CA, the game of

[8]`http://domino.research.ibm.com/comm/research_projects.nsf/pages/bluegene.index.html`
[9]`http://www.nas.nasa.gov/hecc/resources/columbia.html`

life and q-loops, refer to Coppin (2004, Chapter 13). Rossier et al. (2004) adds the Tom Thumb Algorithm to the Von Neumann Universal Constructor to decrease the number of cells required, thus allowing physical implementations. Coppin (2004, Chapter 20) describes how FSA[10] can be used in natural language processing while Coppin (2004, Chapter 19) describes each layer in the *subsumption architecture* for robotics as an FSA. Mendao et al. (2007) uses a CA platform called Netlogo to model B-Cell interaction with antigens. Almeida et al. (2008) merges artificial neural networks and CA for modelling land-usage dynamics. Atrubin (1965) presents a CA-based real-time iterative multiplier.

CA have been shown to be successful in finger-print and traffic analysis, pattern formation, swarm motion, optimal path searching, for modelling biological systems and dynamic systems. They have been used as parallel multipliers and as sorting machines, for image processing and pattern recognition. Maji et al. (2002) implements a general purpose pattern recognition system using CA. The work is based on a class of CA called *General Multiple Attractor Cellular Automata* (GMACA) which use non-linear CA rules. Experiments show this approach to be better than a Hopfield neural network for memorizing unbiased patterns. A review of these applications and more can be found in Ganguly et al. (2003) and Vanag (1999); the latter also discusses the advantages and disadvantages of CA in great detail. However, the greatest difficulty in using CA is determining the local rules for each CA that would produce the desired behavior in the complete automaton. Heuristic models tend to have severe computational constraints. In the past, many techniques have been attempted including Hierarchical Analysis Process, Genetic Algorithms, logistics regression models, Artificial Neural Networks, Decision trees, Kernel-based machine learning, Ant Colony Optimization and so on (Liu et al., 2008). Despite all these efforts, no clear solution has arisen, which is the reason for the waning interest in CA.

Cellular Automata have been shown to be Universal machines (Vanag, 1999, Wolfram, 1984) and as can be seen from a review of applications above, they have been successfully applied to a very large variety of applications. Some of the notable existing Cellular Automata based machines include:

- the Cellular Automata Machine (CAM),

[10]Finite State Automata

- the Connection Machine (CM) which was commercially produced by Thinking Machines, and
- the Massively Parallel Processor (MPP) of Goodyear Aerospace Corporation which was one of the fastest computers in the early 1980s.

2.2.1.1 The Cellular Automata Machine

The 1980s was a period when there was considerable interest in Cellular Automata, and the Cellular Automata Machine (CAM) came out of research done by Tommaso Toffoli at MIT during this time. Simulating CA behavior using general purpose serial computers of the time was not a viable option as it took too long and a VLSI solution, then, as now, was too expensive for experimental implementation. Therefore, the aim was to devise a parallel architecture that would allow simulation, experimentation and research in CA. The CAM was designed to implement 65,536 (64 K) cells arranged in a 256×256 grid, where each cell had an 8 bit state. It included a user port to allow communication with a general purpose computer and a display port to output the simulation to a monitor. Extensions for random number generation and time generation were available to allow the development of CA with non-deterministic or time-dependent rules respectively. Each cell had a unique 16-bit address (64 K address space) to allow space or position dependent rules as well. The rule table was placed in shared memory which meant that the architecture was not truly parallel since cells accessed the table sequentially. Although connections between cells were not fixed, changing connections meant changing positions of jumpers on the cards, so reconfiguration of connections was not dynamic. The architecture was based on $4.5'' \times 6.5''$ cards in a 10-slot card cage connected together using a backplane bus. In its simplest form, this included:

- an *interface card* containing user port, monitor port, system registers, cursor counter, and other interfacing logic,
- a *scanner card* containing all timing, control and addressing circuitry,
- a *planes card* containing 2 planes with memory, pipelines, buffers, multiplexers,
- a *tables card* containing 2 separately addressable 4×1024 bit tables.

For more details about the CAM hardware and programming possibilities the interested reader is referred to (Toffoli, 1984) and (Ganguly et al., 2003).

In essence, the CAM is an architecture dedicated to the simulation of CA and runs much faster than general purpose computers (of the era) programmed to do the same task. This means that the CAM can be used to run CA based solutions to the large range of problems listed in Section 2.2.1. Implications of this are considered in Section 2.3

2.2.1.2 The Connection Machine

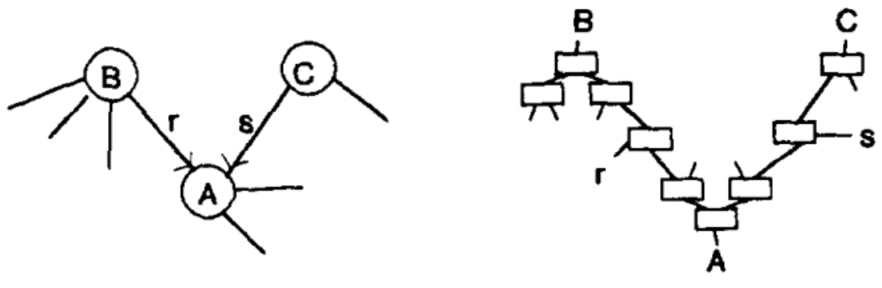
In the 1980s, Daniel Hillis came up with the Connection Machine (CM) which sparked considerable interest for more than a decade and spawned at least 4 commercial versions by Thinking Machines, including the CM-1 (1986), CM-2 (1987), CM-200 (1991) and the CM-5 (1993). Hillis postulated that using serial machines for AI is not a viable solution because even a nano-second cycle time in a serial Von Neumann style machine would not be enough to solve the million-scaled problems of AI[11] in real-time (Hillis, 1984). His solution was to move processing into memory by designing a parallel architecture based on Cellular Automata. This machine was targeted for AI applications; more specifically it was based on the principles of semantic networks (Section 2.1.2) which was the popular AI technique of the time. Best-fit pattern matching, sorting, searching and deduction were identified as typical operations for AI algorithms that needed to be fast. Hillis (1986) identifies the following two requirements for such a machine:

1. Many processors: just as memory is “infinite” in Von Neumann architectures, processing elements should be “infinite”.
2. Programmable connections: interconnections between nodes should be dictated by the problem and should be dynamically changeable.

The Connection Machine consisted of many tiny processing elements or cells connected together in a communication network such that connections mimicked the specific problem. All computation took place through the exchange of messages. The CM was designed before the advent of reconfigurable hardware like Field Programmable Gate Arrays (Section 2.2.4), which meant that cells in the CM had hardwired connections. However, nodes in

[11] AI problems that use millions of facts to come up with solutions.

semantic networks need an arbitrary number of connections. The solution adopted was to implement semantic networks using binary trees such that one of more cells would form a single node in a semantic network. This is illustrated in Figure 2.3. If a node needed



(a) Nodes (b) Cells

Figure 2.3: Representing nodes in terms of cells in the Connection Machine (Hillis, 1984). The node link labels r and s are links from specific cells and the nodes labelled A, B and C are specific cells. This is similar to the way FPGAs use some logic blocks only for routing. See Section 2.2.4 for FPGAs.

more than 3 connections, one parent and one child cell could be used. If 4 connections were needed, one parent and 2 child cells would be able to provide the connections. For 5 connections, 1 parent, 2 children and 1 grand-child cell would be sufficient, and so on. Therefore, each cell in the CM had a fixed number of neighbours – a parent and 2 children. Hardware support was included to facilitate the maintenance of such binary trees. Each cell in the original Connection Machine (CM-1) consisted of the following:

- an Arithmetic and Logic Unit (ALU),
- a state vector which stored condition flags, cell type and markers for marker propagation (Section 2.1.2 has details about semantic networks)
- 5 local 20-bit registers – 1 for each of 3 neighbours and 2 temporary registers.

Cells did not contain local copies of the rules and the rule table was shared, which leads to the obvious bottle-neck. Messages between cells were broadcast using wave propagation (Section 2.1.2) and Hillis (1984) shows how this results in fault tolerance since faulty cells would be ignored. The connection machine operated as a co-processor supervised by a master or a host. The features of the first CM-1 can be summarised as follows (Hillis, 1986):

- 64K cells with 4Kb memory and a simple serial ALU,

- packet switched network – boolean n-cube topology.
- adaptive routing algorithm,
- Single Instruction Multiple Data (SIMD) instruction stream from conventional host controller (Section 2.3.3),
- 1000 MIPS (Millions of Instructions Per Second) in terms of 32-bit additions,
- 12,000 watts running at 4Mhz including cooling mechanism,
- raw processing power:
  - 2.5×10^8 bits memory
  - 2.0×10^{11} bits/second memory bandwidth
  - 3.3×10^{11} bits/second processor bandwidth
  - 3.2×10^7 bits/second worst case and 1.0×10^9 bits/second typical case communication bandwidth
  - 5.0×10^{10} bits/second Input/Output bandwidth

The CM was supported by a customised version of LISP (LISt Programming) called CMLisp (Connection Machine LISt Programming). For complete details of CMLisp and the Connection Machine, the interested reader is referred to (Hillis, 1986). There have been some attempts at parallelizing AI algorithms, particularly using the more mature CM-2. Results from parallel implementations on CM-2 for the ANN perceptron algorithm (Section 2.1.3), the AQ algorithm (Michalski, 1983, cited in Grzymala-Busse (1993)) and the decision tree induction ID3 algorithm (Section 2.1.1) can be found in Cook and Holder (1990). These results as well as other implications of AI implementations on CM are discussed in more detail in Section 2.3

2.2.2 Production System Parallel Architectures

Production systems were introduced in Section 2.1.2.3 as rule-based expert systems and Amaral and Ghosh (1994) presents a parallel hardware architecture for production systems based on the Rete algorithm. It is composed of multiple, identical processing elements, each structured as shown in Figure 2.4. Rules are initialized by an *I/O processor* using the

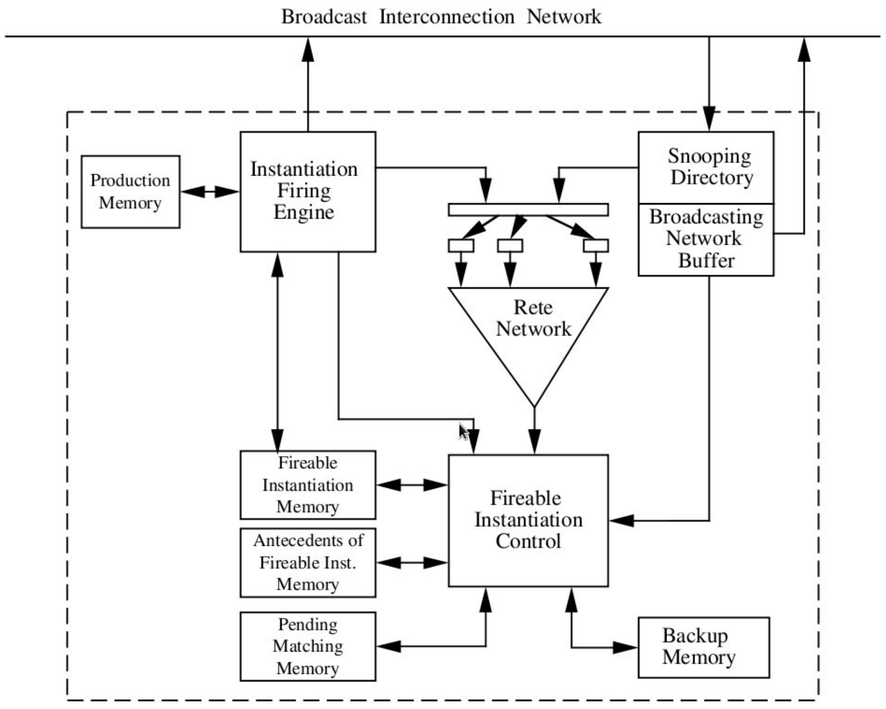

Broadcast Interconnection Network
Production Memory
Instantiation Firing Engine
Snooping Directory
Broadcasting Network Buffer
Rete Network
Fireable Instantiation Memory
Antecedents of Fireable Inst. Memory
Pending Matching Memory
Fireable Instantiation Control
Backup Memory

Figure 2.4: Production system processing element model (Amaral and Ghosh, 1994)

Broadcast Interconnection Network. In order to cater for inter-production dependencies and workload balancing, a partitioning algorithm uniquely assigns rules at compile time to different processors. Each element includes a working memory where antecedents are stored locally and the tokens passed to elements are *deletion*, *addition* or *modification* of a working memory element. Amaral and Ghosh (1994) evaluates performance using a modified version of the travelling salesman problem and shows that there are bottlenecks in the architecture because of which more than 10 elements do not provide a justifiable increase in performance. Therefore, scaling up to larger problems is an issue.

More recently, Lee et al. (2006) presents a parallel hardware accelerator for general purpose

Feature	TOTEM	TOTEM3	SoftTOTEM
Processors	32	128	32
Cycle Time	30ns	25ns	25ns (EPP * limited)
Technology	1.2μm	0.8μm	Virtex 600-E
Transistors	250K	1.5 M	6570 slices
On chip RAM	32 Kb	256 Kb	80 Kb
Performance	1GMAC/s	5 GMAC/s	3.8 GMAC/s
Die Size	70mm	170mm^2	

Table 2.1: Comparison of TOTEM architectures

*Enhanced Parallel Port

processors with hardware support for the Rete algorithm. The work includes software support for programming the custom hardware and also presents results of comparison with C code running on a P4 2.4GHz and an ARM processor running at 400MHz, both with and without the hardware accelerator. Implications of implementing this are discussed in Section 2.3.

2.2.3 TOTEM

Anzellotti et al. (1995) designed the TOTEM architecture to act as a co-processor for applications with inductive learning on Multi-Layered Perceptrons or second generation ANNs. The training of the ANN is viewed as a combinatorial optimization task and solved using *Reactive Tabu Search* (refer to Section 2.3.1 for reactive tabu search). The original TOTEM neuro-chip includes 32 processing elements each with 128×8 bit RAM, a 16-bit broadcast input bus connection, a 32-bit output bus connection and a Multiplier/Accumulator (MAC). A 128×8 weight memory implies that neurons with up to 128 connections can be implemented in one TOTEM chip. The MAC implements the Baugh-Wooley algorithm with a one-stage pipeline for the multiplier, and a ripple carry adder with a 2-stage pipeline. Figure 2.5 shows a simplified version of the TOTEM architecture. The chip has a 32-bit output bus and memory words are 8-bit wide. If sigmoid transfer functions are required, they must be implemented using off-chip RAM based look-up tables.

In successive years, there have been various other versions including the TOTEM3 which integrates 4 TOTEM chips into one, the *SoftTOTEM* (McBader et al., 2002) implemented on an FPGA and the LogTOTEM (Lee et al., 2007) which uses a Hybrid-Logarithmic Number System. Tables 2.1 compares some of the versions in terms of performance and specifications. Note that the SoftTOTEM is implemented on Xilinx Virtex XCV600E FPGA (refer to Section 2.2.4 for details about FPGAs).

2.2.4 Reconfigurable Hardware

The hardware architectures discussed thus far have been fixed at runtime in terms of function and the connections between nodes. Cellular arrays can be said to display a level of *virtual* reconfiguration of function by changing the states of nodes, which allows different

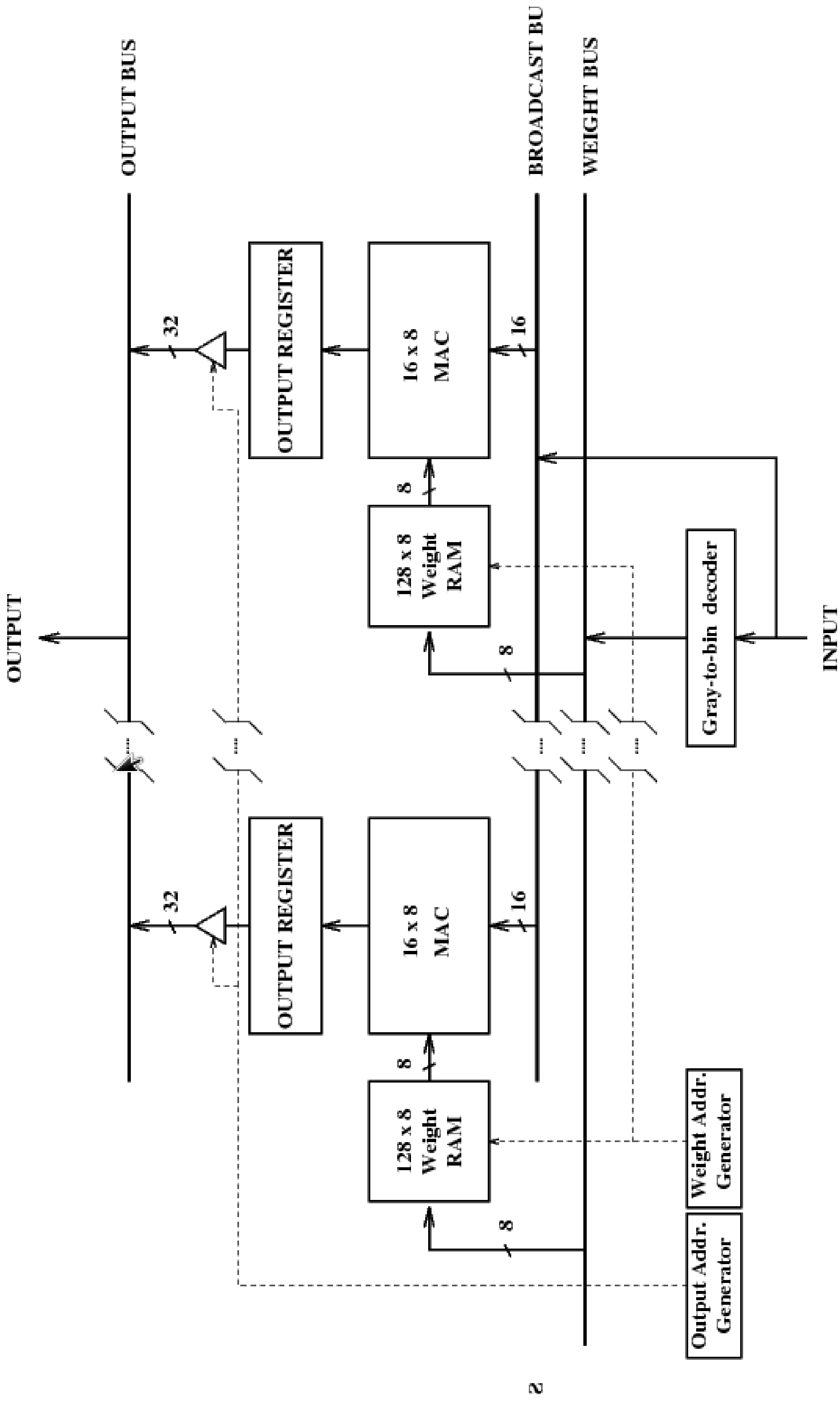


Figure 2.5: Simplified block diagram of the TOTEM chip (Anzellotti et al., 1995)

nodes to perform different operations on the same data. However, this adaptability of function has to be programmed in before the array is brought online and cannot be changed at runtime. Similarly, reconfiguration of virtual connections in architectures presented so far could be achieved where nodes in the array were embedded with packet routers. This has the advantage of allowing dynamic (at runtime) reconfiguration of connections but incurs a (potentially significant) cost in hardware. Although Programmable Logic Devices (PLD) have been in existence since the 1970s, reconfiguration of connections could only be achieved using jumpers and cables in the design which had to be manually shifted and reconnected. Similarly, in the 1980s architectures such as the Connection Machine (described in Section 2.2.1.2) allowed reconfiguration of connections by manually moving jumpers and wires.

The advent of reconfigurable hardware in the late 1980s and 1990s was a major revolution in the field of hardware design. Formally, reconfigurable hardware may be defined as hardware whose functionality can be changed post-fabrication through reconfiguring and reconnecting logic blocks. Xilinx marketed its first Field Programmable Gate Array (FPGA) – the XC2064 – in 1985 (Xilinx, 1985), which boasted 64 configurable logic blocks (CLBs), with two 3-input lookup tables. Figure 2.6 (from Xilinx, 1985) is a section of the FPGA showing how CLBs are connected together using a switching matrix. Although

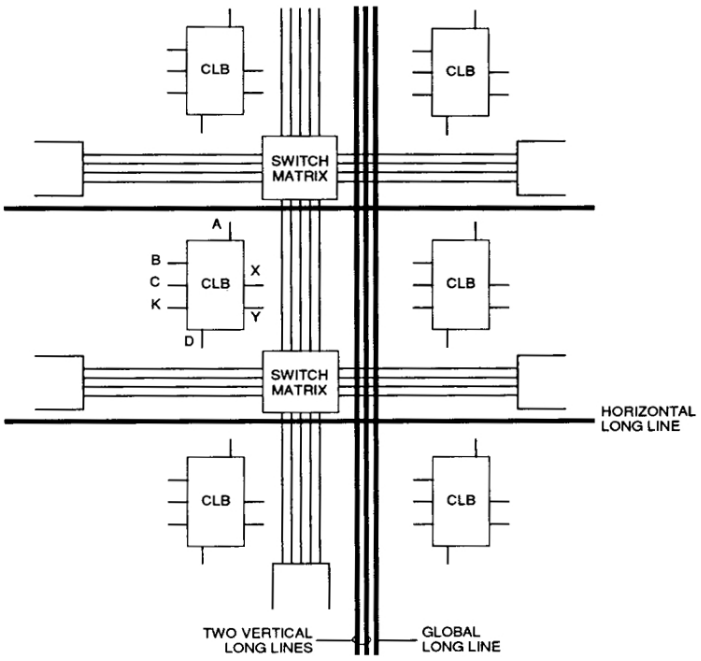


Figure 2.6: Xilinx XC2064 Configuration Logic Block array and interconnect architecture (Xilinx, 1985)

these did not allow reconfiguration at runtime, they provided a significant improvement by combining the flexibility of the *virtual* solutions with the performance of customised hardware solutions. This was primarily due to their ability to modify the data path on demand after fabrication.

Over the years, reconfigurable hardware has come a long way from that first FPGA. Even though today's technology still cannot compete with ASICs in terms of power consumption or clock speed, they usually provide a faster and more power-efficient implementation as compared to software implementations on micro-processors. Coupled with the convenience of upgrading designs and the increase in non-recurring engineering costs for ASICs, FPGA-based implementations are becoming exceedingly popular and cost effective, even for mass produced commercial products. Other notable advantages for the research community include flexibility in design, smaller design time with the ability of fixing post production errors through patches, and increased fault tolerance, both for production faults and those introduced by the environment (Garcia et al., 2006). As such, the advantages of FPGAs for implementing AI algorithms is quite evident from the plethora of publications for FPGA-based AI and robotics implementations in the literature. SoftTOTEM (Section 2.2.3) is just one such example.

FPGAs today are based on memory, which is used to store the function in Look-Up Tables (LUT) and also to control the switches for interconnections between logic blocks. This memory is usually SRAM, but other options like antifuse are also possible (Hamdy et al., 1988). Some devices employ fuses, but these are one-time programmable devices only and technically not *re*configurable hardware (though they are configurable hardware). The most common reason for using such (fused) devices is security, typically in cryptographic applications.

LUT based FPGAs, using the sum-of-products approach, with some level of coarse-grained blocks like multipliers, shifters, memory blocks are the most common form available. Many versions are available today with some providing hard macros for communication structures and even processors.

The flow for the design and implementation of logic in reconfigurable hardware uses CAD tools and is summarised in the following steps.

1. The logic is described in some form of HDL (Verilog/VHDL) and the function is

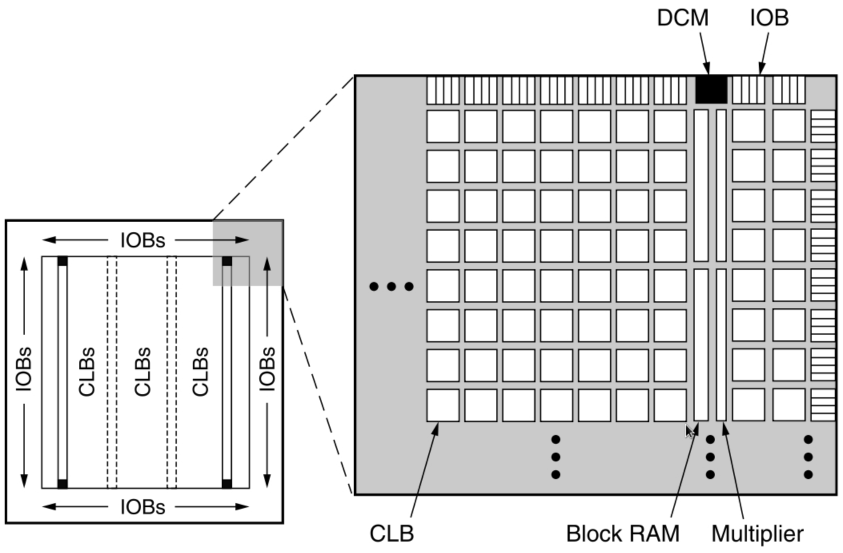


Figure 2.7: Xilinx Spartan 3 FPGA family architecture (Xilinx, 2008). IOBs are Input/Output Blocks and DCMs are Digital Clock Managers.

tested.

2. The circuit is synthesized converting the HDL into a structural circuit netlist.
3. The mapper decomposes the netlist into components matching capabilities of basic blocks, like the LUT or the ALU, in the reconfigurable hardware.
4. The placer determines which netlist components should be assigned to which physical hardware blocks.
5. The router decides how best to use the reconfigurable hardware routing fabric to connect these blocks.
6. A bit-stream generator produces binary values to load into the configuration bit stream for the desired configuration.

For more details and a survey of reconfigurable hardware technologies, refer to Kuon et al. (2007), and for a survey focusing on embedded systems refer to Garcia et al. (2006).

Traditionally, configuring an FPGA would implicitly include clearing any previous configuration in the hardware. As may readily be seen, this is a limiting factor since partial reconfiguration could provide a major benefit in terms of fault-tolerance and flexibility. In appreciation of this, some of the more recent FPGAs – like the Virtex (Xilinx, 2002a) –

do allow partial reconfiguration (Sedcole et al., 2005, 2006) and in most cases the reconfiguration is performed using an on-board micro-processor like the PicoBlaze. This does not preclude other methods for reconfiguration as shown by Legat et al. (2011) where the authors present an architecture that performs partial reconfiguration using custom logic. The amount of configuration memory required in an FPGA varies with FPGA brands and types. The Virtex FPGA series by Xilinx is by far the most popular, particularly in terms of partially reconfigurable FPGAs and therefore these are considered here. Specifically, the Virtex 5 series LX110T (Xilinx, 2010b) is considered as a representative model since it is a medium sized FPGAs that has been used extensively and is still supported by Xilinx Inc. Furthermore, this FPGA was also physically available for experiments later (Chapter 6). The LX110T resources are summarised as follows (Xilinx, 2009, 2010b):

- 160×54 Configurable Logic Blocks,
- $17,280$ slices,
- $1,120$ Kilobits of distributed RAM,
- 64 DSP48E slices[12],
- $5,328$ Kilobits of Block RAM,
- 1 PCI express endpoint block,
- 4 Ethernet MACs,
- 680 maximum amount of User I/O pins
- $69,120$ flip-flops.

As may be seen in Xilinx (2010a), the LX110T requires 3,888,768 bytes of configuration memory and has 1,088 bytes of bit-stream overhead, defined as commands in the bit-stream which are required to perform configuration but do not actually program the memory. This configuration memory adds to the overhead in a number of ways. Firstly, transmitting and configuring the FPGA requires time. Though this problem is somewhat mitigated by the fact that configuration is a one-time process or at least an infrequent process, the problem is only exacerbated with scaling up. Furthermore, for the case of

[12]Containing a 25×18 multiplier, an adder, and an accumulator.

dynamic reconfiguration, the time for reconfiguration can be important. Malik and Diessel (2005) present a comparison of dynamic reconfiguration times using various techniques.

Secondly, for a device with a total of 69,120 flip-flops and 5,328 Kilobits of Block RAM, 31,110,152 bits of configuration memory is certainly quite a high proportion of the design. Thirdly, this configuration memory adds to the possible area where faults can occur and if a fault tolerant design is to be implemented, the functional area must also check for faults in the configuration memory. Fourthly, existing commercial FPGAs do not typically perform any checks on the configuration bit-stream to test its validity. This could result in an incorrectly generated (or transmitted) bit-stream to cause serious damage to the FPGA fabric. Greensted and Tyrrell (2007) presents the Reconfigurable Integrated System Array (RISA) architecture which goes some way towards protecting the reconfigurable fabric by implementing hardware checks to protect against signal contention and combinatorial feedbacks. This makes RISA quite well-suited for implementing evolutionary algorithms. However, such technology is still not commercially available and RISA cannot provide the sort of flexibility, capacity, fine-grained reconfiguration ability or software support that is essential to make it a viable alternative. From this discussion, it can readily be seen that a reconfigurable fabric that maintains the flexibility of an FPGA but removes the necessity for a configuration bit-stream would be useful.

2.3 Implementation of AI techniques on Traditional Hardware

The issue of a formal definition of AI was dealt with in Section 2.1. Informally, an AI application may be defined as one that tries to achieve behaviour displayed by intelligent beings like humans (Turing, 1950), and most will agree that the hallmark of AI applications is complexity. It will readily be seen from Section 2.2 that the state-of-the-art hardware architectures for AI are based on (potentially multiple) conventional processors, where conventional processors are defined as containing:

- an ALU that calculates results,
- a control unit that directs execution based on a program that is fetched from memory,

and

- a memory that stores data (and potentially instructions as well).

Modern general purpose processors implement simple (mathematical) operations in ALUs, sometimes with additional hardware support for more complex but commonly used mathematical operations (like trigonometric functions). These operations are accessed by (machine) instructions to a control unit in the processor, which is further accessed through (potentially multiple) software/firmware wrapper layers, which may be an Operating System or a Virtual Machine or both. Therefore, implementing complex AI applications on traditional machines not only involves mapping an individual complex operation to multiple simple mathematical operations, but also passing each operation through multiple software layers to reach a hardware unit (ALU) that is accessed through yet another hardware wrapper (the control unit). Tsafrir et al. (2005) show that a 2.8GHz P4 processor, with 512MB 400MHz DDR SDRAM[13] and 512KB L2 cache operates 1.6 times slower than its optimum on a benchmark application due to the operating system overheads. Typically, such overheads are considered an acceptable price to pay for having a Turing complete general purpose machine. The following sections examine the implications for AI applications that need to traverse all these layers.

2.3.1 Search

Section 2.1 presented a brief overview of the methods and algorithms in use for Artificial Intelligence and showed how these methods are dominated by search. From this review and from the discussion of the hardware architectures in Section 2.2, it can be inferred that the serial nature in which computers tend to operate makes search a necessity for solutions to a large range of problems; something that is further corroborated by Coppin (2004, Chapter 4). Coupled with the causal nature of human intelligence, this means that researchers tend to talk about “searching the landscape” or “finding a solution in the search space” when they attempt to solve an AI problem. This is the functionalist approach which asserts that intelligence is a result of the process and not the substrate, as long as the substrate has sufficient computational richness to realise the process (Johnson, 2007). Since general purpose processors are Turing Complete, the functionalist also

[13]Double Data Rate Synchronous Dynamic Random Access Memory

typically assumes that the computer is a sufficient medium for the implementation of such algorithms. The following is a review of existing methods used to perform such *searches* with a focus on using conventional processors (as described in Section 2.3).

Search methods may be *data-driven* (forward chaining) or *goal-driven* (backward chaining). Data-driven search starts from an initial state and proceeds using *permitted* actions until a goal state is reached. This is particularly useful when the goal is not clear but the initial state is known. Alternately, goal-driven search begins from the goal state and moves backwards using *permitted* actions, until an initial state is reached. As may be expected, this is useful in situations where the initial state is not clear, but the goal state is known.

2.3.1.1 Brute-Force methods

The most straight-forward place to start when searching for a solution is to use brute-force methods. This implies starting from an initial position (usually one end of search space) and exhaustively testing *all* possibilities to find the right solution. One of the more popular methods in this category is tree-based search, which is used in many game-playing systems. Data first needs to be represented as trees – for example semantic trees (Section 2.1.2) – which can then be traversed to find the correct solution. One way to traverse the tree is *depth-first*, where child nodes are explored down a path until a *leaf* node is reached, and if a solution is not found, the algorithm backtracks to search down another path. This method has the obvious disadvantage of being slow if tree are deep, with the extreme case of getting stuck in infinitely long paths. *Breadth-first* search avoids these pitfalls by parsing through all the nodes at the same depth (or level) before going down to the next level. As opposed to depth-first, breadth-first performs poorly if the trees have a high branching factor so that the number of nodes in each level can increase drastically with deepening depth. Although both of these are brute-force methods, depth-first algorithms typically use less memory and are easier to implement (in terms of writing the code). Furthermore, infinite paths (or effectively infinite paths) can be limited by thresholding the depth to be explored in each path. This forms the basis of the popular *Depth-First Iterative Deepening* (DFID) algorithm.

Depth-First Iterative Deepening iteratively searches the tree by increasing the depth threshold in each successive iteration. Therefore, in the first iteration, the threshold is set to 1 and a depth-first search is conducted. If the goal is not found, the threshold is set to 2 and another search is conducted, and so on. A tree with a branching factor b and depth d will have one root node, b nodes in the first level, b^2 nodes in the second level and so on. Therefore the total number of nodes in such a tree can be described as equation 2.6 which forms the geometric progression shown.

$$1 + b + b^2 + b^3 + \cdots + b^d = \frac{1 - b^{d+1}}{1 - b} \tag{2.6}$$

As may readily be seen, DFID examines nodes more than once. The root node may be examined up to a total of $d + 1$ times; level 1 nodes may be examined up to d times; level 2 nodes may be examined up to $d - 1$ times and so on. Therefore, as shown by Coppin (2004, Chapter 4), the time complexity of DFID is $O(b^d)$ while the space complexity is $O(bd)$, which implies that it is a good solution for large trees but is inefficient for small ones. DFID is *optimal* which means that it is guaranteed to find the best solution and it is *complete* which means that is guaranteed to find a solution if it exists. Nonetheless, note that the DFID is also a form of brute-force search. Although there are other tree search algorithms, these are considered to be variants of the three listed here which are therefore representative of their category. An analysis of hardware implications of such search methods is presented in Section 2.3.1.3.

2.3.1.2 Heuristic methods

Section 2.3.1.1 described some brute-force search methods. Although such methods will *eventually* find the solution, they can usually be improved by using domain knowledge. Methods that rely on such information are called *heuristic* search methods, where a heuristic is a piece of information used to make a search more effective. The search is conducted using a heuristic evaluation method that returns an estimate of the distance of the node from the goal node (or the cost of going from the node to the goal node). Domain knowledge is used to create a search space composed of peaks and valleys. Assuming that the objective is to find the highest point, the algorithm would choose the next step by going to a point higher than its current state. Such algorithms are called hill-climbing algorithms, which were introduced by Minsky (1961). The following is a brief survey of heuristic methods, and the interested reader is referred to Coppin (2004, Chapter 4) for more details.

Tree based search mechanisms are quite popular because they fit the causal nature of the human thought process. It is no surprise therefore that the brute-force tree-based search methods described in Section 2.3.1.1 have been adapted for heuristic search over the years. Beam search is one example of a heuristic breadth-first search such that only the *n best* paths are expanded, thus providing an increase in efficiency for both time and memory. On the other hand, upon discovering a path to the goal node, the British Museum procedure (Newell et al., 1958) stores the path and continues on with the search. Thus if a *better* path is found later, it can replace the one stored.

Perhaps one of most successful algorithm for searching tree-like structures is the popular *A** (pronounced "A star") algorithm. The evaluation function used is *f(node) = g(node) + h(node)*, where *g(node)* is the actual cost incurred so far in reaching the current node and *h(node)* is an (under)estimate of the cost of going from the current node to the goal state. Thus *h(node)* is an *admissible* heuristic evaluation function, which is defined as one that never overestimates the cost of changing the state from a given state to the goal state. If *h(node)* is not admissible, the algorithm is A not A*. Dechter and Pearl (1985) shows that as long as *h(node)* is an admissible heuristic then A* is complete[14], optimal[15] and optimally efficient[16].

Typically, heuristic methods choose the next state that has the least cost associated with it. An obvious failing of such algorithms is their inability to extricate themselves out of local maxima (or minima). One way to resolve this is to maintain a list of all the states that have been visited so as not to visit these again. This is known as *tabu* search, since the visited states become tabu (Glover, 1989, 1990) and is a *metaheuristic*[17] search method. Variants of the tabu search attempt to optimize parameters like the number of time steps for which an observed state is prohibited or the number of states that can be stored. The *Reactive Tabu Search* (Anzellotti et al., 1995, Lee et al., 2007) starts by setting $T = 1$, where T is the number of time steps for which a reverse movement is prohibited, so that only the last move is prohibited. This threshold can be increased or decreased if there

[14]Guaranteed to find solution if it exists.

[15]Guaranteed to find the *best* solution that exists.

[16]Expands the fewest possible paths to the goal node.

[17]Metaheuristic algorithms aim to apply heuristic search to a wide range of problems (Johnson, 2008), which typically do not allow an analytical solution.

is evidence that diversification is required, for instance if repeated states are observed. Although many other algorithms exist that try and resolve the issue of local maxima (or minima) a review of all of these is considered irrelevant since they are similar in terms of their implications on the underlying hardware; therefore these are beyond the scope of this document. A special mention must be made, however, of evolutionary algorithms (Abraham, 2005, Almeida and de Abreu, 2003, Dawkins, 2003, Hotz, 2003, Kumar and Bentley, 2003a, Ray and Hart, 2003, Wolpert, 2003), (Coppin, 2004, Chapter 14) and simulated annealing (Abraham, 2005, Maji et al., 2002) which are two algorithms inspired by processes in the natural world and have had success in recent years. The following section analyses the implications of implementing the search algorithms discussed thus far on current hardware architectures.

2.3.1.3 Implications of search using existing hardware

Sections 2.3.1.1 and 2.3.1.2 discussed some of the existing search algorithms. It was considered unnecessary to present a complete review of search algorithms because the objective of this review was to identify the kinds of operations that the algorithms perform on hardware, and the algorithms discussed were considered to be good representations of their categories.

Using conventional general purpose processors, depth-first algorithms are best implemented using stacks while breadth-first algorithms are best implemented using queues (Coppin, 2004, Chapter 6). Depth-First Iterative Deepening attempts to merge the advantages of depth-first and breadth-first and therefore, typically requires a mixture of queues and stacks. Note that the data structures used only affect the way memory is addressed and the search is dominated by large amounts of memory accesses, even in the case of heuristic search methods. Therefore, performance is once again limited by the Von Neumann bottleneck.

Trees are typically used in symbolic methods (Section 2.1.2) and testing a candidate solution for correctness (during search) is usually not computation expensive, but rather more of a comparison. Over the years, various customised hardware systems have been suggested to mitigate the problems of implementing such tree-based search on serial machines. The Connection Machine (introduced in Section 2.2.1.2) provided hardware support for maintaining and searching binary trees, while production system architectures (Section 2.2.2) typically use the Rete algorithm (Section 2.1.2.3) to search trees. The following deals with both of these separately in an attempt to highlight the overall problems of such parallel arrays.

Despite being a major player in the 1980s and 1990s, the Connection Machine did not stand the test of time. An important factor in this was the financial cost involved, which was more than a million dollars for the CM-5 with 32 nodes in the 1990s. However, equally important were the evaluations of its performance. Cook and Holder (1990) compared parallel implementations of popular AI algorithms like the ANN perceptron algorithm (Section 2.1.3), the AQ algorithm (Michalski, 1983, cited in Grzymala-Busse (1993)) and the decision tree induction ID3 algorithm (Section 2.1.1). Implications for the ANN implementation are discussed in Section 2.3.2; however the results were quite unfavourable for the CM and showed that parallel implementations of popular AI algorithms did not benefit significantly from the potential speed up through the parallelism provided by the CM. Furthermore, it showed that although the CM can parallelize operations, these must be the same for each piece of data, as SIMD (Section 2.2.1.2) limits parallelization. Cook and Holder (1990) also stress the under-development of CMLisp (Connection Machine Lisp), which is a version of LISP (LISt Programming) for the CM. This difficulty in programming the CM is, in fact, not an isolated case, but is indicative of a wider problem relating to the programming of parallel arrays composed of general purpose nodes. However, this is dealt with in more detail in Section 2.3.3.

In the 1970s, Charles Forgy invented the OPS (Official Production Systems) family of computer language for production systems, which was based on his Rete Algorithm (Section 2.1.2.3). This was sufficiently efficient to allow scaling up to thousands of rules and was the first language to be successfully used in the R1/XCON expert system which assisted customers in ordering VAX computers by automatically selecting components (McDermott, 1993). However, as shown by Amaral and Ghosh (1994), programming production systems is far from being a solved problem. The search for an effective parallel Production System machine is composed primarily of two schools of thought. The first believes that little improvement can be obtained in the performance of parallel production systems without the introduction of significant changes to the production systems language semantics; while the second maintains that better performance can be achieved through architectural improvements. Neiman (1991) discusses the problems of conventional conflict resolution algorithms for parallel rule-firing production systems and attempts to use insights from blackboard architectures. The author highlights three main control issues, namely dynamic scheduling, heuristic control and algorithmic control. Note that these issues are based on a parallel rule based system using multi-core serial processor(s). Issues regarding the programming of parallel arrays is discussed further in Section 2.3.3. Regardless, note that (as mentioned in Section 2.2.2) Amaral and Ghosh (1994) show that, in fact, there are still bottlenecks in Rete-based production system architectures.

2.3.2 Sub-symbolic implementations

Sub-symbolic methods (Section 2.1.3) have generated significant interest recently. Chellapilla and Fogel (1999) evolved feed-forward ANNs with 2 hidden layers to play checkers in what later became the famous Blondie24 program (Fogel, 2002). The ANNs used for Blondie24, had 40 neurons in the first hidden layer and 10 neurons in the second one. Yang et al. (2000) described the application of a feed-forward ANN with back-propagation to the problem of image recognition to differentiate between crops and weeds. It uses one hidden layer and, in its various experiments, implements up to 300 neurons in the hidden layer. Almeida et al. (2008) use a feed-forward ANN with 6 hidden neurons to generate transition probabilities for a stochastic Cellular Automaton (Section 2.2.1) that models land use dynamics. Wang et al. (2008) use up to 150 nodes in ANNs used to associate 8×7 pixel images. Sudo et al. (2009) use a variant of the bidirectional associative memory using 99 neurons in the competitive layer to associate the 7×7 pixel IBM PC CGA characters dataset (capital letters versus small letters). Tirdad and Sadeghian (2010) use 100 neurons in a Hopfield network to generate a pseudo-random number generator that passes the 15 critical tests identified by the National Institute of Standards and Technology.

These examples help to illustrate that, using standard hardware architectures based on conventional processors, the typical neural network implementation is of the order of a few hundred neurons, typically using non-linear sigmoid activation functions based on the hyperbolic Tan. The reason for this can be understood from the following.

The number of weights in a fully connected neural network may be calculated as shown in Equation 2.7.

$$\text{Total connections} = (N_i + N_o)N_h \tag{2.7}$$

where N_i, N_o and N_h are the number of neurons in the input, output, and hidden layer respectively. Similarly, for a neural network with 2 hidden layers, the number of total connections are calculated by Equation 2.8.

$$\text{Total connections} = (N_i + N_{h2})N_{h1} + N_{h2}N_o \tag{2.8}$$

where N_{h1} and N_{h2} are the number of neurons in the first and the second hidden layer respectively. As may be seen, the number of connections increases dramatically with the number of nodes, with each connection having a weight associated with it. Even disregarding the sigmoid activation functions and using the simplest step function approach, each neuron must perform at least N_j multiplications and $N_j - 1$ additions, where N_j is the number of inputs to the j^{th} neuron. Considering that weights are typically real numbers, this implies a significant number of floating point arithmetic operations per node. Therefore, despite the advances in processor speeds, it is still infeasible to go beyond a few hundred neurons on a serial machine. This is evident from Long and Gupta (2008) who compare results of various implementations of feed-forward ANNs on serial and parallel machines. For the serial implementations, the authors used IBM's P640 RS6000 Server[18]. The largest ANN attempted was composed of $135 - 70 - 48$ neurons in the input, hidden and output layers respectively and had 12,810 connections. This has led to exploratory research in bespoke hardware solutions for ANNs, such as the TOTEM architecture introduced in Section 2.2.3. Note however, that despite being a customised VLSI solution, TOTEM was only able to implement 32 neurons with 256×8 bit weight RAM, which means a neuron can have a maximum of 256 connections. Furthermore, reactive tabu search requires random write access while modifying weights and sequential read access during network evaluation (Anzellotti et al., 1995). TOTEM3 integrated 4 TOTEM chips into one and therefore implemented 128 neurons while the Virtex 600-E FPGA-based SoftTOTEM also implemented only 32 neurons (Table 2.1). Contrasting this with the brain, it is quite evident that unless implementations can pack many more neurons and connections, complex symbolic AI applications will continue to be beyond reach.

[18]`http://en.wikipedia.org/wiki/IBM_System_p`

Long and Gupta (2008) present a software package called SPANN (Scalable Parallel Artificial Neural Network) in an attempt to make it easier to implement large neural networks on massively parallel cellular arrays. The authors note that, for cellular array implementations, the amount of interprocessor communication in a large-scale fully-connected neural network becomes prohibitive and therefore propose some architectural adjustments to the neural paradigm by moving nodes around to minimise communication. 6-layer feed-forward ANNs are implemented, with the largest one having

- 600,000 inputs,
- 25,000 neurons,
- 4000 neurons per hidden layer,
- 2,400,106,384 weights stored in 4-byte floating point numbers.

Using NASA SGI Columbia supercomputer[19] with 500 Intel 1.5 GHz Itanium2 processors (Mavriplis et al., 2005), the implementation used 19 GB RAM and had a correct response rate of 89%. Scaling down the problem resulted in increasing the correct response rate to 100%, thereby indicating that the machine was operating at peak capacity. Furthermore, a single feed-forward operation had a response time of 0.25 seconds, which, depending upon the application, can be said to already be stretching the bounds of real-time processing. Note that this parallel implementation uses supercomputing resources that are too expensive and cumbersome to be widely available. As suggested by Moravec (1998), proliferation, and therefore cost of the hardware used, is crucial to the success of AI.

2.3.3 Parallel Programming

As mentioned previously, current parallel cellular arrays are composed of Von Neumann style processors. Various parallel architectures have been discussed (Section 2.2) along with implications of implementing AI algorithms on such architectures. Even assuming that the clusters can be miniaturised enough to be carried on the backs of robots, or that robots can access these resources in real-time through some form of cloud computing, programming such parallel architectures is not a straight-forward task, and incurs significant effort. This section attempts to highlight some of these issues.

[19]`http://www.nas.nasa.gov/hecc/resources/columbia.html`

The usual motivation for building parallel systems is to maximise processing power and minimise processing time. In the case of architectures for AI, quite often the objective is to achieve real-time processing. To ensure unambiguity, the definition of real-time processing used by Atrubin (1965) is adopted:

> "Let a sequence of inputs be defined as $i_0, i_1, i_2, \cdots$ and a sequence of outputs be defined as $U_0, U_1, U_2, \cdots$ so that U_n is a function of the inputs up to and including i_n. If U_n is computed within a fixed time after the arrival of i_n and this time is independent of n, then the computation is *real time.* Otherwise, if the delay between receiving i_n and the computation of U_n is arbitrarily large for an arbitrary n, then computation is *general.*"

This section presents an overview of the issues in designing and programming parallel arrays that are composed of conventional processors as defined previously.

In theory, parallel systems can provide massive amounts of computation power and speed. In practice however, the efficiency of parallel architecture can be greatly diminished by conflicting factors and the contention for shared resources. Coppin (2004, Chapter 5) discusses the issue of implementing AI search algorithms on coarse-grained parallel machines and identifies the following three main issues:

- *Task distribution*: distributing the tasks amongst available processors.
- *Load balancing*: ensuring all processors are utilized fully.
- *Tree ordering*: the correct order to process search trees.

Hannig et al. (2004) presents a review of the constraints and methodologies of mapping loops to coarse-grained parallel machine architectures, while Teich (2008) identifies some of the major problems in exploiting multi-processor architectures as follows:

- Lack of architecture aware programming support because parallel programming languages do not support resource awareness.
- Self-mapping of parallel programs to processors is difficult; processing is usually assigned statically as self-adaptiveness is difficult to implement.

- Missing compiler and simulation support for adaptive programs and architectures.
- An optimal mapping may be computed at compile time for a single application, however it is difficult to expand on this for run-time because of time-variant resource constraints or data-dependence.
- Central control of massively parallel computers becomes a performance bottle-neck.

Whereas Veen (1986) indicates that a parallel architecture should have the following aims:

Minimize overhead: minimize processing introduced due to parallelism.

Minimize underutilization: minimize processing that is not utilized in useful computation.

Maximize effective utilization: maximize utilization corrected for overhead.

Scalability: adding more components should increase performance.

Linear speed-up: effective utilization should not drop when the machine is extended.

Flynn (1972) introduced the distinction between parallel computers with a single instruction stream and those with multiple instruction streams. The former are called *Single Instruction Multiple Data* (SIMD) machines and are characterized by the synchronisation between processors at the machine language level. All processing elements get the same instruction stream but can choose to execute an instruction or wait for the next one. This methodology is particularly suited for well structured problems with regular patterns of control signals, such as low level signal processing applications where massive numbers of data have to be processed in a uniform way. On the other hand, SIMD requires skillful programming as scheduling and allocation is performed statically at compile time. Examples of SIMD machines are the CM-1, the CM-2, and the TOTEM architecture (Section 2.2).

In contrast to SIMD, *Multiple Instruction Multiple Data* (MIMD) is where processing elements receive different instruction streams and therefore do not need to execute wait or *no-op* cycles. Therefore, MIMD machines are asynchronous at the machine language level: this does not necessarily imply an asynchronous organisation of the hardware architecture, only that no assumptions can be made about the relative timing of independent concurrent computations. Although scheduling is performed at run-time, the programmer must be aware of segmentation issues for the program and the data. However, this increased utilisation comes at a cost since processing elements in the array would seldom be performing independent computation and would therefore need to be synchronised.

Note that, while all these lists have some overlap, there is a large variety of issues which can broadly be classified as:

- How to make these massively parallel arrays feasible for widespread implementations in electronic devices (or electronic agents) which require artificial intelligence.
- How to program such parallel architectures.

In terms of feasibility, it is not only the financial cost involved, but also the amount of physical area that existing parallel architectures take up that is important. Existing supercomputers are composed of thousands of nodes each of which is a dual-core or a quad-core processor, complete with the peripheral devices required to run it (like RAM, hard-drives, motherboards, fans). Therefore, these clusters cost millions of dollars and take up hundreds of cubic meters, making them infeasible for most AI applications.

The problem of programming such architectures has been studied quite extensively, and although there is no consensus as to the biggest problem facing the parallel programming community, synchronisation is widely agreed to be one of the major problems that needs to be resolved (Beckman et al., 2006, 2008, Jones et al., 2003, Petrini et al., 2003, Radojkovic et al., 2008, Tsafrir et al., 2005).

The issue of accessing the hardware through potentially multiple layers of hardware abstraction was mentioned earlier (in Section 2.3), which is the overhead of using a general purpose processor. This overhead is further enhanced in parallel arrays and is known as the operating system (OS) *noise* or *jitter*, defined as activity *per node* that is unrelated to the parallel application. Note that the nodes in the clusters would typically run a complete operating system, like a flavour of Windows, or Linux. Tsafrir et al. (2005) identifies two major reason for OS noise as, daemon processes that wake up infrequently and, interrupts which need to be serviced; these interrupts make up two-thirds of the slow-down. Figure 2.8 illustrates this effect in a multi-processor environment.

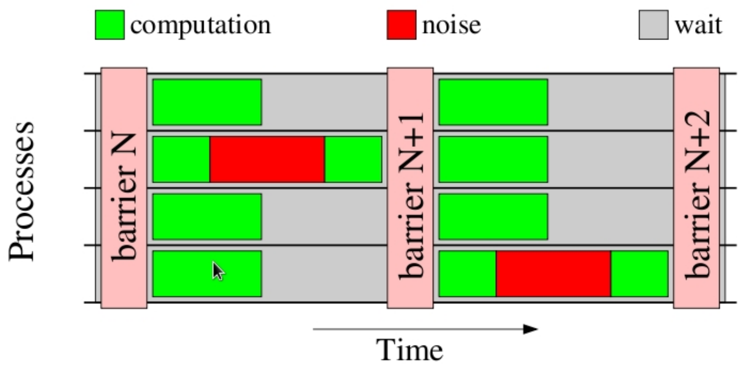


Figure 2.8: Effect of Operating System noise on parallel arrays (Tsafrir et al., 2005). Even when tasks are spawned at the same time (Barrier N), noise in one process can force all other processes to wait (Barrier N+1)

Petrini et al. (2003) observed similar performance degradation while programming the ASCI-Q processor array, composed of 8192 processors, and discovered that utilising 3 out of every 4 cores for their parallel application (rather than all 4) improved performance. This is because the fourth core is left free to deal with the housekeeping tasks thereby increasing the speed with which the entire application executes. As can be seen, this is quite unexpected and counter-intuitive – rather than utilising all available resources for the application, the scheduler must reserve some resources for housekeeping tasks to maximise performance. Radojkovic et al. (2008) corroborated these results and observed an overhead of up to 30% when the operating system timer interrupt service routine used the same hardware context and the same core as the parallel application. If the application used a different hardware context on the same processor core, then the overhead was 10%, while using a completely different core as the parallel application resulted in no overhead.

2.4 Summary

This chapter presented a review of some of the current methods used for artificial intelligence as well as the kinds of hardware architectures used to implement these algorithms. The traditional framework is composed of a serial Von Neumann architecture based on Arithmetic and Logic Units and accessed through (potentially multiple) software layers. This serial nature of machines coupled with the causal nature of human intelligence can be argued to be the reason why AI research has centred around search methods. Thus, for example, Ikeda et al. (2001) is inspired by the description of the human face recognition in

Chellappa et al. (1995) and interprets it to mean that humans brains perform serial search through a database of images, but fails to recognise that this causal nature of the human brain comes at a much higher level of complexity than is within the ability of current AI technology. Serial implementations of AI algorithms with current hardware resources is incapable of achieving the kind of complexity required and there is significant evidence to show that, in fact, a parallel architecture might be necessary to find solutions in real-time. Current parallel arrays are typically just multiple serial Von Neumann machines connected together, which are, once again, accessed via (potentially multiple) software layers. This introduces the (significant) problem of programming and synchronising this parallel array. Although cutting edge parallel machines today are capable of implementing thousands of neurons and millions of connections, the cost of a machine capable of such a task makes it prohibitive. The advantage of using parallel array architectures based on conventional processors is the flexibility it affords, since the general purpose processors are Turing Complete. However, it is by no means definite that a Turing complete general purpose architecture is *required* for performing AI. As indicated by Johnson (2004):

> "[A] complex set of differential equations models the turbulent motion of a seed blowing around in the wind. However a bird moving to catch such a seed doesn't need to solve those equations in order to catch the seed, nor does the seed need to be aware of the equations in order to carry out the movement."

In fact, of late, there has been a push to break the traditional computation paradigm (Stepney et al., 2005) and one way of doing this is to use alternative substrates for performing the computation (Johnson, 2007). Typically radically different substrates like quantum computers or DNA computers are considered as alternative substrates. However, there is no reason silicon based custom electronic hardware architectures should not be considered alternative substrates as long as they are significantly different from the general purpose processor paradigm. It remains to be seen whether such a hardware architecture, based on elements that may or may not be Turing complete, might be more suited for AI applications. Nonetheless, it is worthwhile to explore such alternative architectures in the hopes of finding one that is capable of advancing AI through *the gateway*[20].

[20]A *gateway event* is a change to a system that leads to the possibility of huge increases in kinds and levels of complexity (Stepney et al., 2005)

Chapter 3

Fault Tolerance

Chapter 2 broadly categorised existing techniques for artificial intelligence, with a focus on the implications of using these techniques on existing hardware architectures. The motivation for designing for fault tolerance was identified in Chapter 1, which is as follows. In essence, although a wide range of conventional engineering techniques exist for fault tolerance and error correction, they fail to capture the robustness that can be observed in biological entities (for instance the neuro-plasticity in the human brain) that can recover from extremely severe faults. This chapter reviews some of the techniques used for fault tolerance and discusses their relevance in light of the problem domain, namely, AI applications, and hardware architectures used for AI applications.

This research was partly funded by the SABRE project (introduced in Section 1.2). The motivation for SABRE has been fault tolerance in mission critical environments; i.e. situations where it is imperative that the system continue operation, even in the presence of errors. There have been some attempts in the past at designing hardware architectures for such situations; notably the Embryonics architecture which is reviewed in Section 3.6. Although there are many industrial applications where safety is critical, the focus of this work is more on electronics in hazardous environments. Therefore, this chapter focuses on analysing fault tolerance methods for intelligence in hazardous environments, like the electronics used for space shuttles or for robots in space, underwater or while performing rescue operations after earthquakes.

The main classical fault models used in digital design are:

- stuck-at faults: The output (typically binary) of an element/module is stuck-at a particular value (typically 1 or a 0),

- stuck-open faults: A line (typically binary) is disconnected when it should be connected.
- bridging faults: Two or more (typically binary) signals are connected together, when they should not be.
- delay faults: The timing of a signal does not match the requirements – typically the signal arrives later than expected.

Various techniques exist for handling such fault models which will be discussed briefly in the following sections. From the perspective of hazardous environments however, the most relevant class of faults is Single Event Upsets (SEU). Finn and Lane (1989) define this as a transient state change in a memory cell or a logic circuit caused by a high energy particle; although there is no permanent damage, the total radiation dose can cause a threshold voltage shift and eventual failure. SEUs have not only been observed in space, but also in airborne avionics and even at sea level.

The rest of this chapter is structured as follows. Section 3.1 reviews passive techniques while Section 3.2 presents active techniques and hybrid techniques. Section 3.3 explains data redundancy and Section 3.4 details time redundancy techniques. Section 3.5 talks about the use of timers in fault tolerance, and Section 3.6 is a case study for a recent architecture designed for fault tolerance. Finally, Section 3.7 summarises and analyses the techniques presented in terms of fault tolerance in hazardous environments.

3.1 Passive Hardware Redundancy

One approach to fault tolerance is to mask faults rather than performing any active fault detection and repair. Typically, hardware is triplicated and a voter is used to determine the correct output. This is known as *passive hardware redundancy* or *static hardware redundancy*.

3.1.1 Popular Techniques

Perhaps the most popular method for passive redundancy is *Triple Modular Redundancy* (TMR) where the functional hardware is triplicated and fed to a voter, as illustrated in Figure 3.1.

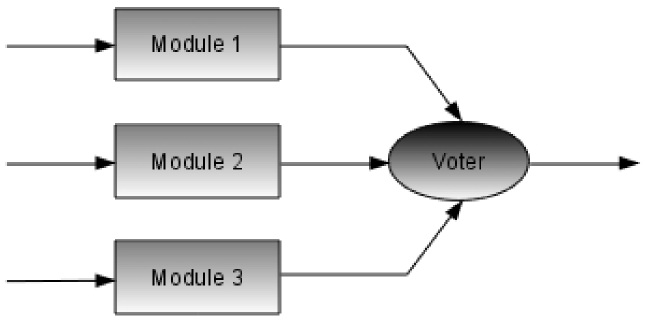


Figure 3.1: Triple Modular Redundancy (TMR)

Iyer and Kalbarczyk (2003) shows that TMR is unsuitable for long missions because, paradoxically, after the first failure, the 2 remaining components *compete* to fail. For higher reliability, TMR can be extended to N-Modular Redundancy.

As may be expected, the voting mechanism is critical to the fault tolerance, and various methods to implement voters have been attempted in the past, including calculating the majority, the average, the mid-value and so on. Figure 3.2 presents a simple 1-bit majority voter for TMR.

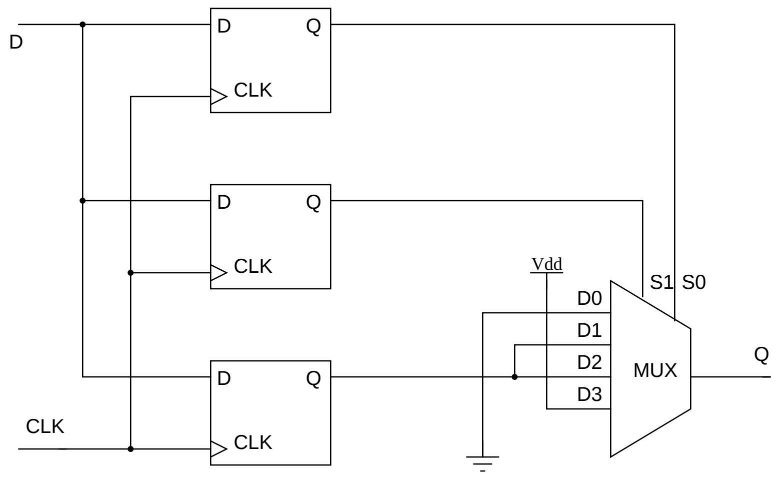


Figure 3.2: Majority voter design

Majority voters are quite common, but have the disadvantage that for many practical applications a majority vote tends to be meaningless. For example sensors sampling external inputs can seldom be manufactured to agree exactly and analogue to digital converters often disagree in the least significant bit. On the other hand, the average voter fails for cases where faulty values can get stuck at extremes, for example in the case of sensors getting stuck at out-of-range (or end-of-range) values. Although the mid-value voter is a good choice, it can be very costly to implement.

Such mechanisms provide a high level of fault-tolerance in the functional portion of the hardware, however, the voter is critical to the correct operation and becomes a single point of failure. One solution is to triplicate the voter, as illustrated by figure 3.3 (Iyer and Kalbarczyk, 2003). Such methods have been proven to be quite successful in hazardous environments, as indicated by the NASA space vehicles which employ 4 computers with 4 copies of a majority voter (Iyer and Kalbarczyk, 2003). Note that passive redundancy is suitable for stuck-at faults and delay faults. However, if there is a stuck-open fault or a bridging fault on the line between the module and the voter, the technique fails. There have been numerous variations on these *voting-schemes* over the years, which provide advantages in their own niche cases.

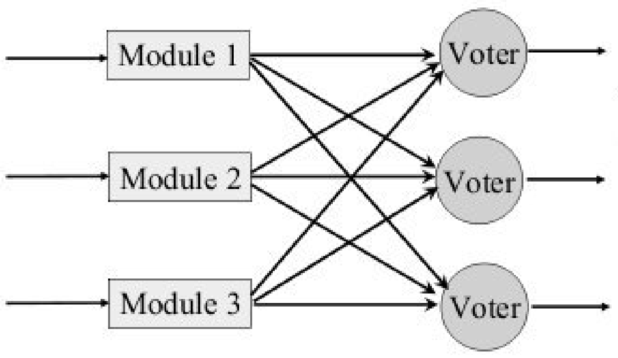


Figure 3.3: Triple Modular Redundancy with triplicate voters (Iyer and Kalbarczyk, 2003)

3.2 Active Hardware Redundancy

In contrast with passive methods, active hardware redundancy includes hardware that actively checks for faults. It is characterized by active fault detection, fault location and fault recovery where possible.

3.2.1 Popular Techniques

The simplest and most popular technique for fault detection is known as *duplicate with compare* (DWC), which simply duplicates the hardware and compares the result to detect if there is a fault in the module. Figure 3.4 illustrates. Note that this is not Double Modular

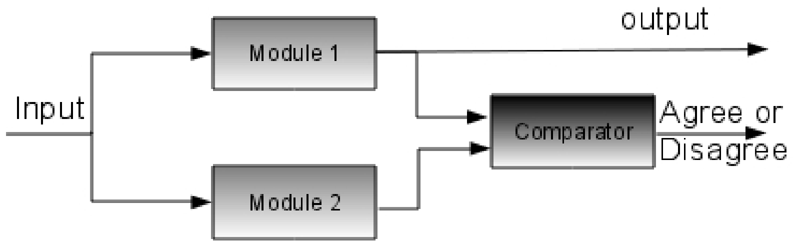


Figure 3.4: Duplicate With Compare (DWC)

Redundancy (as opposed to TMR in Section 3.1). Rather than masking errors, DWC passes errors straight through on the output line and generates another signal to indicate if an error has been detected. Therefore, DWC is simply a fault detection mechanism. Any fault correction or recovery must be performed after the fault has been detected. Note that in terms of applicability to fault models, DWC is similar to TMR discussed in Section 3.1. The method fails if there are stuck-open faults or bridging faults on the line between the modules and the comparators.

One technique for fault recovery relies on the use of spares. *Standby sparing* maintains spare modules on stand-by. When a fault is detected, these modules are brought online. As may be expected, this involves a disruption of service for the time it takes for the standby module to come online. *Hot standby sparing* attempts to minimize this disruption by keeping the spare modules running online, so that only the connections need to be switched when a fault is detected, and the system does not need to wait for the spare to come online. Figure 3.5 illustrates a complete system using such an active redundancy technique (combining DWC with standby sparing). This is popularly known as *Pair and Spare.* Internet domain registrars typically use two domain name servers: the primary being the one actively used, while the secondary one provides a hot spare in case of errors

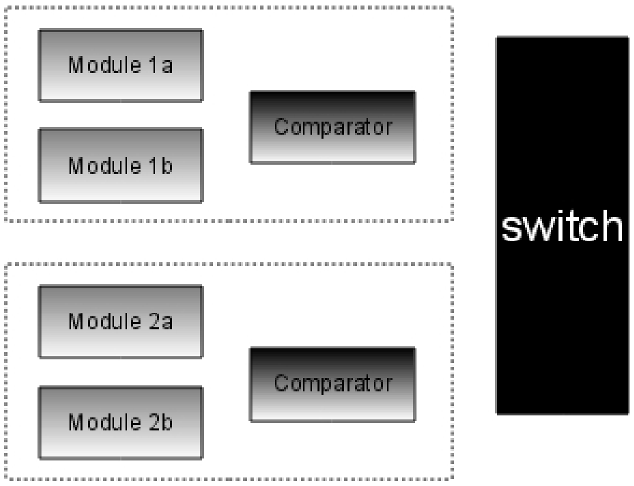


Figure 3.5: Pair and a spare

in the primary. As can be expected, sparing is not typical of space applications where weight and space are at a premium.

Such methods have been used widely and form the basis of much recent research including the Embryonics and POEtic architectures (Section 3.6). In fact, Johnson et al. (2008) show that DWC, if applied to selected areas of an FPGA, is capable of detecting 99.85% of the faults with only an approximate increase of twice the hardware. Additionally, many hybrid redundancy solutions have been attempted over the years which combine passive and active redundancy to maximize the advantages of both (Iyer and Kalbarczyk, 2003), with the Self Test And Repair (STAR) computer by the Jet Propulsion Laboratory (JPL) being a notable example (Rohr, 1995). This is discussed further in Section 3.7.

3.3 Information Redundancy

The passive and active techniques described in Sections 3.1 and 3.2 were based on adding redundant hardware, which can be seen as the overhead for improving fault-tolerance. Information redundancy (or data redundancy) operates on the premise of adding redundant information to provide fault tolerance. Traditionally, the techniques were used to provide robustness during communication, however they have significant scope for application in hazardous environments, which is discussed further in Section 3.7. The simplest method is to transmit multiple copies of the data; however, this incurs a large overhead. Therefore, in order to balance overhead with the fault-tolerance requirements, the information to be transmitted is compressed into a representative *word*, which can be used to detect (and possibly recover from) the required level of faults. Such methods are also called Error Correcting Codes (ECC). Typically, this (compressed, representative) word is added at the end of the data which is being protected from faults. Although the method is suitable for either hardware or software implementation, in both cases the data is typically sourced from memory (RAM or flip-flops), as error checking is not done on single-bit binary values, but rather (potentially multiple) long words. Therefore, in contrast to active and passive hardware redundancy techniques, information redundancy is applicable to all 4 fault models, as long as the fault is *before* the memory source of the data. In terms of hardware implementations, the error checking (and correcting) module is the weak link and any errors in this module remain undetected. Karri and Nicolaidis (1998) discuss information redundancy from the perspective of VLSI implementations.

3.3.1 Popular Techniques

Perhaps the best known data redundancy technique is the parity check, where a single bit is added to the packet to make the sum of all the bits even (even parity) or odd (odd parity). It is commonly used in memories and over short, reliable channels, since it is an effective error detection mechanism with low cost for the encoding/decoding scheme. However, it suffers from three drawbacks. Firstly, it does not provide enough information to correct the error; secondly, it is unable to detect many multi-bit errors; and thirdly, parity is not preserved across data transformations so that, for example, it will have to be recalculated after an addition operation.

3.3.1.1 Cyclic Redundancy Checks

There are numerous methods available that address some or all of these issues. In all cases, more information (than the single parity bit) needs to be added to the data. The most popular from amongst these methods is the *Cyclic Redundancy Check* (CRC), invented by Peterson and Brown (1961). A major reason of this popularity is the ease of implementation in hardware. In brief, CRCs are based on a polynomial that can loosely be said to be the divisor in polynomial division with the remainder of the division (called the checksum) being the information used to detect errors in the data. To check for errors, the division is repeated and if the remainder matches the checksum, the data is assumed to be free of errors. Some of the more popular applications of the CRC is TCP/IP communication over the internet and software drivers for hard-disks. As must be evident, the choice of the polynomial is critical to the level of coverage that is provided by the checksum. Various versions of the CRC are in existence, such as the CRC-12, the CRC-16 and the CRC-32. The polynomials used in these various versions are even more varied such that, depending on the application and the specification being followed, a single CRC version can operate on different polynomials.

3.3.1.2 Hamming Codes

The Hamming code is another popular method, since it provides fault location as well as fault correction abilities. It was developed by Richard Hamming in the 1940s and was first mentioned by Shannon (1948) in his ground breaking paper which, incidentally, is also the first publication to use the term *bit* for binary digit. The Hamming code can detect all errors patterns with a Hamming distance of $d-1$ and correct errors patterns with a Hamming distance of $(d-1)/2$; where d is minimum Hamming distance between any two pairs of code words in the set that is being covered.

3.3.1.3 Arithmetic Codes

One method for testing the operation of arithmetic units, like adders or multipliers, is to use arithmetic codes. Some of the common types of arithmetic codes are AN codes, AN+B codes, residue codes, and bi-residue codes. To illustrate, from the basic properties of equalities in mathematics we know Equations 3.1 and 3.2 to be true.

$$\frac{AN + AM}{A} = N + M \tag{3.1}$$

$$\frac{AN \times AM}{A^2} = N \times M \tag{3.2}$$

In order to test an adder, the two inputs of an adder are multiplied by a constant A, thus, effectively *encoding* the inputs. Errors are detected by dividing the output by the same constant A and looking at the remainder. If the remainder is non-zero, a fault has been detected. Similarly, using Equation 3.2 the test for a multiplier can be developed (Patel and Fung, 1982).

Although arithmetic codes may be implemented using additional hardware, in order to minimise overhead, it is also possible to reuse arithmetic units already present by inserting self-test cycles. If separate hardware is used, the fact that the encoding and decoding is a multiplication and division with a constant, allows minimising the overhead hardware by optimising for compactness and/or speed. The choice of A is important for two reasons. Firstly, it governs the width of the two operands that are fed to the unit being tested (for example the adder). Secondly, since it is theoretically possible for aliasing[1] to result in an undetected error, A must be chosen to minimise this effect. Piuri et al. (1992) indicate that $A = 3$ is an optimal choice in that it produces a small overhead in the arithmetic unit while minimizing the aliasing effect.

Although arithmetic codes are an elegant solution, they do have blind spots. Furthermore, arithmetic codes are only applicable on arithmetic units and cannot be used to detect faults in logical operations. Patel and Fung (1982) indicates that in the worst case, arithmetic codes can fail in detecting single bit errors in group carry look-ahead structures and present a system called REcomputing with Shifted Operands (RESO) which addresses both these weaknesses. Instead of encoding the inputs of the adder by multiply-by-A, the authors use a shift operation so that encoding is performed by shifting left and decoding is performed by shifting right. Not only does this work for logical operations but also for arithmetic operations, as left shifting by 1-bit is equivalent to multiplication by 2 and right shifting by 1-bit is equivalent to dividing by 2.

3.4 Time Redundancy

Time redundancy is a technique used primarily for testing transient faults. It does this by providing the same stimulus multiple times and comparing the result. Therefore, it typically has a low spatial overhead as it uses the same hardware to perform an operation multiple (N) times. Alternately, it has a high time overhead because it requires a minimum

[1]Two separate test vectors to result in the same output, thereby making a distinction impossible

of N times the amount of time to perform the same operation. Figure 3.6 illustrates this approach.

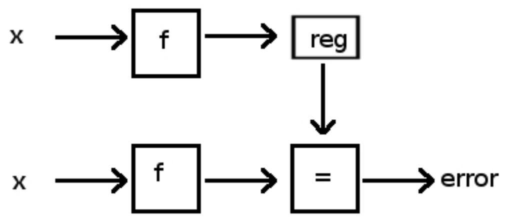


Figure 3.6: Time redundancy

3.4.1 Popular Techniques

Standard time redundancy techniques are limited to detecting transient faults because if a system has a permanent fault, running an operation N times will generate the same result each time. However, as shown by Iyer and Kalbarczyk (2003), merging RESO (Section 3.3) with time redundancy, it is possible to detect permanent faults as well. In the first iteration, a function f (where f is an arithmetic operation like add or multiply) is applied to the inputs x and y generating $f(x, y)$. In the second iteration, the inputs are shifted left by j bits and the output is shifted right by j bits. Note the similarity with degeneracy defined in Section 1.2.

Reynolds and Metze (1978) present another technique for fault detection at the scale of integrated circuits, which depends on *binary alternating variables* implemented using an *alternating circuit*. A binary variable $(x, \bar{x})$ is alternating if its true value in one time interval is followed by its complemented value in the next time interval. An alternating circuit is one that generates a binary alternating value when its inputs are synchronised binary alternating variables. Reynolds and Metze (1978) further shows that although an arbitrary circuit of n variables may not be an alternating circuit, it can always be implemented using $n+1$ variables as an alternating circuit. If the output of an alternating circuit is $(\zeta; \bar{\zeta})$, then, in the presence of an error, the output will become one of:

1. $(\zeta; \zeta)$
2. $(\bar{\zeta}; \bar{\zeta})$

3. $(\bar{\zeta}; \zeta)$

Faults 1 and 2 may be detected, while 3 cannot be detected.

Note that, in its basic form, time redundancy is only applicable to transient errors. Although, with the extensions proposed by Reynolds and Metze (1978), it is capable of detecting long-term errors, the solution is highly intertwined with the architecture of the hardware. Therefore, even though it is almost certain to detect stuck-at faults and delay faults, its performance in cases of stuck-open faults and bridging faults is less clear, and highly dependent on the architecture itself.

3.5 Timers

Timers are often used in digital design to perform fail-safe error recovery, and their operation can be summarised as follows. Two processes run in parallel; the first performing the required function and the second timing the first. If the process being timed is not completed within a predefined interval, the timing process assumes an error has occurred, and triggers a recovery mechanism. Typically however, the recovery mechanism has no information about why the process stalled and so there is no guarantee that if the process is repeated, it will not stall again. More sophisticated techniques attempt to query the process to discover the reason for the fault, but more often than not, human intervention is required.

3.5.1 Popular Techniques

Watchdog timers (Bheevgade and Patrikar, 2008) are an integral part of most microprocessors today. They are designed to run continuously while processors continue executing instructions whether in user space or in administrator space. The operating system or scheduler periodically gains access to the processor and executes an instruction to reset the watchdog timer ensuring that the timer does not expire. The assumption is that if the watchdog timer does not get reset, the scheduler did not get access and so the processor must be stuck somewhere. Therefore, the watchdog timer interrupt is usually an *unmaskable* interrupt of the highest priority, with the possible exception of the reset interrupt (in the cases where reset is handled as an interrupt). It is usually considered good engineering practice to reset the entire system on the expiration of the watchdog timer, regardless of the severity of data loss.

Heartbeat timers (Hong, 2007, Iyer and Kalbarczyk, 2003, Stewart et al., 2000) are similar in operation but are typically run in user-space (most often to time communication protocols) rather than in administrator space. A monitor entity runs a timer while another entity performs an operation. At the expiration of the timer, the monitoring entity transmits a message to the second entity performing the operation, which must respond with an acknowledgment, otherwise an error is assumed. Variants like *adaptive heartbeat* timers periodically negotiate the timeout period while waiting for a response. This is particularly useful in communication protocols where network congestion and other external factors may generate false timeout symptoms (Figure 3.7). *Smart heartbeat* timers ensure that the process being monitored performs a self-check as well before responding with an acknowledgement. This ensures that the acknowledgement not only indicates that monitored process is running, but that it is running correctly.

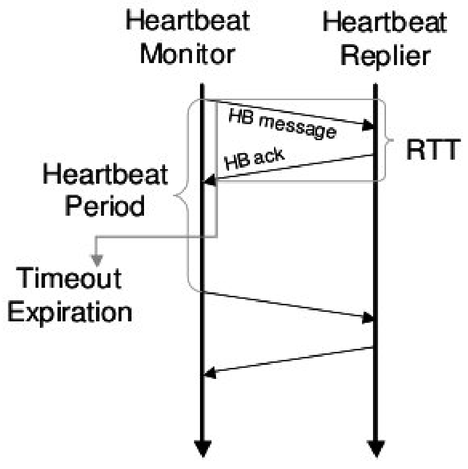


Figure 3.7: Adaptive heartbeat timer (Iyer and Kalbarczyk, 2003)

Note that timers are suitable for fault tolerance at a process level, rather than at the low level of hardware architectures. Whereas they are certainly suitable for delay faults, depending on the location of the fault, the other 3 types of faults could also result in a process getting stalled, thereby allowing the timer to be applicable. However, this is highly dependant upon the location of the fault and is not guaranteed.

3.6 Embryonics: A Case Study

The previous sections have introduced some of the techniques used to achieve fault tolerance in the state of the art today. These are many and varied and as such, are chosen to fit the kind of application that they are being applied to. In recent years, there have been some attempts at adapting these techniques for fault tolerance in mission critical environments. Embryonics is one such architecture that was designed to be general purpose[2] and reconfigurable while also displaying fault tolerance.

The *embryonics* architecture was developed in the 1990s by Mange et al. (2000), Tempesti (1998) and aimed to generate VLSI circuits capable of self-repair and self-replication. It was a novel reconfigurable architecture whose basic elements were "molecules" connected together using a programmable connection network. The molecules were defined such that they were functionally universal and could be combined together to form cells, which could act as simple processors or binary decision machines (Mange et al., 2000, Prodan et al., 2003). Figure 3.8 shows the conceptual hierarchy for the Embryonics framework.

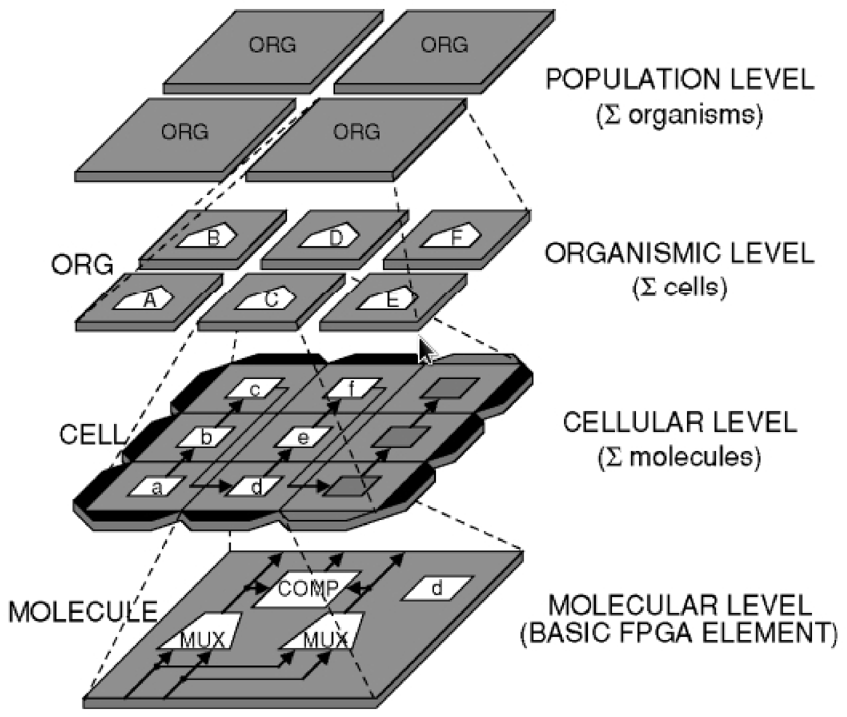


Figure 3.8: Embryonics levels (Tempesti et al., 2007)

[2]Able to implement any digital logic

Mange et al. (2000) presents two different types of building blocks called the *mictree* (at the cellular level) and *muxtree* (at the molecular level). Jackson and Tyrrell (2001) discuss asynchronous design techniques for embryonic architectures, including dual-rail 2-bit bus and acknowledgement techniques and Muller-C gate approaches.

Fault tolerance operates at two levels, in both cases, exploiting spare elements to perform recovery. Errors are detected at the molecular level using duplicate-with-compare (DWC) logic. The desired overall function is typically implemented using a subset of the total number of elements available, thus allowing for spares in the device. Faulty molecules are replaced by one from a set of spare molecules (Figure 3.9) and faulty cells are replaced by one from a set of spare cells. In the case of a faulty cell however, the entire column is replaced with a spare column. An adapted version of time redundancy is used to ensure that cells also contain data from neighbouring cells which can be used during the recovery process to ensure that spare cells receive a "clean-copy"[3] of the data. Triple Modular Redundancy (TMR) using majority voters is used to conduct online self tests (Prodan et al., 2003).

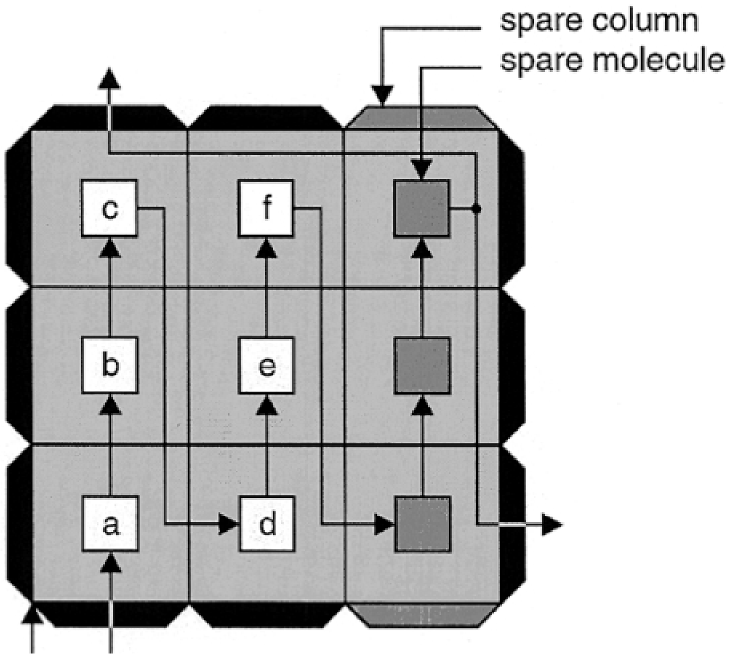


Figure 3.9: Molecular configuration that forms a cell (Mange et al., 2000). The spare column contains molecules that can be used to replace any that are detected as faulty.

In a project that builds on Embryonics, Mange et al. (2001) presents the *POEtic* architecture, which has a more abstract substrate such that cells may be any processing element

[3]not from the faulty cell

including micro-processors, neurons, dedicated circuits and so on, and the POEtic tissue would be an array of such a circuit. The configuration bit-stream for such an array is termed the *genome* and is copied into each cell. Cells decipher their function by using information about their position in the array and extracting the relevant function from the configuration bit-stream. Position information can be fixed or, alternatively, based on a gradient signal. More details about the POEtic architectures can be found in Tempesti (2003), Tempesti et al. (2007) and Barker et al. (2007).

Although Embryonics was a good initial step, it assumes faultless interconnections; an assumption that cannot be justified on reconfigurable hardware where a significant part of the hardware is interconnects. In addition, killing an entire column of cells when a single cell is faulty, is wasteful. Moreover, Embryonics uses DWC at the molecular level to detect faults and in terms of the entire hardware architecture, there is very little overhead (because most of the hardware is connections). However, in terms of functional hardware, this means that for each functional block, there is a duplicate and some extra compare logic making the overhead in terms of functional hardware greater than twice the actual size. On the other hand, Johnson et al. (2008) have shown that using selective DWC, only an approximate overhead of twice the hardware is sufficient to detect 99.85% of the faults. This is further discussed in the following section.

3.7 Analysis and Summary

Fault tolerance has always been an area of interest. As far back as 1949, the Electronic Discrete Variable Automatic Computer (EDVAC) included a duplicate ALU in its design and performed a comparison of the results to detect faults. Similarly, the Universal Automatic Computer (UNIVAC) was built in 1955 (Johnson, 2006) and incorporated a parity checker to detect errors in transmission. Hillis (1986) presents the Connection Machine (introduced in Section 2.2.1.2) which included parity checkers, memory error correction mechanisms, and also hardware fault detection mechanisms. In 1971, the Jet Propulsion Lab (JPL) published work on the Self Test And Repair (STAR) computer (Rohr, 1995), which implemented a number of techniques for fault-tolerance, including passive (Section 3.1), active and hybrid (Section 3.2) hardware redundancy techniques. The JPL-STAR included standby redundancy for functional units and hybrid redundancy for monitoring units. Although it displayed impressive fault-tolerance capabilities for its time, these capabilities were achieved with the help of a customised operating system (STAREX), which provided support for complex interactions between the hardware and software so that the programs were able to resume execution after a repair operation. Note that (in light of the discussion in Chapter 2), such a solution further complicates any real-time applications implemented on such systems as there is yet another layer of abstraction.

This chapter presented a brief review of the state of the art for fault tolerance. An overview of the broad categories was presented as well as details of some of the more popular techniques. Advantages and disadvantages were analysed and their applicability to fault models discussed. This is summarised in Table 3.1. Section 3.6 presented a case study of Embryonics, which was an example of a reconfigurable hardware architecture which combined various techniques for fault tolerance discussed in this chapter. Although it generated significant interest over the years, note that Embryonics implemented hardware redundancy and not degeneracy, in that the spare molecules (and cells) were completely idle and had no functional use until they were used to replace a faulty molecule (or cell). Furthermore, the error checking logic (based on DWC) in each molecule is a critical area that must be assumed to be fault-free.

Technique	**Description**	**Fault-Models**
Passive redundancy	Faults masked not detected	Stuck-at; delay
Active redundancy	Actively test for faults and recover from errors	Stuck-at; delay
Information redundancy	Information or data is duplicated to test for and correct faults	Stuck-at; stuck-open; bridging; delay
Time redundancy	Test transient faults by applying the same stimulus twice	Stuck-at; delay
Timeout	Use parallel processes to test if operations end within a time-frame	delay

Table 3.1: Techniques for fault-tolerance

Most early techniques were based on the stuck-at fault model, which is the reason for these being the most covered in terms of fault tolerance methods. However, stuck-at fault models have been shown to be insufficient (Patel and Fung, 1982), and therefore recent years have seen an increase in popularity for Error Control Codes (ECC), which can cover all fault models. Furthermore, as late as the 1990s, researchers have been assuming interconnects to be fault-free (Piuri et al., 1992), which ignores 2 of the 4 fault models listed above. Note that all the techniques listed above, typically rely on a *critical* area to be fault-free, which performs the error detection and possibly error recovery. Some techniques claim to be able to mitigate this, such as the TMR implementing 3 voters. However, this only defers the decision to some other section that needs to integrate the outputs of these 3 voters and therefore, this section becomes critical. A truly self-sufficient technique should be able to recover from all the different types of faults (or gracefully degrade in performance) without any area requiring to be critical.

Single Event Upsets (SEU) were introduced in the beginning of this chapter as errors that are typical of hazardous environments. Blake and Mandel (1986) record 72 SEUs in a 48-KB satellite memory over a two year period. An important point to note is that multiple SEUs were recorded from single particles such that there were 9 double-faults, 1 triple fault and 1 quadruple fault. They are classified as transient faults since they do not result in physical damage (except over a long period of time) and therefore their effect can be reversed.

SEUs have generated significant interest in recent years, particularly with the advent of space exploration. She and Samudrala (2009) determine the probability distribution for the sensitivity of FPGA resources to SEUs, and use TMR on selective areas to improve fault tolerance, while Johnson et al. (2008) evaluate the use of selectively using DWC in critical areas to improve the fault tolerance against SEUs in SRAM-based FPGAs. Legat et al. (2011) employ the partial reconfiguration ability of the Xilinx Virtex 4 and 5 FPGAs to implement custom logic that checks for errors in the configuration memory of the FPGA, and can reprogram it (assuming that the critical *golden* copy is safe). In fact the topic has generated enough interest to warrant official publications on the subject by FPGA manufacturers[4] (Bridgford et al., 2007). Note however, that the use of these techniques does not free the design from the issue of the critical section that performs fault tolerance.

One method that has commonly been used to address SEUs is radiation hardening by making changes to the process technology at the integrated circuit level. However such methods inevitably increase the area of the circuit and decrease the speed of operation (Finn and Lane, 1989). Another method that is commonly used is *shielding*, however, as

[4]`http://www.altera.com/support/devices/reliability/seu/seu-index.html`

shown by Kerns et al. (1988), this might be counter-productive (particularly in space applications) as high energy particles may deposit more energy rather than less due to nonlinear energy deposition rates. Typically, SEUs in processors are controlled by replication and voting, and watchdog processors; while SEUs in memory are controlled by periodically reading data, using information redundancy techniques (ECC) to correct any errors and writing the data back (Li et al., 1984, Zoutendyk et al., 1987, cited in Finn and Lane (1989)). This raises an important issue, since fault tolerance can be thought of as existing at various levels. Techniques at the transistor level typically use physical properties of the transistors to make them more robust, while at the gate and component level, techniques like redundancy are most often used. At higher levels matters become more complicated since keeping idle, spare processors in a processor array is too expensive. However, cellular computing arrays (or cellular automata) do claim inherent fault tolerant properties. As explained in Section 2.2.1, cellular automata (CA) principles can be extended to processor arrays which can be considered as special cases of cellular automata from the perspective of fault tolerance (Ortega and Tyrrell, 1999). CA theory predicts robustness due to the large number of cells and the similarity in behaviour. Therefore, no individual cell (or processor) is expected be critical to the correct behavior of the entire collection. In practice however, this is not always the case as neighbourhood size plays an important part in determining the influence of a faulty cell. In fact, Bünzli and Capcarrere (2001) shows CA to be fragile and brittle in the presence of faults. Furthermore, as shown by Ortega and Tyrrell (1999), interconnect limitations influence fault-tolerance capabilities for hardware implementations of CA. Ganguly et al. (2003) presents a review of work where the fault tolerant features of CA have been explored.

In conclusion, a hardware architecture that targets artificial intelligence applications for hazardous environments, must be resistant to faults or, in the worst case, be able to degrade gracefully in the presence of faults. From Chapter 2 we note that the architecture needs to be based on nodes operating in parallel. From the analysis of fault tolerant techniques in this chapter, we note that fault tolerance within these nodes should not be implemented in one (or multiple) critical area such that no portion of the hardware should be critical to the whole. Although, ideally, all fault models should be covered, at a minimum the architecture must be able to cater for Single Event Upsets, which can be characterised using stuck-at faults. Chapter 4 presents an architecture which is one attempt at such a solution that displays degeneracy.

Chapter 4

Protein Processor Associative Memory

Chapters 2 and 3 describe the state of the art of fault-tolerant hardware that is used for AI applications. Contrary to the initial excitement and consequent predictions of robots roaming the earth in a few decades, AI has so far failed to pass the Turing test for intelligence, something Turing (1950) famously predicted would occur in the year 2000. As shown in chapter 2 (Section 2.3), current techniques in AI rely (directly or indirectly) on arithmetic operators within the limitations of the classical computation paradigm. With this in mind, this chapter explores the possibility of a novel hardware architecture that does not use arithmetic operators to perform AI related tasks. As explained in Section 4.1, this challenges the Turing paradigm.

In terms of fault tolerance, although current methods are sufficient in many respects, they are designed to cater for well-defined fault-models. On the other hand, fault tolerance in biological systems seems to be more dynamic in terms of being more capable of responding to unexpected faults (for example with methods like neural plasticity which re-routes neural connections). It seems worthwhile, therefore, to test whether this dynamic fault-tolerance is linked to (or is at least more attainable) with a different hardware architecture that does not depend on the uniqueness of hardware modules (chapter 3).

The problem of defining intelligence and artificial intelligence is touched upon in chapter 2, however, a common feature underlying all definitions of AI is the ability to *learn*. Learning and memory are very closely related, and depending on the school of thought that one subscribes to, there can be some overlap. According to Kandel et al. (2000), "Learning is the process by which we acquire knowledge about the world . . . memory is the process by which that knowledge of the world is encoded, stored, and later retrieved". However, Sargent and Stafford (1965), defines it as follows.

> "Memory is a phase of learning . . . learning has three stages:
>
> 1. *Acquiring*; wherein one masters a new activity . . . or memorizes verbal material,
> 2. *Retaining* the new acquisition for a period of time and
> 3. *Remembering*; which enables one to reproduce the learned act or memorized material.
>
> In a narrower sense learning merely means acquiring skill."

Irrespective of how one separates learning and memory, it is agreed that memory is crucial for learning, otherwise the learnt behaviour will be forgotten. Having a memory architecture that stores and also recalls data accurately, robustly and efficiently is therefore crucial for AI. Furthermore, as shown by Coppin (2004, Chapter 11), the human brain performs bidirectional, hetero-associative recall.

4.1 Revisiting the Initial Hypothesis

The initial hypothesis was presented in Section 1.3, and is repeated here.

> " *A novel, non-standard computation method for a bidirectional, hetero-associative memory, implemented using a non-standard hardware architecture, composed of multiple (of the order of hundreds) parallel processing elements, can perform meaningful computation in the context of AI applications, and can perform better and in a more robust way than other traditional techniques based on traditional hardware.* "

Section 1.3 briefly explained some of the terms used in the hypothesis to be able to measure success or failure. These are further specified as part of the following implied objectives:

1. In order to be non-standard, the novel computation method must differ in at least one aspect from the definition of classical computation defined by Johnson (2007), Stepney et al. (2005) (as described in Section 1.1). The following identifies the objectives for the proposed solution to be non-standard.

    - The Turing paradigm is challenged through the use of an alternate substrate for computation. The nodes use memory operations to perform computation, rather than arithmetic operations, thus there is the possibility that such nodes might end up as not being Turing complete. Note however, that the proposed solution is envisioned as a special-purpose companion controller[1]. As indicated by Johnson (2007) this is the method used by nature, and should be considered as an advantage rather than a disadvantage.
    - The Von Neumann paradigm is challenged by using a parallel array of nodes each of which contains data, rather than having information brought to the node.
    - The output paradigm is challenged as the output of the associative memory is the state of all the nodes in the array, rather than a value from one specific output channel.
    - The algorithmic paradigm is challenged through the use of a continuous adaptive process for learning the dataset. Instead of using a configuration bit-stream (as in reconfigurable hardware) adaptation to new data must be automatic and self-regulated.
    - The refinement paradigm is challenged in a number of ways. Firstly, the output of the associative memory (value recalled) is not binary, but a pattern that may also indicate partial association. Secondly, the memory does not depend upon a specification for the relationship between inputs and outputs, but rather extracts the relationship information through observation of inputs over time. Therefore, as inputs change, the specification adapts with it. Thirdly, the associative memory displays emergence as the output of the whole is not the sum (or difference) of the outputs of the individual nodes.
    - Although strictly speaking the computer-as-an-artefact paradigm is not broken as the hardware in the associative memory does not change over time, note

[1]as opposed to a general-purpose processor

that the scalability and fault-tolerance of the architecture means that nodes may be added and removed at runtime. From this perspective, the paradigm can be thought of as having been challenged as hardware can grow (or reduce) over time. Furthermore, neither the nodes, nor the whole network of nodes is a general purpose processor, which also challenges the computer-as-an-artefact paradigm.

2. Since the architecture is non-standard, before testing for *meaningful* computation, the architecture must be tested for *any* computation. This can be tested by measuring the accuracy of memory recall of the training data itself after training.

3. Meaningful computation is defined as the ability to generalise about the underlying dataset from the small sample presented in the training dataset. This must be tested by measuring the accuracy of the recalled values when the data presented for recall is not part of the training set.

4. To determine an increase in performance a comparison must be made with existing techniques implemented on existing architectures of comparable silicon real-estate and at least one of the following must be true:

   - The speed of association of data elements presented must be faster.
   - The speed of recall of data elements must be faster.
   - The capacity in terms of the total number of elements that can be stored and accurately recalled, must be larger.
   - The architecture must be more scalable – in the case of both scaling-up and scaling-down. This also implies that overheads must be lower.

   Details about ways to measure this are presented in Chapter 5 where the experiments are described.

5. Improved robustness can tested in a number of ways, at least one of which must be true:

   - Correct operation or execution must not hinge on any single element so that faults result in graceful degradation of performance. Therefore, nodes must be homogeneous in architecture, but capable of being different from other nodes at runtime.

- Fault tolerance must be achieved through degeneracy rather than redundancy. Therefore, any nodes that duplicate information (or hardware), must also have some way in which they are different from the node whose information (or hardware) is duplicated.

The Protein Processor Associative Memory (PPAM), as proposed in this chapter, is one attempt at such an architecture. It is a bidirectional, hetero-associative memory that combines all three stages of learning listed by Sargent and Stafford (1965) above. As mentioned in Section 2.1.3.4, hetero-associative memories can recall an associated piece of data from *one* category of input upon presentation of data from *another* category of input.

The PPAM is a bio-inspired architecture, where the acquisition and remembering stages take inspiration from the nervous system, while the retaining stage is inspired by the process of protein synthesis and the Genetic Regulatory Networks (GRN). In recent years, inspiration from the biological world is becoming more and more prominent in the world of electronics and computer science. However researchers must be careful when being inspired from biology because, not only is biology based on a different fabric but the processes are extremely complex. It is quite easy to either take too much or take too little from biology (Bolker, 2003, Stewart, 2003). On the other hand, oversimplifying must also be avoided. The test is not how well the model mimics reality, but how well the predictions fit appropriate aspects of reality given what the model leaves in. The following sections describe the biological background that has been the source of inspiration for many of the ideas and how it feeds into a preliminary abstract view of the architecture.

4.2 Cell as a Protein Processor

4.2.1 Biological Inspiration: Genes, Proteins and Transcription

Biological inspiration for the design of computing machines is primarily taken from three models (Tempesti, 2003).

- *Ontogenesis* is the development of individuals from their genetic code.

- *Phylogenesis* is the history of evolution of the entire species.
- *Epigenesis* is the development of individuals using learning processes.

For the purpose of this work, Ontogenesis and Epigenesis are the relevant processes.

4.2.1.1 Proteins

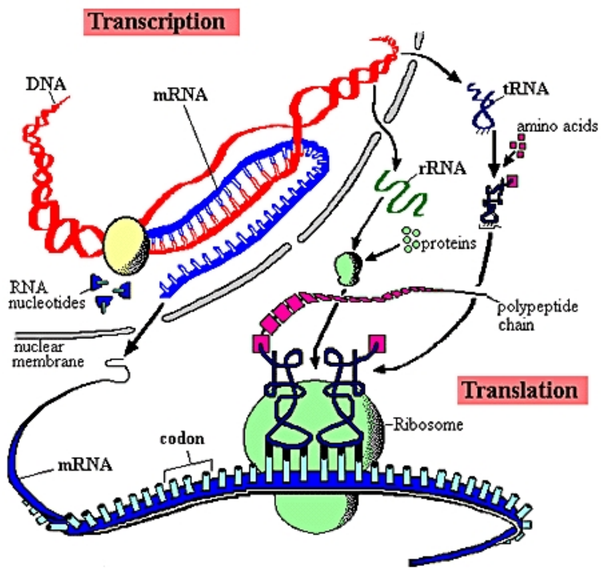
Proteins form a critical part of all communication and processing at the cellular level. Cells receive signals in terms of proteins and after processing the information, respond in terms of other proteins. Proteins may cause cells to change their genetic expression or generate other proteins or both. This forms a complex network of signals central to the Genetic Regulatory Network (GRN) (Jong et al., 2003, Jönsson et al., 2003, Kumar and Bentley, 2003a). At the heart of this GRN is DNA and the process of *transcription*, during which the information in the DNA is translated into proteins by living cells. This is depicted in figure 4.1.

Details of the processes at the membrane of a cell and within it are extremely complex and many are dependent upon the physical shape and structure of proteins, as the biological processes are effected by the laws of physics (Hogeweg, 2003, Zhan et al., 2008). Furthermore, the world at the cellular level is very different from the world at the organism level, so that the effect of forces like gravity is negligible, while forces like surface tension, pressure and viscosity are quite significant (Miodownik, 2003, Stewart, 2003). Since it is not the objective of this work to model the behaviour of a cell, these processes are outside the scope of this document and only a simplified version that is the source of the inspiration will be presented here. The interested reader is referred to Hancock (2003), Hogeweg (2003) for a more detailed discussion of conformational changes inside cells and the processes at the cell receptors.

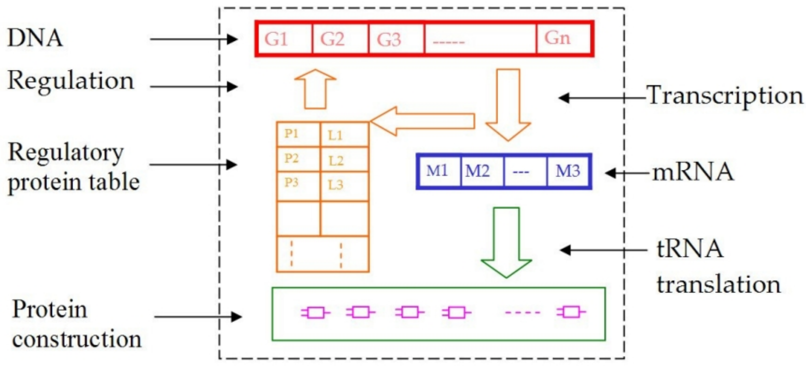
4.2.1.2 Transcription

Proteins enter a cell through various channels or receptors and cause (conformational) changes within the cell. If the protein excites a *cis-regulatory* region[2], it initiates the process of transcription where the DNA is read and a protein is encoded. Genes code for

[2]This can be seen as a precondition or a selection process.



(a) Transcription in a biological cell[*]



(b) Block diagram of the transcription process

Figure 4.1: GRN and transcription (from Zhan et al., 2008)

[*]This diagram is modified from: `http://www.accessexcellence.org/RC/VL/GG/protein_synthesis.html`

a protein or RNA and are composed of *codons* each of which codes for a specific amino acid. Proteins are composed of a variable number of amino acids, which can be as small as 466 (in yeast) or as large as 34,350 (in Titin – the giant protein) (Labeit and Kolmerer, 1995). During transcription, the DNA is read serially in a *reading frame* beginning from a start marker called a *start codon* and ending in a stop marker called a *stop codon*. The sequence of information between the start and stop codons is a description of the protein that is to be produced as a result of receiving the original protein at the cell's input. The resultant output protein may be output from the cell or it may become a further input to the cell itself resulting in another iteration of the transcription process. Proteins may also cause a change in genetic expression, which means that the cis-regulatory regions presented for excitation to incoming proteins can also be altered as a result of processing proteins. Codons are composed of sub-units called *nucleotides* which are labeled U, C, A and G for their chemical names (Turanov et al., 2009). Codons are tri-nucleotide substances so that a combination of 3 nucleotides makes a codon. This means there are a total of 64 (4^3) codon combinations possible. However, there are only 20 valid amino acids and one start and stop codon. Despite this, all 64 combinations are assigned to either amino acids or start/stop markers. The advantage of this becomes obvious when considering the effects of *point mutations* which alter a single codon. Together with natural selection, which controls the effect of *frame-shift mutations*, this results in a highly robust transcription process. For more details of the transcription process, the reader is referred to Nakamoto (2009), Reil (2003). Kumar and Bentley (2003a) and Reil (2003) present the same information from the perspective of computer science while Fleischer (2003) presents a development simulator which can greatly assist in understanding the transcription process through simulation. For a detailed description of genes, chromosomes, DNA and the development process, refer to Wolpert (1991).

4.2.1.3 Compression and Cis-Regulatory Regions

As might be expected, with such a transcription process, the resulting DNA inside the cell is extremely long. Therefore, it needs to be compressed so that instead of being more than a meter in length, it can fit inside a few hundred micrometers. This is achieved by winding the DNA strands around a *histone*, resulting in a selected number of *cis-regulatory interfaces* that can be excited, thereby allowing a pattern-matching process for incoming proteins. Figure 4.2 illustrates. In cell C_1, a protein P_1 may successfully excite a cis-regulatory region and cause transcription of a protein P_2; however, it may not have the same effect in another cell C_2 which has the same DNA, but different cis-regulatory regions exposed. In cell C_2 it may cause the transcription of protein P_3 or it may even result in no action whatsoever. This is the basis of *genetic expression* and *differentiation*. For more details of cis-regulatory regions and transcription regulation by histones see Wolpert (2003), Zhang (2003), Zhang and Reinberg (2001).

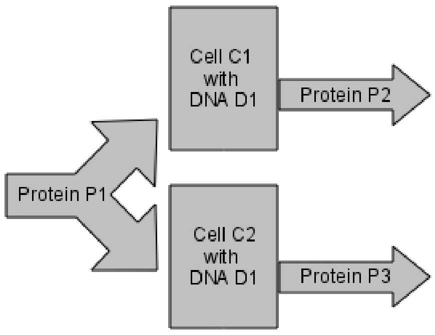


Figure 4.2: Protein synthesis based on cis-regulatory region presented

4.2.1.4 Inductive Communication

The GRN includes a complex network of communication between living cells in an organism. It assists in the development of an embryo into an organism and later in the maintenance of a homeostatic environment inside the organism. One of the methods of cellular communication is a process called induction, which is the result of many cells generating a signal together, thereby inducing other cells into action through a *community effect*. In addition, cell signalling is simple and selective (i.e. selects between a range of simple functions, like stop and go) and not instructive. This means that while signalling is simple, processing is complex.

4.2.2 Principles of Protein Processing

Section 4.2.1 describes some of the properties of the biological world that have inspired the PPAM. Most existing bio-inspired architectures that are inspired by similar principles (Greensted and Tyrrell, 2007, Jackson and Tyrrell, 2001, Mange et al., 2000, Prodan et al., 2003) employ reconfigurable fabrics and store a configuration bit stream (DNA) to (re)configure hardware. Other than being an overhead, such a DNA is also not very consistent with its biological equivalent, since the DNA (in a biological cell) is not just an initial configuration for the cell but is actively read, like a lookup table, as part of the function of the cell (Section 4.2.1). The transcription process performed inside a biological cell can be likened to the looking up of a table of answers. Incoming data is tested to see if any action or data is expected to be produced in response and the expected action or data is looked up in a table or database (DNA) using the incoming data as the unique

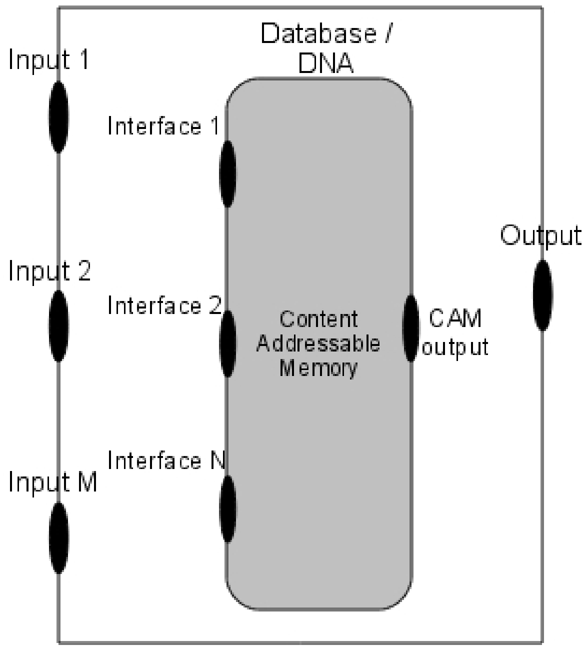


Figure 4.3: Protein Processing conceptual view: Data enters through an input channel and is matched using one of the interfaces (cis-regulatory region) in the CAM. A single output is generated from the CAM which could be sent back to one of the interfaces, or could be sent out of the Protein Processor through the output channel.

field for record identification. As can be seen from section 4.2.1, this seemingly simple lookup-based processing can actually result in an extremely complex system (GRN) which is capable of very complex (developmental) tasks, without having to *calculate* results using mathematical operators (like add, subtract, multiply and so on). In theory, with unlimited memory for the database, lookup tables can be used to perform any kind of processing. If the lookup table contains all possible "answers" to all possible combinations of incoming data, the cell can be used to generate any function. In a real world scenario, however, with limited memory only a limited number of "answers" can be stored. As may be seen from section 4.2.1, the biological cell also suffers from a similar issue and employs histones to *compress* the data, which also facilitates cell differentiation. The use of a custom Content Addressable Memory (CAM) emulates a similar process. As illustrated in Figure 4.3, the CAM has multiple interfaces which is the equivalent of defining the total number of cis-regulatory regions that can be excited in parallel. The contents of the CAM may be seen as the uncompressed DNA that can be matched at the cis-regulatory regions.

The PPAM is envisioned as a network of nodes, receiving input from multiple categories of inputs that are being associated together. This data, that is being associated is presented as input to the entire network of nodes, and so can be called the *external* input; as opposed to the information that each node receives from its neighbours, which can be called the *internal* input. Each node in the network receives one of these (external) inputs along with the outputs from other nodes in its neighbourhood (internal inputs). Figure 4.4 illustrates part of a network of 2 types of nodes (the squares and the circles). The squares

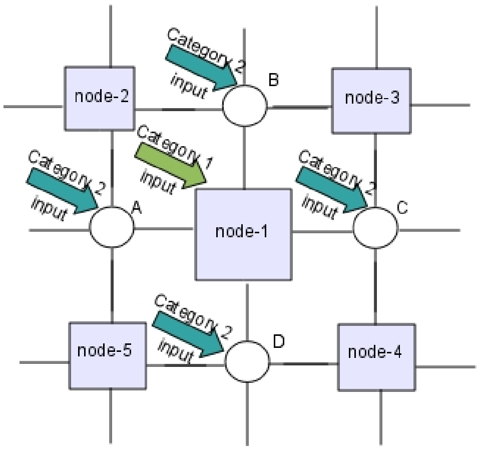


Figure 4.4: Abstract view of nodes: Diagonal arrows are external inputs and the grid connections are internal inputs.

(node-1, node-2, node-3, node-4, node-5) receive external input from the first category of external input (for example movement values for a robot's motors), while the circles (A, B, C, D) receive external input from the second category of external input (for example sonar values from a robot's sensors). External inputs are indicated with the thick diagonal arrows. In addition, all nodes receive inputs from neighbouring nodes, which are indicated by the thinner grid lines connecting nodes. Although the nodes illustrated in Figure 4.4 are arranged in a grid, note that this topology is presented only to facilitate the concept of neighbourhood and that other network configurations are possible. For instance, in the experimental setup in chapter 5, the neighbourhood of each node in one category consists of all nodes of the other category.

Input data from the external world is a time series, which means that it is a sequence of data points sampled at uniform time intervals. In each node, input data (both environment input and neighbouring node input) is looked up in the custom CAM (compare with cis-regulatory excitation described in section 4.2.1). If the data is found in the CAM, a corresponding output value is generated as a result (compare with transcription resulting from cis-regulatory excitation described in section 4.2.1). If a node does not receive an input from the environment, it combines the inputs from its neighbouring nodes into a word and looks this up in the CAM to determine what (if any) output value is to be generated. The value output from the CAM represents the output that the node should generate upon observing the current state of its neighbourhood (details of how this value is placed in the CAM are presented in section 4.3). Therefore in the absence of one category of environmental input, each node recalls a view of its neighbours which is a partial view of the relationship between the two categories of inputs. Together, all the nodes reconstruct the "complete picture".

From another perspective, the CAM can be likened to a two-field look up table, with the difference being that it performs a look-up in both directions. Data is not *calculated* and therefore nodes are free of arithmetic units. Instead, data is considered as abstract symbols and as long as symbols can be uniquely identified, real-world data can be encoded and decoded correctly. Although this is the only encoding *requirement* for the PPAM (unique symbols), in order to enhance robustness, encoding can be composed of symbols in redundant combinations. This translates into the following (based on the codon encoding principles outlined in Section 4.2.1):

> **Requirement:** *" Let data S_1 be defined as being composed of (a subset of) symbols from the set S, with elements $s_1, s_2, s_3, \cdots, s_n$. Let S_1 be encoded into E_1 using (a subset of) symbols from the set E, with elements $e_1, e_2, e_3, \cdots, e_m$. Then there should be multiple encodings $E_1, E_2, E_3, \cdots, E_M$ all using a (possibly different) subset of E that all decode to generate S_1. "*

This approach has two advantages. Firstly, just like genetic code is resistant to point-mutations (section 4.2.1), it adds inherent fault tolerance for transmission errors. Secondly, if decoding results in an unknown pattern of symbols, this is an indication of more severe faults having occurred during transmission that can be signaled to companion controllers – or lower or upper layers in the case of SABRE (Chapter 1). Although at first glance this may seem trivial (and in a sense it is), the important thing to note is that since the PPAM considers input data as abstract symbols, implementing such a scheme on the PPAM is free from overhead, since it does not impact the architecture's memory storage or recall capabilities. On the other hand, an architecture that uses mathematical operators on data (and does not consider data as abstract symbols) might need to augment its implementation with algorithms that allow such an encoding/decoding scheme. Furthermore, for the PPAM, this encoding is not a requirement but rather an optional enhancement.

Nodes do not necessarily have to generate an output upon presentation of an input. A node is said to *fire* if the input causes it to generate an output. Firing nodes *self-regulate* which means that they update the tables in their memory in response to the inputs (compare with proteins causing a change in genetic expression by altering the cis regulatory region presented for excitation). The method by which the nodes update their contents can be seen as the process of *acquiring* defined by Sargent and Stafford (1965). The following sections present the biological basis and the principles of this learning method.

4.3 The Node as a Neuron

4.3.1 Biological Inspiration: The Nervous System

The mammalian nervous system has been the source of a major field of Artificial Intelligence – namely, Artificial Neural Networks (ANN) (Abraham, 2005). The nervous system consists of a network of cells that communicate information about the surroundings of an organism and its state and also processes that information. It can be divided into two classes – both made up of neurons and glia cells.

- central nervous system
- peripheral nervous system

The central nervous system consists primarily of the brain and the spinal cord and coordinates the function of the various parts of the body. The peripheral nervous system connects the central nervous system to the extremities and communicates the sensory information from the limbs to the brain.

4.3.1.1 The Neuron

The human brain is made up of approximately 10% neurons (more than 10 billion), 90% glia cells, and more than 60 trillion connections (Fuster, 2006). Neurons can be one of the following:

- sensory neurons: which respond to sensory organs (sound, light, touch) and send signals to the spinal cord and the brain;
- motor neurons: which receive signals from the brain and spinal cord and cause muscle contraction and affect glands;
- inter-neurons: which connect neurons to other neurons within the brain and the spinal cord.

A typical neuron (depicted in figure 4.5) transmits its signal to over 10,000 other neurons (Vreeken, 2003), which process information at their inputs by summation and if the sum falls above a certain threshold, fire to generate an *action potential.* An action potential

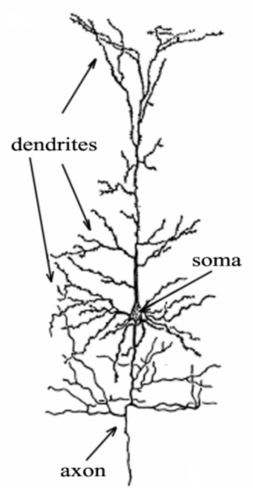


Figure 4.5: A biological neuron (from Vreeken, 2003)

is a short (approx. 1ms) and sudden increase in voltage – a spike – which is transmitted across the body of the neuron. Signal strength is maintained by *bodies of Ranvier* (Vreeken, 2003) and also by duplicating the signal at junctions. Neurons transmit information to other neurons using chemical signals across *synapses* and the flow of information is mostly uni-directional. This is called *neuro-transmission.*

Synapses have proven to be much more complicated than was originally thought (Vreeken, 2003). They are complicated pre-processors and are crucial to learning. When a spike arrives at a synapse, it is *likely* (non-deterministic) that some vesicles fuse with the cell membrane and release neuro-transmitters, which have an excitatory or inhibitory result. Synapses are either excitatory or inhibitory and there has been no evidence so far to suggest that synapses can change effect during their lifetime (Vreeken, 2003). After firing, the neuron enters a state of rest for a short period of time (approx. 10ms) called the refractory period. Summation in the neuron may be spatial where firing of synapses separated in space are summed or it may also be temporal where the same synapse firing repeatedly over time is summed. Sensory nerve endings can be the slowly adapting *tonic* receptors which respond to a steady stimulus with a steady firing rate, or the quickly adapting *phasic* receptors which respond to steady stimulus by stopping or decreasing the firing rate – thus we get used to an ambient smell (Madrid et al., 2003). Recent research shows that neurons code information in the timing of single spikes and not just in their average firing frequency (Vreeken, 2003). A neuron may have many connections communicating its output (axons), allowing communication with multiple cells; thus neural networks may be divergent (one to many) or convergent (many to one). For more details of the structure and operation of neurons, see Li and Du (2007), Vreeken (2003), Kasabov (2007) and Coppin (2004, Chapter 11).

4.3.1.2 Potentiation, Plasticity and Robustness

As shown by Hebb (1949), neurons that *fire together, wire together* which is known as the Hebbian principle. More formally (Coppin, 2004, Chapter 11):

> *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."*

During the development of an embryo, many neural pathways are created so that some may even be repetitive, thereby creating duplicate paths that respond to the same input. Every time one of these repetitive paths wins over the other, its neuron-to-neuron synapses are strengthened, increasing the efficiency of signal transmission over this path. This *long term potentiation* (LTP) is a long lasting improvement in the ability of neurons to communicate with each other and is the basis for learning. Exactly how this improvement is achieved in biological neural networks is a topic for ongoing research. Benuskova and Abraham (2007), Benuskova and Kasabov (2007) and Jedlicka (2002) indicate that the BCM (Bienenstock, Cooper and Munro) rule (Bienenstock et al., 1982), which presents a physical theory of learning in the visual cortex, is still arguably the most accurate model for synaptic learning in the brain. The main difference between BCM and other variants of the Hebbian principle is the shifting synaptic potentiation threshold.

Changes in the organisation of the brain that occur as a result of experience are called *neuro-plasticity*. Neuro-plasticity is also thought to be the source of the robustness of the brain; particularly as it is not known how important cell division is to regeneration and plasticity (Brockes and Kumar, 2003). The effects of plasticity are evident from constraint-induced movement therapy. As far back as 1917 experiments on monkeys by Oden show how the brain can rewire itself so that a paralyzed arm can become useful again (Lashley, 1933). Further information on neuro-plasiticity and LTP can be found in Ros et al. (2006) and Welberg (2008).

4.3.1.3 Organisation

The traditional view of the central nervous system consists of discrete cortical columns and maps dedicated exclusively to individual cognitive functions. A *cortical minicolumn* is a vertical column through the cortical area of the brain, usually consisting of 80 to 120 neurons. Cortical maps are minicolumns that are identified as performing specific information processing. Cortical organisation is often described in terms of cortical maps implying that information from one sensory organ goes to a specific area in the brain. In terms of plasticity, if a cortical map is deprived of its input, it becomes activated at a later stage in response to another input, thus rewiring the inputs. Details about cortical organisation and rewiring of cortical maps can be found in Kasabov (2007). The method of how the cortical maps are rewired is still an open question though.

More recently, this modular or *localizationist* view of cognition is being challenged by the newer distributed network perspective (Fuster, 2006) which points to "wide neuronal maps or networks that extend beyond the confines of anatomically defined areas". The *distributionists* extend the Hebbian learning principle by disassociating it from spatial proximity. Therefore, memory or knowledge is formed by associative synaptic modulation of connections between neurons, however distant they may be in space. Amongst the more recent theories for cognition are the ones based on *cognits* and network memory (Fuster, 1997).

Cognits are networks of neurons that have been associated together because of experience, either of the species or the individual. A neuron can be part of one or more cognits. All cognits derive from primary sensory or motor cortex. Memory or knowledge are cognits organised hierarchically in the cortex of association with the base being phyletic (evolutionary) memory and sensory cognits, and higher strata forming ever more general and abstract concepts, developing primarily in divergent fashion. This deviates radically from the traditional view of sensory hierarchies converging into assemblies that encode progressively complex functions. To illustrate, whereas *representations like power are concentrated at the top of industrial or military hierarchies, they are distributed at the top of cortical hierarchies.* This also means that *higher cognits are more resistant to damage* than are lower cognits, something that is reinforced by observations where patients suffering from motor paralysis continue to philosophize. Such theories have inspired many researchers, like Baxter and Browne (2009a,b), who related embodiment of an agent with cognition, and Kawamura et al. (2004), who implement a model of the working memory system in the human brain as a controller for the upper-torso of a humanoid robot.

4.3.2 Learning in a PPAM Node

Learning in the PPAM is based on the Hebbian principle and is continuous so that there is no separate, distinct training phase. A firing node records a snapshot of its neighbours, indicating which neighbours fired (and their corresponding output values) in conjunction with itself. To illustrate, let the CAM in node-1 (Figure 4.4) contain values as shown in Table 4.1a. Let the environmental inputs be such that when 0x0001[3] is input to node-1,

Input Pattern	**Output Value**
0x0001	0x5
0x0002	0x6
0x0004	0x7

(a) Initial values in CAM

Input Pattern	**Output Value**
0x0001	0x5
0x0002	0x6
0x0004	0x7
.	.
.	.
0xBDCA	0x5

(b) Values in CAM after regulation

Table 4.1: Learning values in CAM

the environment inputs for its neighbouring nodes (the circular nodes) make them fire with values A, B, C and D. Then node-1 fires with value 0x5 and observes 0x$BDCA$ from

[3]0x indicates hexadecimal notation

its neighbours (ordering the data North-South-East-West). It therefore adds a tuple to its CAM to reflect this view of its neighbourhood, resulting in a table as shown in Table 4.1b. Note that the symbol set being used by external inputs (including all categories) must never overlap with the symbol set of the word generated from the neighbourhood inputs. For instance, if the concatenated word from the neighbours of node-1 is 0x0004 (instead of 0x$BDCA$), this could add a tuple to the table where the *input pattern* field contains the word 0x0004 which is the same value as the third tuple in the table. Thus a fallacious conflicting record would be entered into the table which does not stem from any lack of relationship between inputs but an overlap of symbols.

4.3.3 Resolving Conflicts in the PPAM

Conflicting tuples can result if input variables are not related, or if the relationship is one-to-many (or many-to-many or many-to-one). In this case, nodes observe the same neighbourhood inputs in conjunction with different output values from the node itself. Consider two multi-dimensional variables, A and B, that have a one-to-one relationship as shown in Table 4.2. Although A (A_{D1}, A_{D2}) has a one-to-one relationship with B

Table 4.2: Relationship between 2 multi-dimensional variables

A_{D1}	A_{D2}	B_{D1}	B_{D2}	B_{D3}
3	0	5.0	5.0	5.0
0	-1.5	0.5	1.0	0.75
3	-0.25	3.5	4.0	4.75

(B_{D1}, B_{D2}, B_{D3}), this does not imply that A_{Di} also has a one-to-one relationship with any one of B_{Dj} or any combination of them. This is evident from Table 4.2. The memory in an individual node represents a portion of the relationship between one dimension of one category (for example A_{Di}) and the dimensions of the other category (for example B_{Dj}). This means that even when there is a one-to-one mapping in the dataset, the relationship in the nodes may be otherwise, resulting in conflicting tuples. However, this has the advantage that extrapolating from this information does not necessitate solving mathematical equations which require ALUs. Conflicts are resolved by a swapping process illustrated in algorithm 1. Over time, the tuples observed most often will move to the top and will be arranged in the order of the most frequent to the least frequent. The least frequent tuples are the first to be removed when a node runs out of memory.

Algorithm 1: Pseudo-code for frequency detection algorithm

```
Input: neighbourhood firing pattern $p_n$, current node firing value $f$, memory $M$ of size $2 \times N$
// $M[0][x]$ is neighbourhood pattern at memory location $x$
// $M[1][x]$ is firing value in CAM at memory location $x$
// $j$ stores index of record to swap down
$j \leftarrow$ unique init value;
for $i \leftarrow 0$ to $N$ do
    if $p_n == M[0][i]$ then
        if $f == M[1][i]$ then
            if $j != $ unique init value then
                swap tuple at $i$ with tuple at $j$;
                break loop;
        else
            $j \leftarrow i$
if $j ==$ unique init value then
    add new tuple at end of $M$ – location $x$;
    $M[0][x] \leftarrow p_n$;
    $M[1][x] \leftarrow f$;
```

Note that during recall, if variable B has value $(5.0, 5.0, 4.0)$, from the perspective of the PPAM, this is equivalent to $(5.0, 5.0, x)$ where x can take any value not observed before by the CAM – which would therefore not be present in any of the nodes. Thus, this would result in hetero-associative recall. If, on the other hand, the variable B has value $(5.0, 5.0, 5.0)$ then this is auto-associative recall, since B is fully observed before and, considering one tuple in Table 4.2 as one data-point, the complete data-point would have to be recalled from partial information.

4.3.4 Operation of the PPAM Node

In general, most computing methods suffer from the *curse of dimensionality*, which is a deterioration in performance as dimensions are increased. The PPAM, on the other hand depends on there being multiple dimensions in the data; in fact, greater dimensionality helps the PPAM to generalise better. Intuitively, the PPAM uses partially observed data to make a *guess* about the complete input. This is similar to a person observing part of a chair and, having observed the complete chair before, guesses that the complete object is a chair. Therefore, if the PPAM was associating two single dimensional variables, there would be no partially observed data, and the PPAM would be unable to generalise. As illustrated in Figure 4.3, when a node observes no external input, the data from its neighbours is tested against each of the CAM interfaces (Figure 4.3), which are inspired from the cis-regulatory regions in a cell. Each interface indicates how well the incoming data was matched. If the data was fully observed before, an interface would indicate with

100% confidence that it matched the input data; for partially observed data, the interface with the highest confidence of match is the one *selected* by the CAM to create the output via the CAM's output interface. Nodes are initialised with a certain number of interfaces which can operate in parallel and some initial values in the CAM which determine whether a node will *fire* in response to an input or not. For example, the data in Table 4.1a may be the initial contents of the CAM of a particular node of the PPAM, when its brought up from the reset state. Over time, and as the learning continues, the contents might be modified to the values shown in Table 4.1b. The choice of the initial values in the CAM is dependent on the following two questions:

1. Whether these input values are desired to be remapped (or re-encoded) to other values for the internal use of the PPAM. For example, in Table 4.1a, input 0x0001 is being remapped to 0x5. If re-encoding was not desired, then the first tuple would have 0x1 as the output value for input 0x0001.

2. What input values should cause the node to fire. For example, in Table 4.1a, the node is designed to fire for inputs 0x0001, 0x0002, and 0x0004.

The answer to the first question is dependent upon the encoding requirements that were discussed in Sections 4.2.2 and 4.3.3. To answer the second question, two issues need to be considered. Firstly, data should be stored so that it is distributed, thereby allowing parallelization and avoid reducing to a serial search. Secondly, data should be duplicated to allow robustness through degeneracy (as opposed to redundancy). Therefore, although inputs are fixed in terms of their connections to nodes, since the same inputs are provided to multiple nodes, and since *learned* data is duplicated in nodes, the data is not lost when a node is lost. A more systematic method of designing nodes is addressed in the second version of the PPAM presented in Chapter 6.

4.4 Organisation and Robustness

Section 4.2.2 outlined a strategy for fault-tolerant communication based on the codon encoding principles (Section 4.2.1). Briefly summarising, this meant that communication would be resistant to most single-bit errors and, in the case of more severe errors, that faults would at least be detected. In addition, using the principles outlined in section 4.3.1.3 and 4.2 to organise the PPAM further increases its robustness in a number of ways:

1. **Modularity**: A PPAM associating 2 variables may be considered as an individual module (cognit) that connects to other modules (cognits) which together implement a higher function.
2. **Data Degeneracy**: A PPAM is a network of nodes, none of which are individually crucial to the correct operation of the whole.
3. **Configuration Bit-Stream**: The lack of a configuration bit-stream means that a PPAM does not need to be reconfigured to learn a new behaviour.

4.4.1 Modularity

In order to enhance scalability, the PPAM architecture is designed to be modular. A single instantiation of the PPAM, receiving inputs from two categories of inputs, would associate these together and multiple PPAM can be cascaded, associating many different types of environmental inputs. Hence a control algorithm (for example in an intelligent agent) can be distributed over multiple instantiations of the PPAM. Each instantiation can be physically placed at different locations within the *body* of the agent. Thus, for example, in a robot, one instance of a PPAM may be placed close to the sensors above the front-left wheel, another close to the front-right wheel and so on. Each of these instances may associate sensor values and wheel movements, thereby performing simple object-avoidance. Another instance of the PPAM (placed in a more central location within the robot's body) may associate movement with a way-point in physical space and bias the object-avoidance with locations. This would result in the robot navigating to way-points. Such an architecture has a number of implications.

1. As advocated by Neal et al. (2006), partial computation is performed locally (object-avoidance PPAM in the example above), close to the point of action rather than sending data all the way back to a central controller for every control operation.
2. Reflex action strategies may be implemented, allowing faster reaction times.

3. Subsumptive architecture (Coppin, 2004, Chapter 19) principles are implemented where higher functions subsume lower ones. Thus, for example, the PPAM associating way-points with the robot's movement (described above) subsumes the instances of the PPAM providing reflex actions to sensor movements.

4. Overall fault-tolerance is increased as control is distributed. Thus, if the PPAM implementing higher order functions (like associating movement with way-points in physical space) becomes faulty, the reflexive object-avoidance PPAM would still operate correctly. In fact, as the PPAM instances (in the example above) are physically distributed over the body of an agent, individual portions of the body can be broken or removed while other portions continue to operate. This is similar to the distributed nervous systems in biological beings.

4.4.2 Data Degeneracy

As detailed in chapter 3, fault-tolerant techniques can loosely be classified as one of (active / passive / hybrid) hardware redundancy, time redundancy or information redundancy (Iyer and Kalbarczyk, 2003, Karri and Nicolaidis, 1998, Patel and Fung, 1982). Whereas the lower layers of the SABRE architecture (Chapter 1) employ hardware redundancy for fault-tolerance, the primary mechanism in the PPAM is information (or data) redundancy. Data storage is distributed over the nodes and the location of the datum defines its relationship to other data. Data is duplicated to represent reverse direction relationships. This duplication has 3 advantages, which makes this data degeneracy rather than data redundancy:

1. It implements information redundancy thereby increasing fault-tolerance.

2. It improves the recall efficiency since reverse relationships do not have to be calculated.

3. It facilitates quorum sensing.

These 3 features create an important distinction – whereas hardware may store duplicate data, the hardware itself is not duplicated.

Traditional hardware redundancy techniques (active or passive) maintain (functionally) idle hardware. This means that there is no drop in performance as long as redundant hardware is present, as the faulty block can be replaced by spare ones (Iyer and Kalbarczyk, 2003). However, this also means that after all the spare blocks have been used up, even one more fault will break the entire function. In contrast, the PPAM does not duplicate hardware. Areas in hardware that store duplicate data also have a functional use and therefore, are not simply redundant duplicates. Since there is data redundancy, this means that a fault in a block results in a drop in performance (and possibly efficiency), but no node is a single point of failure for the whole system. Therefore if a node dies, although it may result in a drop in performance, there will still be some relevant and useful output from the system, resulting in a graceful degradation of performance as nodes become faulty. Results for experiments that show this are presented in Chapter 5, and more extensively in Chapter 7 (Section 7.4).

As lower levels of the SABRE architecture (Chapter 1) developed at UWE Bristol (Samie et al., 2009a,b) implement more traditional hardware redundancy techniques, this presents two lines of defence against faults. As a first response, faulty blocks can be replaced by spare blocks at lower levels. If the lower levels run out of spare hardware, the PPAM architecture at the higher level can ensure that some functional output continues to be generated in spite of reduced hardware resources.

4.4.3 Configuration Bit-Stream

The PPAM has thus far been described in terms of broad operations which have been abstracted away from hardware implementation. However, it is important to bear in mind that the ultimate object is to design for hardware. The PPAM is envisioned as a hardware architecture that is generic enough to be applied to many different types of control or associative recall problems. In this respect, it is similar to reconfigurable hardware. FPGAs provide major advantages to designers in terms of their flexibility. Hard and soft macros for a wide range of applications are available, including signal processing and floating point operations. In addition, the more recent versions use less power while providing more resources with an improved software tool-chain for fast implementation (more details in Chapter 2). However, in a traditional FPGA, a configuration bitstream is needed to configure a set of LUTs to perform a logic function. This logic function could, for instance, be a controller for a robot that performs object-avoidance. If the same (or similar) FPGA needs to be used as an image processor (or even a slightly different object-avoidance controller), the FPGA would need a different configuration bitstream. The traditional method of reconfiguration is to do it offline, although some advances have been made in recent times towards online (partial) reconfiguration (Mcdonald, 2008). Although the PPAM is not designed to be as flexible in its range of applications, in terms of robust applications for AI, it has some advantages over the FPGA.

Unlike conventional reconfigurable hardware, the PPAM does not require a configuration bitstream, which is a major advantage since it avoids all the problems of (partial or full) reconfiguration at run-time. Assuming the PPAM is of sufficient size (has sufficient number of nodes) for both tasks, the same PPAM can be used for both tasks without any changes in the architecture. This is because computation results from the location of individual data in memory and the PPAM modifies the location of data continuously and online. Therefore, not only is there no need for different configurations, but no extra memory is required within the PPAM for storing such configuration bitstreams. Other than reducing overhead memory requirements, this also means that whereas FPGAs are susceptible to faults through errors in the configuration memory, the PPAM does not have this type of memory which might cause faults. Furthermore, given that the PPAM is generic enough not to require reconfiguration to perform different tasks, this makes it particularly suited for implementing *plasticity* (section 4.3.1). In the context of the PPAM, plasticity is observed if a particular instance of the PPAM gets broken and another instance can take over its operation at run-time without any external assistance. The following illustrates in terms of an example.

Let us assume that four PPAM instances associate sensor values and movement to perform reflex-action object-avoidance for each (cardinal) side of a robot performing a navigation task. Another PPAM instance operates at a higher level and associates way-points with location in physical space. Consider the case where a PPAM from one of the lower-layers has a fatal error so that the robot does not have sensor or motor control on one side. At this point, one of two things should occur. If the fault in the PPAM instance is such that it can still forward the incoming sensor values on to the higher level PPAM, and if the outputs of the higher-level PPAM can be routed to the motors, then the higher level PPAM should be able to adapt to the new input (raw sensor values rather than the lower level

PPAM state). If, on the other hand, the fault is such that the signals cannot be forwarded, the higher-level PPAM would receive erroneous data from the broken lower-layer PPAM. In this situation, the higher layer PPAM should be able to determine that the data from the broken PPAM layer is unrelated to its movement since there will be no relationship or *association* between that data and the robot's location. It should, therefore, be able to update its associations to reflect that it only has control of the other 3 lower-layer PPAM.

4.5 Summary

This chapter presented an abstract first view of an architecture called the Protein Processor Associative Memory (PPAM), designed to perform hetero-associative recall. Sections 4.2.1 and 4.3.1 presented some biological background, justifying the choice of name while Sections 4.2.2, 4.3.2 and 4.3.3 identified the principles that were abstracted from the biological world. Section 4.4 detailed the organisation of the network of nodes in the memory and theorised about the fault tolerant properties.

As mentioned earlier, (chapter 2), current techniques in AI rely (directly or indirectly) on arithmetic operators. On the other hand, the PPAM is designed to extract computation from memory operations without using arithmetic operators. Therefore, a comparison of existing techniques with the PPAM in terms of the number of and type of arithmetic operations performed and resources required should provide some useful insight.

4.5.1 Comparing the structure of BAM with PPAM

Bidirectional Associative Memories (BAM) were described in Section 2.1.3.4. Kosko's BAM and the PRLAB form a useful comparison since most other BAMs are variants that still use arithmetic operators. Table 4.3 compares the structure and the kind of operations required for a BAM and for the PPAM. BAM implementations are assumed to be optimized such that calculations are not repeated and results are considered to be stored in temporary memory (CPU registers). Memory access (read/write) does not include access to these temporary locations or to any other *working* memory. Furthermore, memory requirements (or operations) for stimulating nodes or providing input is not included. In addition, the multiplication and addition operations for calculating indexes of the weight-threshold matrix for the BAM is also not included in Table 4.3 because this is considered to be similar to the significantly simpler and smaller number of memory index counting operations for the PPAM. Table 4.3 is generated for the case where an associative memory is storing a set of vector pairs $T = \{X^{(k)}, Y^{(k)}\}_{k=1,\ldots,S}$. For the BAM, $X^{(k)} \in \{-1,+1\}^N$ and $Y^{(k)} \in \{-1,+1\}^M$, while for the PPAM $X^{(k)} \in \{0,1\}^N$ and $Y^{(k)} \in \{0,1\}^M$.

Table 4.3: Comparison of the structure of the BAM and the PPAM

	BAM	PPAM
Connections	M nodes connected to N and N nodes connected to M	M nodes connected to N and N nodes connected to M
Memory requirements	$(M \times N) + M + N$ floating point numbers	worst case – $S \times (M+N)^2$ bits
Encoding requirements	bipolar preferred	none
Operations per each of N nodes during recall		
Multiplications	M	0
Additions	$M+1$	0
$>$ Comparisons	1	0
$<$ Comparisons	1	0
$==$ Comparisons	0	up to S
Iterations	till stable	till stable
Memory reads	$M+1$	S
Operations per each of N nodes during store		
Multiplications	$(2 \times M) + 3$	0
Additions	$(2 \times M) + 2$	0
$>$ Comparisons	0	0
$<$ Comparisons	1	0
$==$ Comparisons	1	S
Iterations	Epochs $\times S$	1
Memory reads	$M+3$	S
Memory writes	$M+1$	1

Both BAM and PPAM require fully connected networks, where M nodes have N connections and N nodes have M connections. The BAM requires one $M \times N$ connection weight matrix storing floating point numbers, even if the data is not real-valued (as in this case). In addition, it also requires one floating point threshold value for each node. On the other hand, each node of the PPAM requires an M bit wide memory for Y and an N bit wide memory for X, each of which could be up to S locations deep in the worst case

– although in practice it may be much less (Chapter 7). It is likely to be much lower and in the experiments conducted (Chapter 5), the worst case observed was 72 locations for one node while the average requirement was 12 and the minimum was 1 location (where $S = 228$). Depending on the encoding, there can be $M \times N$ nodes in the network. Details of the calculations required to recall values and store associative pairs for the BAM and using PRLAB can be found in Kosko (1988), Oh and Kothari (1994). Note that all memory variables in the BAM are real values and therefore all BAM calculations require floating point arithmetic.

This comparison indicates that the PPAM requires no arithmetic operations and far fewer logical operations than the BAM which may be regarded as a competing technique for the kinds of applications that the PPAM is targeted towards. It remains to test whether the PPAM is able to provide similar or better performance than the BAM. Chapter 5 presents some experiments performed on this architecture and discusses the results observed.

Chapter 5

PPAM: Experiments and Results

This chapter presents the experiments that were performed on the hardware abstracted architecture described in Chapter 4 and discusses the results observed. It is postulated that the PPAM is better suited than the more traditional ALU based architectures to implement artificial intelligence applications, particularly in the domain of associative memory. In addition, it is also postulated that the PPAM degrades gracefully in the presence of faults. In order to test this, experiments need to be conducted to determine the following:

1. Is the PPAM capable of learning associations between two variables?
2. If 1 is true, can this learning of associations result in the PPAM performing a *meaningful* task?
3. If 2 is true, is it better than other traditional AI techniques at performing the task?

These three items are each dealt in Sections 5.1.1, 5.1.2 and 5.1.3 respectively.

Testing for robustness requires faults to be injected and performance to be measured using the same criterion as for the items listed above. Criterion for each of these are specified in more detail in Section 5.1 while Section 5.2 describes the experiment setup used. Section 5.3 presents results for 1 and 2 above. Section 5.4 describes the method and parameters for performing test 3 above and Section 5.5 presents the results. Section 5.6 discusses the results observed while Section 5.7 concludes and summarises.

5.1 Hypotheses and Methods

5.1.1 Hypothesis for Association

A single PPAM, receiving inputs from two categories of inputs, should be able to store the associations and later recall the associated value in the absence of one of the two inputs. A black box approach is used to measure this storage capacity and recall ability of the PPAM and two measures are defined – *Accuracy* and *Difference*. The following experiment was designed to test this hypothesis.

A single agent (robot) is placed in a static environment and initialized with a simple reactive object-avoidance algorithm (Algorithm 2), which controls its movements to randomly navigate the arena. If the sensor values observed indicate that a collision is imminent, the

Algorithm 2: Pseudo-code for innate, object-avoidance algorithm

```
Input: sensor values array $A$
Output: forward speed $s_o$, rotation speed $r_o$
// $s_f$ is constant forward speed used to move forward
// $s_s$ is constant forward speed used to avoid obstacles
// $r_i$ is constant rotation speed to avoid obstacles
$C \leftarrow$ true if any A[i] shows distance to object is less than a threshold value;
if $C == true$ then
    // set forward speed to Object-Avoidance speed
    $s_o \leftarrow s_s$;
    if object to left closer than object to right then
        $r_o \leftarrow -r_i$;
    else
        $r_o \leftarrow r_i$;
else
    $s_o \leftarrow s_f$;
```

agent is rotated to the left or the right depending on which direction indicates that objects are farthest. The movement value generated ($\{s_o, r_o\}$) and the sensor value observed (A), form the two variables in an associative pair. The associative memory can be seen as observing the sensor values (as they are encountered by the agent) in conjunction with the corresponding actions or movement values generated by the *innate* object-avoidance algorithm, and associating these two together. If the agent is now reset back to its initial position and orientation, it will deterministically observe the same sensor values as it did before in the static simulation environment. Instead of using the values from the object avoidance algorithm, this time the agent uses values recalled from memory. This is equivalent to storing the training dataset and then testing if the values in the training dataset can be recalled. *Accuracy* is defined as the percentage of values in the training dataset that are fully and accurately recalled. Furthermore, if the associations have been stored correctly, then the agent will traverse the same path as it did before. Therefore, *Difference* is defined as the difference in the path followed by the agent when using the values recalled from memory. If the dataset has been stored correctly, this difference should be zero.

To test the robustness of the PPAM, errors are injected into nodes and the same performance measure is used. Thus, the PPAM is replaced at the initial position and orientation with some nodes broken and the results are observed. The stuck-at fault model is used so that a node observes the value from its neighbour being stuck at a certain value. The advantage of this is that it can emulate faults in the connecting wires as well as those in the nodes itself. The disadvantage is that it does not emulate transient faults.

5.1.2 Hypothesis for Extracting Relationships

The test outlined in Section 5.1.1 will show whether the PPAM can store associations at all. However, it begs the question whether this is simply rote-learning where only the data points presented in the training set are learnt and no meaningful information is extracted about the overall relationship. Therefore, it is important to test if the PPAM can store association information that is *representative* of the input dataset but not explicitly included as long as it may be extrapolated from the dataset.

In order to test this, the agent is subjected to new and previously unobserved environments. Whereas in Section 5.1.1 the agent was reset to its initial position, for this test, the agent is placed in a location and orientation which it has not observed before. The sensor values observed now would be values that are (completely or partially) previously unobserved. The hypothesis being that if the agent is now able to successfully navigate the arena without collisions, then it may be inferred that it has extracted some *representative* information about the relationship. This success is measured as follows. Firstly, an *accuracy* measure is defined similar to the one used in Section 5.1.1. Whereas in Section 5.1.1 accuracy was tested against the training dataset, in this case accuracy is tested against the *predicted* values that would have been generated by the object-avoidance algorithm had it been subjected to the same sensor values. The recalled movement values are output from the memory and tested against this "absolute correct value", which is the output of the innate object-avoidance algorithm, given accurate sonar inputs. Secondly, *Recall confidence* is defined as the confidence with which the PPAM recalls values. For the PPAM, this is an output of the memory and is directly proportional to the portion of the neighbourhood input that was found in the memory. If a node has a neighbourhood of N nodes, it receives a total N words from its neighbours as input. If during a recall operation, the node is able to locate F of these words in its memory, then the confidence is simply $C = F/N$. Note that this does not imply a division operation within each node, because the nodes only output the value F. The division is optionally performed only to normalize the output and can be implemented (if required) in a companion controller based on the more traditional ALU-based architecture. Actions are deemed *critical* if the action is necessary to avoid collisions – as opposed to actions that will not result in collisions even if they are wrong. The PPAM, as applied here, is incapable of distinguishing between critical and non-critical actions as it lacks the necessary inputs to relate actions with long-term results. However, an analysis of the associativity using this classification should still prove interesting, and so, a third measure called *Critical and Accurate* is defined as the percentage of memory recalls which were critical and also correct. Fourthly, a "Behaviour" measure is defined as the number of collisions with objects that the agent experiences as it navigates the environment. It should be noted that *behaviour* takes precedence over *accuracy*, since the ultimate measure of success is navigating without collision. This will be evident from the case where the agent can make a critical turn to the left to avoid a collision whereas the innate object-avoidance algorithm (Algorithm 2) might have taken a right turn. Therefore, a lack of *accuracy* need not necessarily indicate a negative result, as long as this different value does not result in a collision.

5.1.3 Hypothesis for Comparing with Other Techniques

Comparing the performance of different techniques requires careful thought because a measure for performance that is unbiased to both techniques must be defined. This is the reason for having chosen black box methods of measuring the performance. Therefore, the same measures for *accuracy*, *difference*, *behaviour* and *critical actions* can be reused.

However, since the measure for *recall confidence* is an output of the PPAM, it is dependent upon the structure of the memory. Consequently, it cannot be reused as defined in Section 5.1.2 and is redefined below.

The techniques chosen for the comparison are Bidirectional Associative Memories (BAM) and their variants, which were described in Chapter 2. They have long been the subject of analysis and have formed the basis of many later models. Furthermore, as shown in Section 2.1.3.4, BAMs, particularly with the optimal PRLAB, have long been used for performing and bench-marking hetero-associative recall. Even though some variants exist today that, in some respects, out-perform BAM and PRLAB (discussed in Section 2.1.3.4), this comparison is still performed with the original BAM and PRLAB because:

- They are the most representative of the Bidirectional Associative Memory domain and are rigorously tested,
- PRLAB, if implemented correctly is **optimal**,
- A major point of comparison is the number of arithmetic operations performed (Section 4.5.1) and all BAM variants also perform a similar number of (if not more) arithmetic operations.

The experiment starts by investigating the efficacy of BAMs for hetero-associative recall in the context of AI for robotics. The same experiments (as outlined in Section 5.1.2) are performed using a BAM and two different training algorithms are examined for the generation of the correlation matrix, namely the original BAM training algorithm outlined in Kosko (1988) and the popular Pseudo-Relaxation Learning Algorithm for BAM (PRLAB) first presented in Oh and Kothari (1994). Association is performed between sensor values and the actions needed to avoid obstacles on a robot. Even though the PRLAB guarantees optimal learning, distinguishing between values that are closely spaced is a weakness of the BAM. Nevertheless, such values are realistic inputs for a real-time, noisy environment and the experiment tests the effect of optimising the parameters to maximise recall of the training dataset. A comparison can then be performed of the robustness and noise tolerance of the PPAM, and the noise tolerance as reported by Sudo et al. (2009) for the BAM and Self Organising Incremental Associative Memory (SOIAM).

A recalled value is considered to be perfectly recalled if the value recalled is exactly the same as was trained – the accuracy measure defined in Section 5.1.2. Considering all values trained as attractors, a recalled value is *almost-recalled* if the closest attractor is the correct one. Euclidean distance is chosen as a measure for *closeness* because of the need for orthogonality in BAMs. Since the BAM uses weighted-summation, Euclidean distance is more appropriate than other measures like hamming distance for testing if data points are far apart and, therefore, less likely to be confused with each other. Note that detecting almost-recalled values implies post-processing of the output and *a priori* knowledge of the attractors to determine which attractor is closest. This means that the memory must have another copy of all the associative pairs stored and test the recalled value against all the associative pairs to find the closest one – in essence, duplicating its own behaviour. Despite being unrealistic (particularly for online incremental learning), this method was used to maximise the performance of BAMs.

Recall confidence for the BAM is a measure of the confidence with which the BAM recalls the values. If a value is perfectly recalled, this would be 100% confidence. The distance from the attractor is used to measure the confidence for almost-recalled values. Thus, given that a BAM is trained using a set of training vector pairs $T = \{X^{(k)}, Y^{(k)}\}_{k=1,\dots,S}$, each element of set T can be viewed as an attractor. Furthermore, while recalling X, each of $X^{(k)}_{k=1,\dots,S}$ is an attractor and similarly for Y. Assuming that the BAM is now provided an input $\tilde{X}^{(l)}$ to generate an output $\tilde{Y}^{(l)}$, the confidence of the output $\tilde{Y}^{(l)}$ can be measured as:

$$C = \frac{\lambda - \min\{\text{abs}[||\tilde{Y}^{(l)} - Y^{(k)}||_2]_{k=1,\dots,S}\}}{\lambda}$$

where the subtract operation is performed using Euclidean distance measure for the multi-dimensional vectors Y. λ is the maximum possible distance from the attractor using the current network size and therefore is the lower bound for the confidence measure. For a BAM with n nodes performing recall, it can be represented as $\lambda = 2^n - 1$.

5.2 Experiment Setup

Player/Stage (Gerkey et al., 2003, Owen, 2009) was used to simulate the environment. The agent used was the Pioneer P3-DX Robot[1] with 7 functional sonars pointing towards its relative north with an angular difference of 22.5° from each other. The sonars have

[1]http://www.activrobots.com/ROBOTS/p2dx.html

a range of 5 meters and the area within which the robot navigates is 14×14 meters. The stage driver for the pioneer sonars returns real values in the range $0.0 \leq s \leq 5.0$. Since analogue, real-world data would need to pass through analogue-to-digital converters (ADC) which perform quantisation, this is approximated by discretizing the sonar values into 10 bins so that each bin is 0.5 meters. Since there are 7 sonars, this makes a total of 10^7 or 10 million possible unique values for the 7-dimensional variable that represents sonar input. Movement is represented by a *forward speed* and a *rotation speed* both of which are real values. The object-avoidance algorithm generates one of two possible forward speeds (slow or fast), depending upon the sonar values. Similarly, for rotation, the algorithm can generate one of three possible values – "turn left", "turn right" or "go straight". Therefore, rotation has 3 discrete values and forward speed has 2 discrete values making 6 total possible unique values for the 2-dimensional variable that represents movement. Note the resemblance to a biological GRN where inductive communication (Section 4.2.1) is simple and selective ("slow" or "fast" and so on) instead of being instructive.

The framework is further enhanced by adding another Graphical User Interface (GUI) window designed in OpenGL[2]. This presents a graphical representation of the PPAM and allows the user to interact with it using the mouse pointer and the keyboard. Individual nodes can be selected and *broken* in order to simulate stuck-at faults or broken connections and so on.

5.2.1 Controller Topology

Figure 5.1 illustrates the topology for the controller (BAM or PPAM) in an agent. The

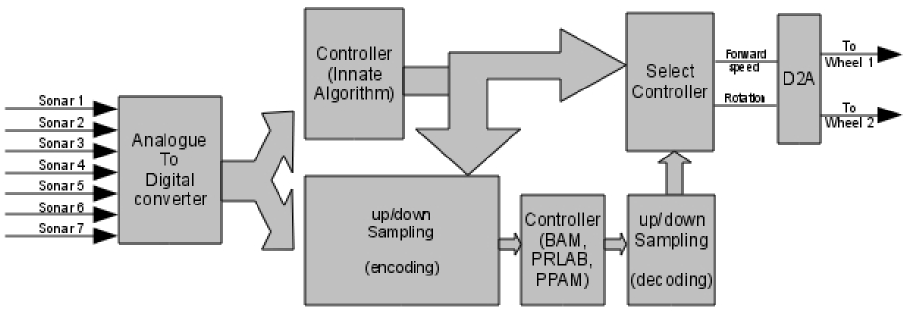

Sonar 1
Sonar 2
Sonar 3
Sonar 4
Sonar 5
Sonar 6
Sonar 7
Analogue To Digital converter
Controller (Innate Algorithm)
up/down Sampling (encoding)
Controller (BAM, PRLAB, PPAM)
up/down Sampling (decoding)
Select Controller
Forward speed
Rotation
D2A
To Wheel 1
To Wheel 2

Figure 5.1: Controller's connection topology in the agent

sonar values are fed to an ADC which outputs digitally encoded values. As stated above,

[2]http://www.opengl.org

the ADC quantises the sonar values into 10 uniform, discrete bins. Thus the output of the ADC is a 4 bit binary value for each sonar, which is directly used by the innate controller (Algorithm 2). Depending on the size of the network forming the associative memory (number of nodes in the BAM or PPAM), the discretized values may need to be up or down sampled. In a BAM with N nodes in a layer, the input vector V for that layer will be $V = I_1, I_2, \ldots I_N$. Assuming a bipolar implementation, I_j may be described as $I_j \in \{-1, +1\}$. The convention used here to describe the (BAM or PPAM) network is Ax + By where the network will have A nodes in the first (movement) layer and B nodes in the second (sonar) layer. For instance, a 3x+7y BAM network has 3 nodes in the movement layer and 7 nodes in the sonar layer, and so $N = 7$ for the sonar layer. This means that the 4-bit output of the ADC needs to be down-sampled to a 1-bit value, since there are 7 sonar values and 7 nodes. For the 3x+28y network, there are 28 nodes in the sonar layer, this means that each sonar value can be represented using 4 nodes ($\frac{28}{7}$), thus re-sampling is not required. Similarly, for the 12x+63y network, an up-sampling is required. The advantage of using this up/down sampling is that it realistically models quantisation errors. This is due to the fact that from the perspective of the associative memory, the movement values generated by the innate object avoidance algorithm (Algorithm 2) are also a source of (analogue) external inputs. Other than this re-sampling (and conversion to bipolar for the case of BAM/PRLAB) no other encoding is performed on the input data. At the output, a corresponding up/down sampling process is performed if one was required at the input. Next, the movement values from either the innate algorithm or the recalled movement values from the associative memory are forwarded on to the actuators in the agent. The choice can be made online and is under user-control. The selected movement values are then converted to analogue values and transmitted to the relevant motors.

Although non-binary nodes are also possible in a PPAM, this implementation uses binary nodes in all the following experiments in this chapter. This not only simplifies the implementation, it also reduces the number of connections between nodes, since binary outputs only require 1-bit connections, which is particularly useful from a hardware perspective. PPAM experiments were performed with networks of column arrays in configurations of:

- 3x+14y,
- 3x+35y,

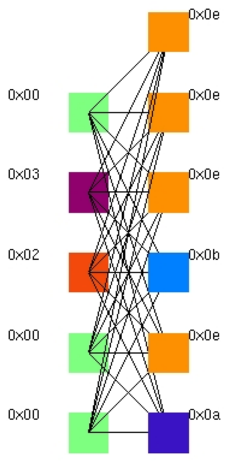


Figure 5.2: Network of column arrays in PPAM

- 3x+70y and
- 3x+60y

The numbers were chosen to be multiples of the movement values and the sonar values, the reasons for which becomes apparent in Section 5.3. An example topology for a column array network is illustrated in Figure 5.2. Each node in one column array receives input from one category of external inputs and is connected to all nodes of the other column array. BAM experiments were performed in configurations of

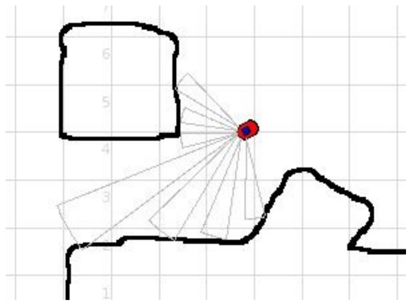
- 3x+7y,
- 3x+28y,
- 6x+28y,
- 12x+28y and
- 12x+63y.

These configurations were chosen to match the PPAM configurations, as explained below.

The 3x+14y PPAM node configuration is equivalent to the 3x+7y BAM configuration because of the following. The 14 (PPAM) nodes may be divided into two sets of 7 each, each of which receive the same seven sonar values so that two nodes (one from each set) receive values from one sonar. The nodes are configured so that one and only one of these two nodes will *fire* (generate an output). Since the two nodes are mutually exclusive, two nodes represent one binary value. Therefore, 14 nodes can distinguish between 2^7 binary values – which is the same as for the 3x+7y BAM configuration. Similarly, the 70 nodes in the 3x+70y PPAM network can be divided into 10 sets of 7 each, with each set receiving the same 7 sonar values. The 10 nodes in each set being mutually exclusive, each set represents one decimal value, thus the 70 nodes can distinguish between 10^7 values which is the same as the 3x+28y BAM configuration. One consideration for simulating BAM using the 6x+28y, 12x+28y and 12x+63y node configurations was to test whether this had any effect on performance, since, like the corresponding PPAM implementation, these configurations also use more bits than is necessary to represent sonar values and movement values.

5.2.2 The Innate Object Avoidance Algorithm

The innate object avoidance algorithm is a simple random walk and was described as Algorithm 2. It should be noted that the object-avoidance algorithm does not have memory. Thus, quantisation errors can cause the agent to get stuck in race conditions where it first turns left, then right, then left and so on. Some of these race conditions are depicted in Figure 5.3. If the "slow" forward speed (s_s in Algorithm 2) is set to a


(a) Passage

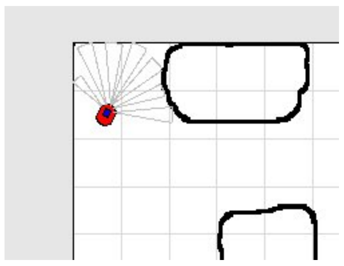

(b) Corner

Figure 5.3: Race conditions for the object avoidance algorithm in Player/Stage

non-zero value, this would result in the agent continuing to move forward while turning, which in turn would result in the agent being able to navigate out of such race conditions without any effort by the object-avoidance algorithm. This is undesirable for two reasons. Firstly, one motivation of using such a simple object avoidance algorithm is to test (in the future) if the associative memory can outperform the innate algorithm, given higher level "goals" as input in associative pairs. Secondly, it would make it harder to detect if a race condition occurs because of the associative memory not storing correct relationships or because of imperfections in the training algorithm.

The trajectory of the agent while being controlled by the object-avoidance algorithm is shown in Figure 5.4 (Objects are shown in black). The end point of the trajectory is

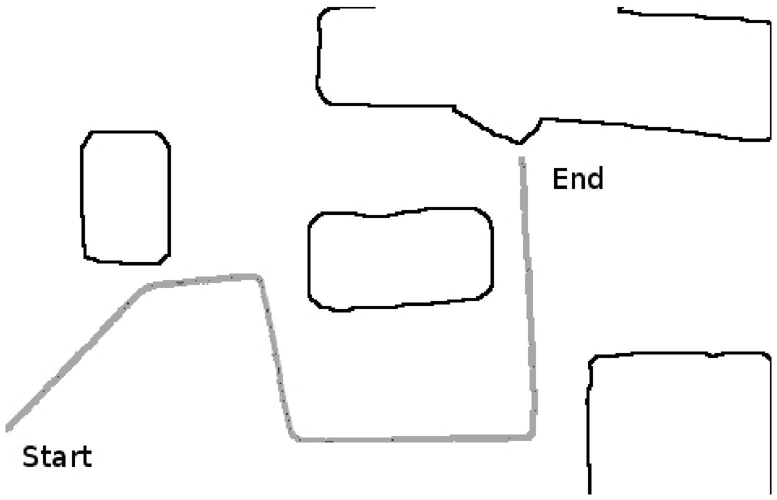


Figure 5.4: Agent trajectory using Algorithm 2

where the algorithm gets stuck in a race condition. This occurs after 670 time steps where, in each time step the agent has observed one 7-D sonar value and the algorithm has generated one 2-D movement value, together making one associative pair. These 670 pairs are the training dataset. Note, firstly, that the training dataset is not pruned to select *key* points but instead includes all observed values. Secondly, the dataset has a one-to-many relationship between the pairs (movement versus sonar); for example, many different sonar readings can result in the agent continuing to move straight ahead. This places the BAM at a disadvantage, since dealing with such relationships is a known weakness. Furthermore, the dataset is not guaranteed or indeed even likely to be composed of orthogonal vectors. However, it is proposed that real-time data is very unlikely to be orthogonal and realistic associative memories cannot depend upon *a priori* knowledge for preprocessing this training data to convert it into orthogonal pairs (as described in Section 2.1.3.4).

5.3 Associativity and Extracting Relationships Results

The following sections present the results of the test to determine if the PPAM is capable of storing associations. The hypothesis and measures were described in more concise terms in Section 5.1, but are reiterated here in more general terms as a reminder.

The general idea is that after running for a predetermined amount of time on the *innate* object-avoidance algorithm, if the agent is now reset back to its initial position and orientation, it will deterministically observe the same sensor values as it did before. Thus, the agent should now be able to switch to the *recalled* object-avoidance values seamlessly which means that there should be no difference in the paths followed if the dataset is stored correctly.

Furthermore, in order to test if the PPAM can extract any meaningful relationship information, the agent is subjected to new and previously unobserved environments. If the agent is now able to navigate the arena successfully (i.e. without collisions), then it may be inferred that it has extracted some *representative* (or meaningful) information about the relationship. Also, the values recalled by the PPAM can be compared with the values from the innate object-avoidance algorithm.

5.3.1 3x+14y

14 nodes get input from the sonars, such that 2 nodes get values from the same sonar, thereby down-sampling the sonar values to 2 discrete values. The total number of possible sonar values that can be distinguished are 2^7 or 128 data values out of 10^7. The trajectory taken by the robot is shown in Figure 5.5 where the light gray trajectory is the one taken by the agent while it was being controlled by the object-avoidance algorithm and the dark-grey trajectory is the one taken by the agent when being controlled by the 3x+14y PPAM. The robot starts out following the path, but very soon gets stuck in a race condition which is not predicted by the object avoidance algorithm. As can be seen in Figure 5.6, the deviation from the path increases over time (y-axis shows deviation in meters while x-axis shows time steps).

Figure 5.7 shows a break-down of the environmental inputs in terms of percentage of the environment observed before. Therefore the 100% *data region* (Figure 5.7) contains those data points where all 7 of the sonar inputs were previously observed. This would occur as the robot follows the path after it is reset to the initial start location and heading. Partially observed sonar inputs would occur after the robot is placed in a new environment. The 85.7% data region contains data points where only 6 of the 7 sonar values have been

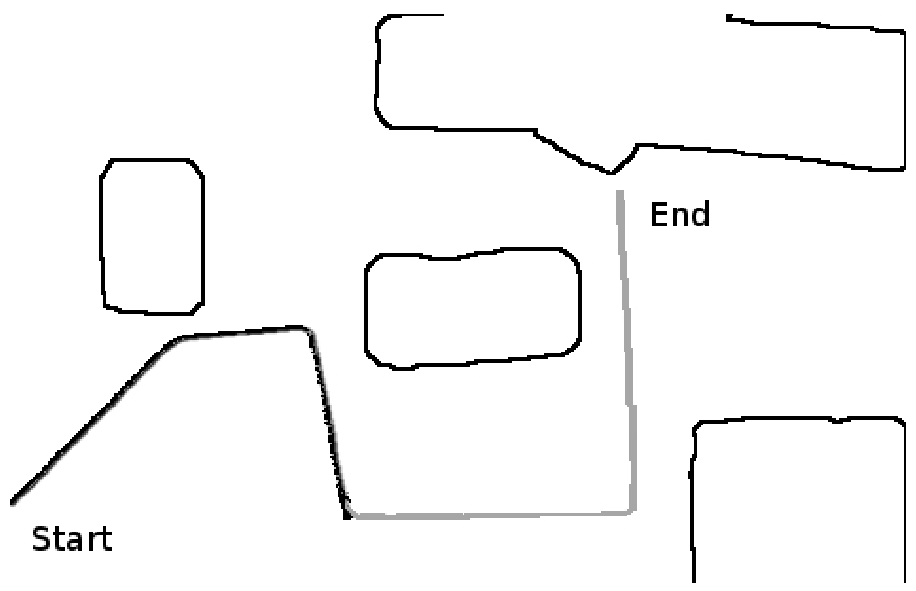


Figure 5.5: Agent trajectory using 3x+14y PPAM

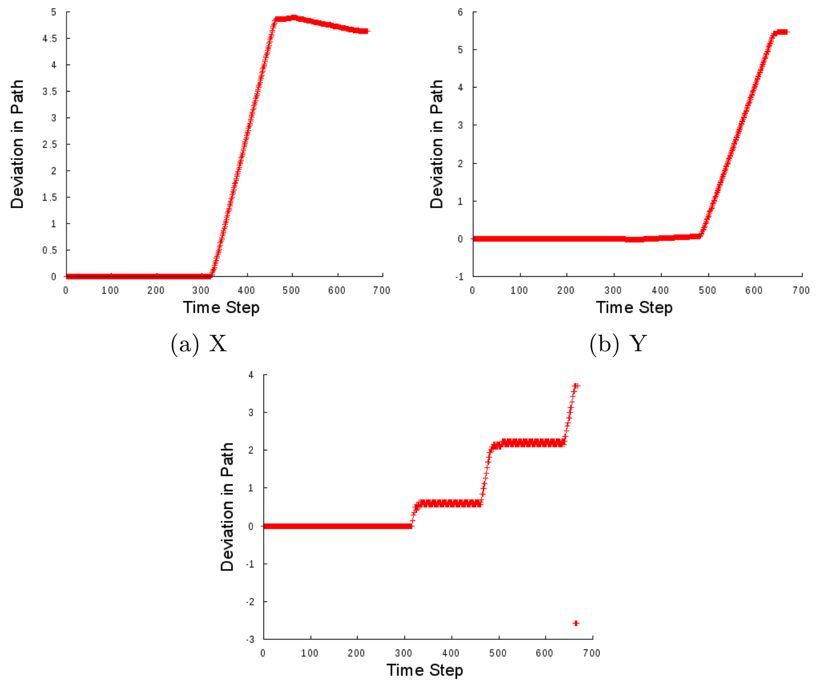
observed before. Although the remaining sonar input actually has a value (as a bus in hardware will always have *some* value), from the perspective of the architecture, this is equivalent to the input being absent. This represents the hetero-associative recall where an extrapolation of relationships is performed to determine the output.

Table 5.1 measures the associativity of the PPAM. "Observed" corresponds to the data

Table 5.1: Associativity in 3x+14y PPAM

Observed	100%	85.7%	71.4%	57.1%	42.8%	28.6%
Recall (confidence)	100%	100%	100%	100%	100%	100%
Accuracy	74.8%	97.0%	99.0%	75.1%	100%	100%
Critical (decisions that avoid collision)	57.7%	12.0%	47.0%	44.2%	12.6%	0.0%
C+A (critical and correct)	56.4%	75.0%	98.0%	43.6%	100%	N/A

regions in the bar chart in Figure 5.7 and indicates the number of sonar values (as a percentage) that had been observed before during training. As explained above, the data regions are the number of inputs that were observed before (during training) and the height of the bar chart in Figure 5.7 indicates the number of data points present in each data region. "Recall" is a measure of the confidence with which the PPAM recalls values and is formally defined in Section 5.1. "Accuracy" indicates the number of memory recalls which matched the outputs of the hardwired algorithm. "Critical" lists the percentage of memory recalls which were critical to avoiding a collision and "C+A" is the percentage of memory recalls which were critical and also correct.



(a) X

(b) Y

(c) θ

Figure 5.6: Error in path following in 3x+14y network

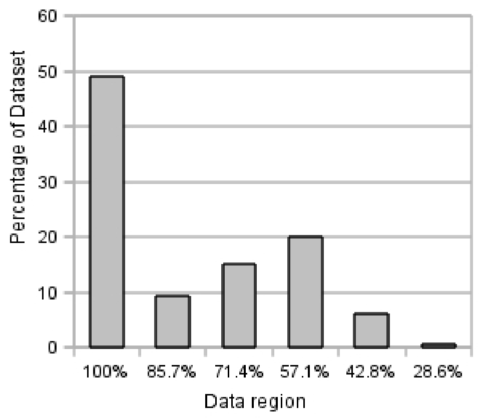
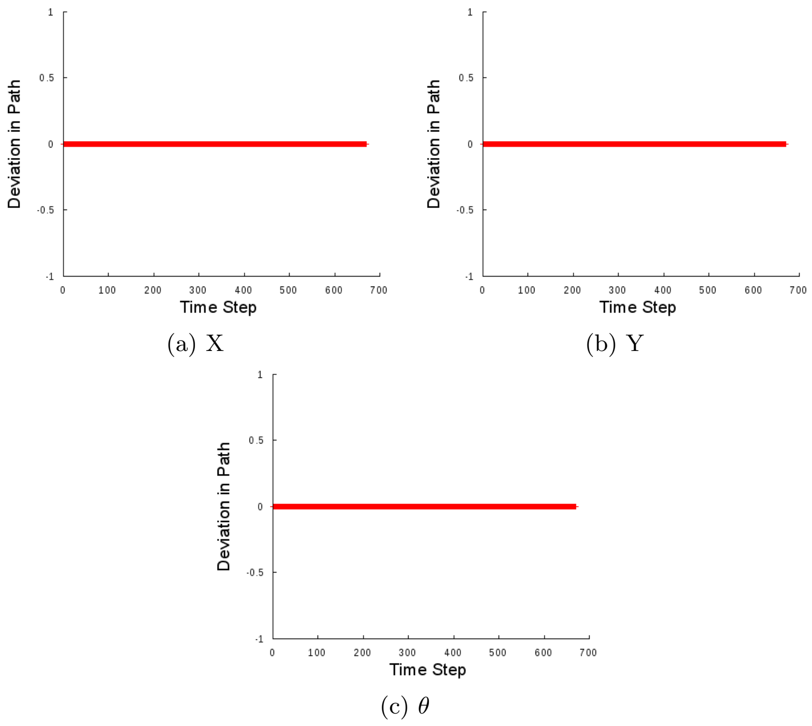


Figure 5.7: Environment input by percentage previously observed for 3x+14y

The first thing of note is that *all* memory recalls performed were considered as accurate memory recalls (100 percent confidence) by the PPAM – even the ones where only 28.6% of the environment had been previously observed. Furthermore, only 74.8% of the recalls were correct from amongst the 49% data points where the environment had been completely observed before (100% column). This indicates that 14 nodes are insufficient to learn the association correctly – a conclusion that is further reinforced by the deviation measure shown in Figure 5.6. The 85.7% observed column shows a better performance with 97% correct actions, however, it forms only 9% of the total as shown in Figure 5.7 and therefore is not a reliable result.

5.3.2 3x+28y

28 nodes get input from the sonars, such that 4 nodes get values from the same sonar, thereby down-sampling the sonar values to 4 discrete values. The total number of sonar values that can be distinguished is 4^7 or 16K data values out of 10^7. As can be seen by the results in Figure 5.8, the robot successfully followed the path, passing through exactly the same coordinates, since the deviation in all 3 dimensions is 0.



(a) X (b) Y (c) θ

Figure 5.8: Error in path following in 3x+28y Network

Figure 5.9 shows a break-down of the environmental inputs in terms of the environment having been observed before. Table 5.2 measures the associativity of the PPAM. “R+C”

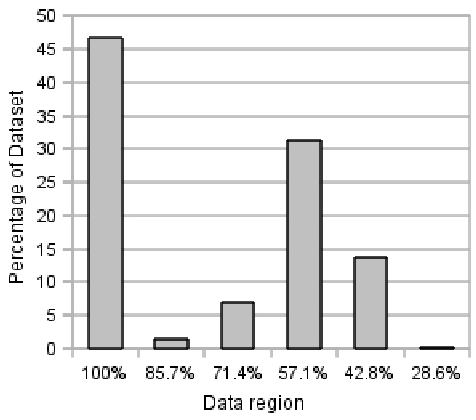


Figure 5.9: Environment input by percentage previously observed for 3x+28y

Table 5.2: Associativity in 3x+28y PPAM

Observed	100%	85.7%	71.4%	57.1%	42.8%	28.6%
Recall (confidence)	100%	30.0%	22.4%	11.4%	24.5%	0.0%
Accuracy	100%	85.0%	65.3%	56.2%	87.7%	50%
Critical (decisions that avoid collision)	15.4%	50.0%	55.1%	52.0%	8.7%	0.0%
C+A (critical and correct)	100%	80.0%	50.0%	50.6%	29.4%	N/A
R+C (critical decisions claimed as correct)	15.4%	83.3%	13.6%	7.8%	2.1%	N/A
R+A (correct when claimed as correct)	100%	100%	100%	100%	100%	N/A

measures the number of memory recalls that were recalled with 100% confidence by the PPAM and were critical actions. “R+A” measures the percentage of memory recalls (both critical and non-critical) which were recalled with 100% confidence by the PPAM and were also accurate in what they recalled. Therefore, these are events where the PPAM recalled with complete confidence that the correct action was, for instance, to “turn left” and this was the correct output as compared with the hardwired algorithm.

Unlike the previous experiment, all memory recalls are not considered as accurate, except for the case when the environment inputs were fully observed (100% column). For the case where 85% of the environment had been previously observed, only 30% of the memory recalls are indicated as being accurate. Environments that had been observed fully before were *always* recalled accurately (100% actions correct). Furthermore, the “R+A” row indicates that any memory recall that was considered by the PPAM as accurate did have the correct outputs. The biggest region of data with incomplete environment information is the 57.1% region (Figure 5.9). Therefore it is the best representative for hetero-associative recall of previously unobserved environments to test whether the PPAM has extracted and stored meaningful and representative information about the dataset. Considering that only 50% of the environment has been seen before, the fact that the PPAM can still recall some values (11%) with 100% confidence, is a respectable result. In addition, 56% of the total recall operations returned the correct outputs. This indicates that 28 nodes are a significant improvement for learning the association – a conclusion that is further reinforced by the deviation measure shown in Figure 5.8. However, the accuracy of hetero-associative recall could do with some improvement.

5.3.3 3x+35y / 3x+70y

35 nodes get input from the sonars, such that 5 nodes get values from the same sonar, thereby down-sampling the sonar values to 5 discrete values. The total number of possible

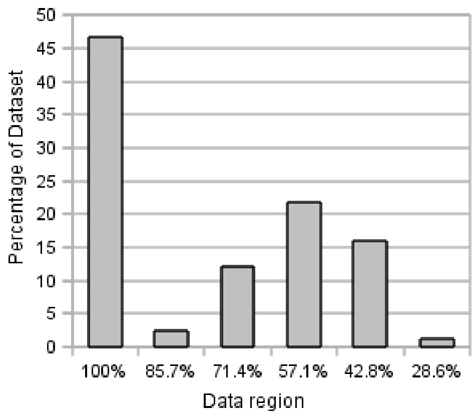


Figure 5.10: Environment input by percentage previously observed for 3x+35y

sonar values that can be distinguished is 5^7 or 78,125 data values out of 10^7. The path deviation measure was zero – the same as for the 3x+28y experiment. Figure 5.10 shows a break-down of the environmental inputs in terms of the environment having been previously observed for the 3x+35y.

Table 5.3 measures the associativity for the 3x+35y PPAM. The results are similar to those for the 3x+28y PPAM array except the memory is more reliable as the number

Table 5.3: Associativity in 3x+35y PPAM

Percentage Previously Observed	100%	85.7%	71.4%	57.1%	42.8%	28.6%
Recall (confidence)	100%	58.3%	24.8%	18.6%	15.8%	68.7%
Accuracy	100%	97.2%	80.9%	77.5%	83.8%	68.7%
Critical (decisions that avoid collision)	15.4%	0.0%	29.5%	13.2%	12.7%	0.0%
C+A (critical and correct)	100%	N/A	54.9%	56.1%	58.6%	N/A
R+C (critical decisions claimed as correct)	15.4%	N/A	2.3%	3.4%	5.5%	N/A
R+A (correct when claimed as correct)	100%	100%	100%	100%	100%	100%

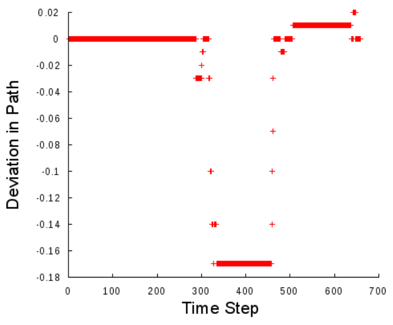
of accurately recalled data is higher. The 57.1% column is still the largest region that performs hetero-associative recall and this also indicates a higher percentage of correct actions. Therefore 35 nodes are an improvement over the 3x+28y node configuration. Results for the 3x+70y nodes configuration indicate a continuation of the trend displayed by increasing the number of nodes.

5.3.4 3x+60y

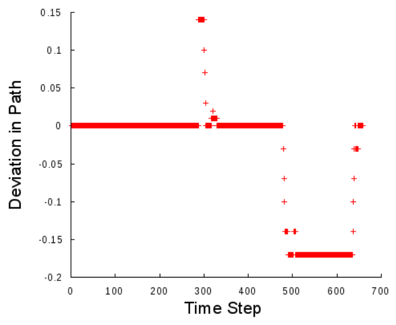
In the 3x+60y network, there were 3 nodes for the movement and 60 nodes for the input from the sonars. Unlike the 3x+14y or the 3x+35y experiment, the sonar values are not down-sampled so that each node only fires for a particular value from amongst the 10 bins.

The 3x+60y PPAM node configuration was simulated to test the case where a 3x+70y node configuration network has 10 nodes broken. Note that this is the same as the case where a 3x+70y network is trained and then 10 nodes stop firing or the 10 nodes get stuck at 0 (where 0 represents not firing in this encoding). This is because during the lookup process in each node, a non-firing node is the same as a node firing with a value not seen before, which would not be found in the CAM. The deviation between the robot's path using the innate algorithm and the path using PPAM is shown in Figure 5.11, where the y-axis is the deviation and the x-axis is the discrete time steps. As can be seen, the deviation is quite small – so small in fact that the difference between the two paths cannot be detected from observation with the naked eye. Table 5.4 measures the associativity of the PPAM. The important thing to note is that, despite the deviation, the associativity is very similar to the 3x+35y configuration.

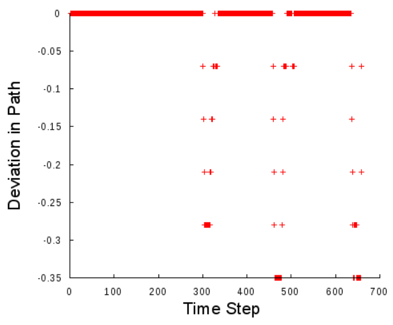
The results presented in this section (Section 5.3), indicate that the PPAM is not only



(a) X



(b) Y



(c) θ

Figure 5.11: Error in path following in 3x+60y Network

Table 5.4: Associativity in 3x+60y PPAM

Percentage Previously Observed	100%	85.7%	71.4%	57.1%	42.8%	28.6%
Recall (confidence)	100%	58.3%	24.8%	18.7%	15.7%	68.7%
Accuracy	100%	97.2%	80.9%	77.4%	83.0%	68.7%
Critical (decisions that avoid collision)	15.5%	0.0%	29.5%	13.2%	14.0%	0.0%
C+A (critical and correct)	100%	N/A	54.9%	56.1%	58.2%	N/A
R+C (critical decisions claimed as correct)	15.5%	N/A	2.3%	3.4%	5.6%	N/A
R+A (correct when claimed as correct)	100%	100%	100%	100%	100%	100%

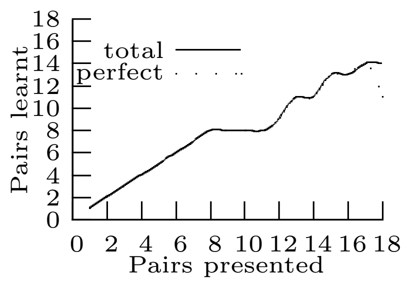
capable of storing the relationships between the sonar values and the movement values, but also that the PPAM can extract meaningful information about the underlying dataset from these relationships. Although the 3x+14y PPAM network was insufficient to duplicate the behaviour of the innate object-avoidance algorithm, it nonetheless, did not result in any collisions; and PPAM configurations with more nodes were successfully able to duplicate the behaviour of the innate algorithm. A measure of confidence was introduced and this was deemed as crucial to evaluating the performance of an associative memory, since it can be argued to be more important to know how likely is the output to be correct, than to simply generate the output. Some initial experiments were performed to evaluate the robustness of the memory (Section 5.3.4) which were quite encouraging. The following sections test the final hypothesis listed in this chapter (Section 5.1.3) to compare the performance of the PPAM with other traditional techniques.

5.4 Tuning the BAM Correlation Matrix

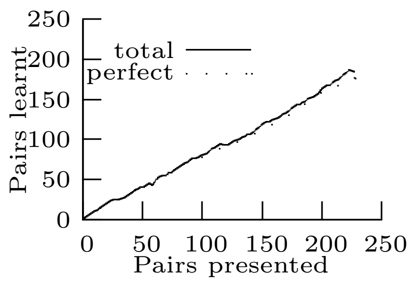
The training dataset (of 670 pairs) was composed of 228 unique associative pairs. Each pair can be represented using a 3 bit value for movement and a 28 bit value for sonar (4 bit value for each of 7 sonars). The perfect associative memory would be able to learn all 228 associative pairs and recall them perfectly so that if the agent is re-initialized at the start coordinates (in Figure 5.4), it would follow the trajectory exactly and get stuck in the race condition at the end. In order to get the best performance from the BAM, parameters for both the original Kosko's learning algorithm and also PRLAB were tuned. Whether *any* BAM correlation matrix using *any* BAM configuration is capable of learning all 228 pairs is dependent upon whether the 228 pairs are orthogonal. The rest of this section presents the attempts to maximize the number of associative pairs that are learnt by the BAM by tuning the parameters. Once the correlation matrix that can remember the maximum number of pairs is found, this is used to control the agent in the player/stage environment (Section 5.5). These player/stage simulations explore the effect of imperfect recall in BAMs and compare them with imperfect recall in the PPAM.

5.4.1 Kosko's Algorithm

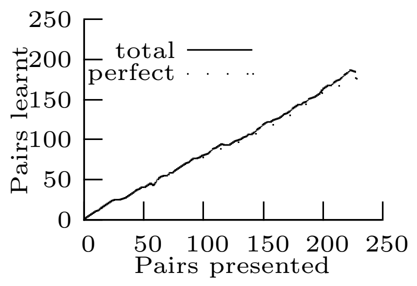
For the original BAM learning algorithm, as presented by Kosko (1988), the parameters that need tuning are the initial values of the weight matrix and the number of nodes in the 2 layers of a BAM. Node configurations of 3x+7y, 3x+28y, 6x+28y, 12x+28y, 12x+63y and 30x+28y were attempted, where 3x+7y means 3 nodes in layer 1 and 7 nodes in layer 2. The numbers were chosen to be multiples of the movement values and the sonar values. Given that the BAM operates on bipolar data, 7 nodes in a layer means that the layer can distinguish between 2^7 or 128 bipolar values. Therefore, the 3x+7y configuration represents a case where sonar values have to be down-sampled. The 3x+28y configuration is sufficient to represent the entire range of values. Higher node configurations are used to test whether up-sampling has any effect on the number of pairs learnt, as the BAM would train better if the associative pairs are farther apart. Where necessary, the training dataset was re-encoded to up-sample or down-sample values to give the BAM maximum advantage. For example, the 3x+7y configuration requires the 4 bit sonar values for each sonar to be down-sampled to 1 bit. In this case, the training dataset turned into 18 unique associative pairs (for all other configurations it remained 228). For higher node configurations (6x+28y, 12x+28y and so on), although re-encoding is not essential, it is preferable since without it data points become closer together. To illustrate, when using 3 nodes, the range of data is 2^3 or 8 values and therefore the maximum distance between any two numbers is 6 integer values. Thus, for instance, 4 is separated from 1 by 3 integer values out of a possible of 8. Alternately, when using 6 nodes, the range of data is 2^6 or 64 values. If the same dataset (as used for 3 nodes) is used without re-encoding (or up-sampling) then in this case, 4 would be separated from 1 by 3 integer values out of a total possible of 64.



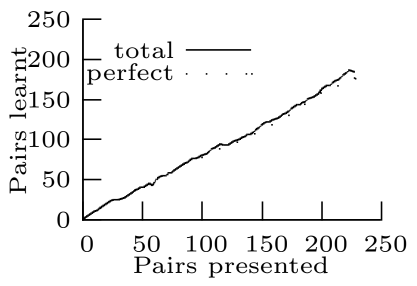
(a) 3x+7y



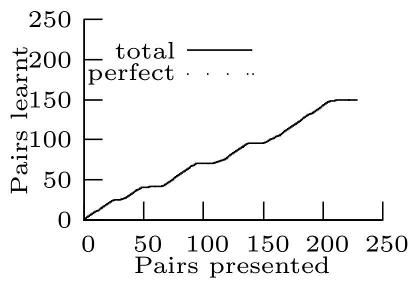
(b) 3x+28y



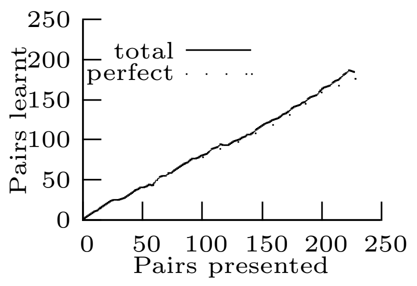
(c) 6x+28y



(d) 12x+28y



(e) 12x+63y



(f) 30x+28y

Figure 5.12: Memory capacity using Kosko learning

For each node configuration, initial weight values were set between −1.0 to +1.0 in steps of 0.02. The 3x+28y node configuration was further tested in an extra long range with weight values in the range of -100.0 to $+100.0$ in steps of 0.02, in order to protect against local minima. Whereas the maximum number of pairs learnt (including almost-recalled pairs) in the smaller range was 186 out of 228, in the larger range, the maximum number of pairs learnt was 187 out of 228, thereby indicating that the effect of initial weights on the correlation matrix is not significant. For each initial weight value, the BAM was trained with an increasing number of associative pairs and the number of pairs successfully recalled was tested in each iteration. As described in Section 5.1, a value is considered to be *perfectly recalled* if the value recalled is exactly the same as was trained. Considering all values trained as attractors, a recalled value is *almost-recalled* if the closest attractor is the correct one. Euclidean distance is chosen as a measure for *closeness*, because of the need for orthogonality in BAMs. Note that detecting almost-recalled values implies post-processing

Table 5.5: Long range search parameters for PRLAB

Parameter	Min	Max	Step
Weight-Threshold	-0.5	$+0.5$	0.1
Epochs	5	70	5
λ	0.5	2.3	0.2
ξ	0.05	0.2	0.015

of the output and *a priori* knowledge of the attractors to determine which attractor is closest. This means that the memory must have another copy of all the associative pairs stored and test the recalled value against all the associative pairs to find the closest one – in essence, duplicating its own behaviour. Despite being unrealistic (particularly for online incremental learning), this method was used to maximise performance. The objective of this tuning is to determine the BAM correlation matrix that can recall the largest number of pairs in the training dataset. This can then be used to control an agent in the player/stage environment. If a correlation matrix can be found that can recall *all* the pairs, it would be able to follow the trajectory of the object-avoidance algorithm exactly.

Table 5.6: Short range search parameters for PRLAB

Parameter	Step	Iterations
Weight-Threshold	0.01	10
Epochs	1	10
λ	0.1	4
ξ	0.002	5

As mentioned above, for each initial weight configuration, an increasing number of associative pairs were presented for training. Figure 5.12 shows how the capacity of the BAM varies as the number of associative pairs in the training dataset is increased. This is an important step for this learning algorithm because, unlike PRLAB, it does not guarantee utilization of maximum storage capacity. Therefore, as can be seen from the plots, if too many pairs are presented, the capacity of the memory begins to fall. Each point in the plots in Figure 5.12 is the one with the *best* initial weight value where the primary fitness criterion is to maximize the number of pairs recalled and the secondary fitness criterion is to maximize the number of pairs recalled perfectly. Note that up to size 8 for configuration 3x+7y and up to about size 25 for other configurations the BAM can successfully recall *all* training pairs presented. However, any pairs presented beyond this point, are only learnt at the expense of some other pair. The maximum number of associative pairs learnt using any configuration of nodes and initial weights is 187.

5.4.2 PRLAB

The parameters that need tuning are the initial values of the weight-threshold matrix, number of nodes, number of epochs, relaxation factor λ and constant ξ deciding the basins of attraction. Oh and Kothari (1994) show that PRLAB is insensitive to parameter values, and indeed other implementations don't train the parameters at all (Sudo et al., 2009). However, as shown by Figure 5.13, if the dataset is not entirely orthogonal, training can result in a difference of up to 68 pairs in the worst case, which is the difference between PRLAB and Kosko's original method. The same node configurations and training dataset

Table 5.7: Extra long range search parameters

Parameter	Min	Max	Step
Weight-Threshold	-5.0	$+5.0$	0.1
Epochs	5	75	3
λ	0.1	2.4	0.1
ξ	0.01	1.0	0.005

were used as for Kosko's algorithm. Details of the training dataset can be found in Sections 5.1 and 5.2. Parameters were tuned in two steps, the first performing a long-range sweep (with larger steps) and the second performing a short-range sweep (with smaller steps) close to the best region found in the long-range sweep. Table 5.5 shows the long-range sweep parameters which result in a total of $10,000$ iterations, while Table 5.6 shows the short range sweep parameters. The same method for perfectly recalled and almost-recalled (as used for tuning Kosko's algorithm) is used here to tune PRLAB. The 6x+28y node

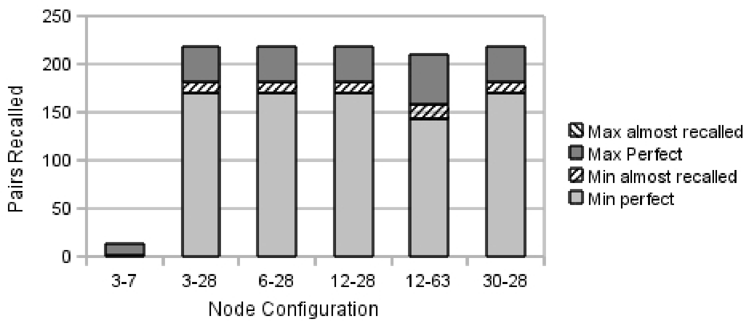


Figure 5.13: Capacity of BAM using PRLAB

configuration was further tested in an extra long range as shown in Table 5.7. Whereas the maximum number of pairs learnt (including almost-recalled pairs) using the long-range/short-range search was 219, using the extra long range search, it was 220. Figure 5.13 is a bar chart showing the capacity of the BAM trained using PRLAB. The *best* parameter values (Table 5.8) are used where the fitness criterion maximizes the number of pairs recalled while maximizing the number of pairs recalled perfectly and minimizing the number of epochs required to learn the associative pairs. The hatched portion indicates

Table 5.8: Best parameters for PRLAB

Network	Epochs	ξ	λ	Weight
3x+7y	11	0.05	0.5	−0.3
3x+28y	20	0.120	1.60	−0.34
6x+28y	35	0.9505	1.54	0.9
12x+28y	23	0.125	1.58	−0.3
12x+63y	65	0.2	0.5	−0.1
30x+28y	23	0.125	1.58	−0.3

the number of values that were almost-recalled correctly. The absolute minimum number of pairs learnt using any combination of parameters is depicted by the light-grey bar plus the hatched bar for “Min Almost Recalled”. The maximum number of pairs learnt is the full height of each bar. Note that max-almost-recalled is 0. This means that when the correlation matrix is fully optimized (so that the maximum number of pairs are stored) all pairs stored are stored perfectly and the Euclidean distance measure (almost recalled) does not enhance performance further. Note also, that, with the exception of the 3x+7y node configuration, the worst-case PRLAB correlation matrix performed as well or better than Kosko’s original correlation matrix. Furthermore, the trained PRLAB correlation matrices all performed approximately equally, indicating that the number of associative pairs learnt by the BAM was unaffected by the number of nodes in the layers and that the

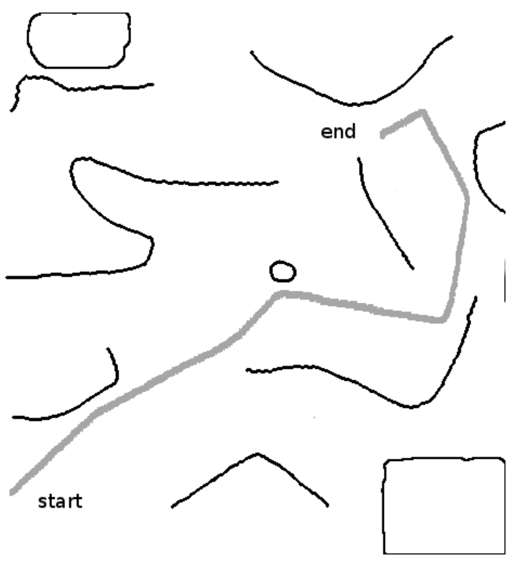


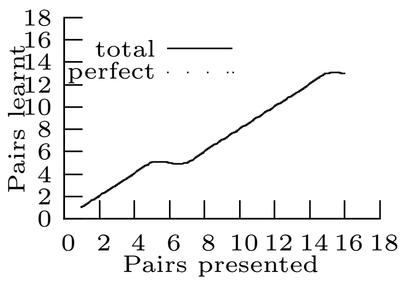
Figure 5.14: Trajectory in alternate environment

maximum number of (orthogonal) pairs that could be learnt had been reached. More pairs can only be learnt by increasing the orthogonality of the pairs. Thus, the dataset would have to be re-encoded. Disregarding the implications of this on the structure of the BAM, and assuming that it is possible online in real-time, no encoding can ensure orthogonality for all real-world datasets – and therefore at some point, the associative memory would have to deal with imperfect recall. The objective here is to explore the behaviour of the BAM and compare it with that of the PPAM in this situation where data is imperfectly stored and recalled as described in Section 5.5.

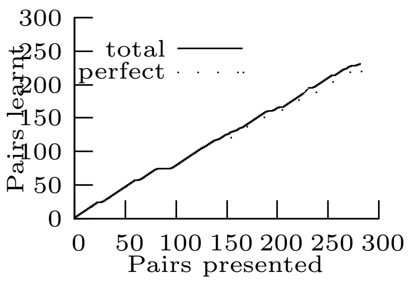
5.4.3 Verifying with alternate dataset

An alternate dataset was also generated and used as a training dataset in the parameter tuning process explained above. The object was to verify that the results observed are not due to some property of this *one* dataset. For this purpose, the agent was placed in a completely different environment. Figure 5.14 shows the modified environment as well as the trajectory taken by the agent. This trajectory was composed of 770 time steps before the agent got stuck in a race condition at the “end” position. These 770 time steps were composed of 283 unique training data points, which formed the alternate dataset. Tuning for both Kosko’s method and PRLAB was performed in the same manner as outlined above. Results from some of the network configurations are shown in Figure 5.15.

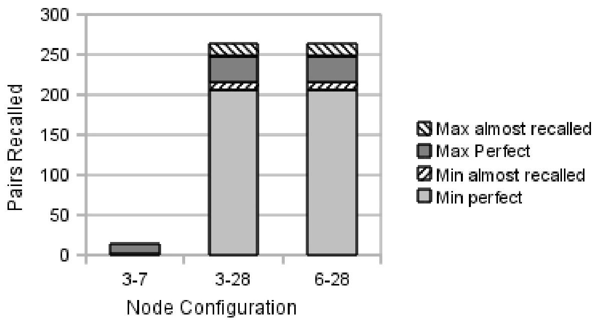
Comparing Figure 5.12 with Figures 5.15a and 5.15b, it can be seen that the results are



(a) 3x+7y using Kosko



(b) 6x+28y using Kosko



(c) PRLAB

Figure 5.15: Alternate dataset memory capacity

similar. For the 3x+7y node configuration, the original dataset had 18 elements while the alternate dataset has 16 elements. In the case of the alternate dataset limited to 16 elements, the maximum number of tuples stored and recalled correctly was 13, which is the same for the alternate dataset. Also, all 13 of these elements were *perfectly recalled* and not *almost recalled* for both datasets. For the 6x+28y node configuration, the original dataset had 228 elements while the alternate dataset has 283 elements. In the original dataset, the maximum number of tuples stored and recalled correctly was 178 (when the dataset was limited to 226 elements), while in the alternate dataset, the maximum number was 221 (when the dataset was limited to 275 elements). In the case when the alternate dataset was limited to 228 elements (the same number as the original dataset) it correctly stored and recalled 180 elements (where 170 were perfectly recalled) while the original dataset correctly stored and recalled 176 elements (where 168 were perfectly recalled). The original dataset correctly stored and recalled 77.19% of the total dataset (176 out of 228), while the alternate dataset correctly stored and recalled 77.74% of the total dataset (220 out of 283).

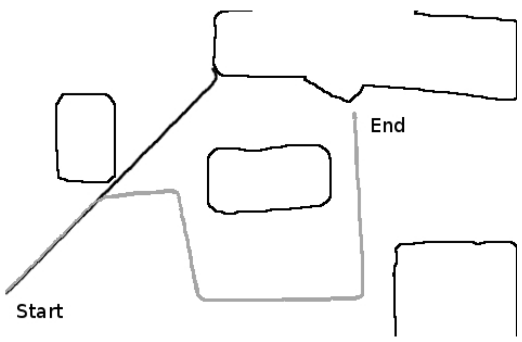
Similarly, comparing Figure 5.13 with Figure 5.15c, for the 3x+7y node configuration using PRLAB, the original dataset accurately stored and recalled 13 out of 16 elements, while for the alternate dataset, it stored 14 out of 18 elements. For both the 3x+28y and 6x+28y node configurations, the original dataset was able to recall 219 out of 228 elements (96.05%) while the alternate dataset was able to recall 266 out of 283 elements (93.99%) elements. These results corroborate the findings from the experiments on the original dataset and indicate that the conclusions drawn were not caused by a feature of the original dataset itself.

5.5 Robot Simulation Results

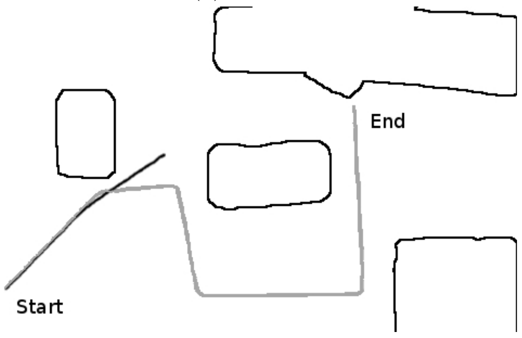
The objective of the previous experiments was to determine the *best* correlation matrix for the BAM. If a correlation matrix had been found that could recall all 228 associative pairs in the training data, then resetting the agent to the initial starting location would have been a trivial experiment, since the agent would have followed the path of the object-avoidance algorithm exactly. In that case, it would only remain to observe the behaviour of the agent in new, previously unobserved environments. However, since no such correlation matrix (that could recall all 228 pairs) was found, experiments in both environments are useful. Although it has already been determined that the maximum number of pairs that can be recalled using *any* BAM configuration is 220 out of 228, the objective of the following experiments is to determine how this effects an agent in a real-world, real-time environment.

In the first phase, the robot in player/stage is reset to the initial starting position and orientation and allowed to run for another 670 time steps. This time, however, the movements are *recalled* from memory and not *calculated*. If the associations have been learnt correctly, the robot should follow the exact same path as it took the first time and at the end of 670 steps, it would be at the "end" position in the correct orientation. The second phase begins after these 670 steps when the robot is placed in new environments. The sonar values encountered now would only have been (at best) partially observed before and therefore the associative memory receives inputs that have been only partially learnt. This is the same as observing a noisy training vector. The robot is placed in three different locations and run for a total of 763 more time steps altogether.

Each of the network configurations described above (3x+7y, 3x+28y, 6x+28y, 12x+28y, 12x+63y and 30x+28y) was used to control the agent in the player stage simulation.



(a) 3x+7y



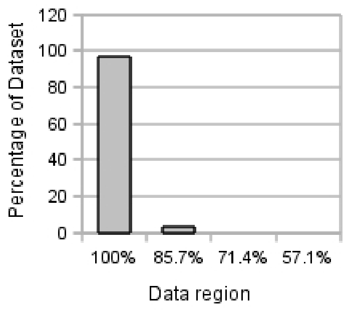
(b) 3x+28y, 6x+28y, 12x+28y

Figure 5.16: Agent trajectory for Kosko's method

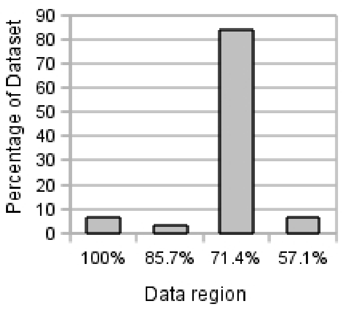
The best trained correlation matrix (Section 5.4.2) was used and the method to detect almost-recalled values was also implemented. Note that for the case of PRLAB, using a trained matrix is not incremental learning. Therefore, as the agent moves through the environment and learns associations, it needs to store another copy of the entire training dataset separately from its associative memory so that this copy can be presented to the associative memory when the weights are to be updated! Nonetheless, this is necessary because implementing an incremental version, as done by Sudo et al. (2009), loses the guaranteed learning capability of PRLAB.

5.5.1 Kosko's Method

The trajectory taken by the agent when it was re-initialized to the original start point is shown in Figure 5.16. The trajectory in light-grey is the one taken by the agent while it was being controlled by the object-avoidance algorithm, while the dark-grey trajectory is the one taken by the agent when being controlled by the BAM. As can be seen from Figure 5.16a, the agent, in fact, had a collision with the wall and stalled. Increasing the nodes from 3x+7y to 3x+28y improved the performance so that the agent now successfully avoided collisions, however, it suffered from race conditions (that the innate object-avoidance



(a) 3x+7y



(b) 3x+28y, 6x+28y, 12x+28y

Figure 5.17: Environment input by percentage previously observed for Kosko's algorithm

algorithm did not predict). The distance travelled by the agent is shorter because the agent stops when it encounters a race condition where the memory first decides to turn left then right and so on. Increasing the nodes further (after 3x+28y) had no effect on the trajectory. This is because the race condition would always occur at a certain set of sonar inputs as the BAM was incapable of storing the required associative pair in a stable state.

Table 5.9: Associativity in 3x+7y Kosko's network

Percentage Previously Observed	100%	85.7%
Recall (confidence)	83.8%	76.1%
Accuracy	34.0%	54.3%
Critical (decisions to avoid collision)	80.2%	50.0%
C+A (critical and correct)	17.8%	8.7%
R+C (critical decisions claimed as correct)	76.4%	34.3%
R+A (correct when claimed as correct)	23.6%	65.7%

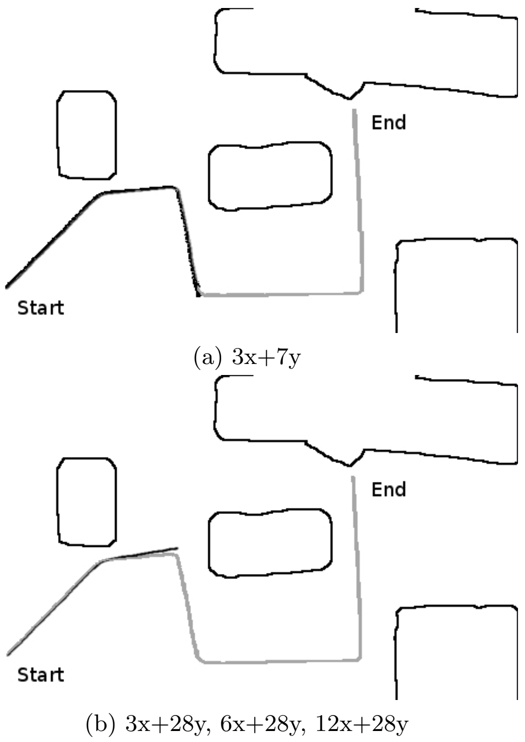
Figure 5.17 shows a break-down of the environmental inputs in terms of percentage of the environment observed before. The 100% region contains data points that would occur as the robot follows the path after it is reset to the initial start location and heading. Partially observed sonar inputs would occur after the robot is placed in a new

Table 5.10: Associativity in 3x+28y, 6x+28y and 12x+28y Kosko's network

Percentage Previously Observed	100%	85.7%	71.4%	57.1%
Recall (confidence)	100%	75.6%	45.9%	38.9%
Accuracy	98.9%	62.2%	20.7%	41.0%
Critical (decisions that avoid collision)	0.0%	37.8%	30.7%	6.3%
C+A (critical and correct)	N/A	5.9%	48.5%	100%
R+C (critical decisions claimed as correct)	N/A	20.6%	35.0%	16.2%
R+A (correct when claimed as correct)	98.9%	82.3%	42.3%	100%

environment or if the robot strays from the original path (as it does in Figure 5.16). The 85.7% region contains data points where only 6 of the 7 sonar values have been observed before. This represents the hetero-associative recall where an extrapolation of relationships is required to determine the output. Table 5.9 shows the *associativity* for the 3x+7y configuration with Kosko's learning algorithm. Data is broken down in terms of "percentage environmental inputs previously observed" so that each column presents data from its corresponding bar in the bar-chart showing a break-down of environmental inputs (Figure 5.17). Therefore, for example, the 100% column in Table 5.9 corresponds to the 100% bar in Figure 5.17a. Since there are no data points in the 71.4% region and below, these columns don't exist in Table 5.9. "Recall confidence" is a measure of the confidence with which the BAM recalls the values and is formally defined in Section 5.1. If a value is perfectly recalled, this would be 100% confidence. The distance from the attractor is used to measure the confidence for almost-recalled values.

"Accuracy" is measured as the number of memory recalls which matched the outputs of the object-avoidance algorithm. Note that according to this definition, an action may

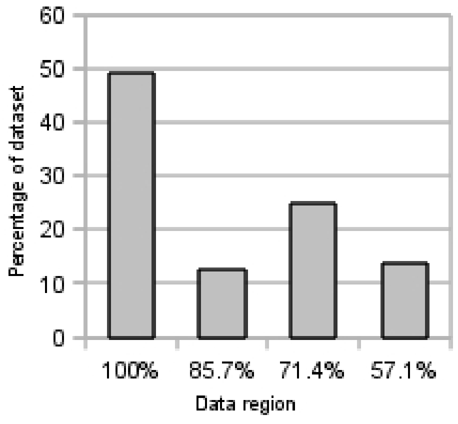


(a) 3x+7y

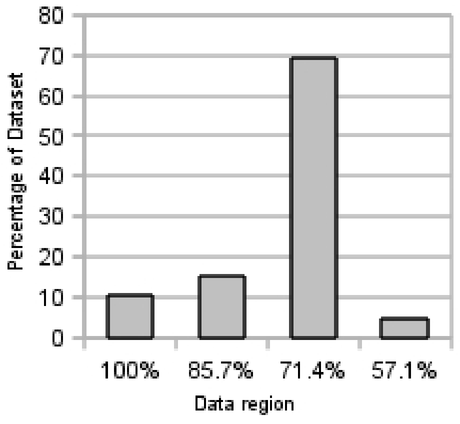
(b) 3x+28y, 6x+28y, 12x+28y

Figure 5.18: Agent trajectory using PRLAB

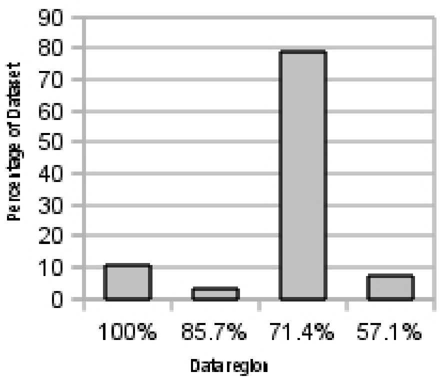
be accurate but still result in a race condition since the object-avoidance algorithm is intentionally simple. "Critical" lists the percentage of memory recalls which were critical to avoiding a collision and "C+A" is the percentage of memory recalls which were critical and also correct. "R+A" measures the percentage of memory recalls (both critical and non-critical) which were recalled with 100% confidence and were also accurate in what they recalled. Therefore, this is a measure of how accurate the memory *thinks* it is, scaled by how accurate it actually was (higher is better). "R+C" measures the number of memory recalls that were recalled with 100% confidence and were critical actions. Therefore this is the same as "R+A" but only for critical actions.



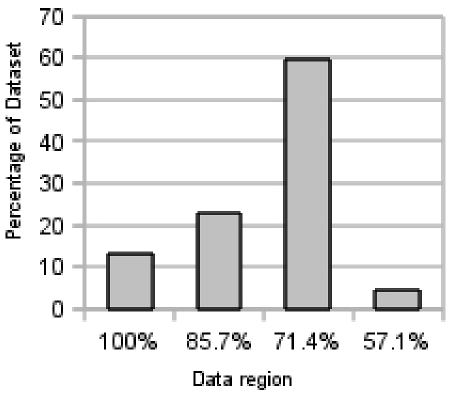
(a) 3x+7y



(b) 3x+28y



(c) 6x+28y



(d) 12x+28y

Figure 5.19: Environment input by percentage previously observed for PRLAB

The associativity tables for the 3x+28y, 6x+28y and 12x+28y configurations are almost identical and are shown in Table 5.10.

5.5.2 PRLAB

The trajectory taken by the robot when it was re-initialized to the original start point after training using PRLAB is shown in Figure 5.18. The performance is decidedly better in all

Table 5.11: Associativity in 3x+28y using PRLAB

Percentage Previously Observed	100%	85.7%	71.4%	57.1%
Recall	98.7%	99.1%	40.7%	66.2%
Accuracy	98.7%	97.7%	34.5%	51.5%
Critical (decisions that avoid collision)	6.6%	88.1%	69.5%	50.0%
C+A (critical and correct)	90.0%	99.0%	36.4%	2.9%
R+C (critical decisions claimed as correct)	6.0%	88.0%	62.0%	37.8%
R+A (correct when claimed as correct)	100%	98.6%	14.1%	64.4%

configurations, particularly since the robot does not collide with objects no matter what the node configuration. Note however, that increasing the nodes from 7 to 28 deteriorates the performance as the robot now gets stuck in a race condition much earlier. This is primarily because of over-training. The 3x+7y training set actually operates on a reduced training set of 18 (unique) pairs. Thus, although the 3x+28y correlation matrix can recall 219 pairs out of 228 pairs trained, because of the many-to-one relationship between the associative pairs, the BAM gets confused in *that one* critical set of sonar inputs. Further increases in the number of nodes had no effect on the trajectory.

Table 5.12: Associativity in 6x+28y using PRLAB

Percentage Previously Observed	100%	85.7%	71.4%	57.1%
Recall	99.3%	91.1%	67.4%	60.9%
Accuracy	99.3%	95.6%	44.2%	60.9%
Critical (decisions that avoid collision)	6.7%	13.3%	75.0%	78.1%
C+A (critical and correct)	90.0%	100%	50.7%	50.0%
R+C (critical decisions claimed as correct)	6.0%	9.8%	77.6%	64.1%
R+A (correct when claimed as correct)	100%	100%	33.4%	100%

Figure 5.19 shows a break-down of the environmental inputs in terms of percentage of the environment observed before. Table 5.14 shows the *associativity* for the 3x+7y configuration trained using PRLAB while Table 5.11, 5.12 and 5.13 show the *associativity* for the 3x+28y, 6x+28y and 12x+28y configurations respectively.

Table 5.13: Associativity in 12x+28y using PRLAB

Percentage Previously Observed	100%	85.7%	71.4%	57.1%
Recall	54.8%	99.4%	43.1%	58.5%
Accuracy	54.8%	72.8%	45.9%	47.7%
Critical (decisions that avoid collision)	2.7%	61.1%	82.4%	53.8%
C+A (critical and correct)	80.0%	99.0%	36.2%	2.9%
R+C (critical decisions claimed as correct)	3.9%	60.9%	71.5%	44.7%
R+A (correct when claimed as correct)	100%	73.3%	30.6%	57.9%

Table 5.14: Associativity in 3x+7y using PRLAB

Percentage Previously Observed	100%	85.7%	71.4%	57.1%
Recall	100%	100%	100%	100%
Accuracy	74.8%	97.7%	80.7%	97.9%
Critical (decisions that avoid collision)	57.7%	15.7%	58.0%	9.7%
C+A (critical and correct)	56.4%	85.7%	66.7%	78.9%
R+C (critical decisions claimed as correct)	57.7%	15.7%	58.0%	9.7%
R+A (correct when claimed as correct)	74.8%	97.7%	80.7%	97.9%

5.6 Discussion

A structural comparison (in terms of the number and type of operations) between the BAM and the PPAM was presented in Chapter 4 (Section 4.5.1). The following sections compare the BAM with the PPAM in terms of performance and fault-tolerance. When comparing the associativity, all results must be biased according to the number of vectors present in that category (which can be seen from the bar chart with the break-down of the environment). Furthermore, for the case when the environmental input has been fully observed (100% column), an ideal associative memory should be able to recall the correct output *every time* and with *full* confidence. Otherwise, it means that the memory has forgotten the training pair. Even if some of the pairs have been forgotten, the memory should at least be able to correctly indicate a confidence level with the value recalled. The "R+A" values are critical since they measure how accurate the memory *thinks* it is, scaled by how accurate it actually was (higher is better).

5.6.1 Comparing Kosko's Method with PRLAB

The relevant fields from Table 5.10 for Kosko's method, Tables 5.11, 5.12 and 5.13 for PRLAB, and Tables 5.2 and 5.3 for PPAM are summarised in Table 5.15.

Table 5.15: Comparing associativity

Percentage Previously Observed	100%	85.7%	71.4%	57.1%	42.8%	28.6%
Kosko's Method for 3x+28y, 6x+28y and 12x+28y (Table 5.10)						
Accuracy	98.9%	62.2%	20.7%	41.0%		
R+A (correct when claimed as correct)	98.9%	82.3%	42.3%	100%		
PRLAB for 3x+28y (Table 5.11)						
Accuracy	98.7%	97.7%	34.5%	51.5%		
R+A (correct when claimed as correct)	100%	98.6%	14.1%	64.4%		
PRLAB for 6x+28y (Table 5.12)						
Accuracy	99.3%	95.6%	44.2%	60.9%		
R+A (correct when claimed as correct)	100%	100%	33.4%	100%		
PRLAB for 6x+28y (Table 5.12)						
Accuracy	54.8%	72.8%	45.9%	47.7%		
R+A (correct when claimed as correct)	100%	73.3%	30.6%	57.9%		
PPAM for 3x+28y (Table 5.2)						
Accuracy	100%	85.0%	65.3%	56.2%	87.7%	50%
R+A (correct when claimed as correct)	100%	100%	100%	100%	100%	N/A
PPAM for 3x+35y (Table 5.3)						
Accuracy	100%	97.2%	80.9%	77.5%	83.8%	68.7%
R+A (correct when claimed as correct)	100%	100%	100%	100%	100%	100%

Comparing the "accuracy" fields for Kosko's method and PRLAB, it is evident that PRLAB is more accurate (with 2 exceptions). The two anomalous cases (100% columns in 3x+28y and 6x+28y configurations) are unreliable because the number of test vectors in these regions is very low (less than 10% – Figure 5.17 and 5.19). In none of the node configurations, with either Kosko's original method or using PRLAB was the memory able to successfully recall all the training pairs. This was as expected from the results of the experiments conducted to train the correlation matrix (Section 5.4). Using Kosko's method, it can be seen that even for the 100% observed case, a value recalled with complete confidence could still be wrong ("R+A" field in the 100% observed column in associativity tables is less than 100%). Although the 3x+7y PRLAB node configuration also suffered from the same problem, higher PRLAB node-configurations overcame this and if a value recalled was claimed to be correct, it *was* correct. Furthermore, in general, the "R+A" measure for PRLAB was higher than the "R+A" measure for Kosko's method. For the one anomalous case where "R+A" for the 71.4% column in Table 5.10 is higher than its corresponding PRLAB associativity table, note that it is better to compare the 6x+28y PRLAB configuration with the 6x+28y Kosko configuration, since these are closest in terms of environment input break-down (79.06% and 83.74%). Comparing these two, even though the PRLAB measures at 33.4% "R+A" compared to Kosko's 42.3%, bearing in mind that Kosko's method is only confident about 45.9% of the recalls, while PRLAB is confident about 67.4% of the recalls, the PRLAB result is still better. In addition, the agent controlled by PRLAB was able to follow the trajectory better than the agent controlled by Kosko's algorithm (Figure 5.16 and 5.18).

From the above comparison of Kosko's method with PRLAB, it can be seen that PRLAB performs better, not only in terms of the number of associative pairs recalled (Section 5.4) but also in terms of the agent's behaviour in a real-time environment. Therefore, it should be sufficient to compare PPAM with PRLAB.

5.6.2 Comparing PRLAB with PPAM

As may be seen from Figures 5.5 and 5.18a, the trajectory of the agent controlled using the 3x+7y PRLAB configuration is almost identical to the trajectory of the agent controlled using the 3x+14y PPAM configuration. However, for higher configurations, the performance of the PPAM is significantly better. Furthermore, from the summarised associativity in Table 5.15, it can be seen that in higher configurations, the accuracy of the PPAM for the 100% previously observed sonar values was perfect. Also, the "R+A" field indicates that whenever the PPAM indicated 100% confidence in the value recalled, it would always be the correct value. This is true not only for the 100% previously observed sonar values, but even when only 28.6% of the environmental input had been previously observed (one out of 7 sonar values).

From the perspective of the PPAM, inputs with less than 100% observed sonar values are exactly the same as 100% observed values, corrupted with noise. Therefore, less than 100% observed sonar values trigger a hetero-associative recall. This is because, although the remaining sonar inputs actually have a value, from the perspective of the architecture, this is equivalent to the inputs being absent, since values not seen before are all treated the same. Since the BAM does not operate on abstract symbols but uses arithmetic calculations, a proper noise test would involve using a random noise pattern based on a known distribution. Nonetheless, for the following discussion, the aforementioned, much simpler noise test is considered where the performance of the BAM is expected to be better than with the random noise source. In addition, a more accurate estimate of the performance of BAM in the presence of noise is included from Sudo et al. (2009).

As can be seen from the results (Section 5.3.4 and 5.5), the PPAM is more noise tolerant. Although the 3x+70y PPAM configuration is the equivalent of the 3x+28y BAM node configuration, comparing the 3x+35y PPAM with any of 3x+28y, 6x+28y or 12x+28y BAM, PPAM has an accuracy of 80.9% when noise reaches 28.6% while that of the BAM is 45.9% at best. This is one of the two types of noise considered by Sudo et al. (2009) – namely, “noise-added original patterns”. As shown in Sudo et al. (2009), for the case of random noise, when the noise level reaches 15%, accuracy drops to approximately 95% for SOIAM (Section 2.1.3.4), 85% for KFMAM-FW (Kohonen Feature Map Associative Memory with Fixed Weights), 60% for KFMAM with batch learning, 38% for KFMAM with sequential learning and 0% for BAM with batch or sequential learning. This implementation of BAM with PRLAB uses a redundant memory to store all the associative pairs. Although this defeats the purpose of having an online associative memory, it does result in the BAM displaying a much higher level of noise tolerance – up to 97.7% accuracy in the best case and 72.8% in the worst case (with 15% noise). The accuracy for the 3x+35y PPAM with 15% noise is 97.2% (Table 5.3) and for the 3x+70y PPAM (the equivalent of the SOIAM and 3x+28y and higher BAMs) is 99.1%. When noise is at 28%, SOIAM has less than 75% accuracy and all others are lower than 30% (from Sudo et al., 2009). Our implementation of the BAM with PRLAB has at best 46% and at worst 35% accuracy. PPAM, on the other hand, has 81% accuracy for the 3x+35y configuration and 90% accuracy for the 3x+70y case. Note that an increase in the number of nodes always results in a corresponding increase in performance, indicating that the PPAM architecture does not easily succumb to issues of over-training.

The second type of noise indicated by Sudo et al. (2009) is “faultily presented random patterns” that must be identified as noise or unknown patterns. In the SOIAM, this is achieved by generating an “unknown” output. In the PPAM, this is achieved by the confidence value in the output. Upon presentation of an input pattern, the PPAM generates an output and a confidence measure. In cases where the input pattern is unrecognisable, the confidence measure is zero or almost zero. Note that results reported in Sudo et al. (2009) do not include “R+A”. To our knowledge, results reported in other existing literature on associative memories also do not include any measure equivalent to “R+A”, which measures the *claimed* accuracy of recall against an *absolute* accuracy of recall. In our opinion, such a measure is critical to determining performance of associative memories.

5.7 Summary

The objective of this chapter was to determine whether the principles of protein processing, as outlined in Section 4.2.2 were capable of acting as a hetero-associative memory while also exhibiting fault-tolerance. The deviation graphs in Figure 5.8 show that the architecture is capable of auto-associative recall, while the associativity Tables 5.2 and 5.3 indicate that it is capable of hetero-associative recall and learning relationships between inputs. Furthermore, faults result in a degradation of performance but operation continues without the faulty nodes having to be replaced, as shown by the results in Table 5.4. Therefore, the architecture does exhibit fault-tolerance without requiring spare nodes to replace faulty ones. As long as there are enough nodes, the PPAM can indicate the *confidence-level* of a memory recall so that if a memory recall is 100% certain, the data recalled is sure to be correct with respect to the past experience of the PPAM. On the other hand, although there are no spare nodes, the firing of nodes is sparse so that many nodes remain idle for long periods between firing. In addition, the architecture has only been tested with a limited fault-model (Section 5.3) and methods that are closer to modeling hardware faults need to be incorporated in the future.

A mobile robot in the player/stage environment was controlled using Bidirectional Associative Memory and also Protein Processor Associative Memory. Two training algorithms were attempted for the BAM – the original learning algorithm presented by Kosko (1988) and also the more popular PRLAB (Oh and Kothari, 1994) that guarantees recall of all training pairs if it is possible to store them in a BAM correlation matrix. Parameters were tuned in an attempt to maximize memory utilization of BAMs. Although the PRLAB is relatively insensitive to initial parameter values (as claimed), we found that these values can, in the worst case (when the dataset is not orthogonal), drop the performance of the PRLAB to the level of the original Kosko's algorithm. Various network configurations for the BAM were attempted with the number of nodes being selected based on the level of quantisation desired for input pairs. However, none of the configurations were successfully able to recall *all* associative pairs despite tuning the parameters. This is because of the fact that BAMs require associative pairs to be orthogonal. Therefore, although increasing the number of nodes in the 2 layers increased the capacity of the BAM, it was still unable to store and recall all the pairs. In fact, in some cases, increasing the number of nodes had the opposite effect because the associative pairs became *less orthogonal* because of the new encoding required by a higher node count in each layer. In order to store all pairs in a BAM, either the data should be re-encoded to become orthogonal (Simpson, 1988, cited in Oh and Kothari (1994)) or the training dataset should be parsed to select critical data points that are orthogonal and also sufficient to describe the underlying distribution/algorithm. For both solutions, this means that generating the training dataset for a BAM requires *a priori* knowledge of the associative pairs, which is not a realistic requirement for online, real-time learning environments. Furthermore, there is no such requirement for the PPAM.

Equivalent PPAM and BAM configurations were attempted as controllers for the agent and the PPAM was shown to perform better in all cases, displaying higher accuracy and noise tolerance. Results were also compared with SOIAM and the PPAM was shown to be more tolerant to both types of noise identified in Sudo et al. (2009), thus achieving higher accuracy. The advantages of the player/stage environment were utilized in transferring the player implementation to the E-Puck mobile robot (Mondada et al., 2009) and the results were verified on the embedded environment. A close-up of the E-Puck in its maze is shown in Figure 5.20a while an overall view of the robot in the maze can be seen from Figure 5.20b.

The PPAM architecture described in Chapter 4 has been shown to be better at associativity and noise tolerance. It has also been shown to be fault-tolerant so that a node configuration of 3x+70y degrades gracefully even with 10 nodes *broken* (Section 5.3.4). Furthermore, the PPAM architecture achieves this without the use of arithmetic operations. Nonetheless, there is significant room for improvement, particularly since the architecture has issues for hardware design. These are outlined in Section 5.7.1. A second version of the PPAM is presented in Chapter 6, which is expected to retain (or improve) the level of associativity and address these hardware issues to make the architecture more suitable for a hardware implementation.

5.7.1 Hardware Implementability

The architecture has features which give the hardware designer a number of issues:

1. The number of nodes required to accurately learn the relationships is related to the number of separable bins in the dataset.

(a) E-Puck mobile robot

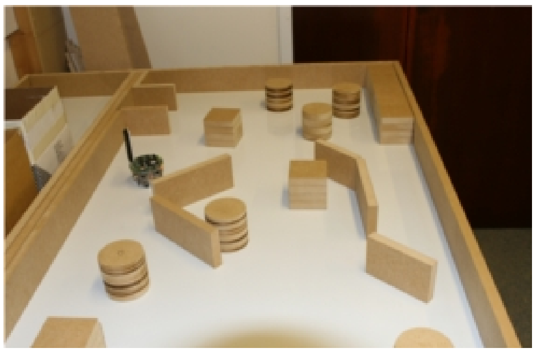

(b) E-Puck maze

Figure 5.20: E-Puck platform

2. Each node in one column array (Figure 5.2) needs to be fully connected to nodes in the other column array. Attempts with fewer than fully connected networks failed to learn the relationships accurately. Although preliminary designs of nodes indicated that they were quite small, the number of connections to create full connectivity became prohibitive. Alternate solutions using routers with a common bus could not be justified considering the granularity of the nodes.

3. The memory width of the CAM in each node is dependent upon the neighbourhood that the node observes – which is the entire column array as each node is fully connected. This is discussed further below.

For the experiments described in this chapter, the nodes were binary, thus limiting the memory width in nodes. Therefore, for example, for the 3x+28y network configuration, the nodes had 28-bit wide memory. As will be obvious, however, scaling up quickly becomes infeasible in hardware as a neighbourhood of 100 nodes means the memory has to be 100-bits wide. Furthermore, and more critically, increasing the symbol size (even for smaller sized neighbourhoods) means memory width becomes impractical. Note that this is memory *width* and not memory *depth*, which is the subject of the discussion. Therefore, even if points 1 and 2 (above) are overcome, point 3 is a major barrier to hardware implementation for larger networks.

It should be noted that in all the experiments – even the 3x+14y node configuration – the PPAM learnt the most important relationships (critical decisions) in a few time-steps (around 100). Wrong critical decisions were only wrong in the direction of turn, so that no matter how different a new environment may have been, the robot never collided with an object. Although problematic for a hardware implementation, the technique is still feasible for software implementations.

In order to improve the architecture for a hardware implementation, a modified version is explored in Chapter 6, where the number of nodes are related to the number of variables and the number of dimensions of each variable rather than the number of quantised bins. Furthermore, the CAM in each node stores the output value of a single neighbour along with the output value of the node itself so that the width of the memory is not dependent upon the neighbourhood of the node. Simulation using a second PPAM network – to relate actions with long-term results – is left as future work.

Chapter 6

Hardware Architecture for the Protein Processor Associative Memory

The PPAM stems from research into parallel architectures targeted towards problems in the domain of Artificial Intelligence (AI). An AI application may be defined as one that tries to achieve behaviour displayed by intelligent beings like humans (Turing, 1950), and most will agree that the hallmark of AI applications is complexity. Some of the desired features of such applications are adaptability (in the presence of new stimuli), robustness (in the presence of errors), real-time results and scalability. Although many hardware architectures exist that attempt solutions for problems in this domain (refer to Chapter 2 for a comprehensive review), these are based on Arithmetic and Logic Units (ALUs) and often are simply Von Neumann style processors in parallel, thus also suffering from the well known Von Neumann bottleneck. As mentioned in Chapter 2, the traditional computation paradigm is to implement simple, mathematical operations in ALUs, which are accessed by (machine) instructions to a control unit in the processor. This is further accessed through (potentially multiple) software/firmware wrapper layers, which may be an Operating System or a Virtual Machine or both. Therefore, implementing complex AI applications on traditional machines not only involves mapping an individual complex operation to multiple simple mathematical operations, but also each operation passes through multiple software layers to reach a hardware unit (ALU) that is accessed through yet another hardware wrapper (the control unit). Traversing all these layers is a severe handicap for an application that intends to solve million-scaled AI problems[1] in real-time.

[1]AI problems that use millions of facts to come up with solutions; e.g. natural language processing

Chapter 4 proposes a novel architecture called the Protein Processor Associative Memory (PPAM), that uses hetero-associative recall to achieve artificial intelligence and explores the effect of moving computation into memory, while Chapter 5 describes experiments performed to test the performance of the PPAM and presents the results of these experiments. From these results it can be seen that:

1. The PPAM is capable of learning associations between two variables.
2. This learning of associations can result in the PPAM performing a *meaningful* task.
3. It is better than other traditional AI techniques at performing robust, associative recall.

However, the architecture (as presented in Chapter 4) has issues for a hardware implementation – as shown in Section 5.7.1. This hardware implementability is crucial, since a software implementation of the PPAM implies using the traditional ALU based hardware architecture, whereas the argument for the PPAM is based on moving computation away from arithmetic operators.

This chapter describes the modifications to the PPAM that make it more suitable for a hardware implementation. These reduce the number of connections per node and dissociate the width of the memory in each node from the dimensions of the data. Whereas in the previous version (Chapter 4) each node needed to be fully connected to nodes in the other category (number of connections were proportional to the number of dimensions of data), in this version, nodes only need to be connected to $1 + C$ nodes, where C is the number of Conflict-Resolving nodes (Figure 6.2) and is independent of the number of dimensions of the variables. Also, whereas in the previous version each node needed to implement memory at least $b \times D$ bits wide, where b is the number of bits that encode each symbol and D is the number of dimensions of the other category of data, this version implements memory that is only b bits wide. Section 6.1 lists the considerations that went into some of the early design decisions. Section 6.2 uses a worked example to describe the connection topology and node structure. Section 6.3 presents details about the physical hardware architecture and includes synthesis reports and some thoughts on implementation on various fabrics. Verification of performance is carried out in Chapter 7.

6.1 Trade-offs and Design Decisions

Trade-offs are a key aspect of any engineering process. Options need to be weighed and the impact of choosing one against the other must be evaluated with a view of the target applications. Considerations of time, cost and effort constrain and shape the final result. This section lists these design decisions for the hardware design and presents the reasoning for each. Section 6.1.1 presents a short review of the advantages and disadvantages of asynchronous versus synchronous design techniques to show the reasoning for the each sort of method in the PPAM. Section 6.1.2 is a brief overview of some of the issues in conventional parallel systems and shows that the PPAM avoids all these issues of designing and programming parallel architectures.

6.1.1 Asynchronous Vs Synchronous

The components of most modern machines and systems operate in synchronization with one (or more) central clocks. The primary reason for this is simplicity of design. The following is a brief review of the issues involved in the choice of synchronous vs asynchronous design and the advantages and disadvantages of each.

Asynchronous circuits suffer from Static and Dynamic hazards. A *static-1 hazard* is defined as the case where the output is meant to remain stable at logic-1, but momentarily glitches to logic-0 because of the asynchronous nature of signals without clock synchronization. *Static-0 hazards* are the complementary situation where the output is meant to remain stable at logic-0 but momentarily glitches to logic-1. When the output is meant to transition once ($0 \rightarrow 1$ or $1 \rightarrow 0$) but instead the output makes three or more transitions ($0 \rightarrow 1 \rightarrow 0 \rightarrow 1$), these are defined as *dynamic hazards.* All static and dynamic hazards *due to a single input change* can be eliminated by adding additional cubes covering all adjacent logic-1s in a K-map (Hauck, 1995). Multiple simultaneous input changes, on the other hand, may cause unavoidable hazards. Therefore the policy in asynchronous circuits is usually to only allow single input changes. However, this has the effect of limiting the speed of asynchronous circuits, which otherwise could operate much faster.

Another method to resolve hazards is to assume *Bounded delay models*, which are similar to the model used in synchronous digital circuits in that they assume that the delay in all circuit elements and wires is bounded. *Huffman circuits* are bounded delay circuits with the *fundamental mode* assumption, which assumes that an external input cannot occur until the entire system has settled to a stable state because of the previous input. Note however, that this also slows down the operation of the circuit. To design circuits without the fundamental mode assumption, detailed knowledge of the circuit (false-paths, multi-cycle paths and so on) is required so that the designer can implement *circuit-specific* strategies to allow new transitions to arrive earlier. Since the techniques are circuit-specific, they are not reusable and therefore, are not the preferred method for designers.

Burst mode circuits allow a pre-specified set of inputs to transition in any order by maintaining their state so that the circuit reacts only after all the inputs have changed. The machine then transitions outputs and changes state. Such circuits allow multiple inputs to change simultaneously, since the output does not change until all inputs have changed. Therefore there are no unavoidable hazards like in Huffman circuits and the technique of adding redundant sum of products (from K-Map) is sufficient to remove hazards. However, note that this too slows down the circuit since the state of the circuit is maintained until all inputs have changed.

Delay insensitive circuits assume that delays in both elements and wires are unbounded. Therefore, irrespective of how long the circuit waits, there is no guarantee that the inputs will be properly received. Consequentially, receivers and transmitters perform a handshake using a completion detection circuit and must wait for this completion signal before transmitting the next signal. Other techniques include I-Nets which are based on Petri Nets, Trace based circuits, Signal Transition Graphs which are one of the more well-known design methodologies under study and are also based on Petri-Nets and Change Diagrams which are similar to Signal Transition Graphs. Hauck (1995) presents a detailed study of these and others techniques including Micropipelines and methods based on the use of Muller-C elements, while Navaridas et al. (2009) presents the design for a fast and area efficient multi-input Muller C-element. However, under the current state of the art, these techniques still suffer from the disadvantages of slowing down the operation of the asynchronous circuit and requiring the designer to employ circuit-specific techniques to overcome hazards. In contrast, for synchronous logic design, synthesis tools do most of the work to translate high-level HDL into the physical circuit.

In terms of memory, although latch-based asynchronous-write RAMs are in existence, these aren't commercially available as packages; and neither are sufficiently reliable, non-volatile analog memories (Vreeken, 2003). Asynchronous-read RAMs, on the other hand, are quite wide-spread. Since a major portion of the PPAM is memory, and the amount of memory determines the capability and capacity of the PPAM, it is essential to use a memory architecture that is reliable and is widely and cheaply available. Therefore each node in the PPAM performs synchronous write operations and asynchronous read operations.

A concise summary of the issues discussed above is presented in table 6.1 (Hauck, 1995, Hillis, 1986, Jackson and Tyrrell, 2001, Vreeken, 2003). This table is not an exhaustive

#	**Synchronous**	**Asynchronous**
1	Design is simple with fast, precise and small equipment, and reusable techniques.	Design is hard, and solutions are usually not reusable and specific to the particular problems. Therefore, designers must remove hazards by hand.
2	Size and speed of hardware is well defined and predictable.	Designs should be smaller and faster theoretically, however, this is seldom the case because signalling policies end up requiring extra time.
3	Many CAD tools to assist in the design process.	Existing tools do not support techniques for asynchronous circuits.
4	Many reliable memories are commercially available.	Sufficiently reliable non-volatile analog memory does not (yet) exist.
5	Clock toggle consumes more power.	There is no clock so power consumption is low.
6	Does not scale well because it becomes harder to synchronise components in large designs because of clock skew.	No synchronisation issues for larger designs but in practice removing hazards for large designs is very difficult.
7	Clock speed is determined by the critical path, so all components operate at the speed of the slowest one.	Average case instead of worst case performance as circuits detect completion of computation.
8	All paths must be optimized otherwise critical path may be very slow. Therefore all paths must be optimized during place and route (PnR).	If the critical path is used infrequently, PnR effort level can be set low, since it does not effect speed of frequent paths.
9	Impossible to input real-world (asynchronous) data without the possibility of synchronisation failure.	Easily interfaced with analog world.

Table 6.1: Synchronous vs asynchronous design techniques in VLSI

list, however, note that there are many points in favour of asynchronous design such that, at first glance, it might seem better to use asynchronous techniques. In practice, however, synchronous design is preferred in general (particularly in the industry) as the ease and reusability of synchronous design far outweighs any potential advantages of the other. This has meant that there hasn't been a lot commercial interest in asynchronous design techniques, although, there has been some interest in the research community.

The PPAM combines *Delay insensitive* and *Burst mode* asynchronous design techniques with a synchronous-write RAM. Thus memory recall is an asynchronous operation, while learning is synchronous to avoid the requirement of asynchronous RAMs in hardware. As discussed above, the use of these asynchronous methods means that the circuit operates at less than the theoretical maximum. Note however, that the PPAM is envisioned as a system that operates on real-time inputs obtained from the real world. Since sensory data from the real world is typically slow – in the range of KHz, rather than MHz – this slowing down does not reduce the performance of the system. On the contrary, it means that the system can meet the data rates while at the same time not waste power on a clock that is operating faster than the data rates require. Section 6.3.1 explains the theoretical design and presents circuit diagrams and schematics, while Section 6.3.2 presents an adapted implementation that only uses digital logic elements available in existing FPGAs. Section 6.3.2 also shows that using a small number of user-defined constraints specifying false-paths and multi-cycle paths, the asynchronous logic can be synthesised using a standard commercial synthesis tool (Xilinx ISE) which is not specifically designed for implementing asynchronous logic. Before that, however, the following section discusses the design considerations for conventional parallel architectures and explains why they are irrelevant for the PPAM.

6.1.2 Trade-offs in Parameters for Parallel Systems

The PPAM is a parallel architecture since it is composed of nodes operating in parallel. However, unlike other parallel architectures (reviewed in Chapter 2) nodes are not based on *traditional processors*, where traditional processors are defined as containing (as mentioned previously in Section 2.3):

- an ALU that calculates results,
- a control unit that directs execution based on a program that is fetched from memory and

- a memory that stores results (data) and instructions.

The usual motivation for building parallel systems is to increase the amount of processing power and decrease the time required to perform the processing. In the case of architectures for AI, quite often the objective is to achieve real-time processing. To ensure unambiguity, the definition of real-time processing used by Atrubin (1965) is adopted (Section 2.3.3):

> "Let a sequence of inputs be defined as $i_0, i_1, i_2, \cdots$ and a sequence of outputs be defined as $U_0, U_1, U_2, \cdots$ so that U_n is a function of the inputs up to and including i_n. If U_n is computed within a fixed time after the arrival of i_n and this time is independent of n, then the computation is *real time*. Otherwise, if the delay between receiving i_n and the computation of U_n is arbitrarily large for an arbitrary n, then computation is *general*."

This section presents an overview of the issues in designing and programming conventional parallel architectures; the purpose being to show that the PPAM is able to avoid all these through its novel computation method.

From Section 2.2 it can be seen that most existing parallel architectures are simply networks of traditional processors as defined above. In theory, parallel systems can provide massive amounts of computation power and speed. In practice however, the efficiency of parallel architecture can be greatly diminished by conflicting factors and the contention for shared resources. From the discussion in Section 2.3.3 it can readily be seen that most issues pertain to the programming of parallel architectures. The PPAM, on the other hand, does not execute a program or need a configuration bit-stream. This allows it to circumvent the issues faced by parallel architectures based on traditional processors (listed in Section 2.3.3). However, it must be noted that the PPAM compromises by being much less general purpose. The most significant advantage of the traditional processor architecture is its flexibility, since these are general purpose processors and are Turing Complete. Although the PPAM does not claim Turing Completeness, it does display a range of flexibility. As explained in Chapter 4 and particularly in Section 4.4, the PPAM can adapt online and learn the associations of whatever data is presented in real-time, discarding any old, redundant information. Therefore, assuming that the problem *can* be solved by an associative memory, the PPAM should be applicable. However, as will be seen from Section 6.3.1 which presents the theoretical hardware design, there are parameters of the circuit which are dependent upon the dataset being associated. Therefore, one fundamental assumption in the claim for the flexibility of the PPAM is that it is implemented either on:

- a reconfigurable fabric like the FPGA, which is less desirable because of reasons which will be explained in Section 6.3.1 or
- an ASIC which includes certain dynamic routing capabilities – either through packet switching routers, virtual connections, or physical multiplexers to dynamically redirect packets. It must be noted however, that this dynamic routing capability is not required in real-time since it will happen only once at the beginning of the association when the PPAM is informed of the characteristics of the dataset.

Due to cost and time considerations, the ASIC option was not actually implemented, but Section 6.3.2 does present details of implementing on a reconfigurable fabric. Before that, however, Section 6.2 uses a worked example to explain the changes to the design of the PPAM as presented in Chapter 4 which allow the PPAM to be better suited for a hardware implementation.

6.2 Logical View

Some changes were required in the design of the PPAM in order to resolve the issues raised in Section 5.7.1 about the hardware implementability. Before presenting the circuit design in Section 6.3, this section attempts to provide a logical view of the altered version, and uses a worked example to illustrate the changes to the design.

6.2.1 Sample Dataset

The following dataset is used to illustrate the operation of the PPAM. The variables being associated are the Cartesian coordinates (X, Y) against the polar coordinates (R, θ), both of which are two-dimensional variables, whose relationship can be described with the simple equations in 6.1

$$X = R\cos\theta$$
$$Y = R\sin\theta \tag{6.1}$$

Table 6.2 is an excerpt from the complete dataset that describes these two variables.

Cartesian Coordinates		**Polar Coordinates**	
X	**Y**	**R**	θ
1	1	1.41	0.78
1	2	2.23	1.10
1	3	3.16	1.24
1	4	4.12	1.32
2	1	2.23	0.46
2	2	2.82	0.78
2	3	3.60	0.98
2	4	4.47	1.10
3	1	3.16	0.32
3	2	3.60	0.58
3	3	4.24	0.78
3	4	5.00	0.92
4	1	4.12	0.24
4	2	4.47	0.46
4	3	5.00	0.64
4	4	5.65	0.78

Table 6.2: Cartesian-polar dataset

Recalling that the PPAM operates on abstract symbols, the simplest symbols that can be used are binary (integer) values. Since the values for X are integers from 1 to 4, they can be used as they are and do not need to be re-encoded. Although the same logic applies to Y, in order to ease identification of firing nodes for debugging, the values for Y are re-encoded as shown in table 6.3a.

Each dimension of the Cartesian variable has 4 symbols. Therefore, dividing the complete range of values in R (and θ) into 4 equal-sized categories, the bins and symbols are as shown in tables 6.3b and 6.3c. The resulting table with the dataset encoded in abstract symbols is shown in table 6.4.

Y	Symbol
1	5
2	6
3	7
4	8

(a) Cartesian Y encoding

Min	Mid	Max	Symbol
1.410	1.940	2.470	0x4
2.470	3.000	3.530	0x3
3.530	4.060	4.590	0x2
4.590	5.120	5.650	0x1

(b) Polar R encoding

Min	Mid	Max	Symbol
0.240	0.375	0.510	0xC
0.510	0.645	0.780	0xB
0.780	0.915	1.050	0xA
1.050	1.185	1.320	0x9

(c) Polar θ encoding

Table 6.3: Dataset encoding

6.2.2 Network Topology and Structure

In order to store the dataset shown in section 6.2.1 (table 6.4) consider a network of 4 nodes as shown in figure 6.1. This represents the sort of network that would be required in the previous version of PPAM described in Chapter 4. Note that an individual node (for

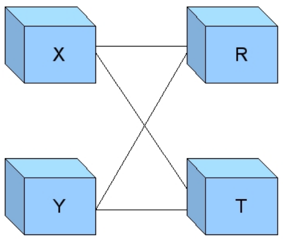


Figure 6.1: Abstract view of unexpanded nodes

example X) would be connected to D other nodes, where D is the number of dimensions of the other variable (which is 2 in this case). Although this is not an issue for $D = 2$, as dimensions are scaled, this high connectivity would become a problem for a hardware implementation. Also, note that the width of the memory in a node must be wide enough to store D symbols from its neighbours. In order to make the PPAM more suitable for a hardware implementation, these two design considerations are resolved as follows. Each node in figure 6.1 (for example X) is expanded into multiple other nodes as shown in figure 6.2. The horizontal axis has D number of nodes, where D is the number of dimensions of the other variable and the vertical axis has C number of nodes where C is the maximum number of conflicts (Section 4.3.3) that are being catered for. All nodes in figure 6.2 receive the same external (environmental) input – and therefore fire with the same output value

Cartesian Coordinates		**Polar Coordinates**	
X	**Y**	**R**	θ
1	5	0x4	0xA
1	6	0x4	0x9
1	7	0x3	0x9
1	8	0x2	0x9
2	5	0x4	0xC
2	6	0x3	0xA
2	7	0x2	0xA
2	8	0x2	0x9
3	5	0x3	0xC
3	6	0x2	0xB
3	7	0x2	0xA
3	8	0x1	0xA
4	5	0x2	0xC
4	6	0x2	0xC
4	7	0x1	0xB
4	8	0x1	0xA

Table 6.4: Cartesian-polar encoded dataset

when an external input is applied. Each node receives only one neighbour-node input – where neighbour-node is a node from the other variable which is being associated with this variable. In addition, each node in the same column receives the same neighbour-node input. Therefore, with reference to the sample dataset, there would be 2 columns in figure 6.2, since there are 2 dimensions in the polar coordinate variable (R, θ). Also, the nodes in column A would receive input from nodes in one dimension of the polar coordinate variable (for instance R) while the nodes in column B would receive input from nodes in the other dimension (for example θ). Note that since each node now only observes the

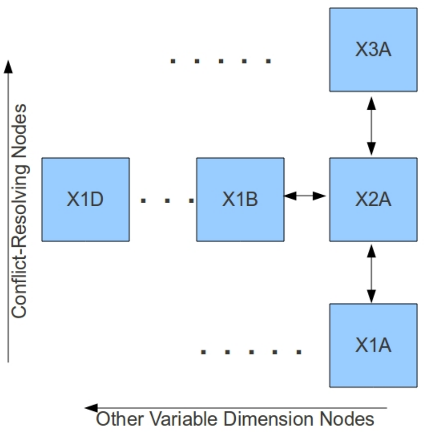


Figure 6.2: Abstract view of expanded nodes

input from one neighbour-node, the memory width required to record the neighbour-node value is also restricted to just one symbol wide. The connections between nodes within the same dimension (nodes in figure 6.2) are explained in more detail in section 6.2.3.

Although at first it may seem that the structure of the PPAM is still related to the dimensions of the data being associated, there is a crucial difference in this new version that allows better hardware implementability. The structure of a *node* and particularly the width of memory inside a node is independent of the dimensions of data, even if the connection network is related to it. Although for small PPAMs (like the one described for this example) this may be mostly irrelevant, for large scaled PPAMs however, it becomes a major advantage in terms of hardware implementability.

6.2.3 Learning and Conflict Resolution

Conflicts occur when a node observes two different neighbour-node inputs along with the same external input or vice versa (the same neighbour-node input with different external inputs). The first is called a *conflict-in-input*, while the latter is a *conflict-in-output*. Although conflicts may occur when there is a many-to-one (or one-to-many or many-to-many) relationship in the dataset (as exists in the sample dataset in table 6.4), conflicts may also occur in a dataset where there is only a one-to-one relationship. Consider the first two tuples in table 6.4. Although (X, Y) together are unique, a single X node that has a neighbour-node from amongst the θ nodes (as described in section 6.2.2) would observe first A and then 9 while its external input would stay 1. Thus it would consider this as a conflict. If there are C Conflict-Resolving nodes (Figure 6.2), then C conflicts in the dataset can be resolved.

When a conflict is observed, nodes *differentiate* (a process inspired from GRNs) to ensure that no node stores conflicting tuples (explained further in section 6.2.4). Until the first conflict is observed, all nodes in one column (for example column A in figure 6.2) store the same values. Upon observing the conflict, some nodes update to the new (conflict generating) tuple, while other nodes maintain the old (previous) tuple. If the memory in each node is likened to DNA in a cell, this self-modification of memory can be likened to changing genetic expression, which is the basis of genetic differentiation. However, this differentiation also means that during memory recall, it is possible for nodes to generate two (or more) different, conflicting outputs. To resolve this, all nodes in the same row (for example row 1 in figure 6.2) reinforce outputs. Therefore, if node $X1B$ generates the same output as node $X1A$, the output may be considered to be more likely to be correct than if only one node produced it. And so, as more nodes reinforce the outputs, the *confidence* of the PPAM in its output increases. In the extreme case, this confidence can be used to make a binary decision so that only if the output of a node is reinforced, is it output from the PPAM; otherwise it is ignored. As a consequence, if one node in a row decides to update to a new conflict generating tuple, all nodes in the row update to the new tuple. However, if even one of the nodes in a row is unable to update, all nodes in the row are unable to update. Section 6.2.4 works through the sample dataset using this binary decision method.

During conflict resolution, nodes are updated in a cascaded fashion. All nodes in one row may be updated simultaneously or in any sequence as long as they are all updated before the nodes in other rows start, or after the nodes in other rows finish updating. Therefore, memory writes are synchronous, however memory recalls are asynchronous and occur as soon as input is applied.

6.2.4 Worked Example

For simplicity, assume the same C (number of Conflict-Resolving nodes) for X, Y, R and θ, and let it be arbitrarily chosen as 4. There would be $2C$ nodes (8 nodes) in each of X, Y, R and θ, resulting in a total of 32 nodes in the PPAM. This can be compared with the previous version of the PPAM (described in Chapters 5 and 4) which would require a PPAM of 16 binary nodes, however, scaling up would be infeasible in hardware. Section 5.7.1 describes in detail the reasons why the previous version was infeasible.

We will use the same convention as used in Figure 6.2 to identify nodes – i.e. columns (for dimensions) are identified using letters A and B, while rows (for Conflict-Resolving nodes) are identified using numbers $1, 2, \ldots C$ Also, for simplification, when an external input is applied, the output of a node can be assumed to be the same value as the external input, regardless of memory contents. The following will work through the dataset for the Y nodes which should be sufficient to illustrate the operation of the PPAM.

Let all Y nodes in column A (YiA nodes) have R nodes as neighbour-nodes, and let all column B nodes (YiB nodes) have θ nodes as neighbour-nodes. Then, when the first tuple is applied, YiA nodes observe 0x4 as neighbour-node input and 0x5 as external input and so update their memory as shown in time step 1 of table 6.5.

Time-Step	Mem-address	Y1A	Y2A	Y3A	Y4A
1	0x5	0x4	0x4	0x4	0x4
	0x6				
	0x7				
	0x8				
2	0x5			0x4	0x4
	0x6	0x4	0x4		
	0x7				
	0x8				
3	0x5			0x4	0x4
	0x6	0x4	0x4		
	0x7	0x3		0x3	0x3
	0x8				

Table 6.5: Memory contents in YiA nodes during the worked example

Similarly, YiB nodes observe 0xA as the neighbour-node input and 0x5 as the external input and so update their memory as shown in time step 1 of table 6.5. Upon presentation

Time-Step	Mem-address	Y1B	Y2B	Y3B	Y4B
1	0x5	0xA	0xA	0xA	0xA
	0x6				
	0x7				
	0x8				
2	0x5			0xA	0xA
	0x6	0x9	0x9		
	0x7				
	0x8				
3	0x5			0xA	0xA
	0x6		0x9		
	0x7	0x9		0x9	0x9
	0x8				

Table 6.6: Memory contents in YiB nodes during the worked example

of the second tuple of the dataset (in table 6.4), YiA nodes fire with 0x6 and observe 0x3 from their neighbour-nodes again. Correspondingly, they detect a conflict as 0x4 is once again the neighbour-node input while the external input is not 0x5. To resolve this, half of the nodes with conflict (which in this case is all of the YiA nodes) decide to update to the new tuple, while the other half decide to retain the old value. At the same time, YiB

nodes fire with 0x6 and observe 0x9 from their neighbour-nodes, which is not a conflict and therefore, none of the YiB nodes *object* to updating to the new tuple. However, only those YiB nodes update to the new tuple whose corresponding YiA nodes indicated that they can update to the new tuple as well. Thus, the Yij nodes update as shown in time step 2 of Tables 6.5 and 6.6.

A similar case (with 2 differences) occurs when the next tuple from the dataset is presented. Firstly, YiB nodes detect a conflict instead of YiA nodes. Secondly, only $Y1B$ and $Y2B$ nodes have a conflict and not $Y3B$ and $Y4B$. This is because these nodes have already differentiated. In order to resolve the conflict, half of the nodes with the conflict, update to the new conflict-generating tuple, while the other half retain the old tuple. Therefore $Y1B$ updates its memory contents and $Y2B$ does not. Since $Y3B$ and $Y4B$ do not have a conflict, they determine that they are willing to update to the new tuple. In the meantime, all of the YiA nodes indicate that they are willing to update to the new tuple. However, like the previous time step, only those YiA nodes update to the new tuple whose corresponding YiB nodes indicated that they can update to the new tuple as well. Thus, the memory values in Yij nodes is as shown in time step 3 of Tables 6.5 and 6.6.

After 3 time steps, if the Cartesian-coordinate external input is removed, and a polar-coordinate external input of $R \to$ 0x4 and $\theta \to$ 0xA (first tuple in the dataset) is provided, the PPAM should recall 0x5 from the Y nodes. Nodes $Y1A$ and $Y2A$ fire with 0x6 while nodes $Y3A$ and $Y4A$ fire with 0x5. In the meantime, nodes $Y1B$ and $Y2B$ do not generate any output since these nodes do not have 0xA in their memory. Nodes $Y3B$ and $Y4B$, however, fire with 0x5, thus reinforcing the outputs of $Y3A$ and $Y4B$. In this way, the correct value of 0x5 is recalled from the Y nodes in the PPAM.

6.3 Hardware Implementation

The PPAM is a distributed system such that each node makes its own decision to update its memory contents or not. To make this decision, it does need some information from other nodes. As shown in Section 6.2.4, this includes information from Conflict-Resolving nodes and also Other-Dimension-Neighbour nodes. To limit the number of connections per node, information from Other-Dimension nodes is propagated sequentially over a pipeline while information from Conflict-Resolving nodes continues to be transmitted in parallel.

Therefore, nodes make their decision in $D+1$ steps, where D is the number of dimensions in the other variable. For a digital logic implementation, this would translate to $D+1$ clock edges which must be applied after each tuple is presented as input before the PPAM can store the associations correctly. In each step, nodes in one column transmit an address and data to each other, and use this information to determine if there are other nodes that store the same information as themselves. The total number of these *other nodes* (that store the same information) is used to generate a threshold which is later used to determine if the current node should update its value or maintain the old one. A node n_0, generating an output q_0, compares this against the outputs of its Conflict-Resolving neighbours $(q_1, q_2, \cdots q_{c-1})$. If the total number of nodes where $q_i = q_0$ is greater than the threshold, then n_0 decides to update to the new value. This process will hereafter be known as the "threshold and compare" operation and can be implemented in a variety of ways described in more detail in Section 6.3.1.

If there is a conflict in input, and a node decides to update, two (or more) tuples may have to be updated, as one tuple has to be removed while another has to be entered. For example, in time step 2 of table 6.5, the memory at address 0x5 in node $Y1A$ has to be cleared (zero-ed) while the location at 0x6 has to be updated to 0x4. Thus if a node decides to update, it must ensure not only that there is no conflict in writing the new value at the new location, but also that it is not the only node maintaining the old value. For example, in time step 2 of table 6.6, $Y1B$ must ensure that address 0x6 either does not contain any meaningful data or that this data is also stored somewhere else. It must also ensure that removing the value at address 0x5 does not remove the only copy of that data. Therefore, in the first D cycles after a new tuple is applied, nodes transmit information about tuples that will need to be removed (if any) if the memory is updated. During these cycles, the threshold and compare operation is used to determine which nodes may remove the old tuples (if there is a conflict). This has to be done in D cycles because, in the worst case, there might be D different conflict-in-input cases for a single new tuple and therefore D tuples might need to be removed. In the last cycle, nodes transmit the new tuple information so that the threshold and compare operation can determine which nodes can add the new tuple and update the node accordingly.

6.3.1 Circuit Design and Schematic

Figure 6.3 illustrates the overall schematic for a node in the PPAM. In order to determine when a node may update its memory, it needs to be able to detect when an external input is being presented. Since all Conflict-Resolving nodes receive the same external input, the circuit for this does not need to be duplicated for each of these. The presence of external input is detected using the standard method of pulling down lines from inside the chip, so that when an external source drives with a higher strength, it can be detected. Note that this implies that the symbol 0 is reserved to indicate that no external input is present. A sample circuit for this is shown in Figure 6.4.

The external input LUT at the top of Figure 6.3 is optional. This is because the LUT is useful only if the PPAM desires to remap the symbol encoding. For example, in the case of the worked example in Section 6.2.4, if the external input is 0x1 and the PPAM desired to internally remap 0x1 to 0x5, this could be implemented using an internal LUT. However, this would require another LUT to be implemented at the output of the PPAM when it was recalling the values to reverse the mapping for the external reader.

When an external input is detected, this value is input as an address to the neighbour-node LUT in Figure 6.3, while the value from the other variable is fed as data to the LUT. Simultaneously, the combinatorial cloud labeled "match all locations" uses N XNOR gates to compare all the locations of the neighbour-node LUT with the value from the other variable; where N is the size of the memory in bits. Consider the case when the example dataset in Section 6.2.4 is presented to the nodes in Figure 6.2, and node Y1A fires with 0x5 external input and observes 0x4 from the other variable in step 1 (Table 6.5). In this situation, 0x5 is input as the address to the LUT and 0x4 is input as data, while the "match all locations" block compares 0x4 with the values in all the locations in the LUT. The circuit for this block is shown in Figure 6.5. In case a match is found and there is no external input present, this matched address is fed as an input address to the neighbour node LUT. Note that this results in a combinatorial feedback loop where the address of the matched location is generated from the output of the asynchronous read LUT. However, since the LUT is never written when this combinatorial loop is active, any potential ringing on the lines is not destructive and will always result in an eventual stable output. The neighbour node LUT is multi-port for the situation where there are conflicts

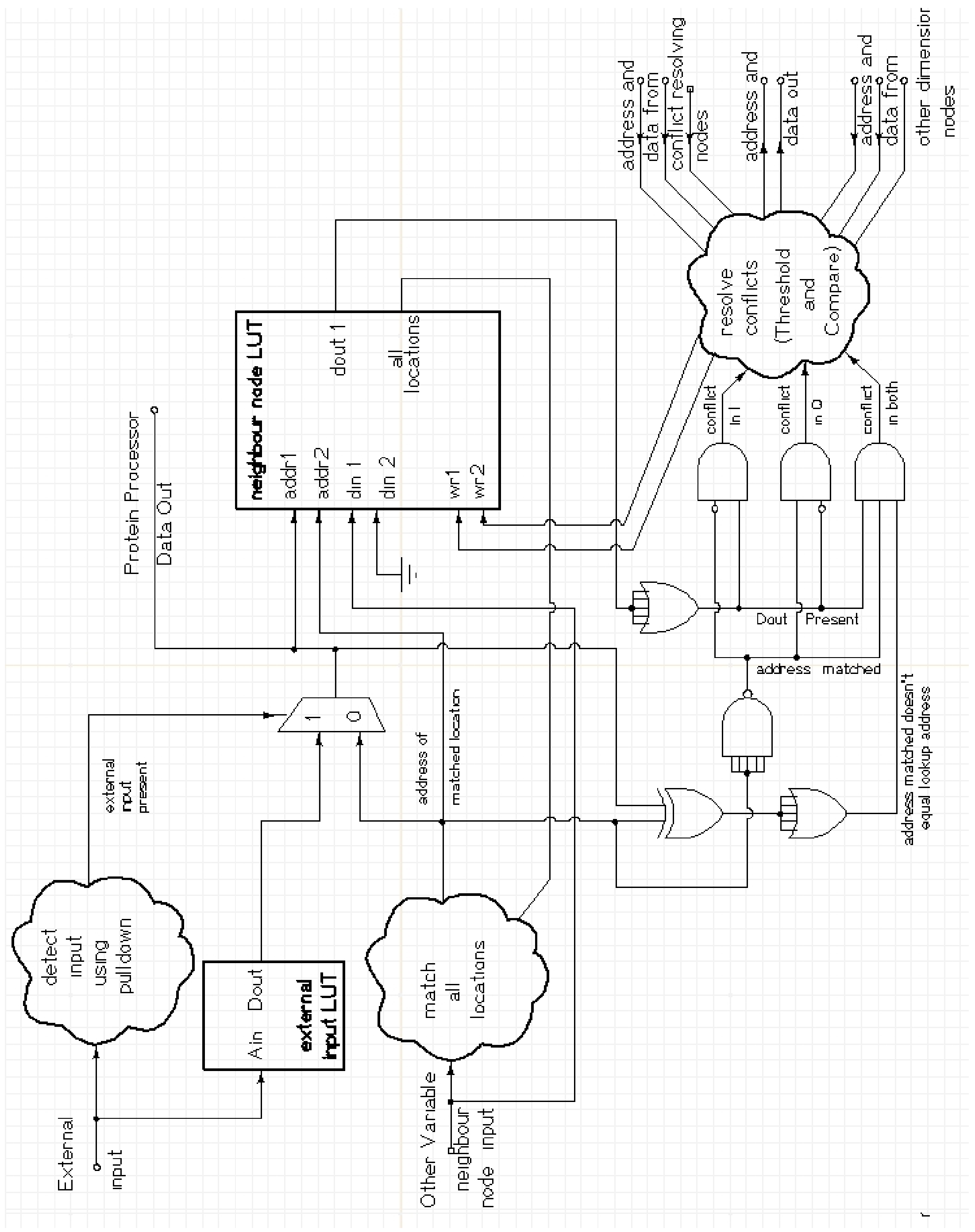

External input
detect input using pulldown
external input present
Ain
Dout
external input LUT
1
0
Protein Processor Data Out
neighbour node LUT
addr1
addr2
din 1
din 2
wr1
wr2
dout 1
all locations
address of matched location
match all locations
Other Variable neighbour node input
Dout
Present
address matched
address matched doesn't equal lookup address
conflict In I
conflict in O
conflict in both
resolve conflicts (Threshold and Compare)
address and data from conflict resolving nodes
address and data out
address and data from other dimension nodes

Figure 6.3: Node schematic

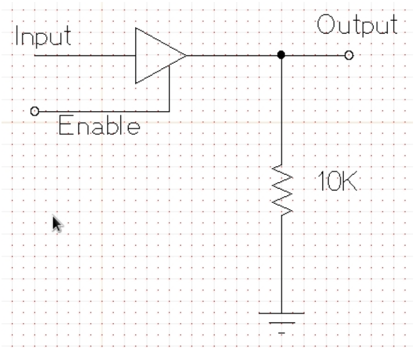


Figure 6.4: Detect input

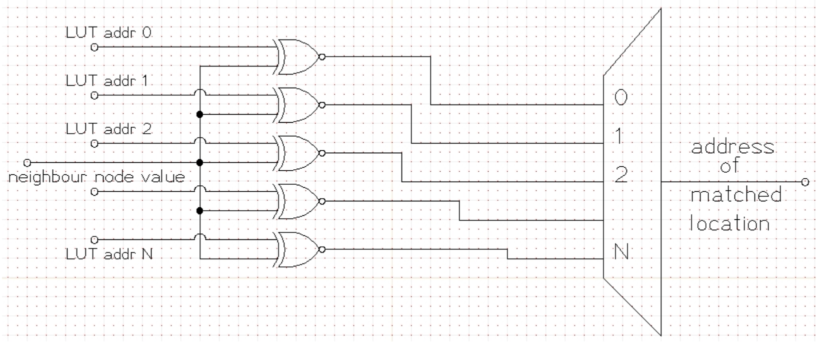


Figure 6.5: Match all locations

and some tuples need to be removed to update the memory in a node (as explained in Section 6.2.4). As mentioned in Section 6.3, in the worst case, this could result in up to D tuples having to be removed. Therefore, in the worst case, the LUT needs to have up to D write ports. This is one area of weakness and will need to be addressed in future designs for example by pipelining the write operations to the LUT. Options for this are explored in Chapter 8.

Conflicts are detected using a small set of gates as shown in the bottom right section of Figure 6.3. A conflict is defined as having been detected if there is a *conflict-in-input* or a *conflict-in-output*. A *conflict in input* is defined as the case when the "match all locations" circuit detects a match (the other variable value is detected as being present at some address in the neighbour node LUT), but the address of the matched value is not the same as the external input to the node. A *conflict in output* is defined as the case when there is data present at the input address (external input) but it does not match the value of the other variable. Once a conflict is detected, the conflict resolving circuit determines whether to update the node to the new tuple or to maintain the old tuple. The conflict resolving circuit is based on the "threshold and compare" logic described earlier in Section 6.3.

Two methods for implementing this circuit are considered. The first is based on analogue methods which are deemed to be better suited for the operation but require mixed signal design techniques; while the second is based entirely on digital logic which has the advantage of being easier to implement with greater support in terms of tool-chains and also transistor level design. The advantages and disadvantages of both are considered, however only the digital logic design was actually implemented, due to the limitations of current FPGA architectures (which only allow digital logic design).

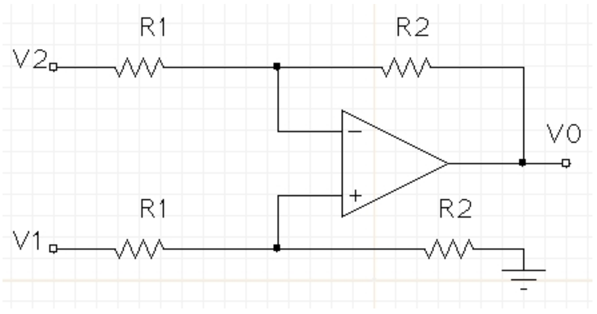
6.3.1.1 Analogue Threshold and Compare

The analogue threshold and compare operates as follows. A Protein Processor node receives address and data values from its Conflict-Resolving neighbours as an analogue voltage. The node compares this value with itself by *subtracting* the 2 voltages. A regenerative amplifier is used at the output so that if the result is small, it tends to zero. The output is then *inverted*. This is performed for all the values received from all the Conflict-Resolving neighbours and these results are *added* together. *Half* of the sum is *stored*. As the clock edges arrive, each node decides to update or not depending upon whether the current result of the summed values is *greater than* the stored threshold. Note that it is essential that the summation circuit should respond faster than the clock skew between Conflict-Resolving nodes.

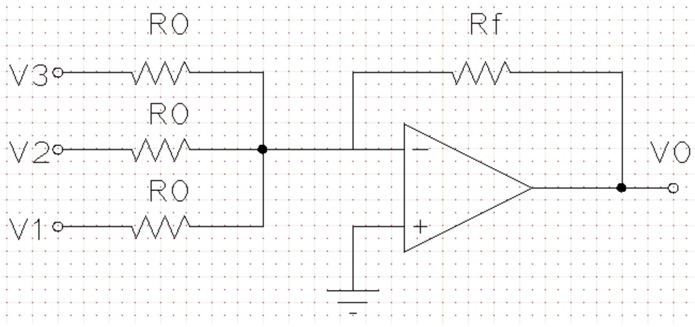
The italicised words represent the operations that need to be performed in the analogue domain; these are:

- Analogue subtracter (Regenerative)
- Analogue inverter
- Analogue adder
- Analogue potential divider
- Analogue comparator
- A to D and D to A converters

The analogue circuit is not presented as a novel design but rather as a rehash of existing analogue circuits. Figure 6.6 shows a portion of the analogue threshold and compare circuit for illustration. The equations governing the operation of these circuits is presented as



(a) Subtracter



(b) Adder

Figure 6.6: Analogue circuits for threshold and compare

Equations 6.2 and 6.3 respectively.

$$V_0 = \frac{R_2}{R_1}(V_1 - V_2) \tag{6.2}$$

$$V_0 = -\frac{R_f}{R_0}(V_1 + V_2 + V3) \tag{6.3}$$

The thresholding is based on the well studied and frequently used sample-and-hold circuit and the voltage adder. The voltage adder is a more natural choice rather than the digital adder for a number of reasons. Firstly, it better mimics the operation of the biological counterparts which are the source of inspiration (Section 4.3.1). The output of the voltage adder is a voltage which increases as the voltages are summed, which is similar to the way neuronal synapses accumulate chemical charge until a threshold level is reached and the synapse fires. The digital implementation, on the other hand, uses a binary representation with each bit using the same voltage to represent logic levels. Therefore, only when all these bits are observed together, is the output of the digital adder seen to be performing the same operation.

Secondly, the analogue circuit ends up using far fewer transistors and what is more important, very few physical connections. It must be noted that analogue transistors are much larger than their digital counterpart and coupled with the extra hardware required for mixed signal design, this usually outweighs the considerations for making analogue designs. In the case of the PPAM however, there is the added consideration of the number of connections between nodes, which becomes more important as the network is scaled. An N-bit symbol may be transmitted using N parallel lines or serially over a single line. The network topology described in Section 6.2.2 (Figures 6.1 and 6.2) shows that using the digital logic parallel communication method is infeasible as the number of connections per node becomes prohibitive with any realistically sized network. Furthermore, using the digital logic serial method has the disadvantage that it significantly increases the complexity and size of the hardware and also the time required to transmit data, particularly as the network is scaled. These are described in more detail below.

Complexity and Size Serial communication involves the use of registers (typically shift registers) to store the data as it is received. It also requires a state machine to control the overall communication which could, in the best case, be a binary counter (the simplest form of a state machine). Note that counters include adders. The number of flip-flops in the (shift) registers and the counter are dependent upon the width of the symbols; 16-bit symbols are quite conceivable. Furthermore, from Figure 6.2, it can be seen that a node communicates simultaneously with C Conflict-Resolving nodes and one Other-Dimension node. Therefore, given that symbols are represented using N-bits, each node must be composed of $C + 1$ serial communicators, each of which would be composed of an N-bit shift register and a ($\log_2 N$)-bit counter.

Timing and Power The communication between Other-Dimension nodes (horizontal neighbours in Figure 6.2) is performed sequentially. This means it takes a minimum of $D + 1$ *steps* for a node to make a decision when updating its value. Since the PPAM is expected to be dealing with real-world events which typically don't occur in the range of MHz, it is expected that a slow clock signal would be sufficient for counting (or clocking) these steps. However, serial communication is also dependent upon "clocking in" data as it is received. Assuming 16-bit symbols, and the simplest serial communication with 1 start-bit and 1 stop-bit, this would mean that the clock would now have to be at least 18 times faster to be able to meet the same timing requirements. Since power dissipation is directly related to signal transition, using serial communication would have a major impact on this as well – the exact value is dependent on the number of transistors, the length of interconnects, capacitance and other implementation specific factors. Furthermore, in the absence of stimulus from external inputs, a node uses the outputs of Other-Variable neighbour nodes to recall the associated value. If this communication is also done over serial lines, this would mean that the clock input would be required for recall operations as well. This would have a serious impact on the nature and features of the PPAM. Previously the clock input was only required for a slow memory update and a fast memory recall could be performed asynchronously and with low power dissipation since it was done without a clock input. However, with serial communication, it would now be subject to all the considerations of traditional digital logic design which would require a fast clock for fast memory recall, thereby further exacerbating the issue of timing and power. For these reasons, the digital alternative, though possible, is considered less desirable.

6.3.1.2 Digital Threshold and Compare

The digital logic threshold and compare is based on counters. Even though the parallel communication option is prohibitive for large PPAM networks, this method was chosen in order to closely mimic the analogue circuit behaviour and also to ensure asynchronous recall. The digital threshold and compare circuit is based on counters that receive values from Conflict-Resolving neighbours and compare them with their own values to generate a count c of the number of nodes that store redundant information. In the presence of conflicts, a threshold t is used to decide if a node updates to the new value, or maintains the old one. This threshold is stored locally in each node and is defined as $t = c/2$. The count c is generated combinatorially, and is constantly updated to represent the current number of nodes that store redundant information. As nodes update to the new value, the value of c in a node that has not updated decreases since fewer of its Conflict-Resolving neighbours now store the old information. In this way, for all nodes where c drops below t, the old value is maintained. Since t is half of c, half of the nodes with redundant information update to the new value. Implicit in this description is that the Conflict-Resolving nodes must update in a staggered manner and not simultaneously. Updating to the new value means writing to the neighbour-node LUT (Figure 6.3), which is done at the edge of a clock. Therefore, it is essential that Conflict-Resolving nodes in the PPAM receive skewed (delayed) versions of each other's clock; although there is no such requirement for other nodes. This means that for the network shown in Figure 6.2, each row can operate on one clock but this must be a skewed version of the clock in another row. Note that adding serial communication between nodes to such an architecture would have a major impact on the complexity of the clock relationships and consequently the hardware design. It would be necessary to consider the nodes as operating on clocks that are not synchronised with each other and the serial communication as asynchronous therefore adding further complexity to the design. In addition, the combinatorial circuit for generating count c would become much more complex since it would not be able to depend upon continuously generated values from the Conflict-Resolving nodes. Instead, a handshake would need to be established to ensure that updated values from the Conflict-Resolving neighbours have been received before a node reads the current count c and makes its decision to update or not.

Note that in traditional logic design, clock skew is seen as a unwanted side-effect of large designs and considerable effort and expense is usually undertaken to remedy this, as designs are scaled. The PPAM, on the other hand, embraces and utilises clock skew thereby making it more easily scalable. Furthermore, although this implementation is based on counters, which are based on adders, these adders are not being used to perform addition operations on the *data*; they are being used to count the number of nodes. The difference may be subtle, but it is crucial for the definition of the computation paradigm which required computation to be implemented without arithmetic operators.

6.3.2 Digital Logic Synthesis and Implementation Results

As stated before, the PPAM is a distributed system where each node makes its own decision to update its memory contents or not. Nodes make their decision in $D+1$ cycles, where D is the number of dimensions in the other variable. Thus, $D+1$ clock cycles must be applied after each new tuple is input to the PPAM before it can store the associations correctly. The digital logic version of the threshold and compare operation is based on a staggered clock design as described in Section 6.3.1.2.

Including full schematics of the hardware architecture or a listing of the HDL code would not necessarily add to the understanding of the reader, therefore these have been excluded from this thesis. However, a fully parameterized HDL (verilog) code including the test bench is available online[2] for the interested reader. The following sections present results from the synthesis of this code.

Elements	used	percentage
External IOBs	62 out of 112	55
Flops	4	
Latches	0	
IOBs driving Global Buffers	1 out of 8	12
CLBs	163 out of 196	83
Latches	0 out of 392	0
CLB Flops	36 out of 392	9
4 input LUTs	306 out of 392	78
3 input LUTs	34 out of 196	17
BUFGLSs	2 out of 8	25
STARTUPs	1 out of 1	100

Table 6.7: XCS10XL device utilisation summary

6.3.2.1 Implementation on FPGAs

The HDL code for PPAM was synthesized for two different Xilinx FPGA architectures. The first was the XCS10XL of which there are 2000 FPGAs in the BioWall (Tempesti et al., 2002). A single node was synthesized per FPGA, and the parameters were defined as follows: The number of Conflict-Resolving nodes (C) is set to 7, and the number of bits to encode symbols was 2. In addition, each node implemented 16 locations deep memory. The device utilisation summary is presented in table 6.7.

The second FPGA used was the one on the Xilinx University Program[3] development board XUPV5LX110T. The entire PPAM was implemented on one Virtex 5 FPGA. Implementing a PPAM for the sample dataset presented in section 6.2.1, each node implemented a 4 bit wide and 16 locations deep memory (symbols encoded in 4-bit binary). The number of Conflict-Resolving nodes was set to 6. The device utilisation summary is presented in table 6.8. The resource utilisation is shown in table 6.9.

[2] `http://www-users.york.ac.uk/~oq500/blog.html#PPAMverilogHDL`
[3] `http://www.xilinx.com/university/`

Elements	used	percentage
Slice Logic Utilization		
Slice Registers	3648	5
Slice LUTs	14330	20
used as Logic	14330	20
Slice Logic Distribution		
LUT Flip Flop pairs used	14330	
with an unused Flip Flop	10682	74
with an unused LUT	0	0
fully used LUT-FF pairs	3648	25
unique control sets	774	
IO Utilization		
IOs	314	
bonded IOBs	312	48
Specific Feature Utilization		
BUFG/BUFGCTRLs	12	37
PLL_ADVs	1	16

Table 6.8: 5VLX110T device utilisation summary

6.3.2.2 Implementation on the Unitronics Fabric

The PPAM was also implemented on the Unitronics fabric being developed by the project partners at UWE Bristol as part of the SABRE project (Chapter 1). For details about the fabric, its features and capabilities, refer to Samie et al. (2009a,b). Bremner et al. (2010, 2011a) explore methods of using genetic programming to evolve circuits and make generic synthesis tools for the Unitronics architecture. In its current state, although the Unitronics architecture is theoretically capable of implementing sequential circuits, the synthesis tool is only capable of inferring combinatorial logic. Therefore, the combinatorial logic portion of a single node was extracted from the design and implemented on the fabric. The simplest possible node was attempted to get an idea of the smallest number of resources that would be required. The parameters for such a node are presented in Table 6.10. This resulted in a truth table with 17 inputs and 16 outputs that completely describes the operation of the combinatorial portion of the PPAM; Circuits were evolved for these tables using the synthesis tool at UWE Bristol which uses reduced Unitronics cells as nodes for Cartesian Genetic Programming (CGP) (Bremner et al., 2011b). There are three available node configurations:

- 3-input LUT
- full adder

Elements	used
BELS (Basic Elements of Logic)	14703
GND	1
LUT2	3356
LUT3	634
LUT4	2339
LUT5	2490
LUT6	5511
MUXF7	371
VCC	1
FlipFlops/Latches	3648
FDC (Flip Flop with Async clear)	480
FDC_1 (negedge FDC)	96
FDE (Flip Flop with Clock Enable)	3072
Clock Buffers	12
BUFG	12
IO Buffers	312
IBUF	19
IBUFG	1
OBUF	292
PLL_ADV	1

Table 6.9: 5VLX110T resource utilisation summary

Parameter	Number of Bits
Conflict-Resolving Nodes (C)	2
Variable 1 Dimensions ($D1$)	2
Variable 2 Dimensions ($D2$)	2
Data bus width	1
Address bus width	2

Table 6.10: Parameters for a simple node implemented on the Unitronics fabric

- 3-input LUT with its output multiplexed with an additional node input (*big-nodes*). The multiplexer makes these big node more versatile, but also more resource hungry in terms of routing.

Mutation operations change

- node type (3-input LUT, Full adder or 3-input LUT with extra MUX)
- LUT contents
- node connections

Multi-Objective Optimisation (MOO) is run as a second step after functional correctness has been achieved. The objectives are to minimise the total number of nodes and also the number of big-nodes; the aim being to minimise the Unitronics resource utilisation. Using 10 independent MOO runs for each output in the truth table, resource utilisation was calculated to be either 56 nodes, none of which are big-nodes, or 47 nodes, 7 of which are big-nodes. Optimal routing software is still under development at UWE Bristol therefore, as yet, it is uncertain which of these two options will result in the smallest routed implementation. Note that this implementation on Unitronics, including CGP implementations and MOO, was performed by the project partners at BRL, which is why details of these experiments are not included in this thesis.

6.4 Summary

PPAM implementations on FPGAs are handicapped because of their inability to customise RAM blocks (in the way required by the PPAM) and also because of the lack of analogue resources. However, due to cost and time considerations, ASIC implementations were not attempted and instead, FPGA implementation results are used to estimate granularity.

The exact number of nodes that may be packed in an FPGA is dependent upon:

- the size of the nodes, which is influenced by the size of the custom LUT inside each node and the width of the abstract symbols and so on;
- the network configuration of the PPAM – i.e. values of the various parameters, like number of dimensions and the number of Conflict-Resolving nodes and so on.

Experiments with various configurations indicate that using the Virtex 5 LX110T FPGA, it is possible to fit at least 200 nodes with 32 memory locations in each node. A PPAM of this size can be used to associate variables with low dimensionality; therefore it should be able to perform basic reflex-action-like tasks – for example, object avoidance during robot navigation, or controlling the movements of a robotic arm and so on. Cascading 10 such FPGAs would generate a PPAM of around 1000 Protein Processors. This should be capable of associating variables with medium sized dimensionality and should be able to perform slightly more complex tasks like Optical Character Recognition and so on.

32 such FPGAs would generate a PPAM of around 6400 Protein Processors, capable of associating variables of even higher dimensionality and performing much more complex tasks like associating sounds (encoded as symbols) with images (encoded as symbols).

Correcting for the difference between FPGA and VLSI implementations, synthesis results presented in section 6.3.2 show that the individual Protein Processors do meet the granularity requirements to be able create a large network within the hardware resources of current technology. Section 6.3.2 also shows that using a small number of user-defined constraints specifying false-paths and multi-cycle paths, the asynchronous logic can be synthesised using a standard commercial synthesis tool (Xilinx ISE) which is not specifically designed for implementing such logic. Thus, it has been shown that this improved version of the PPAM (as presented in this chapter) is more suitable for hardware implementation. Furthermore, the architecture is designed for scalability as implementing the PPAM on higher dimensional variables only requires the following 2 changes:

- Increase the number of nodes in the network and
- Increase the number of clock cycles required to update the memory.

In terms of HDL code, it is simply a matter of changing parameter values. Chapter 7 details the experiments performed by simulating this HDL code to explore the performance of the PPAM and also presents the results obtained.

Chapter 7

Profiling the PPAM

Chapter 6 presented the modified version for the Protein Processor Associative Memory (PPAM) which was shown to be better suited for a hardware implementation. It also presented various options for a hardware implementation along with a discussion of the advantages and disadvantages of each. This chapter presents the results obtained from experiments performed to explore the performance and capacity of the PPAM on different types of datasets. Section 7.1 tests the performance and capacity of the PPAM using a simple toy dataset that can be described using a simple mathematical formula and would typically not be considered an AI application. Section 7.2 further explores the behaviour of the PPAM using a larger, more realistic dataset obtained from a physical robotic arm-and-vision system experiments performed by researchers at the University of Aberystwyth. Section 7.3 verifies the inferences and observations from the experiments in simulation by learning the inverse kinematics for the BERT2-Vicon system at the Bristol Robotics Laboratory (BRL) and moving the robotic arm in the physical world to observe the effects on performance of quantisation and noise from the real world. Finally, Section 7.4 presents results from experiments performed to test the robustness of the PPAM in the presence of faults.

7.1 Memory Capacity on a Toy Dataset

A simple toy dataset was presented in Section 6.2.1 of Chapter 6, and Section 6.2.4 used this dataset as a worked example to illustrate the operation of the PPAM. As a first step, it is useful to apply the PPAM on this dataset and ensure that the results are the same as predicted in the worked example. Therefore, this dataset is used in the experiments outlined here. The objectives are as follows:

- Verify the results in the worked example through simulation.
- Vary parameters and collect data to analyse the performance of the PPAM.
- Implement the architecture on an FPGA and compare the results of the physical implementation with results from simulation. This increases confidence in future simulation results as being true reflections of the implemented hardware.

The key features of the sample dataset are repeated here to briefly summarise the dataset and the 16 element dataset originally presented in Table 6.4 is repeated in Table 7.1. The

Cartesian Coordinates		**Polar Coordinates**	
X	**Y**	**R**	θ
1	5	0x4	0xA
1	6	0x4	0x9
1	7	0x3	0x9
1	8	0x2	0x9
2	5	0x4	0xC
2	6	0x3	0xA
2	7	0x2	0xA
2	8	0x2	0x9
3	5	0x3	0xC
3	6	0x2	0xB
3	7	0x2	0xA
3	8	0x1	0xA
4	5	0x2	0xC
4	6	0x2	0xC
4	7	0x1	0xB
4	8	0x1	0xA

Table 7.1: Cartesian-polar encoded dataset

16 element dataset associates Cartesian coordinates (X, Y) with the polar coordinates (R, θ), both of which are two-dimensional variables whose relationship can be described with the simple equations in 6.1. The following section (7.1.1) explains the PPAM setup used in the experiments.

7.1.1 Experiment Setup

A 4-bit data encoding was used (which is sufficient for the data in table 7.1), while each Protein Processor was configured with 16 locations in the custom lookup-table. The number of conflict-resolving nodes (C) was increased iteratively from 2 up to 7 for each experiment. For each PPAM configuration, 15 experiments were conducted such that, in each experiment, the system attempted to store (and recall) an increasing number of tuples from the dataset. Thus, for example, in the case of the PPAM configuration where $C = 2$, first 2 tuples were stored (and recalled), then 3, then 4 and so on. The performance is measured by assigning a *score* for each experiment. The score is calculated by giving each *complete recall* 2 marks and each *partial recall* 1 mark. A tuple is considered to be completely recalled if the values recalled for all the dimensions (two in this case) are correct. A tuple is considered to be partially recalled if the values for at least half the dimensions (one in this case) are correct. For these experiments, only values recalled with a *confidence* of 1 are considered, where confidence is measured as the number of Other-Dimension nodes that recall a value divided by the total number of Other-Dimension nodes. Since the variables being associated are both two-dimensional, this means that if both nodes in one row of Figure 6.2 (repeated here as Figure 7.1) reinforce each other's output, the value is recalled with a confidence of 1.

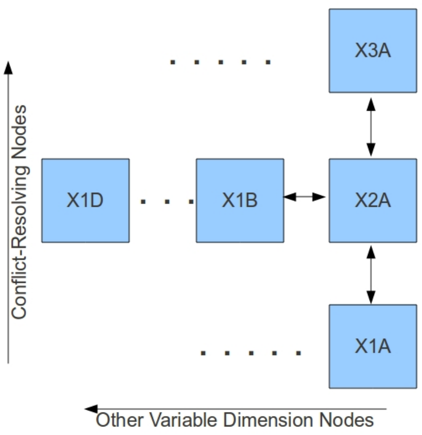


Figure 7.1: Abstract view of nodes

7.1.2 Results

Table 7.2 and Table 7.3 show the results for the 15 experiments where $C = 2$ and $C = 3$ respectively.

Dataset Size	Complete Recall	Partial Recall	Score
02	2	0	04
03	2	1	05
04	2	1	05
05	2	1	05
06	2	2	06
07	2	3	07
08	2	5	09
09	3	5	11
10	4	5	13
11	4	6	14
12	4	6	14
13	4	6	14
14	4	6	14
15	4	7	15
16	4	7	15

Table 7.2: PPAM results for the Cartesian-polar coordinate dataset ($C = 2$)

Dataset Size	Complete Recall	Partial Recall	Score
02	2	0	04
03	3	0	06
04	3	1	07
05	4	1	09
06	5	1	11
07	5	2	12
08	6	2	14
09	7	2	16
10	8	2	18
11	8	3	19
12	8	3	19
13	8	4	20
14	8	4	20
15	9	4	22
16	9	6	24

Table 7.3: PPAM results for the Cartesian-polar coordinate dataset ($C = 3$)

Each row indicates one experiment where *Dataset Size* is the number of tuples presented to the memory for storage. Figure 7.2 shows how complete recall of a PPAM configuration increases as the number of conflict resolving nodes is increased, while figure 7.3 and figure

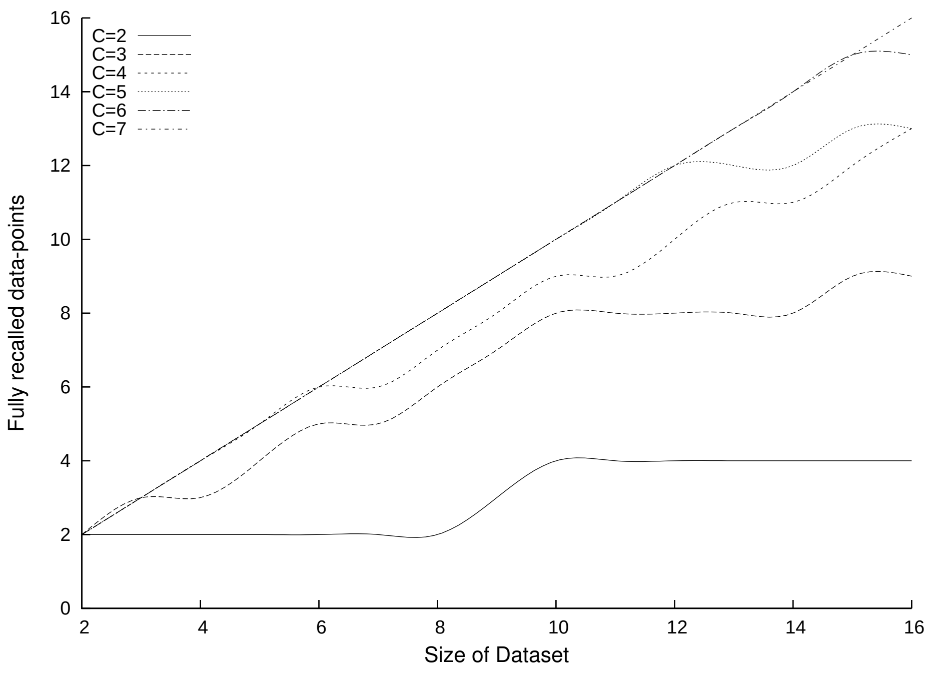


Figure 7.2: PPAM complete recall capacity for various C values

7.4 plot *partial recall* and *score* respectively. The configuration where $C = 7$ provides sufficient memory capacity to store all tuples in the dataset completely. This can be seen from the fact that the plots for $C = 7$ are a straight line defined by $y = x$ in figures 7.2 and 7.4, and a straight line defined by $y = 0$ for figure 7.3, showing a perfect score, i.e. 100% complete recall and no partial recall.

7.1.3 Discussion

As can be seen from Table 7.2 the highest score for $C = 2$ with 16 elements is 15 while increasing C by 1 increases the score to 24. The memory capacity of the PPAM can be seen to increase quite quickly with C so that the configuration where $C = 4$ scores 29 out of a maximum of 32. However, as can be seen from Figure 7.5, this effect slows down with increasing C.

The dip at $C = 5$ suggests over-training, however, increasing C beyond 5 overcomes this and memory capacity continues to increase. Therefore, it is considered to be insignificant. In addition, note that Figure 7.5 has been drawn for only one direction of association. This means that after training, the Cartesian coordinates were recalled by presenting the polar coordinates as input but not the other way around as well. Table 7.4 presents the

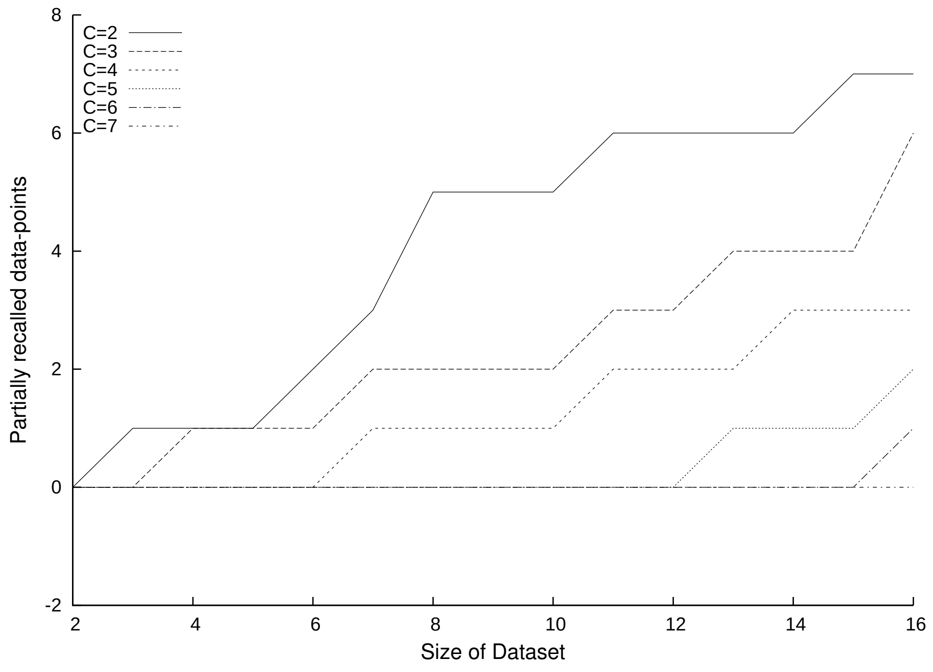


Figure 7.3: PPAM partial recall capacity, for various C values

results for when data is recalled in both directions and it can be seen that performance increases with C as expected.

C	Complete Recall	Partial Recall	Score
4	23	7	53
5	25	4	54

Table 7.4: 16 element Cartesian-polar coordinate dataset results for bidirectional association

Figure 7.6 shows the total number of memory locations used by all the nodes in the different PPAM configurations. In the worst case ($C = 7$) the total number of memory locations needed by all the nodes was 152. Since each memory location in these experiments was 4-bits wide, this means that the PPAM needs 76 bytes to store and fully recall the tuples in this dataset.

Post place-and-route simulations were conducted on this dataset to verify the hardware implementability of this architecture. In addition, a Virtex 5 LX110T FPGA (on the XUP development board) was programmed and the operation of the system was verified. Since this design had a maximum of $C = 7$, this meant that 7 independent clocks were required (as explained in Sections 6.3.1 and 6.3.2), which is a feasible target using the Virtex 5 series FPGAs. FPGA experiments confirm the results presented in this section.

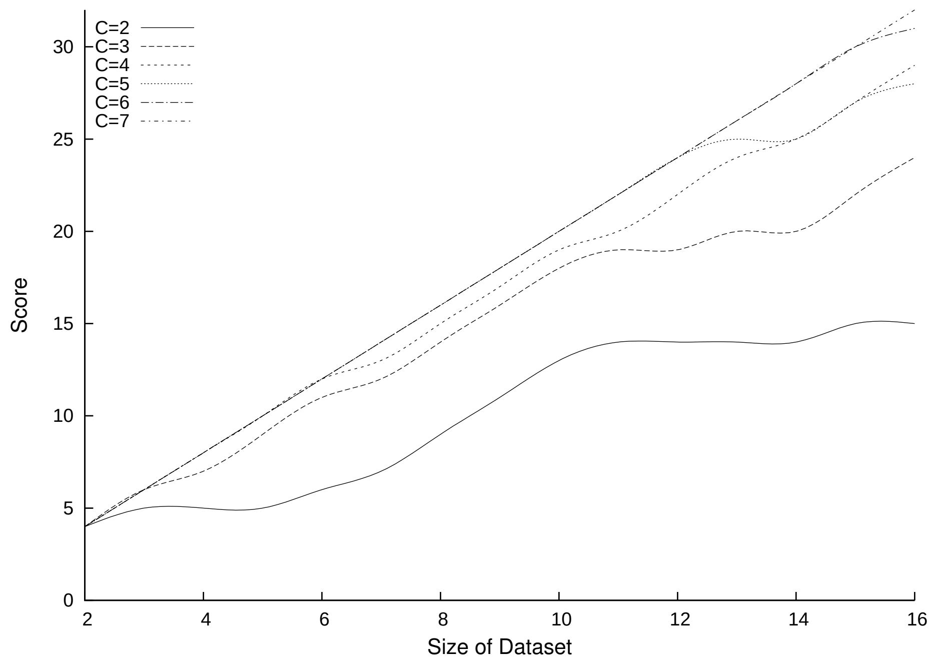


Figure 7.4: Capacity score (2 × complete recall + partial recall), for various PPAM configurations (C values)

These have not been presented here because black-box experiments were conducted with no information about the inner processing of data, in order to simply verify the simulation results.

Although this toy dataset was a useful first experiment, realistic datasets would be much more complex. Furthermore, although the PPAM can be used to associate the variables in this dataset, the PPAM suffers in comparison with other techniques since the dataset is too simple and small to display the strengths of the PPAM. In addition, a larger dataset is needed to have more confidence in the inferences made from the observations in this experiment. In the next sections, more complex and realistic datasets are used to explore the performance of the PPAM.

7.2 Performance on Large Datasets – Catcher-in-the-Rye

The *catcher-in-the-rye* dataset was used in order to test the performance of the PPAM on a larger, more complex dataset that performs a real-world task. The catcher-in-the-

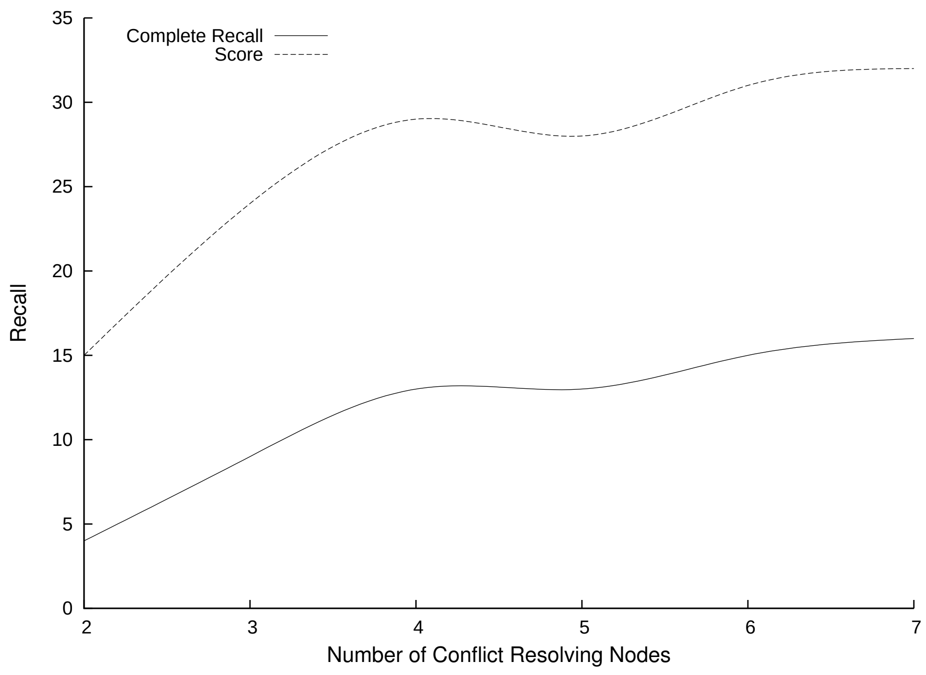


Figure 7.5: Effect of C on capacity – score for the 16 element dataset for each PPAM configuration.

rye project (Hülse et al., 2009, Hülse et al., 2010) was conducted at the University of Aberystwyth and essentially performed a hand-eye coordination task. A robotic arm places objects in (and retrieves them from) its *reach space*, while a stereo vision system tries to identify the location of the object in its *vision space*. The task was to associate the vision space with the reach space so that when one is known, the other can be *recalled*. The hardware platform described in Hülse et al. (2009), Hülse et al. (2010) was used at the University of Aberystwyth for this task and is the source of the dataset used in this section. The objectives of applying the PPAM to this dataset are:

1. To test if *any* configuration of the PPAM can accurately store and recall all the elements of this larger and more realistic dataset.
2. To compare the number and types of operations performed by the PPAM as opposed to other more traditional techniques.
3. To verify the inferences drawn about memory usage from experiments in Section 7.1.

Section 7.2.1 describes the experimental setup at the university of Aberystwyth, while 7.2.2 presents the results of the performance of the PPAM on this dataset and compares the number and type of operations required to achieve similar results. Section 7.2.3 ends

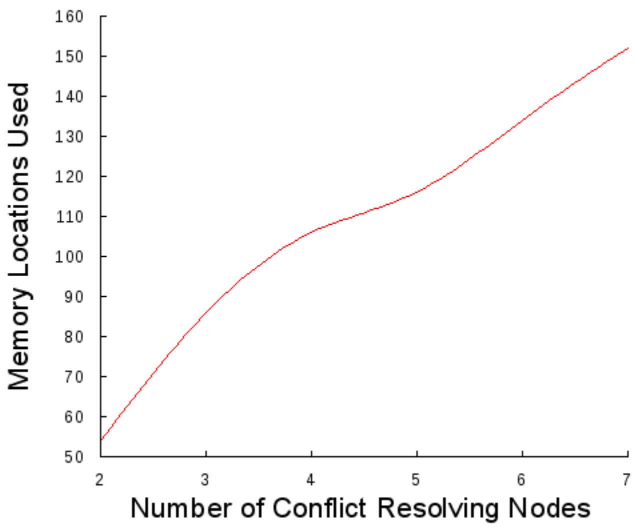


Figure 7.6: Total number of memory locations used

with some thoughts on the larger implications of this on the hardware resources required for such a task.

7.2.1 Experiment Setup

Figure 7.7 illustrates the setup used to perform the experiment in the original Catcher-in-the-rye project (Hülse et al., 2009, Hülse et al., 2010). As mentioned above, the task is to associate the vision space of the camera with the reach space of the robotic arm. Thus, when one is known, the other can be *recalled*. The vision system is mounted on a

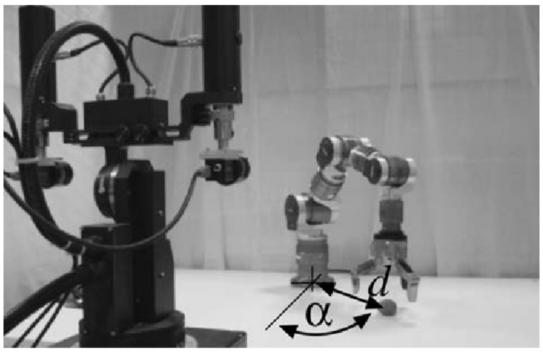


Figure 7.7: Robotic arm and camera setup (Hülse et al., 2010)

pan–tilt–verge unit and is composed of 2 cameras, each providing 1032 × 778 RGB pixels at the rate of 25 fps. Hülse et al. (2010) reduced the images to 129 × 62 pixels by taking the mean of 8 × 8 pixel blocks. Object detection was performed by filtering the image for blue colour, and tilt–verge movements were performed in the camera to saccade to the detected object. Since the pan movement for the camera is not used, the vision system has 3 degrees of freedom (one verge movement for each of 2 cameras and one tilt movement for both cameras together) and thus 3 dimensions in the variable. Although the robotic arm has 7 degrees of freedom, the location of an object in the *reach space* of the arm can be represented using polar coordinates on a 2-dimensional plane. Thus the second variable has 2 dimensions. Therefore, each entry in the dataset is composed of two variables; the first is a 3-dimensional variable representing motor control for saccading the vision system and the second is a 2-dimensional variable representing polar coordinates for the robotic arm reach location.

7.2.1.1 Original Catcher-in-the-rye

The approach used by Hülse et al. (2010) is to store selected associative pairs from the dataset and use Euclidean distance to recall the closest matching pair. Each associative pair is tested to see if it should be added to the *map* and if so a new *link* is created. Old links that haven't been refreshed in a specified number of cycles are removed from the map. The algorithm is summarised in Algorithm 3.

Algorithm 3: Pseudo-code for Catcher in the Rye inferred from the publications about the project

```
// $T$ is Tolerance for acceptable error in mapping
// $Q$ is age threshold for old links
// $M$ is map of links -- empty initially
// $V$ is the 3 dimensional variable representing the vision system movement
// $R$ is the 2 dimensional variable representing the Robotic arm movement
// $N$ is the total number of elements in the training set
for each of $N$ elements in the training set do
    Arm places object at a known position;
    Vision saccades to focus on the object;
    for $i \leftarrow 0$ to $N$ do
        // Calculate Euclidean distance between $V$ and $M[i]$
        $D \leftarrow V - M[i]$;
        if $D <= T$ then
            break loop as link found;
        if age of $M[i] > Q$ then
            remove $M[i]$ from $M$;
    if link found then
        zero age of link;
    else
        add a new link $\{V, R\}$ to $M$;
```

The location of objects in the reach space is represented by polar coordinates (d, α) (as shown in Figure 7.7), that range as follows: $30 \leq d \leq 60$ cm; $-1.4 \leq \alpha \leq 1.4$ radians. This defines a total area of $3,944cm^2$. The distance D between points in the reach space in Algorithm 3 can be measured according to equation 7.1, which can be seen to include several mathematical operations.

$$D = \sqrt{[d_1 \cdot sin\alpha_1 - d_2 \cdot sin\alpha_2]^2 + [d_1 \cdot cos\alpha_1 - d_2 \cdot cos\alpha_2]^2} \qquad (7.1)$$

Benchmarks to evaluate the computational cost of trigonometric functions vary greatly, depending upon the hardware architecture and the method used to compute the function. Intel architectures are used as the standard here, since they provide hardware support for calculating trigonometric functions using native instructions (i.e. *fsin, fcos* and *fsincos*). Despite the hardware support however, Intel (2011) shows that clock latency for sine is 160–200 cycles with a throughput of 130 cycles; the latency for cosine is 180–280 cycles with a throughput of 130 cycles; and the latency for sine and cosine together (of the same angle) is 170–250 cycles with a throughput of 140 cycles. Here Intel (2011) defines clock latency as the number of cycles to complete execution of the instruction and throughput as the number of cycles to wait before the instruction can be issued again. The exact number of cycles is dependent upon the range of the input data and the processor model. This estimate does not include the overhead from the branch instructions or the load/store instructions required to manipulate the data. Furthermore, the calculations are performed on floating point numbers, thus requiring a Floating Point Unit. Hülse et al. (2010) uses a training set of 300 elements, and shows that setting $Q = 300$ and $T = 0.0$ produces the best results, since these values result in the map storing all the associative pairs in the training set. Thus, for an arbitrary associative pair, at most N (300 in this case) iterations of the loop (with trigonometric calculations) may need to be executed to find the closest matching link. A more detailed comparison of the number and type of instructions is presented in Section 7.2.3.

7.2.1.2 PPAM

The 300 element dataset used by the catcher-in-the-rye experiments described in Section 7.2.1.1, was used as input to the associative memory. The dataset was composed of 3-dimensional floating point numbers for camera movement and 2-dimensional floating point numbers for arm movement. Tables 7.5 and 7.6 present the relevant features of the dataset for the camera movement variable and the robot arm movement respectively. The data

Property	**tilt**	**verge left**	**verge right**
Min	-0.486	-0.334	-0.524
Max	-0.322	0.436	0.247
Range	0.164	0.770	0.771
number of discrete values	164	770	771

Table 7.5: Camera movement dataset features

Property	**distance** (d)	**angle** (α)
Min	33.0	-01.40
Max	58.0	01.40
Range	25.0	02.80
number of discrete values	25	280

Table 7.6: Robot arm movement dataset features

captured as part of the original Catcher-in-the-rye experiment had been rounded to varying precision by by Hülse et al. (2010). Camera movement data was rounded to 3 decimal places and the angle α in robotic arm movement was rounded to 2 decimal places. The distance d did not need to be rounded. The data was encoded into abstract symbols as follows:

- Each point in the data was normalised by subtracting the corresponding minimum value and then converted to abstract integer symbols.
- All the data was multiplied by 1000 to convert it to integer numbers (each of which represents an abstract symbol).

At first glance, this might seem like the PPAM is simply off-loading the computation (for encoding from real numbers to integers) to a companion controller, and claiming freedom from floating point units because they are implemented elsewhere in the system. However, an analysis of the system as a whole indicates that this is not the case. Analogue inputs arrive from the physical world and need to be encoded into digital data using some representation (typically 32-bit floating point) before they can be used by the system. Even when using integer representation for inputs, it is still necessary to perform floating point arithmetic because often division is involved. Note that regardless of the representation used, *some* precision will be lost through the process of quantisation, therefore from this perspective alone, both representations are equally valid. Although the process described above of encoding the dataset into abstract symbols appears to be re-encoding the dataset, in a complete PPAM system, the external inputs would not need to be represented as floating point numbers first and could directly be encoded as integers.

Experiments were performed with an increasing number of conflict resolving nodes to determine the number of nodes required to correctly store and recall the complete dataset. Data was collected to profile memory usage for the PPAM and results are presented in Section 7.2.2.

7.2.2 Results

In order to determine the number of nodes required to accurately store and recall the 300 element dataset, experiments were performed using node configurations with 10, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 50, 100 and 200 conflict resolving nodes. This set

C	**Errors**	**Correct Recalls**
10	199	104
20	10	293
21	7	296
22	3	300
23	2	301
24	2	301
25	1	302
26	2	301
27	1	302
28	1	302
29	0	303
30	0	303
50	0	303
100	0	303
200	0	303

Table 7.7: Catcher-in-the-rye results for varying C

was chosen empirically while searching for the value of C (number of Conflict-Resolving nodes) that would accurately recall the entire dataset. This can be thought of as a hill-climbing search with varying step sizes, so that initially, the value of C was varied by large amounts, and later by small amounts. Results are shown in Table 7.7 and Figure 7.8 plots the number of data points that were correctly stored and recalled for the different node configurations.

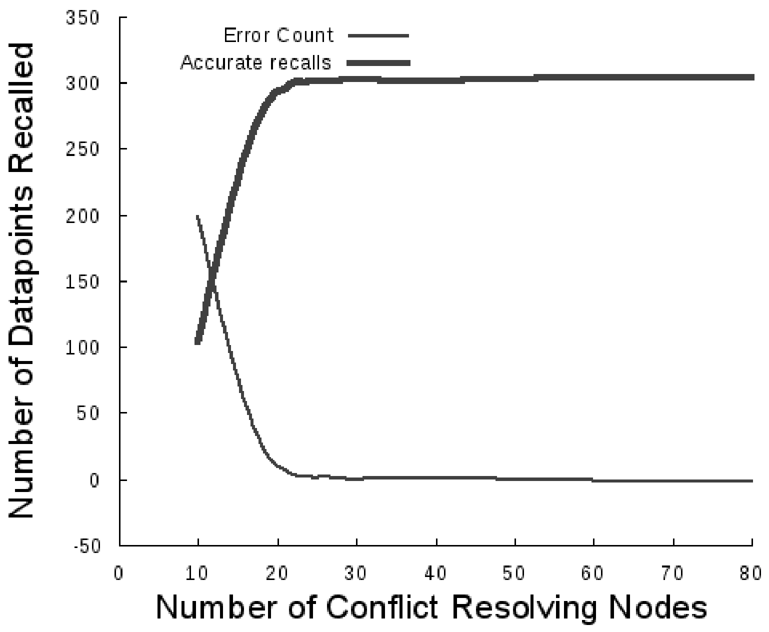
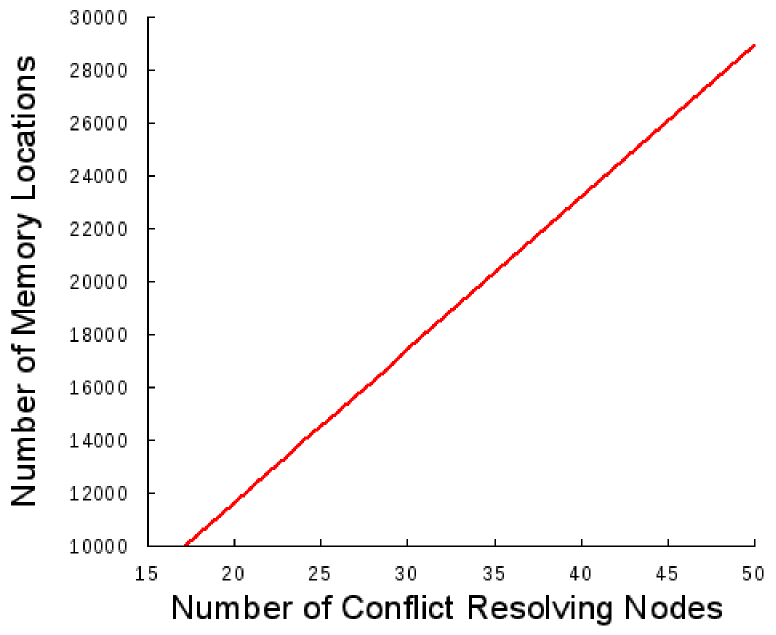


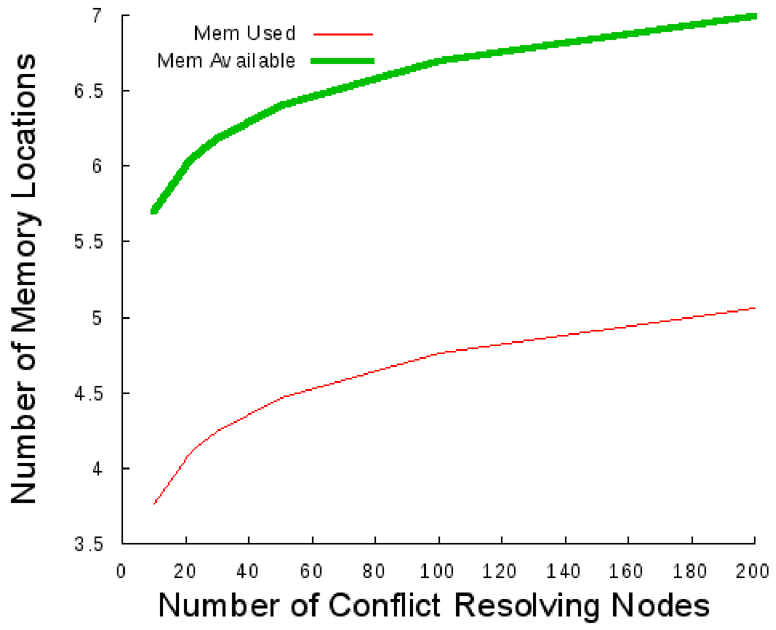
Figure 7.8: Performance of PPAM as C is varied in the configurations. This plots the number of data points that were correctly stored and recalled. The point with maximum recalls and zero errors is with $C = 29$.

As observed for the toy dataset (Section 7.1) adding more conflict resolving nodes initially makes a large impact on the memory capacity but provides diminishing returns. Figure 7.9b shows the total number of memory locations available and the number actually used in the PPAM on a logarithmic scale. This "total number of locations used" is the sum of all memory locations used in each node. The distribution for the number of memory locations used in each node can be seen in Figure 7.10, which shows that the maximum number of memory locations used in *any* node for *any* configuration was 160 whereas the upper quartile is half that and the median is even lower.

Two unexpected results can be seen from the analysis of the memory usage. First, the ratio of "memory locations used", to "the total number of memory locations" in the PPAM is constant irrespective of the size of the PPAM (Figure 7.9b). Second, although the structure of the PPAM dictates that there is a linear relation between the number of nodes and the amount of memory in the PPAM, Figure 7.9a shows that there is also a linear relationship between the number of memory locations actually used and the size of the network. The reason for this can be seen from Figure 7.10 which shows that the distribution of memory utilization per node does not change significantly with respect to the number of conflict resolving nodes used. Thus, increasing the number of nodes



(a) Used (20 to 29)



(b) Log Scale

Figure 7.9: Memory profiling: 7.9a shows the "total number of locations used" (sum of all memory locations used in each node) as C is increased; 7.9b shows the total number of memory locations available and the number actually used in the PPAM on a logarithmic scale.

seems to have a linear effect on the memory utilisation but an exponential effect on the performance (Figure 7.8). This means that increasing the number of nodes initially has a small cost in terms of memory and a large advantage in terms of performance; whereas the opposite is true in order to learn the last few elements of a dataset. Implications of this are discussed further in Section 7.2.3.

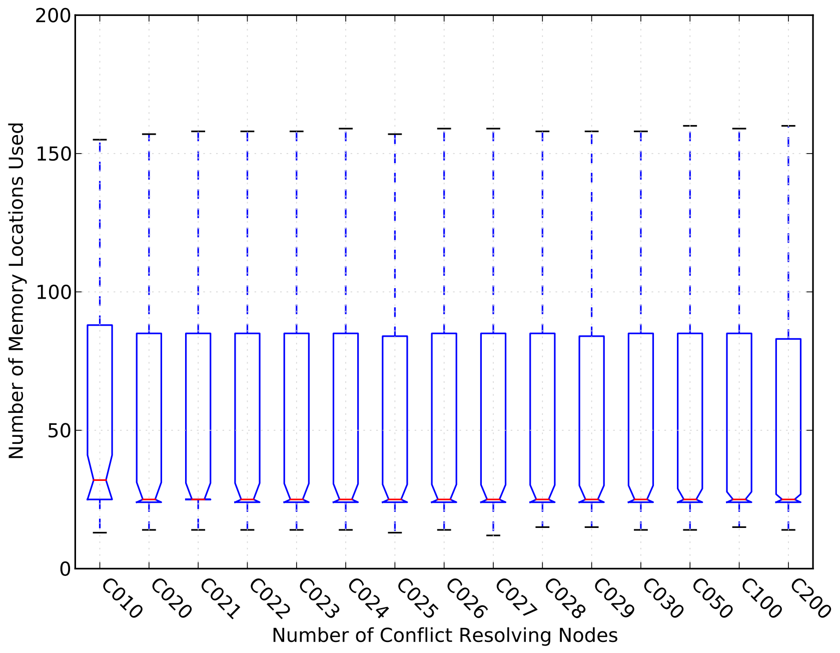


Figure 7.10: Memory locations used per node as C is varied. Note that the theoretical maximum in the Y-Axis is 4096 (12-bit symbols). This plot uses a smaller range for the Y-Axis so that the slight differences between the different values of C become visible.

As can be seen from Figure 7.8, the number of errors approach zero around the 20 conflict resolving node configuration mark. The $C = 20$ node configuration was chosen arbitrarily as representative of this range in order to further analyse it to determine its capacity. A detailed view of the performance for this configuration is shown in Figure 7.11. Note that the optimal configuration (least number of nodes with zero errors) has 29 conflict resolving nodes. Therefore, the 20 node configuration displays a similar performance with 108 fewer nodes and 5194 fewer memory locations (Figure 7.9a). Implications of this are further discussed in the next section.

7.2.3 Discussion

Section 7.2.2 shows that the PPAM is capable of storing and correctly recalling the entire dataset. Table 7.8 compares the number and types of operations used in the PPAM, with

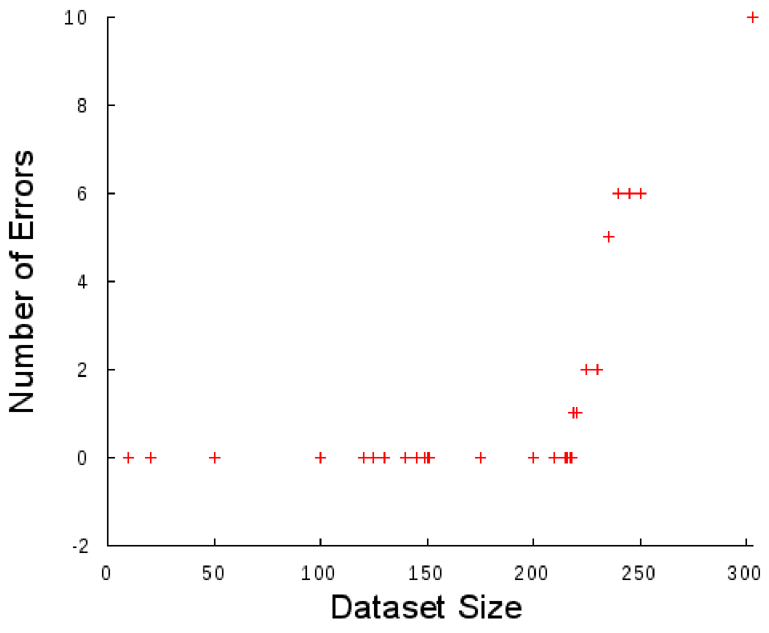


Figure 7.11: Error count for $C = 20$ node PPAM configuration

the more traditional method used by the catcher-in-the-rye. During this comparison, it does not consider the branch operations implicit in any software implementation of the algorithm, since these would be removed if the algorithm was implemented in hardware. The catcher-in-the-rye implementation is assumed to be optimized such that calculations are not repeated and results are considered to be stored in temporary memory (CPU registers). Memory access (read/write) does not include access to these temporary locations or to any other *working* memory. Furthermore, memory requirements (or operations) for stimulating nodes or providing input is not included. In addition, the multiplication and addition operations for calculating indexes for the values in the link-map for the catcher-in-the-rye are also not included in Table 7.8, because this is considered to be similar to the significantly simpler and smaller number of memory index counting operations for the PPAM.

As mentioned in Section 7.2.1, assuming there are 300 elements in the map for the catcher-in-the-rye ($Q = 300$, $T = 0.0$), it can be seen from algorithm 3 that a single memory recall operation will take 300 iterations in the worst case and 1 iteration in the best case.

Assuming an evenly distributed random distribution for the memory recall operations, the average can be assumed to be 150 iterations. Each iteration requires a number of floating point operations, which depend on whether the robot arm movement is being recalled or the vision movement is being recalled. Since the framework used in Hülse et al. (2010) looks up robot arm movements using the vision system as input, the same is used for the purpose of this comparison. Note that this assumption is not required for the PPAM since

Table 7.8: Comparison of Catcher-in-the-rye and PPAM. S is the size of the dataset and H is the number of nodes in the PPAM.

	Catcher-in-the-rye	PPAM
Trigonometric	up to $S \times 4$	0
Multiplications	up to $S \times 6$	0
Square root	up to S	0
Additions	up to S	0
Subtractions	up to $S \times 2$	0
$>$ Comparisons	0^* or up to $S^\dagger$	0
$<=$ Comparisons	up to S	0
$==$ Comparisons	0	up to $S \times H$
Memory reads	up to $S \times 3^*$ or up to $S \times 4^\dagger$	$S \times H$
Memory writes	1^* or up to $S \times 5 + 6^\dagger$	0^* or up to $H^\dagger$

*for recall
†for store

the PPAM treats every variable in exactly the same way. During memory recall, each iteration of the loop in algorithm 3 includes:

- 1 Euclidean distance measurement operation (for D)
- 1 threshold comparison operation (T)
- 3 memory read operations (3 dimensions of the vision variable in $M[i]$)

At the end of the loop, there is one conditional memory write operation (for zeroing the age of the link). For adding a new link to the map, each iteration of the loop in algorithm 3 includes:

- 1 Euclidean distance measurement operation (for D)
- 2 threshold comparison operations (T and Q)
- 4 memory read operations (3-dimensional vision variable in $M[i]$ and age of $M[i]$)

- 5 conditional memory write operation (remove 3-dimensional value for the vision and 2-dimensional value for the arm in $M[i]$)

At the end of the loop, there are 6 conditional memory write operations (3 for vision, 2 for arm and 1 for the age of the link). Each Euclidean distance measurement operation is composed of:

- 2 sine calculations
- 2 cosine calculations
- 2 subtractions
- 1 addition
- 6 multiplications
- 1 square root

Table 7.8 is generated for the case where an associative memory stores a set of vector pairs $T = \{X^{(k)}, Y^{(k)}\}_{k=1,\ldots,S}$. For the catcher-in-the-rye, $X^{(k)} \in \mathbb{R}^N$ and $Y^{(k)} \in \mathbb{R}^M$, while for the PPAM $X^{(k)}$ and $Y^{(k)}$ are encoded values of the same. The best case values for the catcher-in-the-rye may be obtained by using $S = 1$ while in the worst case S retains its value as the number of elements in the dataset. H in Table 7.8 represents the number of nodes in the PPAM. Note that each of the H operations execute in parallel in each of the H nodes in the PPAM. Further note that all memory variables in the catcher-in-the-rye are real values and therefore all catcher-in-the-rye calculations require floating point arithmetic.

Despite all these drawbacks of the method used in the catcher-in-the-rye, in terms of sheer speed of operation, it still runs quite fast – primarily because the traditional processors it operates on run at clock frequencies in the range of GHz. However, these traditional processors are produced using components which have become highly refined and suited to the architecture through decades of commercial pressure. As mentioned in Chapter 6, digital logic implementations of the PPAM are at a disadvantage and therefore comparing the speed of a PPAM implemented over digital logic with such highly optimised architectures is unfair. Nonetheless, it must be noted that memory recall in the PPAM is asynchronous and, assuming use of efficient electronic components, asynchronous outputs are faster than synchronous outputs.

7.2.3.1 Encoding and Quantisation

The catcher-in-the-rye uses an error threshold to control acceptable error levels in the predicted (or recalled) values from the controller. As shown by Hülse et al. (2010) mappings accumulate uncertainty so that, eventually, the error values distort predictions. For the PPAM, the equivalent of this error threshold is performed implicitly in the encoding. Upon observation of an associative pair, if the PPAM determines that the pair is a new one, it is added to the memory. If, during the encoding process, more values are placed in the same *bin* such that a larger range of values are encoded using the same symbol, the error threshold is increased. However, unlike the catcher-in-the-rye, minimising this error does not maximise the performance of the PPAM. As shown in Chapter 5 (Section 5.2), this (quantisation) error is a parameter that may be tuned.

7.2.3.2 Memory Analysis

As shown in Section 7.2.2 (Figure 7.8), the memory capacity of the PPAM increases with the number of conflict resolving nodes. Furthermore, the results in Section 7.2.2 indicate that the number of memory locations used in each node is independent of the number of conflict resolving nodes. Together, these two observations imply a high level of scalability since increasing the number of nodes in the network does not effect the structure of individual nodes but it does increase the performance of the PPAM.

One concern is the size of the memory required to be implemented in each node. This is dependent upon the width of the data symbols used, since the symbols are used as addresses as well as data in the content addressable memory. The PPAM experiments conducted used 12-bit wide symbols thereby implying an address space of 2^{12} or 4K locations in each node. However, as shown by Figure 7.10, the maximum number of locations used in any node was 160, while most nodes used far fewer locations. This result agrees with previous observations in Sections 7.1 and 4.5.1 and show that although the theoretical maximum number of calculations in Table 7.8 may be up to S, the actual number in a real application are far fewer. Therefore, the memory inside nodes in the

PPAM can safely be limited to a much smaller number than the theoretical maximum, thereby allowing more nodes to be implemented. Considering that external data is encoded into abstract symbols, it should be possible to predict which memory locations will be required. Furthermore, an intelligent encoding algorithm should be able to cluster the symbols together thereby requiring only a small contiguous block of memory to be implemented within each node.

7.2.3.3 Summary

The task was to associate the vision space with the reach space of a robot. The first objective was to test if any PPAM configuration can be found that can accurately store and recall all the elements of this larger and more realistic dataset. Results in Section 7.2.2 show that a PPAM configuration of 29 Conflict-Resolving nodes ($C = 29$) can accurately store and recall the dataset, while a configuration of $C = 20$ can closely approximate this performance.

The second objective was to compare the number and types of operations performed by the PPAM as opposed to other more traditional techniques. A comparison of the number and type of operations performed for the PPAM and the catcher-in-the-rye was performed and it was shown that the PPAM utilises the errors from the quantisation process rather than being hindered by it (Section 7.2.3.1).

The third objective was to verify that the inferences drawn from Section 7.1 about the relationship between Conflict-Resolving nodes and memory usage were not an anomaly of the toy dataset and could be reproduced in this dataset as well. The conclusions drawn from Section 7.1.3 were as follows:

- Memory capacity increases rapidly (initially) with an increase in C but this has diminishing returns.
- The number of memory locations actually used in each node was far less than the theoretical maximum.

It can be seen that the results presented in Section 7.2.2 reinforce both these inferences.

In Chapter 5 it was shown that the previous version of the PPAM was capable of extracting meaningful relationship information from the data presented to it and successfully predict values for previously unseen data. So far it has been assumed that these properties have carried over to the second version because it operates on the same paradigm. However, this assumption needs to be verified. In addition, the results presented in this section have been generated using simulations. Repeating the experiment on a physical robot–vision system would test any implicit assumptions and further verify the results. Furthermore, it would be fruitful to explore the effects on performance of varying the quantisation levels when encoding the dataset. Section 7.3 addresses these issues.

7.3 Performance on Real-World Dataset – Hand Eye Coordination Task

It is important to test whether the PPAM as described in Chapter 6 is capable of extracting relationship information about the underlying dataset from the training dataset. In Chapter 5, this was shown (for the version of the PPAM presented in Chapter 4) using player/stage experiments. A robot was trained in a training arena and then successfully navigated through previously unseen environments. In order to test this behaviour for this version of the PPAM, the Catcher-in-the-rye experiments (Section 7.2) were recreated in the Bristol Robotics Laboratory[1] (BRL) with the help of the SABRE project partners. The Bristol Elumotion Robot Torso 2 (BERT2) and the Vicon motion capture system were used for these experiments. This setup is described in more detail in Section 7.3.1 and Section 7.3.2 presents the results.

7.3.1 Experimental Setup

The hand-eye coordination task was described in Section 7.2 as the situation where the location of objects in *vision space* is associated with their location in *reach space*. In essence, the task is one of *learning* the kinematics (forward[2] and inverse[3]) of a robotic arm. Although this is by no means a task outside the capabilities of current machines,

[1]www.brl.ac.uk

[2]calculate the position (and orientation) of a robotic arm from its joint angles

[3]calculate the joint angles from the position (and orientation) of a robotic arm

the traditional approach is to use mathematics, particularly trigonometry and geometry, to solve such tasks. In Johnson (1996), the authors calculate the forward kinematics and model the end-effector[4] space using NURBS[5]. Note that in conclusion the authors state an intention to implement on parallel computers and of using dedicated hardware support; which is suggestive of the kinds of issues highlighted in Section 2.3.

Rather than *calculating* the forward or inverse kinematics of the robotic arm, the PPAM attempts to *learn* kinematics from the dataset. Such a task is well within the domain of AI and as such is a good application to test the behaviour and performance of the PPAM. Lenz et al. (2010) provides a detailed description of the hardware platform used, the relevant points of which are summarised here.

BERT2 (Figure 7.12) is the second version of the upper-body humanoid torso at the Bristol Robotics Laboratory produced in co-operation with Elumotion[6]. The torso has four joints,

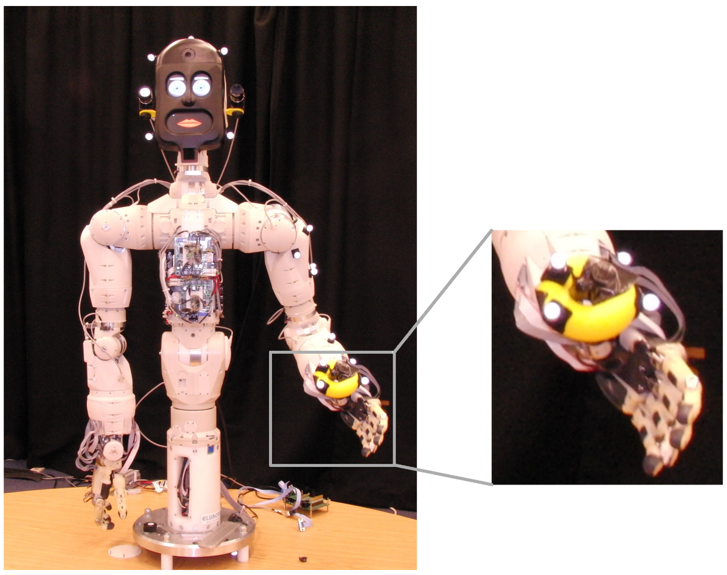


Figure 7.12: Bristol Elumotion Robot Torso 2. Note the toroid object attached to the left thumb.

namely hip rotation, hip flexion, neck rotation and neck flexion where hip rotation is the most proximal joint. The torso also has 2 arms and each arm has seven degrees of freedom, namely shoulder flexion, shoulder abduction, humeral rotation, elbow flexion, wrist pronation, wrist abduction and wrist flexion. This means that there is significant overlap in the movement such that many different orientations of the arm can be used to

[4]The end of a robotic arm designed to interact with the environment
[5]Non-Uniform Rational B-Splines
[6]www.elumotion.com

reach the same location. Humeral and wrist movement were considered irrelevant to the task and these motors were not used. Furthermore, in order to simplify the shape of the reach space, only hip rotation was used from amongst the four torso joints. This meant that the reach space could be defined using a 4-dimensional variable, with each dimension being an angle value for each motor used – namely, hip rotation, shoulder flexion, shoulder abduction, and elbow flexion.

Vision is provided by the 3-dimensional Vicon[7] Motion Capture system. The proprietary Vicon hardware, software, and model templates, store information about the marker topology of the objects to detect and localise objects in 3-D space. An Object Property Database (OPDB) stores the static properties of all objects in the environment, while an *EgoSphere* maintains updated real-time information about the dynamic properties of the objects, like their location and orientation. This means that the EgoSphere can be used to retrieve Cartesian coordinates of objects in real-time, and this forms the 3-dimensional variable representing vision space.

The various subsystems are composed of proprietary and non-proprietary software and hardware all of which needed to be connected together. The “glue” used to connect all these heterogeneous modules was a C++ open source software library called Yet Another Robot Platform (YARP)[8]. Fitzpatrick et al. (2008) has full details about YARP, but in essence, it operates on a platform independent client/server paradigm and allows the development of software modules which communicate using named TCP/IP or UDP channels. Data from the OPDB and the EgoSphere can be read by reading from specific YARP ports and commands for the various BERT2 motors can be sent by writing to specific YARP ports.

The overall aim was to use the PPAM to associate the reach space of the BERT2 robot with the vision space of the Vicon system, to test if the PPAM can extract relationship information, and analyse the effects of quantisation on the performance of the PPAM. The following objectives were identified to achieve this aim.

- Generate a sparse dataset containing Cartesian coordinates (evenly distributed over the vision space) along with corresponding BERT2 motor movements needed to reach those coordinates.

[7]www.vicon.com
[8]http://eris.liralab.it/yarp/

- Determine the PPAM configuration required to store and accurately recall the dataset.
- Test whether the PPAM is able to extract underlying relationship information by placing an object in a new location in the 3-D space and using the object coordinates provided by the EgoSphere to reach for the object.

An important aspect of these experiments was to explore the effect of varying quantisation levels on the performance of the PPAM in terms of its optimal configuration and of being able to extract relationship information.

7.3.1.1 Generating the dataset

It is necessary to have a well-defined 3-D space from which the dataset is generated to be able to determine the sparsity (or otherwise) of the dataset. For this purpose, the range of motor movements was restricted according to Table 7.9. These values were chosen

Motor	**Min Angle**	**Max Angle**
Hip Rotation	$-45.0°$	$45.0°$
Shoulder Flexion	$-30.0°$	$30.0°$
Shoulder Abduction	$0.0°$	$45.0°$
Elbow Flexion	$-30.0°$	$30.0°$

Table 7.9: BERT2 motor angular movement range (in degrees)

empirically by observing the movements of the robot and define a complex 3-D region. In order to simplify this region, the object coordinate space was limited as specified by Table 7.10, which is a subset of the 3-D region defined by the Table 7.9. Table 7.10 defines a

Axis	**Min Value**	**Max Value**
X	-45cm	30cm
Y	-21cm	75cm
Z	14cm	87cm

Table 7.10: Vicon object coordinate range (in centimeters)

cuboid with base $b = 75$ cm, length $l = 96$ cm and height $h = 73$ cm, with a total volume of 5.256×10^5 cubic centimeters ($b \times l \times h$).

The sparse dataset was produced by generating 500 random values for each of the 4 dimensions of the BERT2 Motor movement variable (500×4 random values), which were evenly distributed within their corresponding ranges (Table 7.9). For each of these 500 values, the BERT2 motors were actuated to effect the corresponding pose while an object was attached to the hand of the robot. After the robot achieved its pose, the coordinates of the object (attached to the hand of the robot) were read back from the EgoSphere. If the coordinates returned were within the coordinate range described by Table 7.10, the data-point was accepted and added to the final dataset; otherwise the data-point was discarded. Thus the resulting dataset was composed of 430 real-valued tuples each composed of 2 variables; the first being a 4-dimensional variable with angular motor values and the second being a 3-dimensional variable with Cartesian coordinates. This dataset was encoded into integer symbols to be used by the PPAM. The size of the symbols is discussed in the following sections along with the effect of choosing different symbol sizes.

7.3.1.2 Determining the optimal configuration

The real-valued data is encoded into integers which are abstract symbols from the perspective of the PPAM. The number of bits (b) used to encode the data into symbols is the width (or size) of the symbols and is inversely proportional to the noise introduced by this quantisation process. Decreasing the size of the symbols means that more real-valued data points are encoded in the same symbol with the extreme case being a single-bit symbol encoding the entire dataset into one symbol. Conversely, increasing the size of the symbols means that fewer real-valued data points are encoded into one symbols with the extreme case being an infinite width symbol so that each data point is encoded using a unique symbol. For low values of b, the PPAM would be more likely to recall a value corresponding to the input as data presented would be more likely to be previously observed data. However, the values recalled would suffer from a large quantisation error when mapped back to the real world. Alternately, for high values of b, the PPAM would be less likely to recall a value corresponding to the input, even though the values that are recalled would have a high accuracy when mapped back to the real world. Therefore, as b moves from 1 to infinity, the resolution of recalled values (when mapped back to the real-world) increases, however the PPAM is able to recall fewer and fewer values as more and more data is previously unobserved.

Therefore, increasing the size of the symbols is expected to have the following effects:

1. Increase the width of the memory in the nodes as each node needs to store b bits

wide data in memory.

2. Increase the depth of the memory in the nodes as the data of one variable is used as the address in the other variable, and so nodes in the other variable need to store data to b-bit wide addresses.

3. Change the number of conflicts in the dataset as varying the quantisation levels modifies the number of real-valued points encoded in each symbol. The nature of the change (increase or decrease in the number of conflicts) is dependent upon the actual dataset.

4. Increase the accuracy when mapping back to real-valued data as quantisation noise is reduced.

5. Decrease the ability to extract relationship information and predict values when presented with previously unseen data.

In order to isolate these effects, an optimal configuration is defined as one that is successfully able to store and recall the entire *training* dataset. Note that the performance for previously unseen data-points is not considered in this definition so that the last two points about the accuracy of decoded symbols and extracting relationships can be addressed separately (Section 7.3.1.3).

Various experiments with symbol sizes of 4-bit, 5-bit, 6-bit, 7-bit and 8-bit were used to encode the dataset and the real-valued data was encoded using equally sized quantisation levels. This means that each symbol encodes $(M - m)/(2^b - 1)$ real-values; where M is the maximum real-value, m is the minimum real-value and b is the number of bits used to encode the symbol. The divisor is $2^b - 1$ instead of 2^b because the symbol 0 is reserved to indicate that external data is not present (Section 6.3.1). Thus, for an 8-bit symbol size, each Cartesian coordinate X-symbol encodes (from Table 7.10), $(30 - (-45))/255 = 0.294$ cm. Similar calculations for Y-symbols (0.376) and Z-symbols (0.286) show that the Cartesian coordinate variable encoded with 8-bits means a single 3-dimensional value encodes a cubic volume of 0.032 cubic centimeters. Volumes for all the symbol sizes are shown in Table 7.11 and the effect of this, along with results of the experiments using the different symbol sizes, are presented in Section 7.3.2.

The memory depth shown in Table 7.11 is the absolute theoretical maximum number of

Symbol Size (b)	Volume(cm^3)	Memory Depth (per node)
4	155.733	16
5	17.643	32
6	2.102	64
7	0.257	128
8	0.032	256

Table 7.11: Effect of symbol size

locations in each node, whereas the actual number of locations used in each node may be less (as shown by the results in Sections 7.1 and 7.2). The total number of memory locations in the entire PPAM is dependent upon the number of Conflict-Resolving nodes (C) in the network. Experiments were conducted to find the optimal configurations for each of the symbols sizes listed in Table 7.11 and results are presented in Section 7.3.2.

7.3.1.3 Extracting relationships

In order to determine whether the PPAM can extract relationship information about the underlying dataset from the training dataset, object coordinate values not present in the dataset need to be presented. Two measures were used to test this. Firstly, the training dataset was divided into 2 parts with the first 420 elements being part of the training set and the last 10 elements being used to test the PPAM in the presence of previously unseen data. This unbalanced ratio was used because the dataset already provided only a very sparse coverage and decreasing the number of training elements meant that the coverage would decrease even further. Sparsity and its effects are discussed in Section 7.3.2.1.

The simulations provided a controlled environment which eased analysis of the behaviour of the PPAM. Using different symbol sizes the volume of space that the PPAM recognises as being within each symbol could be varied (as shown in Table 7.11) and the effects of this explored, by repeating the same experiments with each of the 5 symbol sizes listed in Table 7.11.

Ultimately, however, the final test is how well the PPAM performs in the real-world and so further experiments were conducted using the physical BERT2 platform described in Section 7.3.1. Objects were placed randomly within the arena and the object coordinates in the vision space were retrieved from the Vicon system. These coordinates were presented to the PPAM and recalled values were used to move the BERT2 arm. The measure for success in this case was simply whether the arm would be able to successfully touch the object. Note that there are many possible motor configuration where the fingers of the hand would be located in the same space, despite the arm being in different poses. Therefore, in the experiments with the physical platform, there are many correct solutions, unlike the experiments performed in the simulation. Results are presented in Section 7.3.2

7.3.2 Results

7.3.2.1 Generating the dataset

As shown in Table 7.11, the symbol size determines the volume of space that is represented by each data-point. Therefore, the sparsity of the training dataset depends upon the symbol size used. Part of Table 7.11 is reproduced in Table 7.12, which also includes the "Total Volume Covered" column to indicate how sparsity varies with symbol size. Each

Symbol Size (b)	**Number of Unique Coordinates**	**Volume per Symbol (cm^3)**	**Total Volume Covered (cm^3)**	**Percentage Coverage**
4	291	155.733	45318	8.622
5	410	17.643	7234	1.376
6	428	2.102	900	0.171
7	430	0.257	110	0.021
8	430	0.032	14	0.003

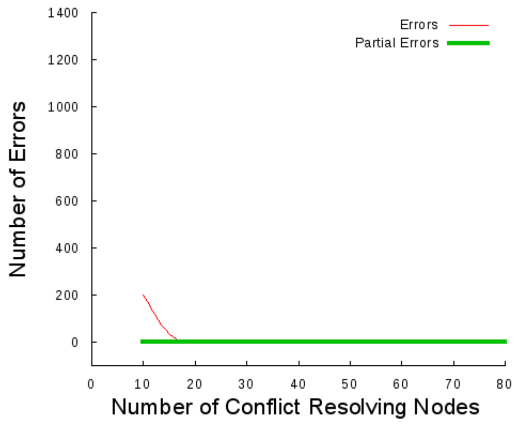
Table 7.12: Sparsity of coverage in the dataset

point in the 430 element (encoded) dataset represents a region (cuboid) of space whose volume is shown in Table 7.12. Points that are close (in 3-D space) could overlap, and therefore the actual coverage of the 3-D space depends upon the training dataset itself. Maximum coverage of the volume is achieved, however, if we assume that the volume of space for each data-point does not overlap with the volume of space for any other data-point. Note that this assumption generates the least sparse case as coverage is maximised for that particular symbol size. The values in the "Total Volume Covered" column are generated using this assumption, therefore they are simply the total number of unique coordinates in the dataset multiplied by the volume of a data-point in that dataset. Note that the number of unique coordinates in the dataset changes with the encoding symbol size. This is the result of there being more volume of space represented by smaller symbol sizes. The total volume of the entire region is 5.256×10^5 cubic centimeters (Section 7.3.1). Therefore, the percentage coverage column is the "Total Volume Covered" as a percentage of the total volume of the entire region. The sparsity of the dataset can now be observed from this percentage, which shows that even with 4-bit wide symbols, only 8.622 percent of the total volume of space is covered. Note that a 4-bit wide symbol represents 155.733 cubic centimeters. Alternately, using 8-bit wide symbols, only 0.003 percent of the entire region is covered.

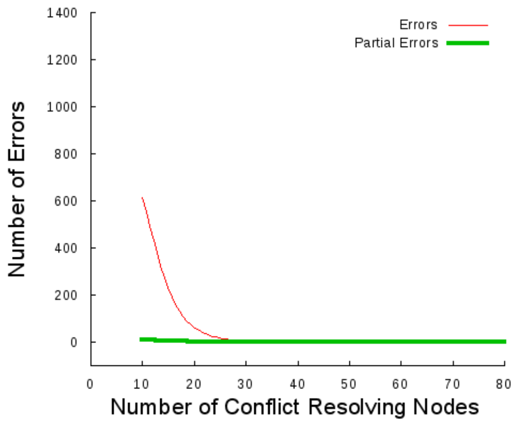
7.3.2.2 Determining the optimal configuration

In order to determine the optimal configuration for storing the training dataset (Section 7.3.1.2), experiments were conducted with varying number of Conflict-Resolving node (C) for each of the 5 different symbol sizes (b). Figure 7.13 shows the results. Values recalled by the PPAM have a confidence associated with them, and a successful recall is when the value indicated by the PPAM as having the highest confidence, is the correct value. *Partial Errors* are defined as the case where the correct value was recalled, but the PPAM indicated a higher confidence for some value other than the correct value. *Errors* are defined as the case where the correct value was not recalled at all. These (errors and partial errors) were recorded for each dimension of data recalled and the entire dataset (of 430 elements) was stored and recalled twice – once for each variable. Therefore, for each network configuration (each value of C), first the 430 element training dataset was stored; next the 430 Cartesian coordinate values were presented and corresponding motor movement values recalled; and finally 430 motor movement values were presented and corresponding Cartesian coordinates recalled. Thus, the maximum possible errors are $430 \times 4 + 430 \times 3 = 3010$. Figure 7.14 plots the score for each symbol size as C is varied. Score is calculated as $2E + e$, where E is errors and e is partial errors, so that the objective is to minimise the score.

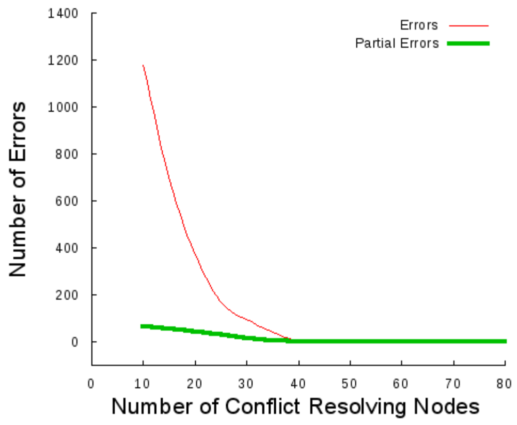
The number of errors in Figure 7.13 reflect the capacity of the memory. Therefore, past trends can be seen to be repeated, namely memory capacity increases rapidly with C but with diminishing returns. Furthermore, from Figure 7.14 it can be seen that larger symbol sizes have fewer errors for the same C, with the exception of $b = 5$. Therefore, $b = 8$ achieves zero errors and zero partial errors for $C = 24$, while $b = 7$ achieves the same for $C = 36$ and so on. Symbol sizes of $b = 4$ and $b = 5$ are exceptions in a couple of



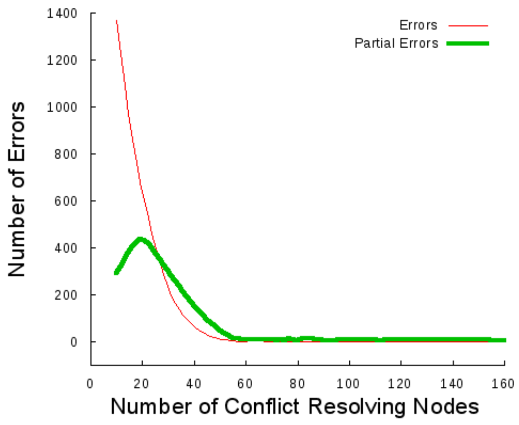
(a) $b = 8$



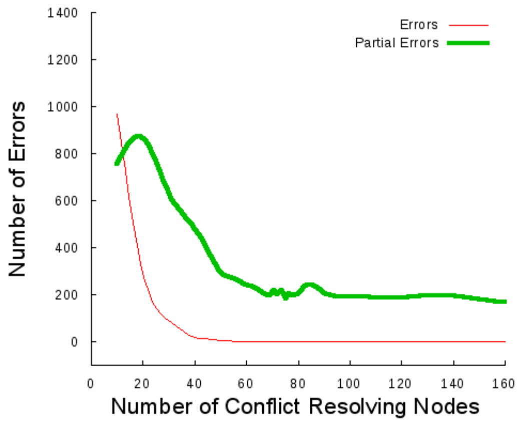
(b) $b = 7$



(c) $b = 6$



(d) $b = 5$



(e) $b = 4$

Figure 7.13: For each symbol size (b), the number of Conflict-Resolving nodes (C) is varied and errors and partial errors are plotted to find the optimal configuration for the BERT2 dataset.

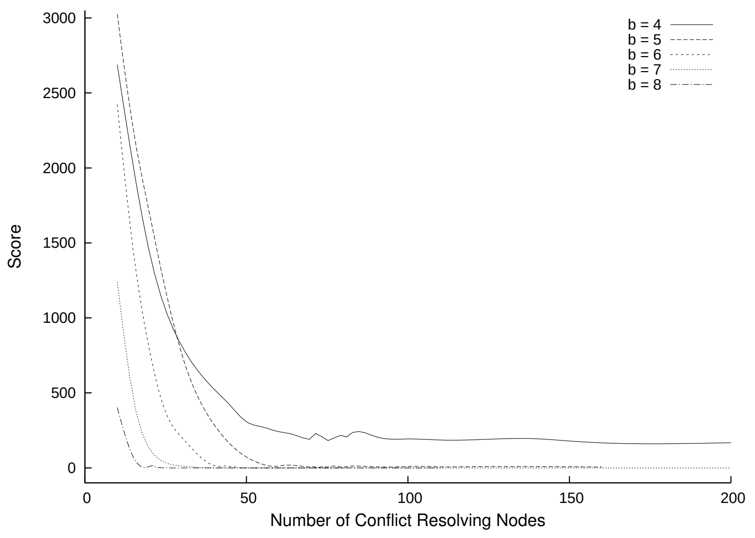


Figure 7.14: Score for each symbol size while searching for the optimal C

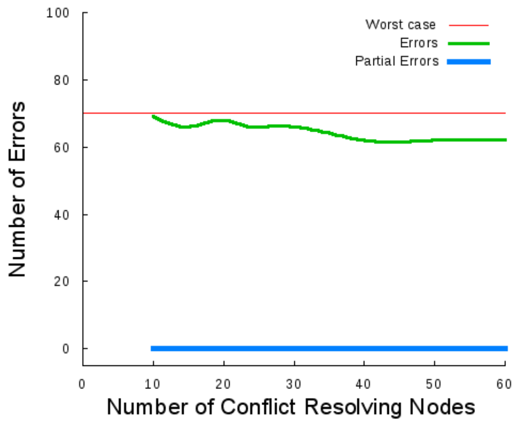
ways. Firstly, for both these cases, although zero errors were achieved, zero partial errors were not achieved for any configuration. The minimum for $b = 4$ was with a configuration of $C = 60$ while that for $b = 5$ was observed for $C = 69$. Secondly, $b = 5$ has more errors for the same value of C which is in opposition to the trend for the rest of the symbol sizes. Note, however, that the number of partial errors for $b = 5$ approaches zero (though it does not attain it) while that for $b = 4$ remains quite high (around the 200 mark). Therefore, although the curve for $b = 4$ starts below the curve for $b = 5$, it ends up above it. In this, it does follow the overall trend.

Both these anomalies can be explained because of the effect symbol size has on the training dataset. As the symbol size is reduced to a sufficiently small value, the nature of the relationship of the two variables being associated also changes. As shown in Table 7.12, $b = 4$ and $b = 5$ both have significantly fewer unique number of coordinates. However, the number of unique motor commands remain at 430 for $b = 5$ and drops just a little to 428 for $b = 4$. Therefore, whereas for $b = 6$ and onwards the dataset has a one-to-one relationship (or almost that for $b = 6$), the dataset when $b = 4$ and $b = 5$ has a one-to-many relationship. This means, that when a particular value is presented for recall, there are multiple correct answers. As may be expected, in such a situation the PPAM can get confused.

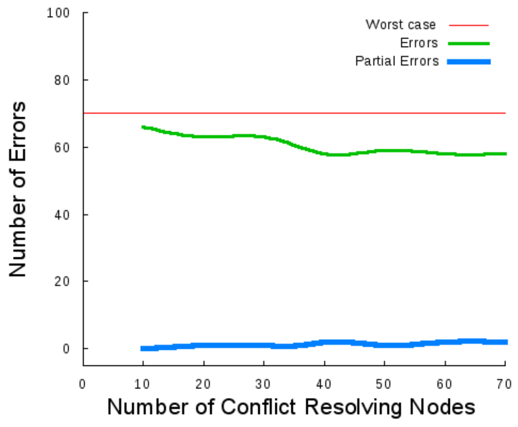
7.3.2.3 Extracting relationships

In order to determine whether the PPAM was able to extract information from the training dataset about the relationship of the underlying dataset, previously unseen values were presented. As stated in Section 7.3.1.3, the PPAM was trained using the first 420 elements and the last 10 elements were used as the test set. As for Section 7.3.2.2, errors and partial errors were recorded for each dimension of data recalled and the test set (of 10 elements) was recalled twice – once for each variable. Therefore, for each network configuration (each value of C), first the 420 element training dataset was stored. Next the 10 Cartesian coordinate values were presented and corresponding motor movement values recalled; and finally 10 motor movement values were presented and corresponding Cartesian coordinates recalled. Thus, the maximum possible errors are $10 \times 4 + 10 \times 3 = 70$. Figure 7.15 plots the errors for each symbol size as C is varied, while Figure 7.16 plots the score for each symbol size as C is varied. As in Section 7.3.2.2, score is calculated as $2E + e$, where E is errors and e is partial errors, so that the objective is to minimise the score.

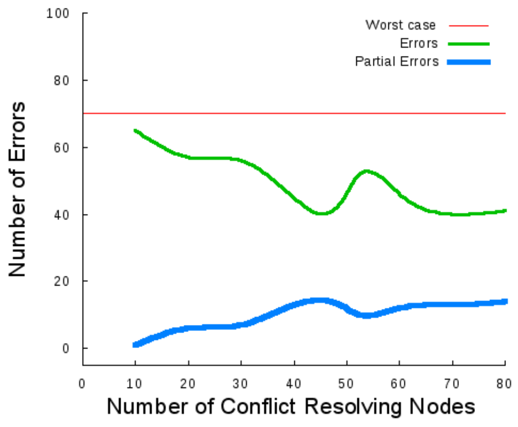
From the plots in Figure 7.15, it can be seen that varying C does not have quite as much effect on the memory capacity of the PPAM as it did in Figure 7.13. In fact, from Figure 7.15a, it may be seen that $b = 8$ performs quite poorly in the case of previously unseen data, with errors being very close to the worst case value (as is the corresponding score in Figure 7.16). The performance of $b = 7$ (Figure 7.15b) is quite similar to that of $b = 8$ and its only with $b = 5$ that errors fall below 50%. $b = 4$ is the only case where errors are reduced to zero and this trend is quite obvious from Figure 7.16 which shows the score for $b = 4$ being far better than the rest. The reason for this is related to the volume of space represented by each symbol. As symbol size changes, the volume of space represented by each symbol also changes, as discussed in Section 7.3.1. As shown in Table 7.12, the volume of space represented by each unique symbol when $b = 4$ is 155.733 cubic centimeters, whereas this drops to 17.643 cubic centimeters for $b = 5$ and only 0.032 cubic centimeters for $b = 8$. This means that while $b = 8$ is extremely accurate in motor control, new data points are quite unlikely to fall within this covered region of space since a maximum of only 0.003 percent of the space is actually covered by the training dataset (Table 7.12). On the other hand, with $b = 4$, the motor control is quite grainy as each movement locates objects within 156 cubic centimeters, new data points are more likely to fall within the covered region of space since a maximum of 8.6 percent of the space is



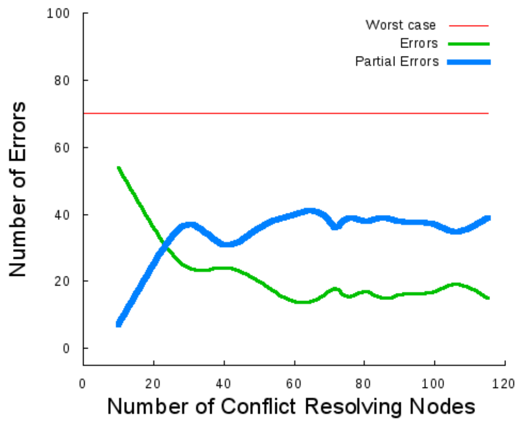
(a) $b = 8$



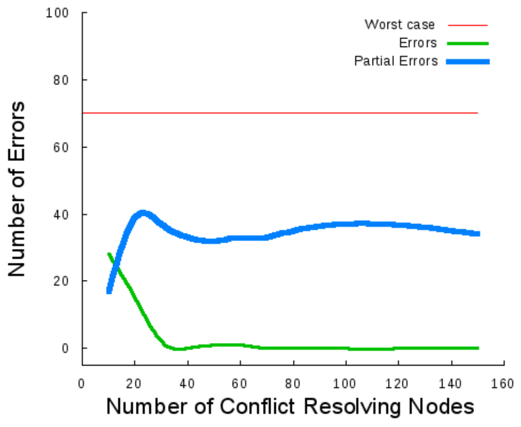
(b) $b = 7$



(c) $b = 6$



(d) $b = 5$



(e) $b = 4$

Figure 7.15: For each symbol size (b), the number of Conflict-Resolving nodes (C) is varied and previously unseen data is presented to test whether the PPAM was able to extract relationship information about the underlying dataset.

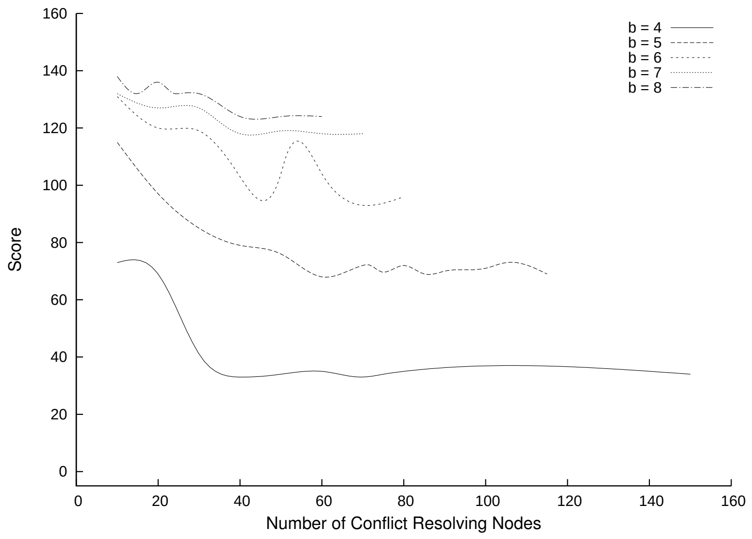


Figure 7.16: Score for each symbol size while testing on previously unseen data

covered by the training dataset. Implications of this are discussed in Section 7.3.3.

These results do not take into account the noise inherent in any real-world system interacting with the physical world. This includes such things as sensor reading errors, motor movement granularity, and play or flexibility in joints and motors. As stated in Section 7.3.1.3, experiments were conducted on the BERT2 platform to test the recall of unseen data. Since the simulations with $b = 8$ and $b = 7$ indicated a high error rate, only symbol sizes of 4, 5 and 6 were attempted. An experiment is considered a *complete success* if the index finger or thumb of the hand touches the object. It is considered a *partial success* if any part of the robots arm touches the object. 10 objects were placed randomly in the environment and results of these experiments are shown in Table 7.13 and discussed in the following section.

Symbol Size (b)	**Complete Success**	**Partial Success**
4	0	3
5	0	1
6	0	0

Table 7.13: BERT2 experiments using previously unseen object locations

7.3.3 Discussion

The overall aim was to use the PPAM to associate the reach space of the BERT2 robot with the vision space of the Vicon system and to test if the PPAM was able to extract relationship information about the underlying dataset. In addition, the effect of quantisation on the performance of the PPAM needed to be explored and this was emulated by varying the symbol size for encoding real-values. A sparse training dataset was generated using the BERT2 platform and it was shown how the relationship of the variables in the training dataset changed with changes in the symbol size. Results from Section 7.3.2.2 also show that as the relationship between the 2 variables being associated becomes more well-defined (by increasing the symbol size), fewer number of Conflict-Resolving nodes (C) are required to store those associations. On the other hand, results from Section 7.3.2.3 show that although increasing the symbol size sharpens the relationship between the variables, it also reduces the coverage of the overall underlying dataset, which in turn means that extracting relationship information becomes harder. In essence, this means that although the PPAM can extract relationship information, it is a function of the quantisation parameter (b) which needs to be tuned along with the number of Conflict-Resolving nodes (C) to maximise performance.

Note that the performance of the PPAM for previously unseen data converges with lower C than the value needed to store the complete dataset accurately. Therefore, for $b = 4$, the score converges at $C = 40$ (Figure 7.16) while it converges at $C = 60$ for storing the complete training dataset (Figure 7.14). These results are summarised in Table 7.14 and support the results in previous sections indicating that the PPAM with fewer nodes than needed for the optimal node configuration is able to display a similar performance.

Symbol Size (b)	**Unseen Data**	**Entire Training Dataset**
4	40	60
5	60	72
6	40	50
7	35	38
8	15	24

Table 7.14: Convergence of C for unseen data and the entire training set

Other than emulating noise through quantisation, the effect of noise from real-world systems was tested by running experiments on the BERT2 platform. As can be seen from Table 7.13, none of the 30 cases resulted in complete success (thumb touching the object) while only 4 of the 30 cases resulted in partial success (any part of the hand touching the object). Although these results are quite poor, they can be understood better in light of the discussion in Section 7.3.2.1 about the effect of sparsity of the dataset on the PPAM. With $b = 4$ there were a few cases of partial success (some portion of the hand touching the object). The reason these were partial successes (and not complete successes) was because, at this bit-width, the quantisation error is significant, since each symbol represents a volume of 155.733 cubic centimeters. However, as mentioned before, the likelihood of a previously unseen value falling into this cubic region of space is larger than with higher bit-widths. At $b = 4$, the coverage was at 8.622 percent (Table 7.12). At $b = 5$ only 1 case of partial success was recorded out of 10 and the percentage of space covered by the training dataset was 1.376 (Table 7.12). Therefore, it can be concluded that a coverage of 8.622 percent is insufficient. It remains to discover the relationship between spatial coverage and the accuracy of recall; in particular, what is the level of coverage at which the PPAM can provide 95 percent accuracy? The logical next step would be to conduct more experiments in order to generate a higher coverage of the space and evaluate the performance of the PPAM in this situation. However, due to limited time and access to the shared resource (BERT2) at the Bristol Robotics Laboratory, it was not possible to gather more data. An imprecise projection can nonetheless be made from the current data. Figure 7.17 uses linear regression to predict how coverage varies with the success rate. Spatial coverage is normalised to range between 0 and 1.

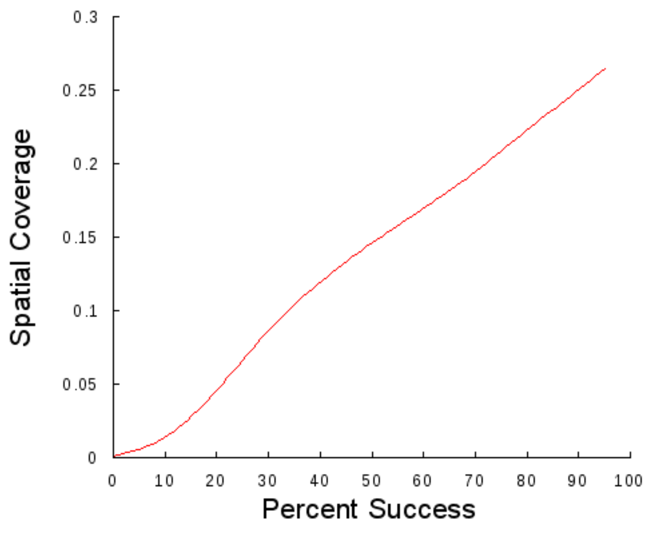


Figure 7.17: Projected coverage requirements to improve success rate (generated using linear regression)

The experiments performed so far have been aimed at analysing the performance of the PPAM in terms of memory capacity and recall. Another critical aspect of the PPAM is its fault tolerance. This is addressed in Section 7.4.

7.4 Performance in the Presence of Faults

The PPAM has been designed for fault-tolerance. Experiments conducted to test the robustness of the design are described in Section 7.4.1. Note that the PPAM implemented on the SABRE fabric also inherits the fault-tolerance features of the lower layers developed at BRL (Section 1.2).

7.4.1 Experiment Setup

The objective was to design an architecture that displayed *graceful degradation* in performance in the presence of faults. This can be broken into the following:

1. No individual node should be critical to the correct operation of the whole.
2. Increasing the number of nodes should increase fault tolerance.

The toy dataset presented in Section 7.1 (associating Cartesian and Polar coordinates) was used to test these objectives. It can be seen from Section 7.1.2 that the network configuration with $C = 7$ provides sufficient nodes to store the entire dataset of 16 elements.

To test the first hypothesis, experiments were conducted with node configurations ranging from 7 to 51 Conflict-Resolving nodes. In each configuration, the following steps were performed iteratively for each node:

- A fault was injected into the node.
- the entire contents of the memory were recalled using variable 2 as the lookup
- the entire contents of the memory were recalled using variable 1 as the lookup
- the injected fault was removed from the node.

In this way, the effect of injecting faults into each node was analysed. To test the second hypothesis, the effects of increasing the number of Conflict-Resolving nodes was noted, while exhaustively injecting faults in all the nodes. In addition, a second set of experiments were performed where two nodes were injected with faults (instead of just one), and the contents of the memory were similarly recalled. These experiments were not performed exhaustively (for each and every combination of two nodes) because the simulation time for such experiments was prohibitive (months worth of runtime). Node configurations ranging from 7 to 21 Conflict-Resolving nodes were used, with a total of E different combinations of nodes being injected with faults, where E is a constant (Section 7.4.2). This number was arbitrarily chosen in order to minimise the simulation time while testing enough nodes to ensure an even spread of faults over the network. Furthermore, injecting faults in the same number of nodes (as the number of Conflict-Resolving nodes was increased) can be used to illustrate the effects of increasing the number of nodes on fault tolerance.

Note that although the faults injected can be categorised as stuck-at faults for the node, this also models transient faults as nodes become faulty and later recover as well. Furthermore, the faults also encompass connection faults, since an erroneous value being received as input can be a result of a faulty connection or a faulty node. From the perspective of the PPAM, bridging faults (Chapter 3) on the lines between nodes are the same as two nodes getting stuck-at faults together, which is also analysed. Delay faults are nodes stuck-at the previous value for a little longer than desired, and therefore from the perspective of the PPAM, these are also modeled. Stuck-open faults, however are not encompassed in these experiments and indeed are a cause of concern for the PPAM architecture. However, these faults are usually induced during the (VLSI) fabrication process rather than post fabrication and therefore can be thought of as not being entirely relevant for the focus of this thesis, since it deals with faults that occur in the field, after a chip has passed post-production testing.

7.4.2 Results

Figure 7.18 shows the results for the case where each node was injected with faults (one at a time) when $C = 7$. The two axes represent the ID of the node that is injected with a fault. The y-axis represents the Conflict-Resolving ID while the x-axis represents the dimension ID. The thick black vertical line down the centre separates the cases where faults were injected in variable 1 nodes (on the left) and in variable 2 nodes (on the right). The thin black vertical lines at the quartiles represent the cases where variable 1 is being recalled (on the left) and variable 2 is being recalled (on the right). From Chapter 6 it

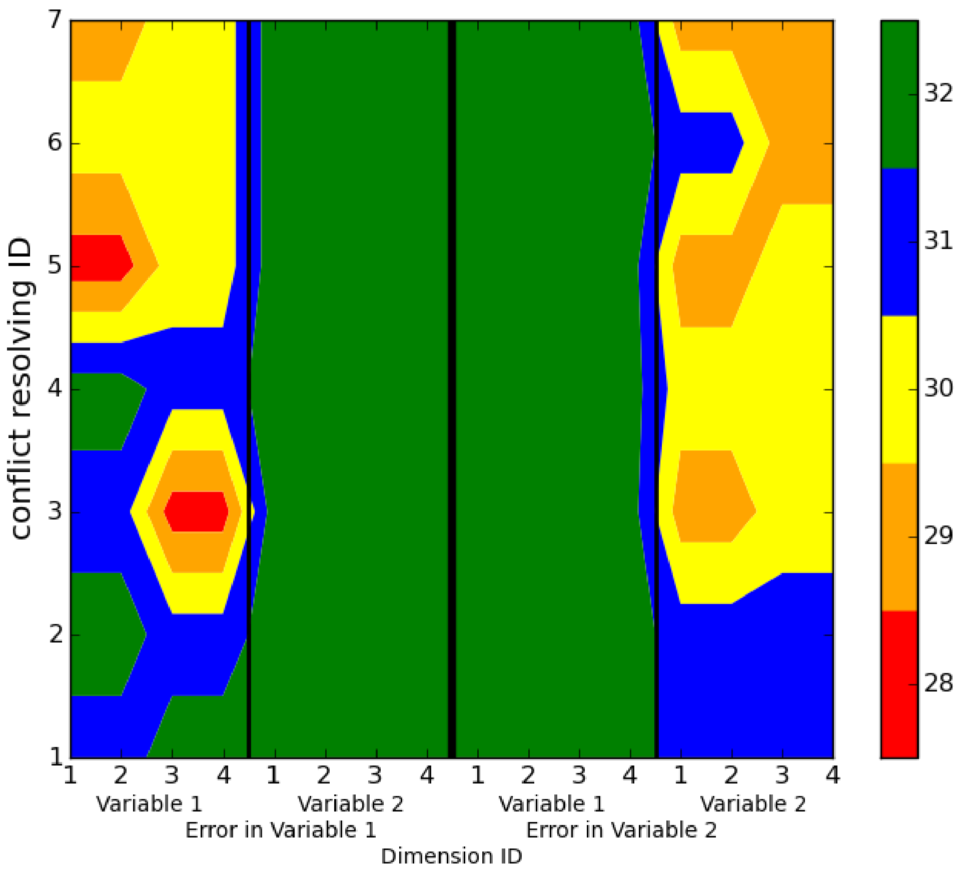
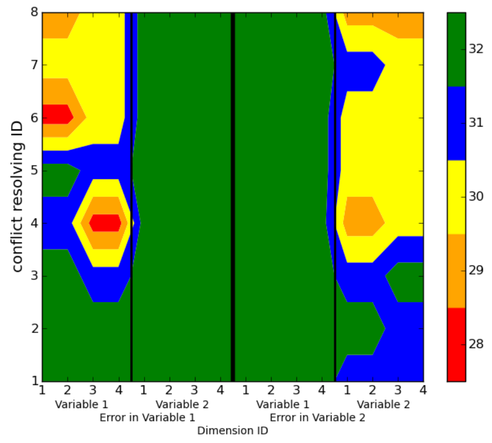


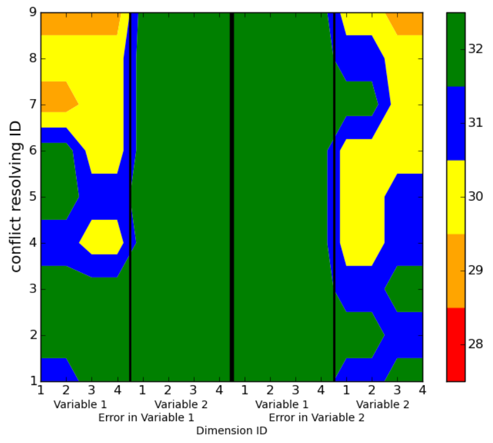
Figure 7.18: Exhaustively injecting faults in each node one at a time for $C = 7$

can be seen that each variable is stored in $D_1 \times D_2 \times C$ nodes, where D_1 is the number of dimensions in variable 1, D_2 is the number of dimensions in variable 2 and C is the number of Conflict-Resolving nodes. The numbers 1 to 4 along the x-axis represent the $D_1 \times D_2$ nodes. Therefore, if a fault is injected in a variable 1 node with $D_1 = 1$ and $D_2 = 2$ and $C = 3$, this would be the point represented by the intersection of the horizontal line along $y = 3$ on the y-axis and the vertical line along $x = 2$ in the first quartile on the x-axis. The colours of the contours represent the number of correct memory recalls. Since the dataset is composed of 16 elements (Sections 7.1 and 6.2.1), and bidirectional recall is performed, this means that a total of 32 elements are recalled from memory. Therefore, the areas shaded green represent the case where faults resulted in no degradation of memory recall. Figures 7.19 and 7.20 present selected results from the remaining node configurations. These are discussed in more detail in Section 7.4.3.

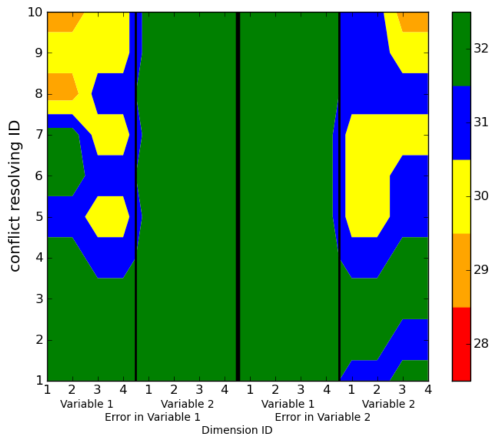
Figure 7.21 shows the results for the second set of experiments performed where 2 nodes were injected with faults simultaneously. As mentioned in Section 7.4.1, the number of



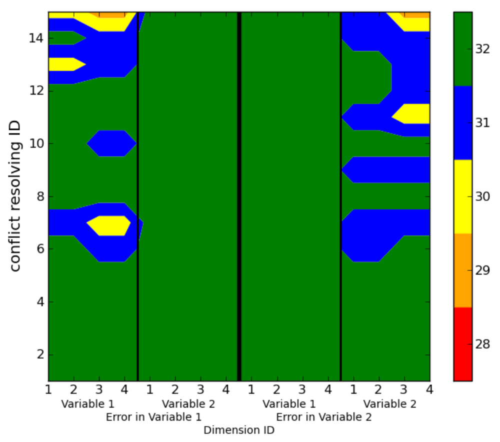
(a) $C = 8$



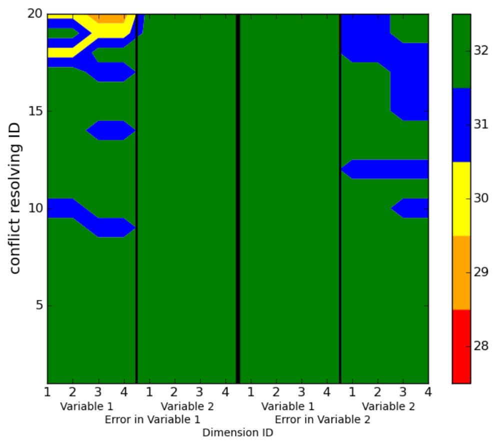
(b) $C = 9$



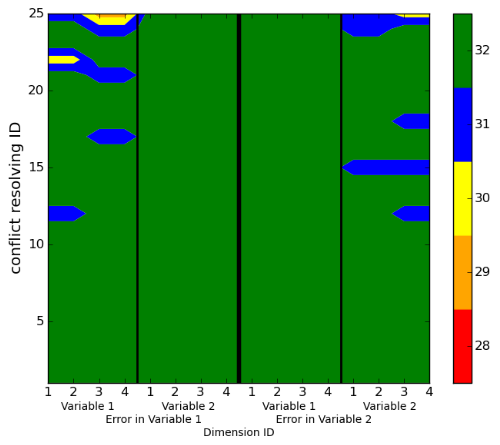
(c) $C = 10$



(d) $C = 15$

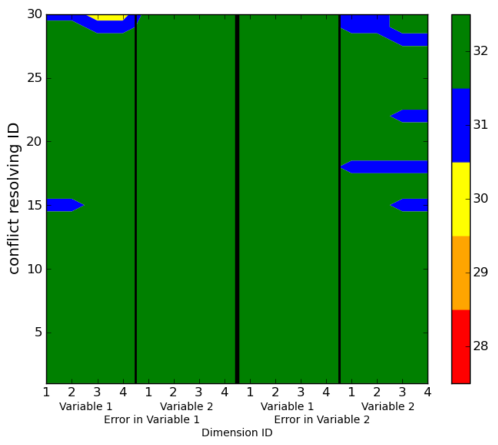


(e) $C = 20$

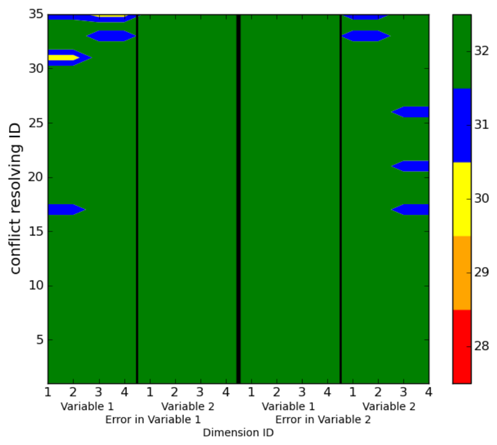


(f) $C = 25$

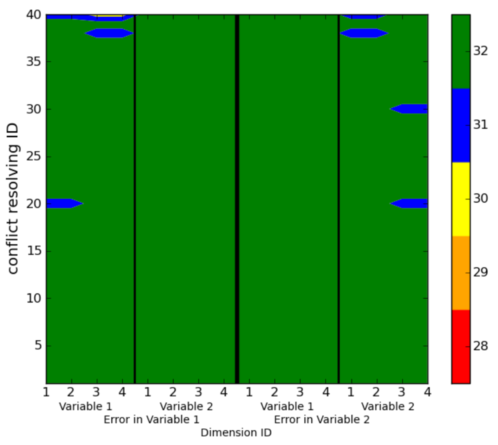
Figure 7.19: Exhaustively injecting faults in nodes one at a time (lower half)



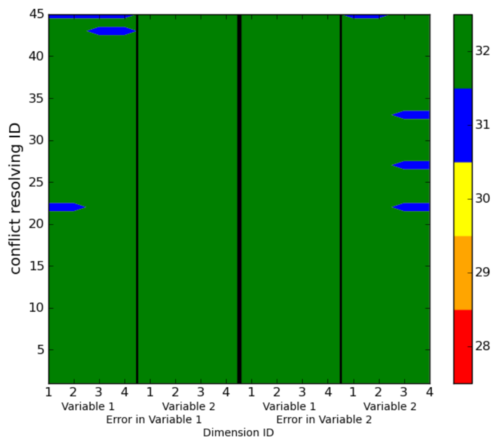
(a) $C = 30$



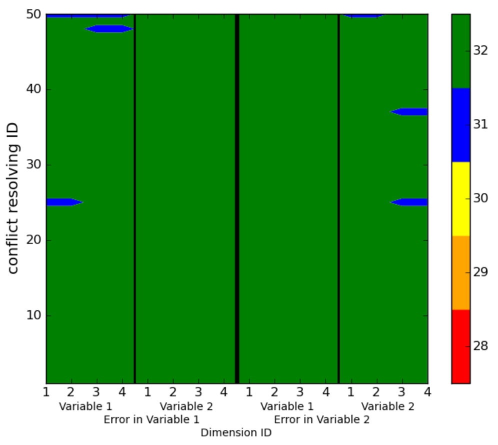
(b) $C = 35$



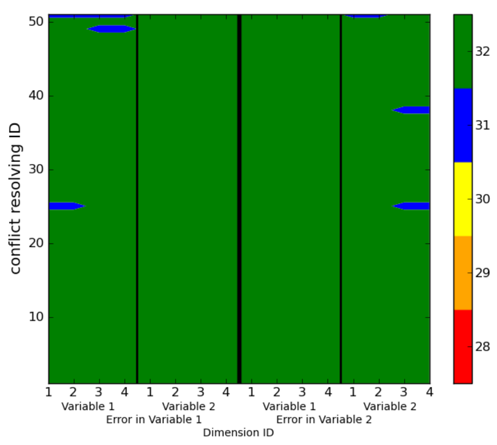
(c) $C = 40$



(d) $C = 45$

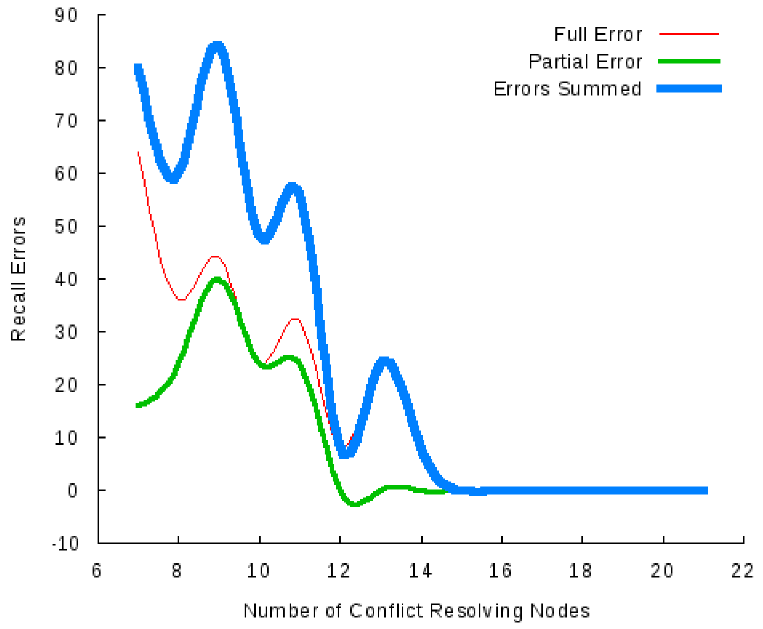


(e) $C = 50$

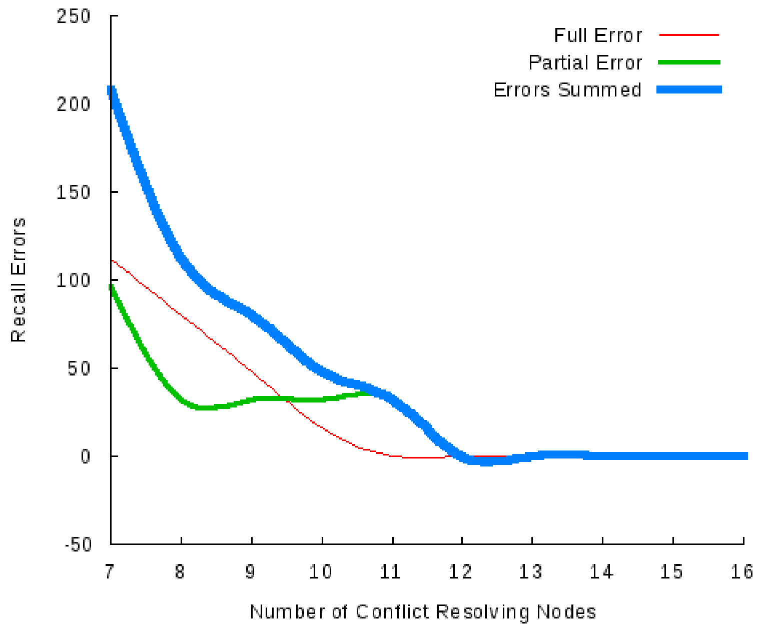


(f) $C = 51$

Figure 7.20: Exhaustively injecting faults in nodes one at a time (upper half)



(a) Errors evenly distributed over nodes



(b) Errors in the same nodes every time

Figure 7.21: Injecting faults in 2 node at a time

errors injected in this experiment was constant (E). In the first step, 192 errors were evenly distributed in pairs over the network of nodes (a total of 96 combinations of faulty nodes). These nodes are different for each experiment so that the faulty nodes when $C = 7$ are different from when $C = 8$ and so on. However, both directions of recall are tested with the same faulty nodes. Figure 7.21a shows the results for this scenario. Errors are recorded for each dimension of data. A “full error” is when the PPAM was completely unable to recall the correct data (for each dimension), while a “partial error” occurs when the PPAM does recall the correct value, but suggests a different value with a higher confidence value. Figure 7.21b shows the case when the same nodes are injected with faults in every experiment so that the faulty nodes when $C = 7$ are the same as for when $C = 8$. For this experiment, the number of errors injected (E) was 256 (a total of 128 combinations of faulty nodes).

7.4.3 Discussion

Figure 7.18 shows that if a fault is injected into variable 1, the recall of variable 2 is completely unaffected and vice versa. On the other hand, accurate memory recall is more sensitive to some nodes in Figure 7.18 than to others. This can be observed from the different shaded areas which indicates that some nodes cause fewer failures than others. However, this does not necessarily mean a failure of the first hypothesis (no single node should be critical to the correct operation). In order to understand why this is so, consider that $C = 7$ is the minimum number of nodes required to store the dataset – any fewer and the PPAM is unable to store all the elements because of conflicts. In the presence of conflicts, nodes differentiate to store the different conflict-generating values. Therefore, if $C = 7$, some nodes would, of necessity, be storing data that is not stored in any other node. Since the PPAM provides fault-tolerance through data redundancy, this means that if a *small* number of Conflict-Resolving nodes is implemented, some nodes become more critical than others. Injecting a fault in a node has an isolated impact such that only the specific data stored in that node is lost, while recall of data that is also present in other nodes is not affected. Therefore, if a *large* number of Conflict-Resolving nodes are implemented, this data is duplicated elsewhere and therefore not lost if the node is lost. This means that although some nodes are more critical than others, if there are *enough* nodes, no node becomes a single point-of-failure. This can be observed from the fact that as the number of nodes is increased (Figures 7.19 and 7.20) the data redundancy is increased, thereby increasing fault tolerance.

The oscillatory effect observed in Figure 7.21a can now be understood more easily. If C is small, some nodes become more critical than others, therefore, if errors are randomly distributed over the network, sometimes these errors are present in more critical nodes, while at other times they occur in less critical nodes. However, note that as C is increased, the overall effect is a dampening of errors and eventually errors are completely removed. In Figure 7.21b the effect is more stable as the same nodes are injected with faults every time so that the variation in *criticality* is removed.

The first hypothesis (regarding nodes being critical) has not been clearly proven since it is dependent upon the value of C. With a large enough value for C however, nodes are not critical. Nonetheless, whether a value is large enough is dependent upon the dataset being stored. The second hypothesis has been proven, since increasing the number of nodes clearly increases fault tolerance. In general, graceful degradation in performance can be achieved by providing a sufficiently large number of Conflict-Resolving nodes (C). Being able to analytically determine a value for C that provides a given level of fault tolerance would be quite helpful, though it is uncertain whether such a (analytical) method exists.

7.5 Summary

Chapter 4 presented a novel, bio-inspired computation method for performing bidirectional hetero-associative recall, while Chapter 5 described experiments to show that the PPAM was capable of storing associations and extracting relationship information about the underlying dataset without the need for selection of a training dataset. However, the PPAM (as described in Chapter 4) was not feasible to be implemented in hardware as the size of the problem (and consequently the PPAM network) is scaled up. Chapter 6 described a novel hardware architecture for the Protein Processor Associative Memory and used a worked example on a toy dataset of Cartesian-Polar coordinates to illustrate its operation. This chapter presents the setup and results of extensive experiments on the hardware architecture (implemented using Verilog) to:

- profile the memory capacity (Section 7.1),

- ensure that the PPAM hardware could be scaled up for large datasets (Section 7.2),
- verify that the memory capacity results were not a feature of a particular dataset, but would scale up with large datasets (Section 7.2),
- ensure that the PPAM continued to be able to extract relationship information about the underlying dataset (Section 7.3),
- explore the effects of quantisation and noise on the performance of the PPAM (Section 7.3),
- test the robustness of the PPAM (Section 7.4).

In general, the results of Sections 7.1, 7.2 and 7.3 show that increasing C increases the *associativity* of the PPAM, while the results of Section 7.4 show that the fault tolerance of the PPAM also increases by increasing the value of the independent parameter C. Therefore, as a rule-of-thumb, C should be set to the maximum possible value allowed by the hardware resources. Furthermore, Section 7.3 indicates that the quantisation level has a significant effect on the performance of the PPAM and that this is also a parameter that needs to be tuned. Note however, that as long as the number of bits used to encode the dataset during quantisation is less than (or equal to) the memory width of the nodes (fixed during hardware design), the hardware does not need any change.

Although the results are promising in many respects, it should be noted that the current (GRN inspired) memory update method is not optimal. This means that although it might be possible for a tuple to be stored in memory without erasing an older tuple, whether the PPAM is actually able to do so depends on the sequence in which the data is presented. Therefore, other update algorithms should be explored to find alternate, potentially better solutions.

Chapter 8

Conclusions and Future Work

The previous chapters have described a novel hardware architecture for bi-directional hetero-associative recall, along with experiments and their results. This chapter presents the conclusions and some possible future research directions. Section 8.1 summarises the contents of each chapter with a brief description of the novel contributions in each, while Section 8.2 discusses the results against the hypothesis. Section 8.3 presents some thoughts on further work and finally, Section 8.4 ends with some conjectures about the future of AI.

8.1 Thesis summary and novel contribution

Artificial Intelligence is a vast area and hard to define analytically; in fact, there is no evidence to suggest that it even *can* be defined analytically. As noted by Hernández-Orallo and Dowe (2010), in the absence of concrete measures, it is hard to precisely determine even how far AI has progressed over the years. Therefore, the method used most often to evaluate solutions is to compare contending methods which are determined through some subjective measure of *similarity*. Therefore, any architecture that claims to be targeted towards artificial intelligence must be very careful in defining its scope. Chapter 1 very briefly identified the problem domain as the way to implement robust AI applications on existing hardware and presented an initial definition of the hypothesis. Chapter 2 elaborated on the problem domain, by first broadly categorising the algorithms in AI, then reviewing the strengths and weaknesses of existing (parallel) hardware architectures that attempt to solve problems in this domain. Such hardware architectures were seen to be typically multiple traditional processors connected together, where traditional processors are defined as being composed of:

- an ALU that calculates results,
- a control unit that directs execution based on a program that is fetched from memory and
- a memory that stores data (and potentially instructions as well).

The major advantage of these traditional processors is their *universality* and *completeness*, which means that they are extremely flexible and can be applied to a wide range of problems. This does come at a cost, however – the application must pass through the various layers of abstraction to access the hardware. This can have major performance implications for real-time applications as shown by Song et al. (2007) who use custom hardware to offload some of the scheduling and interrupt handling tasks to display a 60% performance improvement.

Regardless of the overhead of such operating system housekeeping tasks, Hillis (1984) shows that the serial single processor environment is incapable of solving the million scaled problems in AI[1]. Therefore realistic hardware architectures for this domain have often relied on some sort of parallelism. Chapter 2 went on to discuss AI implementations on existing parallel hardware architectures, focusing on the sorts of operations that AI applications perform and the implications of executing these operations on parallel arrays. These problems were broadly classified as:

- How to make these massively parallel arrays feasible for widespread implementations in electronic devices (or electronic agents) which require artificial intelligence.
- How to program such parallel architectures

With this in mind, Chapter 2 further argued that an exploration into alternate hardware architectures, that are inspired from the way computation is "naturally" achieved in the real world, is worthwhile.

[1]AI problems that use millions of facts to come up with solutions.

Chapter 3 reviewed some existing fault tolerance methods with a focus on relevant hardware architectures and AI applications (as described in Chapter 2) for hazardous environments. It therefore presented techniques commonly used in such applications as space shuttles and robotics; and discussed the advantages and disadvantages of each. It revisited the requirement for graceful degradation and degeneracy identified in Chapter 1, in the context of the hierarchical structure proposed by the SABRE project and suggested some areas that need to be supplemented.

Chapter 4 started by reexamining the hypothesis presented in Chapter 1, and providing further details, thus presenting a final, formal hypothesis. It then presented the biological inspiration and an abstract view for a novel architecture of a bidirectional hetero-associative memory called the Protein Processor Associative Memory (PPAM). The proposed solution was based on multiple nodes, each of which only stored partial information thereby creating a distributed network. It was designed to learn firing patterns through self-regulation principles (inspired from the Genetic Regulatory Network) to achieve continuous, life-long learning. The PPAM was also designed for improved plasticity and online adaptability through modularity, data degeneracy (not redundancy) and the lack of a configuration bit-stream. Finally Chapter 4 compared the structure and types of operations performed by the PPAM with the Bidirectional Associative Memory which has long been used as a benchmark for comparing associative memory architectures. It showed that the BAM required a large number of floating point arithmetic operations (multiplications and additions) even when using integer inputs, whereas the PPAM only required integer equality comparisons. However, the theoretical maximum number of equality comparisons and the memory requirements for the PPAM was greater than the BAM.

Chapter 5 presented the experiments performed on the PPAM presented in Chapter 4. It started by describing the ways to measure the performance of the PPAM as defined in the final hypothesis in Section 4.1 and then presented results from simulations using Player/Stage. The BAM and the PPAM were alternately used to associate the sonar values and the motor movement values of a robot navigating a static environment. Two training algorithms were used for the BAM; the original Kosko's algorithm and the optimal PRLAB. In both cases (for BAM), it was shown that no correlation matrix was possible that accurately stored and recalled all 228 pairs of the training dataset, whereas the PPAM was successfully able to store all the training pairs. The concept of a *confidence* value was introduced which indicated the level of confidence that the PPAM had in its outputs and a corresponding way to measure confidence for the BAM was also indicated. The significance of a confidence value must be stressed since it can be argued that it is more important than being accurate, to have an indication of how accurate the response is. In addition, unlike the BAM, the PPAM did not need a unique and representative training dataset. Therefore, it could train on all 670 tuples of the training set in real-time which included repeated tuples. An addendum was that although PRLAB has been claimed to be insensitive to parameter values, results showed that the worst case performance of PRLAB was the same as the original Kosko's algorithm.

The PPAM was shown to be better at extracting information about the underlying dataset from the training, by comparing the *predicted* values with the values that should have been generated as determined by the object-avoidance algorithm. It should be noted that the BAM may have performed better if the training dataset had been parsed to select a representative training set composed of orthogonal pairs that encompassed the essential elements of the underlying dataset. However, this is counter to the real-time nature of the kind of applications being targeted.

Chapter 5 also described some initial experiments to test the fault tolerance of the architecture and showed that it out-performed BAM as well as the more recent SOIAM. Experiments were repeated using a real E-Puck robot over Player/Stage and the results were confirmed. However, it was shown that the architecture presented, had issues for a hardware implementation which can be summarised as:

- The number of nodes required to accurately learn the relationships was related to the number of separable bins in the dataset.
- A fully connected network was required.
- Memory *width* (not depth) in each node was related to the neighbourhood of the node.

As may be readily seen, this had a detrimental effect on the scalability of the design.

Chapter 6 attempted to resolve the issues for hardware identified in Chapter 5 by *exploding* nodes, such that the location of data determined its relationship to other data. The choice of the node as well as the choice of memory location within the node both were relevant for this relationship. In the presence of conflicts in the dataset (for instance because of many-to-many relationships), nodes would differentiate (inspired from the Genetic Regulatory Network) to express different data. During recall, the final output value emerged from the constructive or destructive interference of individual node outputs. A sample toy dataset of the relationship between the Cartesian and polar coordinates was used to illustrate the changes with a worked example. Chapter 6 also presented details about the hardware design including a schematic for nodes. It theorised about possible analogue implementations and showed that portions of the hardware design lend themselves to an analogue implementation. However, due to time and resource considerations, a full VLSI implementation was considered beyond the scope of this work, which could therefore only use the resources available on current FPGAs to implement the design. Although the memory designed within each node was of standard width, it still required a non-standard method of access and therefore FPGA implementations were highly inefficient in terms of resource utilisation. Despite this synthesis and implementation reports indicated quite favourable device utilisation in order to be able to implement multiple (of the order of thousands) of nodes on an ASIC. Thus the novel hardware architecture which was described in fully parameterised HDL code, was shown to be synthesisable. It maintained its features of plasticity and online adaptability through modularity, data degeneracy (not redundancy) and the lack of a configuration bit-stream. This last point made it superior to existing FPGAs which require a configuration bit-stream to perform dynamic reconfiguration to change their function. The PPAM was also implemented on the novel Unitronics architecture developed at the Bristol Robotics Laboratory as part of the SABRE project.

Chapter 7 presented results from experiments performed on the hardware architecture with the objective of profiling the memory capacity, exploring the effects of quantisation on performance, testing scaling up for larger datasets, and robustness in the presence of faults. Experiments were conducted on the Cartesian-Polar dataset introduced in Chapter 6, the catcher-in-the-rye dataset from the University of Aberystwyth, and the BERT2-Vicon dataset from the Bristol Robotics Laboratory. The results showed that increasing the number of Conflict-Resolving nodes not only increased the associativity (capacity) of the memory, but it also increased the fault tolerance. In addition, the effect of quantisation on the performance of the PPAM was determined to be significant thereby requiring that this parameter be tuned when encoding datasets. Note, however, that this would normally not require a redesign of the hardware.

8.1.1 Novel Contributions

The novel contributions of this thesis are as follows:

1. A novel non-standard computation method that is an amalgamation of inspirations from the biological neural network and the biological genetic regulatory network (Chapter 4). Individual novel features are as follows.

    - Development of decentralised network of nodes for hetero-associative recall, that negates the Von Neumann paradigm of classical computation.
    - Individual node operation performed without any arithmetic operations and inspired from the protein transcription process inside biological cells which negates the Turing paradigm of classical computation and the computer-as-an-artefact paradigm.
    - Ability to dynamically update the learnt algorithm on a continuous basis from observed inputs without the need of a configuration bit stream thus negating the output paradigm and the refinement paradigm.
    - Use of abstract symbols for computation which is completely free from any encoding requirements – thus allowing analog implementations in theory.
    - Novel real-time sorting algorithm for conflict resolution based on the frequency with which data is observed.
    - Modular design for scalability through the use of novel homogeneous nodes.
    - Use of degeneracy rather than redundancy to provide fault tolerance.
    - Comparison of structure of BAM with the structure of the novel hetero-associative memory.

2. Experimentally shown that the novel computation paradigm is able to perform better than the Bidirectional Associative Memory (BAM) on the same problem (Chapter 5). Individual novel features are as follows.

    - Analysis of the claimed insensitivity of PRLAB to parameter values.

- Comparison of Kosko's training algorithm with the optimal PRLAB.
- Analysis of the ability of the novel hetero-associative memory to extract relationship information about the underlying dataset,
- Analysis to show that the novel hetero-associative memory has higher capacity than the BAM.

3. Novel hardware architecture for a slightly modified version of the computation paradigm tested through actual hardware implementations (Chapter 6). Individual novel features are as follows.

   - Method of "exploding" large nodes into smaller nodes to improve hardware implementability.
   - Cell differentiation to resolve conflicts.
   - Novel hardware architecture described in fully synthesisable, parameterised HDL code.
   - Novel method of using clock skew to the advantage of the design rather than having to reduce clock skew.
   - Ability of hardware to adopt function without needing a configuration bitstream.
   - Ability of hardware to dynamically reconfigure with no additional time delay for reconfiguration.
   - Novel threshold-and-compare method that allows for harnessing the power of analogue circuits for making decisions using local information.
   - Implementation on Unitronics array developed at BRL.

4. Experimental analysis of the capacity and performance of the hardware architecture for hetero-associative memory (Chapter 7). Individual novel features are as follows.

   - Memory capacity analysis on a toy dataset.
   - Measurement of associativity on a dataset from robot experiments on a hand-eye coordination task performed at the University of Aberystwyth.
   - Measurement of associativity and effect of quantisation using a dataset from physical robot experiments performed at BRL.

- Analysis of fault tolerance by injecting faults in each node exhaustively, but one node at a time.
- Analysis of fault tolerance by injecting faults in two random nodes simultaneously.

8.2 Testing the Hypothesis

An initial hypothesis was presented in Section 1.3, which was further formalised in Section 4.1 and is repeated here:

> " *A novel, non-standard computation method for a bidirectional, hetero-associative memory, implemented using a non-standard hardware architecture, composed of multiple (of the order of hundreds) parallel processing elements, can perform meaningful computation in the context of AI applications, and can perform better and in a more robust way than other traditional techniques based on traditional hardware.* "

Section 4.1 also presented definitions for the terms *non-standard*, *meaningful*, *better performance* and *more robustness* as well as discussed in detail how to measure these. This hypothesis was considered, in order to explore the possibility of alternate hardware architectures, which are not based on the standard Turing complete Von Neumann model. Typically, breaking free of the standard computation substrate is envisioned as using a substrate other than silicon-based electronic chips. Although this would indeed be different, this work contends that silicon-based electronic chips, with an architecture that does not use arithmetic operations, is also sufficiently distinct to be considered a different substrate. The definition for standard computation provided by Johnson (2007), Stepney et al. (2005) was used for this purpose, and details of how each of the standard computation paradigm was challenged, were presented in the implied objectives listed in the beginning of Chapter 4. It is therefore asserted that a novel architecture for non-standard computation was successfully designed.

The next step is to test the performance of this novel architecture and measure if there is an increase in performance. Results in Chapter 5 showed that the PPAM is able to store and accurately recall hetero-associative pairs in the training dataset, thereby showing that the PPAM can perform meaningful computation. In order to determine if there was an increase in performance, attention must be given to choosing the correct method for comparison. Existing solutions that perform hetero-associative recall are typically sub-symbolic algorithms implemented on Von Neumann processors. Since any comparison between a software solution and a hardware solution is always dubious, the processor and algorithm together must be considered as one system solving the problem of hetero-associative recall. However, these Von Neumann processors are produced using components which have become highly refined and suited to the architecture through decades of commercial pressure. Therefore comparing the speed of operation of this (Von Neumann system) with a hardware architecture (PPAM) that can only be implemented using components not suited for it, is unfair. Similarly, the Von Neumann processors today have access to gigabytes of memory and the standard memory available has been designed to be used with such processors; thereby making a memory capacity comparison unfair as well. Nonetheless, the only option is to use such a system for comparison because the novelty of the PPAM architecture means that there is no other existing hardware architecture that it can be compared with. One method to try and use comparable hardware resources is to implement the sub-symbolic algorithm as well as the PPAM on the same FPGA and measure the speed and capacity of both architectures on the same problem. Note however, that this method is also not completely fair because the PPAM suffers from a highly inefficient utilisation of FPGA hardware resources, due to its non-standard memory requirements (Chapter 6). Since the SoftTOTEM (Section 2.2.3) is a commercially produced neural network hardware architecture for the Xilinx Virtex XCV600E FPGA (Xilinx, 2002b), this can be used to estimate the number of neurons that may be comparable to the PPAM. SoftTOTEM boasts 32 neurons with 8-bit weights and operating at 40MHz. Note that this maximum number of neurons (32) is inclusive of all layers in the neural network. Results of tuning the BAM correlation matrix in Section 5.4 indicate that 31 neurons (3x+28y network configuration) were the minimum that provided reasonable results for both the BAM and the PPAM. However, note that for the experiments described in Chapter 5, the BAM was incapable of learning the entire training dataset, while the PPAM stored and successfully recalled all 228 values. Therefore this indicated a definite increase in capacity from the BAM thereby satisfying the requirement for an increase in performance.

In terms of fault tolerance, results from experiments conducted to inject faults into nodes (Chapter 7) showed that, assuming sufficient number of nodes, no individual node was critical to the whole and faults resulted in graceful degradation. Since all of this was achieved through degeneracy rather than redundancy, the hypothesis can be declared to have been passed. However, it must be noted that in terms of the more abstract aim of discovering an alternate computational paradigm that could be applied more widely to a large range of AI applications, the results are less favourable. Most notably, the results of Section 7.3 which analyse the effect of quantisation indicate that the ability of the PPAM to extract information about the underlying dataset hinges on the quantisation level used. Although this is an advantage in one sense, since quantisation errors can be seen to be utilised to increase performance, however, incorrectly chosen quantisation levels (or bit-width for symbols) could drastically reduce the performance of the PPAM.

8.3 Further Work

The previous section (8.2) hinted at a possible area of further work. Unless a mechanism can be devised for the appropriate quantisation level to be determined (or at least recommended) automatically from within the PPAM, the scope of applications would be quite limited. Furthermore, as mentioned in Section 7.5, the GRN inspired memory update algorithm is sub-optimal. This is because, although it might be possible for a tuple to be stored in memory without erasing an older tuple, whether the PPAM is actually able to do so is dependent upon the sequence in which the data is presented. Therefore, other update algorithms need to be researched in order to find one that may be optimal. In addition, although results in Chapter 7 indicate that the number of Conflict-Resolving nodes (C) should be maximised, as this increases memory capacity and also robustness, it would be very helpful to be able to analytically derive a value for C that would be suitable for a given dataset.

In terms of the hardware architecture, one obvious area that needs exploring is the analogue circuits described in Section 6.3.1, where it was theorised that these are better suited for the operation. Although there is a large body of literature that describes the operation of the analogue circuits mathematically, it would be worthwhile to explore the effect of using these circuits in conjunction with the other digital logic circuits in a mixed signal design. Furthermore, fabricating the design using VLSI and customised components would allow a fair comparison to be made with existing techniques running on fast computers. Finally, as mentioned in Chapter 6, in the worst case, if the data has D dimensions, then the memory inside each Protein Processor would require D write ports. Although the issue of alternate memory architectures has already been discussed, it would be worthwhile to explore methods of reducing this requirement for D write ports, to as little as possible. One technique for doing so is to pipeline the write operations – however this would have the effect of further slowing down the *memorising* process.

One other possible area for further work is further inspired from the biological neural network and can be used to expand the complexity of the architecture. As mentioned in Section 4.3 nodes in the PPAM observe output values from their neighbours and use this to determine how their memory is updated, which is based on the Hebbian principle. However, note that the nodes do not use any weights for their neighbours. A major reason for avoiding weights was to avoid using standard weighted summation techniques which would employ ALUs. An alternate method that emulates weighted connections without using multipliers or adders is described below. It would be worthwhile to implement this and test what, if any, sophistication is added to the architecture.

Figure 8.1 shows three nodes that are connected through links. In order to implement

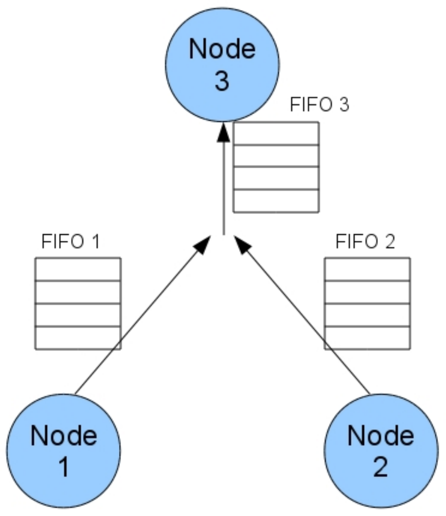


Figure 8.1: Neural network training

weights, instead of feeding the output of a node (for example node 1) to the other node (for example node 3), it is fed into a customised FIFO[2]; such that the output is always read from the last location of the FIFO (for example address 3, in a 4-element deep FIFO). Therefore until the FIFO is full (4 values in the 4-element deep FIFO) the node does not receive any input. The length (or depth) of the FIFO can denote the weight, such that more closely related nodes can have FIFOs with smaller depths, while unrelated nodes can have FIFOs with larger depths. Furthermore Spike Timing Dependent Plasticity (Vreeken, 2003) principles can also be incorporated by including a timeout period for values in the FIFO, such that over time values are removed from memory. Thus, for a 4-element deep FIFO, only if 4 inputs are received within a specific time interval, will the value be passed on, otherwise the values would *decay* over time. Moreover, summation of inputs can be implemented by feeding all inputs from nodes into one FIFO (FIFO 3 in Figure 8.1).

In the biological neural network paradigm, the neuron output is binary (fire or not-fire), as are the neuron inputs. FIFOs typically need adders to increment addresses or maintain a count of elements in the FIFO, which should preferably be avoided in the PPAM. One solution to this is to use Micro-pipelines (Hauck, 1995). The design shown in Figure 8.2 is a micro-pipeline implementing an asynchronous single-bit transition FIFO with depth 3.

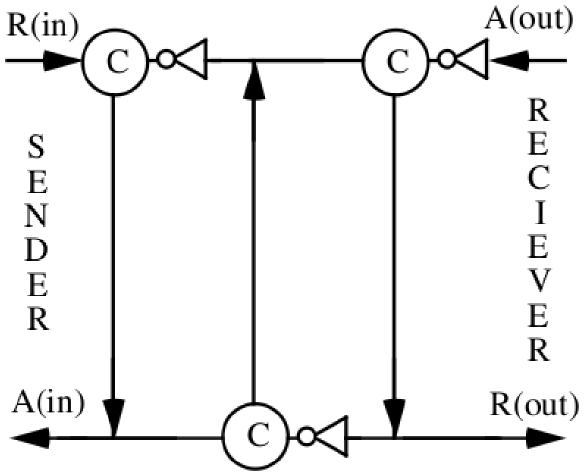


Figure 8.2: Micro-Pipelines (Hauck, 1995)

Gates labeled *C* are Muller C-elements which can be defined by the digital logic circuit shown Figure 8.3 Transitions to the micro-pipeline FIFO are fed via R(in), and output on R(out). The first transition passes straight through to R(out) and all further transitions get held at successively earlier C-Elements. If transitions input at R(in) appear at A(in),

[2]First-In-First-Out

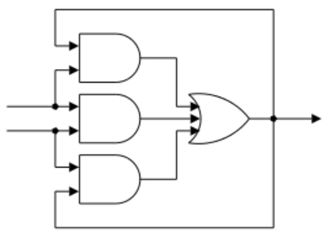


Figure 8.3: Digital logic circuit for a Muller C-Element (Hauck, 1995)

this indicates that the FIFO has space for more transitions (not full). The receiver can pop transitions by transitioning A(out). If a transition appears at R(out) as a result of the transition on A(out), the FIFO is not empty.

As can be seen from figure 8.3 a single 2-input Muller C-element can be constructed from three 2-input AND-gates and one 3-input OR-gate. Alternately, it can also be constructed using the transistor level design shown in figure 8.4, which uses 4 transistors and 2 inverters. Furthermore, since the 3 element FIFO micro-pipeline structure in figure

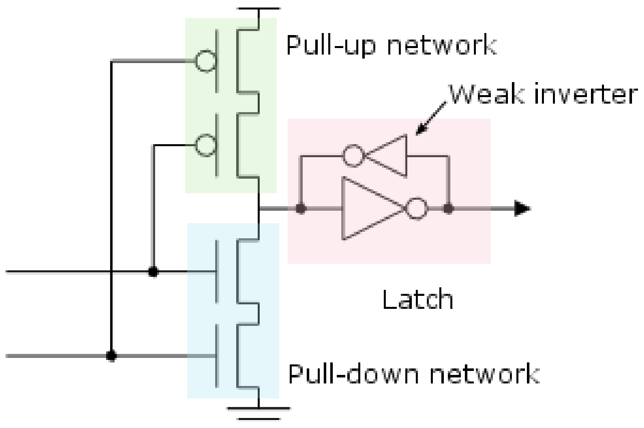


Figure 8.4: Transistor design for a Muller C-element (Hauck, 1995)

8.2 is repeatable, the gate count is as shown in table 8.1. As can be seen from the table,

FIFO depth	2-input AND-Gates	2-input OR-Gates	NOT-Gates	Gate Count per FIFO	Gate Count for 100 FIFOs
3	9	3	3	15	1500
6	18	6	6	30	3000
9	27	9	9	45	4500
12	36	12	12	60	6000

Table 8.1: Gate count for micro-pipeline FIFOs of varying depths

this initial research seems quite promising in terms of allowing a compact implementation of links that perform the equivalent of weighted summation and should therefore form an interesting area for further work. In addition, the weighted sum operation results from a non-standard method that is free of adders and multipliers. It is suggested that such non-standard methods are key to the success of AI, particularly for *online* applications.

8.4 Looking into the crystal ball

Artificial intelligence applications can be either online, where results (or intelligence) is required in real-time; or off-line, where the time constraints are relaxed. Indeed, this applies to human intelligence as well, where off-line intelligence can be defined as when a person can take their time to analyse (potentially) large amounts of data in order to gain some insight. Since there are no time-constraints, the person can wait for any resources that are required, and therefore, in a sense, can be said to have the opportunity to access any and all resources. Alternately, online intelligence can be defined as the human's ability to instantly recall the words of a song from the tune (or not, as is sometimes the case), or to recall the taste of a dish from hearing its name.

There is a plethora of algorithms that are suited for off-line artificial intelligence. In fact, it can be argued that all methods that treat AI as search, are better suited for off-line artificial intelligence. When applied to realistic problems[3] however, the kind of resource requirements for these algorithms dictates that only supercomputers can be used to obtain results; though even these, with their massive teraflops cannot provide the results in real-time. Since there are no resource constraints for off-line artificial intelligence, this can perhaps be justified; particularly as general-purpose, reusable machines that can be applied to *everything* are considered to be elegant designs. It is this search for elegant solutions that has created the general-purpose processors which use mathematical operators as a *substrate* in the standard (or classical) computation paradigm (Stepney et al., 2005). However, just because it is possible to describe a process using a mathematical equation does not mean that it is also the correct substrate to use for creating that process. After all, a simple thresholded accumulator can be created using a bucket and a tap of water. Whereas the mathematician might be tempted to describe the process in an equation to determine when the bucket will overflow and the engineer might be tempted to use an adder, a register

[3]Problems operating on millions of pieces of information.

and a comparator to model this process, and determine when the bucket will overflow, the bucket itself uses none of these when it does overflow. This raises an interesting question. Could it be that in fact, this use of general purpose (mathematical) operators has placed us in a local minimum and that there is another global minimum out there? It is universally assumed that a Turing complete architecture is better than an incomplete one and that a modular, reusable design is better than a non-reusable one. However no investigation into the efficacy of general-purpose, Turing complete architectures has been carried out and no comparisons with other methods is performed. This is primarily because no electronic, silicon-based architectures exists that challenge the arithmetic general-purpose paradigm. FPGA implementations of neural networks cannot be held up as examples of non-standard hardware architectures because firstly, they are a hardware implementation of an algorithm that is based on this general-purpose operator paradigm. Secondly, non-standard hardware FPGA implementations are greatly hindered by the resources available to them, as FPGAs are (understandably) suited for standard architectures. To be truly non-standard, an electronic architecture needs to be implemented using non-standard elements that are best suited for the operation, regardless of whether they are universal or not. Therefore although the NAND gate is universal and being able to describe any digital circuit in terms of this is interesting, a non-standard architecture implementation described only in terms of NAND gates is not really non-standard. Whether special-purpose architectures can provide the solution to online artificial intelligence remains to be seen, however it is certainly worthy of investigation, particularly for online AI.

Acronyms

ADC Analogue to Digital Converter

ACO Ant Colony Optimization

AES Artificial Evolution System

AI Artificial Intelligence

AIRS Artificial Immune Recognition System

AIS Artificial Immune System

ALU Arithmetic and Logic Unit

ANN Artificial Neural Network

APC Antigen Presenting Cells

API Application Programming Interface

ASIC Application Specific Integrated Chip

BAM Bidirectional Associative Memory

BERT2 Bristol Elumotion Robot Torso 2

BIN Broadcast Interconnection Network

BCA B-Cell Algorithm

BRL Bristol Robotics Laboratory

CA Cellular Automata

CAD Computer Aided Design

CGP Cartesian Genetic Programming

CISC Complex Instruction Set Computer

CM Connection Machine

CMLisp Connection Machine LISt Programming

CRC Cyclic Redundancy Check

DCA Dendritic Cell Algorithm

DCGP Developmental Cartesian Genetic Programming

DFID Depth-First Iterative Deepening

DNA Deoxyribo-Nucleic Acid

DWC Duplicate With Compare

ECC Error Control Code

EDVAC Electronic Discrete Variable Automatic Computer

ENIAC Electronic Numerical Integrator And Computer

EP Evolutionary Progamming

EPP Enhanced Parallel Port

ES Evolutionary Strategies

FIFO First In First Out

FLS Fuzzy Logic System

FOPL First Order Predicate Logic

FOPC First Order Predicate Calculus

FPGA Field Programmable Gate Array

FSA Finite State Automata

GA Genetic Algorithms

GMACA General Multiple Attractor Cellular Automata

GMAC/s Giga Multiply ACumulate operations/second

GP Genetic Programming

GPS Global Positioning System

GRN Genetic Regulatory Network

GUI Graphical User Interface

HDL Hardware Description Language

IDA* Iterative Deepening A*

IDS Iterative Deepening Search

I/O Input/Output

LCM Logical Computing Machine

LCS Learning Classifier Systems

LISP LISt Programming

LSb Least Significant Bit

LTP Long Term Potentiation

LUT Look-Up Tables

MAC Multiplier/Accumulator

MIMD Multiple Instruction Multiple Data

MIPS Microprocessor without Interlocked Pipeline Stages

MIPS Millions of Instructions Per Second

MHC Major Histocompatibility Complex

MSb Most Significant Bit

NASA National Aeronautics and Space Administration

NK Natural Killer

ODE Ordinary Differential Equations

OPS Official Production System

OPDB Object Property Database

PCA Probabilistic Cellular Automata (Automaton)

PCB Printed Circuit Board

PLD Programmable Logic Device

PnR Place and Route

PPAM Protein Processing Associative Memory

PS Production System

RAM Random Access Memory

RBN Random Boolean Network

RESO REcomputing with Shifted Operands

RISA Reconfigurable Integrated System Array

RISC Reduced Instruction Set Computer

RNA Ribo-Nucleic acid

RTS Reactive Tabu Search

SABRE Self-healing cellular Architectures for Biologically-inspired highly Reliable Electronic systems

SEU Single Event Upset

SIMD Single Instruction Multiple Data

SNN Spiking Neural Networks

SOM Self-Organising Map

SPANN Scalable Parallel Artificial Neural Network

SRAM Static Random Access Memory

STDP Spike Timing Dependant synaptic Plasticity

TCR T Cell Receptor

TMR Triple Modular Redundancy

TTA Transport Triggered Architecture

UNIVAC Universal Automatic Computer

VLIW Very Large Instruction Word

YARP Yet Another Robot Platform

Bibliography

Abraham, A. (2005). *Nature and Scope of AI Techniques*, chapter 128, pages 893–900. John Wiley & Sons.

Acevedo-Mosqueda, M. E., Yáñez-Márquez, C., and López-Yáñez, I. (2006). Alpha-Beta Bidirectional Associative Memories Based Translator. *International Journal of Computer Science and Network Security*, 6(5A):190–194.

Almeida, M., C., Gleriani, M., J., Castejon, F., E., Soares-Filho, and S., B. (2008). Using neural networks and cellular automata for modelling intra-urban land-use dynamics. *International Journal of Geographical Information Science*, 22(9):943–963.

Almeida, C. R. and de Abreu, F. V. (2003). Dynamical instabilities lead to sympatric speciation. *Evolutionary Ecology Research*, 5:739757.

Amaral, J. and Ghosh, J. (1994). An associative memory architecture for concurrent production systems. In *Proc. IEEE International Conference on Systems, Man, and Cybernetics 'Humans, Information and Technology'*, volume 3, pages 2219–2224 vol.3.

Andrews, P. S., Timmis, J., Owens, N. D., Aickelin, U., Hart, E., Hone, A., and Tyrrell, A. M., editors (2009). *Artificial Immune Systems, 8th International Conference, ICARIS 2009, York, UK, August 09-14, 2009, Proceedings*, number 5666 in Lecture Notes in Computer Science. Springer.

Anzellotti, G., Battiti, R., Lazzizzera, I., Soncini, G., Zorat, A., Sartori, A., Tecchiolli, G., and Lee, P. (1995). Totem: a highly parallel chip for triggering applications with inductive learning based on the reactive tabu search. *International Journal of Modern Physics C*, 6(4):555–560.

Atrubin, A. (1965). A One-Dimensional Real-Time Iterative Multiplier. *IEEE Trans. Electron. Comput.*, EC-14(3):394–399.

Backus, J. (1978). Can programming be liberated from the Von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641.

Barker, W., Halliday, D., Thoma, Y., Sanchez, E., Tempesti, G., and Tyrrell, A. (2007). Fault Tolerance Using Dynamic Reconfiguration on the POEtic Tissue. *IEEE Trans. Evol. Comput.*, 11(5):666–684.

Baxter, P. and Browne, W. (2009a). Memory-Based Cognitive Framework: a Low-Level Association Approach to Cognitive Architectures. In *European Conference on Artificial Life (ECAL'09)*. Springer.

Baxter, P. and Browne, W. (2009b). Perspectives on Robotic Embodiment from a Developmental Cognitive Architecture. In *International Conference on Adaptive and Intelligent Systems (ICAIS'09)*.

Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. (2006). The influence of operating systems on the performance of collective operations at extreme scale. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1 –12.

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., and Nataraj, A. (2008). Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11:3–16.

Benuskova, L. and Abraham, W. C. (2007). STDP rule endowed with the BCM sliding threshold accounts for hippocampal heterosynaptic plasticity. *Journal of Computational Neuroscience*, 22(2):129–133.

Benuskova, L. and Kasabov, N. (2007). Modeling L-LTP based on changes in concentration of pCREB transcription factor. *Neurocomputing*, 70(10-12):2035 – 2040. Computational Neuroscience: Trends in Research 2007, Computational Neuroscience 2006.

Berger, T., Sallez, Y., and Tahon, C. (2006). Bio-inspired approach for autonomous routing in FMS. In V. Kordic, A. L. and Merdan, M., editors, *Manufacturing the Future*, chapter 04. Pro Literatur Verlag, Germany / ARS, Austria.

Bersini, H. and Carneiro, J., editors (2006). *Artificial Immune Systems, 5th International Conference, ICARIS 2006, Oeiras, Portugal, September 4-6, 2006, Proceedings*, volume 4163 of *Lecture Notes in Computer Science*. Springer-Verlag. Revised and Extended Version.

Bheevgade, M. and Patrikar, R. M. (2008). Implementation of watch dog timer for fault tolerant computing on cluster server. *World Academy of Science, Engineering and Technology*, 38:265–268.

Bienenstock, E. L., Cooper, L. N., and Munro, P. W. (1982). Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48.

Blake, J. B. and Mandel, R. (1986). On-orbit observations of single event upset in harris hm-6508 1k rams. *Nuclear Science, IEEE Transactions on*, 33(6):1616 –1619.

Bolker, J. A. (2003). *From genotype to phenotype: looking into the black box*, chapter 4, pages 82–91. In Kumar and Bentley (2003b), 1st edition.

Bremner, D., Demaine, E., Erickson, J., Iacono, J., Langerman, S., Morin, P., and Toussaint, G. (2005). Output-sensitive algorithms for computing nearest-neighbour decision boundaries. *Discrete & Computational Geometry*, 33:593–604.

Bremner, P., Samie, M., Dragffy, G., Pipe, T., Walker, J., and Tyrrell, A. (2010). Evolving digital circuits using complex building blocks. In Tempesti, G., Tyrrell, A., and Miller, J., editors, *Evolvable Systems: From Biology to Hardware*, volume 6274 of *Lecture Notes in Computer Science*, pages 37–48. Springer Berlin / Heidelberg.

Bremner, P., Samie, M., Pipe, A. G., Dragffy, G., and Liu, Y. (2011a). Evolving cell array configurations using CGP. In Silva, S., Foster, J. A., Nicolau, M., Giacobini, M., and Machado, P., editors, *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*, volume 6621 of *LNCS*, pages 73–84, Turin, Italy. Springer Verlag.

Bremner, P., Samie, M., Pipe, A. G., and Tyrrell, A. (2011b). Multi-objective optimisation of cell-array circuit evolution. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 440 –446.

Bridgford, B., Carmichael, C., and Tseng, C. W. (2007). Correcting single-event upsets in virtex-ii platform fpga configuration memory. Application Note XAPP779, Xilinx.

Brockes, J. P. and Kumar, A. (2003). *Plasticity and reprogramming of differentiated cells in amphibian regeneration*, chapter 5, pages 92–106. In Kumar and Bentley (2003b), 1st edition.

Bünzli, D. and Capcarrere, M. (2001). Fault-tolerant structures: Towards robust self-replication in a probabilistic environment. In Kelemen and Sosik, editors, *European Conference on Artificial Life VII, the proceedings*, pages 90–99. Spinger-Verlag.

Chellapilla, K. and Fogel, D. (1999). Evolving neural networks to play checkers without relying on expert knowledge. *Neural Networks, IEEE Transactions on*, 10(6):1382–1391.

Chellappa, R., Wilson, C., and Sirohey, S. (1995). Human and machine recognition of faces: a survey. *Proceedings of the IEEE*, 83(5):705 –741.

Chen, S., Gao, H., and Yan, W. (1997). Improved exponential bidirectional associative memory. *Electronics Letters*, 33(3):223 –224.

Cook, D. and Holder, L. (1990). Accelerated learning on the connection machine. In *Proc. Second IEEE Symposium on Parallel and Distributed Processing*, pages 448–454.

Coppin, B. (2004). *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Inc., USA, 1st edition.

Corkill, D. D. (1991). Blackboard systems. *AI Expert*, 6(9):40–47.

Dawkins, R. (2003). *The evolution of evolvability*, chapter 13, pages 239–255. In Kumar and Bentley (2003b), 1st edition.

Dayan, P. (1999). Unsupervised Learning. In Wilson, R. A. and Keil, F. C., editors, *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, Cambridge, Mass.

de Abreu, F. V., 'Hoen, E. N. M. N., Almeida, C. R., and Davis, D. M. (2006). Cellular Frustration: A New Conceptual Framework for Understanding Cell-Mediated Immune Responses. In Bersini and Carneiro (2006), pages 37–51. Revised and Extended Version.

de Abreu, F. V. and Mostardinha, P. (2009). Nonself detection in a two-component cellular frustrated system. In Andrews et al. (2009), pages 19–21.

de Castro, L. N. and Timmis, J. (2002). *Artificial immune systems : a new computational intelligence approach.* Springer, London.

Dechter, R. and Pearl, J. (1985). Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32:505–536.

Elberfeld, M. and Textor, J. (2009). Efficient Algorithm for String-Based Negative Selection. In Andrews et al. (2009), pages 109–121.

Eom, T.-D., Oh, S.-K., and Lee, J.-J. (2001). Guaranteed recall of all training pairs for exponential bidirectional associative memory. *Electronics Letters*, 37(3):153 –154.

Erman, L. D., Roth, F. H., Lesser, V. R., and Reddy, D. R. (1980). The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213–253.

Faber, V. (1994). Clustering and the Continuous k-Means Algorithm. *Los Alamos Science*, 22:138–149.

Finn, A. and Lane, S. (1989). System effects of single event upsets. In *Proc of 7th Computers in Aerospace, Monterey, CA*, pages 994–1002.

Fitzpatrick, P. M., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1):29–45.

Fleischer, K. W. (2003). *How synthetic biology provides insights into contact-mediated lateral inhibition and other mechanisms*, chapter 12, pages 220–236. In Kumar and Bentley (2003b), 1st edition.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960.

Fogel, D. B. (2002). *Blondie24: playing at the edge of AI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Forgy, C. L. (1979). *On the efficient implementation of production systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Fuster, J. M. (1997). Network Memory. *rends in Neuroscience*, 20:451–459.

Fuster, J. M. (2006). The cognit: A network model of cortical representation. *Int J Psychophysiol*, 60(2):125–132.

Ganguly, N., Sikdar, B. K., Deutsch, A., Canright, G., and Chaudhuri, P. P. (2003). A Survey on Cellular Automata. Technical report, Centre for High Performance Computing, Dresden University of Technology.

Gara, A., Blumrich, M. A., Chen, D., Chiu, G. L.-T., Coteus, P., Giampapa, M. E., Haring, R. A., Heidelberger, P., Hoenicke, D., Kopcsay, G. V., Liebsch, T. A., Ohmacht, M., Steinmacher-Burow, B. D., Takken, T., and Vranas, P. (2005). Overview of the blue gene/l system architecture. *IBM Journal of R&D*, 49(2/3):195–212.

Garcia, P., Compton, K., Schulte, M., Blem, E., and Fu, W. (2006). An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*, 2006(1):13–32. Article ID 56320.

Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123.

Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323.

Glover, F. (1989). Tabu search – part i. *ORSA Journal on Computing*, 1(3):19–206.

Glover, F. (1990). Tabu search – part ii. *ORSA Journal on Computing*, 2(1):4–32.

Greensmith, J., Aickelin, U., and Cayzer, S. (2005). Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection. In Jacob, C., Pilat, M. L., Bentley, P. J., and Timmis, J., editors, *ICARIS*, volume 3627 of *Lecture Notes in Computer Science*, pages 153–167. Springer.

Greensmith, J., Oates, R., Aickelin, U., Garibaldi, J. M., and Kendall, G. (2007). The Application of a Dendritic Cell Algorithm to a Robotic Classifier. In de Castro, L. N., Zuben, F. J. V., and Knidel, H., editors, *ICARIS*, volume 4628 of *Lecture Notes in Computer Science*, pages 204–215. Springer.

Greensted, A. J. and Tyrrell, A. M. (2007). Extrinsic Evolvable hardware on the RISA architecture. In *Proceedings of ICES 2007, 7th International Conference on Evolvable Hardware*, pages 244–255.

Grzymala-Busse, J. W. (1993). Selected algorithms of machine learning from examples. *Fundamenta Informaticae*, 18:193–207.

Hamdy, E., McCollum, J., Chen, S.-O., Chiang, S., Eltoukhy, S., Chang, J., Speers, T., and Mohsen, A. (1988). Dielectric based antifuse for logic and memory ics. In *Electron Devices Meeting, 1988. IEDM '88. Technical Digest., International*, pages 786–789.

Hancock, J. T. (2003). *The principles of cell signaling*, chapter 3, pages 64–81. In Kumar and Bentley (2003b), 1st edition.

Hannig, F., Dutta, H., and Teich, J. (2004). Mapping of regular nested loop programs to coarse-grained reconfigurable arrays - constraints and methodology. In *Proc. 18th International Parallel and Distributed Processing Symposium*, pages 148–.

Hart, E. and Davoudani, D. (2009). Dendritic Cell Trafficking: From Immunology to Engineering. In Andrews et al. (2009), pages 11–13.

Hauck, S. (1995). Asynchronous design methodologies: an overview. *Proceedings of the IEEE*, 83(1):69–93.

Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory.* Wiley, New York.

Held, J., Bautista, J., and Koehl, S. (2007). From a Few Cores to Many: A Tera-scale Computing Research Overview. White paper, Intel Tera-scale Computing Research Program, www.intel.com/go/terascale.

Hernández-Orallo, J. and Dowe, D. L. (2010). Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, 174(18):1508–1539.

Hillis, W. D. (1984). The Connection Machine: A Computer Architecture Based on Cellular Automata. *Physica D*, 10:213–228.

Hillis, W. D. (1986). *The connection machine.* MIT Press, Cambridge, MA, USA.

Hofmeyr, S. A. and Forrest, S. (1999). Immunity by design: An artificial immune system. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1289–1296. Morgan-Kaufmann.

Hogeweg, P. (2003). *Computing an Organism: on the interface between informatic and dynamic processes*, chapter 9, pages 162–178. In Kumar and Bentley (2003b), 1st edition.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. R. (1986). *Induction: Processes of Inference, Learning, and Discovery.* MIT Press, Cambridge.

Hong, S. H. (2007). Implementation of fault tolerant mechanism in the bacnet/ip protocol. In *Proceedings of the Sixth International Conference on Advanced Language Processing and Web Information Technology (ALPIT 2007)*, pages 589–594, Washington, DC, USA. IEEE Computer Society.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558.

Hotz, P. E. (2003). *Combining developmental processes and their physics in an artificial evolutionary system to evolve shapes*, chapter 16, pages 302–318. In Kumar and Bentley (2003b), 1st edition.

Hülse, M., McBride, S., and Lee, M. (2009). Robotic hand-eye coordination without global reference: A biologically inspired learning scheme. In *Proc. Int. Conf. on Developmental Learing 2009, China, 2009, IEEE Catalog Number: CFP09294*.

Hülse, M., McBride, S., and Lee, M. (2010). Fast Learning Mapping Schemes for Robotic HandEye Coordination. *Cognitive Computation*, 2:1–16.

Ikeda, N., Watta, P., Artiklar, M., and Hassoun, M. H. (2001). A two-level hamming network for high performance associative memory. *Neural Networks*, 14(9):1189–1200.

Intel (2011). *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel, 248966-024 edition.

Iyer, R. K. and Kalbarczyk, Z. (2003). Hardware and Software Error Detection.

Jackson, A. H. and Tyrrell, A. M. (2001). Asynchronous Embryonics with Reconfiguration. In *Proceedings of 4th International Conference on Evolvable Systems*, volume 2210/2001, pages 88–99. Springer Berlin / Heidelberg.

Jedlicka, P. (2002). Synaptic plasticity, metaplasticity and BCM theory. *Bratisl Lek Listy*, 103(4-5):137–43.

Johnson, C. G. (1996). Modelling robot manipulators in a CAD environment using B-splines. In Bourbakis, N. G., editor, *IEEE International Joint Symposia on Intelligence* *and Systems*, pages 194–201. Institute of Electrical and Electronic Engineers, IEEE Press.

Johnson, C. G. (2004). What kinds of natural processes can be regarded as computations? In Paton, R., editor, *Computation in Cells and Tissues: Perspectives and Tools of Thought*, Natural computing series, pages 327–336. Springer, Heidelberg.

Johnson, C. G. (2007). The non-classical mind: Cognitive science and non-classical computing. In Schuster, A., editor, *Intelligent Computing Everywhere*, chapter 3, pages 45–59. Springer.

Johnson, C. G. (2008). A design framework for metaheuristics. *Artif. Intell. Rev.*, 29:163–178.

Johnson, J., Howes, W., Wirthlin, M., McMurtrey, D., Caffrey, M., Graham, P., and Morgan, K. (2008). Using Duplication with Compare for On-line Error Detection in FPGA-based Designs. In *Proc. IEEE Aerospace Conference*, pages 1–11.

Johnson, L. (2006). Coming to grips with univac. *Annals of the History of Computing, IEEE*, 28(2):32 – 42.

Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., Maskell, B., Tomlinson, P., and Roberts, M. (2003). Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 10–30, New York, NY, USA. ACM.

Jong, H. D., Geiselmann, J., and Thieffry, D. (2003). *Qualitative modeling and simulation of developmental regulatory networks*, chapter 6, pages 109–134. In Kumar and Bentley (2003b), 1st edition.

Jönsson, H., Shapiro, B. E., Meyerowitz, E. M., and Mjolsness, E. (2003). *Signalling in mutlicellular models of plan development*, chapter 8, pages 156–161. In Kumar and Bentley (2003b), 1st edition.

Kandel, E. R., Schwartz, J. H., and Jessell, T. M. (2000). *Principles of Neural Science.* McGraw-Hill Medical, 4th edition.

Karri, R. and Nicolaidis, M. (1998). Guest Editors' Introduction: Online VLSI Testing. *IEEE Des. Test*, 15(4):12–16.

Kasabov, N. (2007). *Dynamic Modeling of Brain Functions and Cognitive Processes*, volume 2, chapter 9, pages 275–302. Springer London, 2 edition.

Kawamura, K., Peters II, R. A., Bodenheimer, R. E., Sarkar, N., Park, J., Clifton, C. A., Spratley, A. W., and Hambuchen, K. (2004). Distributed Cognitive Control Systems for a Humanoid Robot. *International Journal of Humanoid Robotics*, 1(1):65–95.

Kerns, S., Shafer, B., Rockett, L.R., J., Pridmore, J., Berndt, D., van Vonno, N., and Barber, F. (1988). The design of radiation-hardened ics for space: a compendium of approaches. *Proceedings of the IEEE*, 76(11):1470 –1509.

Kohonen, T. (1972). Correlation matrix memories. *Computers, IEEE Transactions on*, C-21(4):353 –359.

Kosko, B. (1988). Bidirectional associative memories. *IEEE Trans. Syst. Man Cybern.*, 18(1):49–60.

Kumar, S. and Bentley, P. J. (2003a). *An introduction to computational development*, chapter 1, pages 1–44. In Kumar and Bentley (2003b), 1st edition.

Kumar, S. and Bentley, P. J. (2003b). *On Growth, Form and Computers*. Academic Press, Amsterdam, London, 1st edition.

Kuon, I., Tessier, R., and Rose, J. (2007). FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253.

Labeit, S. and Kolmerer, B. (1995). Titins: Giant Proteins in Charge of Muscle Ultrastructure and Elasticity. *Science*, 270(5234):293–296.

Lashley, K. S. (1933). Integrative functions of cerebral cortex. *Physiological Reviews*, 13(1):1–42.

Lee, P., Costa, E., McBader, S., Clementel, L., and Sartori, A. (2007). LogTOTEM: A Logarithmic Neural Processor and its Implementation on an FPGA Fabric. *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 2764–2769.

Lee, S. W., Kim, J. T., Wang, H., Bae, D. J., Lee, K.-M., Lee, J.-H., and Jeon, J. W. (2006). Architecture of RETE Network Hardware Accelerator for Real-Time Context-Aware System. In Gabrys, B., Howlett, R. J., and Jain, L. C., editors, *KES (1)*, volume 4251 of *Lecture Notes in Computer Science*, pages 401–408. Springer.

Legat, U., Biasizzo, A., and Novak, F. (2011). Fpga soft error recovery mechanism with small hardware overhead. In *European Test Symposium (ETS), 2011 16th IEEE*, page 207.

Lenz, A., Skachek, S., Hamann, K., Steinwender, J., Pipe, A., and Melhuish, C. (2010). The bert2 infrastructure: An integrated system for the study of human-robot interaction. In *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, pages 346 –351.

Li, D. and Du, Y. (2007). *Methodologies of AI*, chapter 02, pages 21–42. CRC Press, Boca Raton, Fla. ; London, 1st edition.

Li, K. W., Armstrong, J. R., and Tront, J. G. (1984). An hdl simulation of the effects of single event upsets on microprocessor program flow. *Nuclear Science, IEEE Transactions on*, 31(6):1139–1144.

Liu, X., Li, X., Liu, L., He, J., and Ai, B. (2008). A bottom-up approach to discover transition rules of cellular automata using ant intelligence. *Int. J. Geogr. Inf. Sci.*, 22(11-12):1247–1269.

Liu, Y. (2009). *Object Tracking using AIS*. Draft, University of York. Chap 3: Reviews on Bio-Inspired Artificial Systems.

Long, L. N. and Gupta, A. (2008). Scalable massively parallel artificial neural networks. *Journal of Aerospace Computing, Information, and Communication*, 5:3–15.

MacQueen, J. B. (1967). Some Methods for Classification and Analysis of MultiVariate Observations. In Cam, L. M. L. and Neyman, J., editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.

Madrid, R., Sanhueza, M., Alvarez, O., and Bacigalupo, J. (2003). Tonic and Phasic Receptor Neurons in the Vertebrate Olfactory Epithelium. *Biophysical Journal*, 84(6):4167–4181.

Maji, P., Ganguly, N., Saha, S., Roy, A. K., and Chaudhuri, P. P. (2002). Cellular Automata Machine for Pattern Recognition. In *ACRI '01: Proceedings of the 5th International Conference on Cellular Automata for Research and Industry*, pages 270–281, London, UK. Springer-Verlag.

Malik, U. and Diessel, O. (2005). A configuration memory architecture for fast run-time-reconfiguration of fpgas. In Rissa, T., Wilton, S. J. E., and Leong, P. H. W., editors, *FPL*, pages 636–639. IEEE.

Mange, D., Sipper, M., Stauffer, A., and Tempesti, G. (2000). Toward robust integrated circuits: The embryonics approach. *Proc. IEEE*, 88(4):516–543.

Mange, D., Stauffer, A., Tempesti, G., and Teuscher, C. (2001). From embryonics to poetic machines. In *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks: Bio-inspired Applications of Connectionism-Part II*, IWANN '01, pages 1–13, London, UK. Springer-Verlag.

Mavriplis, D., Aftosmis, M., and Berger, M. (2005). High resolution aerospace applications using the nasa columbia supercomputer. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, page 61.

McBader, S., Clementel, L., Sartori, A., Boni, A., and Lee, P. (2002). SoftTOTEM: An FPGA Implementation of the TOTEM Parallel Processor. In Glesner, M., Zipf, P., and Renovell, M., editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, volume 2438 of *Lecture Notes in Computer Science*, pages 63–70. Springer Berlin / Heidelberg.

McDermott, J. (1993). R1 ("XCON") at age 12: lessons from an elementary school achiever. *Artificial Intelligence*, 59(1-2):241–247.

Mcdonald, E. (2008). Runtime FPGA partial reconfiguration. *Aerospace and Electronic Systems Magazine, IEEE*, 23(7):10–15.

McEwan, C. and Hart, E. (2009). On AIRS and Clonal Selection for Machine Learning. In Andrews et al. (2009), pages 67–79.

Mendao, M., Timmis, J., Andrews, P., and Davies, M. (2007). The Immune System in Pieces: Computational Lessons from Degeneracy in the Immune System. In *Proc. IEEE Symposium on Foundations of Computational Intelligence FOCI 2007*, pages 394–400.

Michalski, R. S. (1983). A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111 – 161.

Miller, W. T., Sutton, R. S., and Werbos, P. J., editors (1990). *Neural Network for Control.* MIT Press.

Minsky, M. (1961). Steps toward Artificial Intelligence. *Proceedings of the IRE*, 49(1):8–30.

Minsky, M. (1975). A framework for representing knowledge. In Winston, P., editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York.

Miodownik, M. A. (2003). *Using mechanics to map genotype to phenotype*, chapter 11, pages 203–219. In Kumar and Bentley (2003b), 1st edition.

Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.-C., Floreano, D., and Martinoli, A. (2009). The e-puck, a Robot Designed for Education in Engineering. In Gonalves, P., Torres, P., and Alves, C., editors, *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, volume 1, pages 59–65, Portugal. IPCB:InstitutoPolitcnicodeCasteloBranco.

Moravec, H. (1998). When will Computer Hardware Match the Human Brain? *Journal of Evolution and Technology*, 1.

Nakamoto, T. (2009). Evolution and the universality of the mechanism of initiation of protein synthesis. *Gene*, 432(1-2):1 – 6.

Navaridas, J., Luján, M., Miguel-Alonso, J., Plana, L. A., and Furber, S. (2009). Understanding the interconnection network of SpiNNaker. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 286–295, New York, NY, USA. ACM.

Neal, M., Feyereisl, J., Rascuna, R., and Wang, X. (2006). Don't Touch Me, I'm Fine: Robot Autonomy Using an Artificial Innate Immune System. In Bersini and Carneiro (2006), pages 349–361. Revised and Extended Version.

Neiman, D. (1991). Control Issues in Parallel Rule-Firing Production Systems. In *in Proceedings of National Conference on Artificial Intelligence*, pages 310–316.

Newell, A., Shaw, J., and Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, 65(3):151–166.

Oh, H. and Kothari, S. (1994). Adaptation of the relaxation method for learning in bidirectional associative memory. *IEEE Trans. Neural Networks*, 5(4):576–583.

Ortega, C. and Tyrrell, A. (1999). Biologically inspired fault-tolerant architectures for real-time control applications. *Applications, Control Engineering Practice*, 7:673–678.

Owen, J. (2009). How to Use Player/Stage. On Player/Stage website – http://playerstage.sourceforge.net.

Pagnoni, A. and Visconti, A. (2005). An innate immune system for the protection of computer networks. In *WISICT '05: Proceedings of the 4th international symposium on Information and communication technologies*, pages 63–68. Trinity College Dublin.

Patel, J. and Fung, L. (1982). Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. *IEEE Trans. Comput.*, C-31(7):589–595.

Peterson, W. W. and Brown, D. (1961). Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235.

Petrini, F., Kerbyson, D. J., and Pakin, S. (2003). The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC*, page 55. ACM.

Pipe, A. G. and Carse, B. (2007). Fuzzy classifier system architectures for mobile robotics: An experimental comparison: Research Articles. *Int. J. Intell. Syst.*, 22(9):993–1019.

Piuri, V., Sami, M., and Stefanelli, R. (1992). Arithmetic codes for concurrent error detection in artificial neural networks: the case of AN+B codes. In *Proc. IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 127–136.

Prodan, L., Tempesti, G., Mange, D., and Stauffer, A. (2003). Embryonics: electronic stem cells. In *ICAL 2003: Proceedings of the eighth international conference on Artificial life*, pages 101–105, Cambridge, MA, USA. MIT Press.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Radojkovic, P., Cakarevic, V., Verdú, J., Pajuelo, A., Gioiosa, R., Cazorla, F. J., Nemirovsky, M., and Valero, M. (2008). Measuring operating system overhead on cmt processors. In *Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pages 133–140, Washington, DC, USA. IEEE Computer Society.

Ray, T. and Hart, J. (2003). *Evolution of multi-threaded digital organisms*, chapter 17, pages 319–336. In Kumar and Bentley (2003b), 1st edition.

Reil, T. (2003). *Artificial genomes as models of gene regulation*, chapter 14, pages 256–277. In Kumar and Bentley (2003b), 1st edition.

Reynolds, D. and Metze, G. (1978). Fault Detection Capabilities of Alternating Logic. *IEEE Trans. Comput.*, C-27(12):1093–1098.

Ritter, G. X., Diaz-de Leon, J. L., and Sussner, P. (1999). Morphological bidirectional associative memories. *Neural Netw.*, 12(6):851–867.

Rohr, J. (1995). STAREX Self-repair Routines: Software Recovery in the JPL-STAR Computer. In *Proc. Twenty-Fifth International Symposium on Fault-Tolerant Computing, ' Highlights from Twenty-Five Years'*, pages 201–.

Ros, E., Carrillo, R., Ortigosa, E. M., Barbour, B., and Agís, R. (2006). Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics. *Neural Comput.*, 18(12):2959–2993.

Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

Rossier, J., Petraglio, E., Stauffer, A., and Tempesti, G. (2004). Tom Thumb Algorithm and Von Neumann Universal Constructor. In *The Sixth International conference on Cellular Automata for Research and Industry (ACRI 2004)*, Lecture Notes in Computer Science, pages 1–10.

Sahba, F., Tizhoosh, H., and Salama, M. (2005). A coarse-to-fine approach to prostate boundary segmentation in ultrasound images. *BioMedical Engineering OnLine*, 4(1):58.

Samie, M., Dragffy, G., and Pipe, T. (2009a). Novel bio-inspired self-repair algorithm for evolvable fault tolerant hardware systems. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2143–2148, New York, NY, USA. ACM.

Samie, M., Dragffy, G., Popescu, A., Pipe, T., and Melhuish, C. (2009b). Prokaryotic Bio-Inspired Model for Embryonics. In *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '09, pages 163–170, Washington, DC, USA. IEEE Computer Society.

Sargent, S. S. and Stafford, K. R. (1965). *Basic Teachings of the Great Psychologists*. Dolphin Books, Garden City, N.Y.

Sedcole, N. P., Blodget, B., Becker, T., Anderson, J., and Lysaght, P. (2005). Modular partial reconfiguration in virtex fpgas. In *Proceedings of Field Programmable Logic and Applications 2005*, pages 211–216.

Sedcole, P., Blodget, B., Becker, T., Anderson, J., and Lysaght, P. (2006). Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques, IEE Proceedings -*, 153(3):157–164.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, 27:379–423.

She, X. and Samudrala, P. (2009). Selective triple modular redundancy for single event upset (seu) mitigation. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 344 –350.

Shen, D. and Cruz, J.B., J. (2005). Encoding strategy for maximum noise tolerance bidirectional associative memory. *Neural Networks, IEEE Transactions on*, 16(2):293 –300.

Shepard, D. (1968). A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, pages 517–524, New York, NY, USA. ACM.

Simpson, P. (1988). Bidirectional associative memory systems. Technical Report GDE-ISG-PKS-02, General Dynamics Electronics Div.

Song, M., Hong, S. H., and Chung, Y. (2007). Reducing the overhead of real-time operating system through reconfigurable hardware. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 311 –316.

Stallings, W. (2000). *Computer Organization and Architecture*. Prentice Hall International, 5th edition.

Stelzer, R., Proll, T., and John, R. (2007). Fuzzy Logic Control System for Autonomous Sailboats. In *Proc. IEEE International Fuzzy Systems Conference FUZZ-IEEE 2007*, pages 1–6.

Stepney, S., Braunstein, S. L., Clark, J. A., Tyrrell, A., Adamatzky, A., Smith, R. E., Addis, T., Johnson, C., Timmis, J., Welch, P., Milner, R., and Partridge, D. (2005).

Journeys in non-classical computation I: A grand challenge for computing research. *International Journal of Parallel, Emergent and Distributed Systems*, 20(1):5–19.

Stewart, I. (2003). *Broken symmetries and biological patterns*, chapter 10, pages 181–202. In Kumar and Bentley (2003b), 1st edition.

Stewart, R., Xie, Q., Motorola, Morneault, K., Sharp, C., Cisco, Schwarzbauer, H., Siemens, Taylor, T., Networks, N., Rytina, I., Ericsson, Kalla, M., Telcordia, and Zhang, L. (2000). RFC2960: Stream Control Transmission Protocol. Request For Comments.

Stibor, T., Timmis, J., and Eckert, C. (2006). On Permutation Masks in Hamming Negative Selection. In Bersini and Carneiro (2006), pages 122–135. Revised and Extended Version.

Storey, N. (1999). Design for safety. In *Towards System Safety: Proc. 7th Safety-Critical Systems Symposium, Huntington, UK*, pages 1–25, Huntington, UK.

Sudo, A., Sato, A., and Hasegawa, O. (2009). Associative Memory for Online Learning in Noisy Environments Using Self-Organizing Incremental Neural Network. *IEEE Trans. Neural Networks*, 20(6):964–972.

Tanaka, T., Kakiya, S., and Kabashima, Y. (2000). Capacity Analysis of Bidirectional Associative Memory.

Teich, J. (2008). Invasive Algorithms and Architectures (Invasive Algorithmen und Architekturen). *it - Information Technology*, 50(5):300–310.

Tempesti, G. (1998). *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. PhD thesis, Logic Systems Laboratory, Computer Science Department, Swiss Fedaral Institute of Technology, Lausanne, Switzerland.

Tempesti, G. (2003). Architectures and design methodologies for bio-inspired computing machines. SNF Professorship Application Research Plan.

Tempesti, G., Mange, D., Mudry, P.-A., Rossier, J., and Stauffer, A. (2007). Self-replicating hardware for reliability: The embryonics project. *J. Emerg. Technol. Comput. Syst.*, 3.

Tempesti, G., Mange, D., Stauffer, A., and Teuscher, C. (2002). The BioWall: an electronic tissue for prototyping bio-inspired systems. In *Proc. NASA/DoD Conference on Evolvable Hardware*, pages 221–230. IEEE Computer Society, Los Alamitos, Calif.

Thorpe, S. J., Delorme, A., and van Rullen, R. (2001). Spike-based strategies for rapid processing. *Neural Networks*, 14(6-7):715–725.

Timmis, J., Hart, E., Hone, A., Neal, M., Robins, A., Stepney, S., and Tyrrell, A. (2008). Immuno-Engineering. In *2nd IFIP International Conference on Biologically Inspired Collaborative Computing, 20th IFIPWorld Computer Congress, Milan, Italy, September 2008*.

Tirdad, K. and Sadeghian, A. (2010). Hopfield neural networks as pseudo random number generators. In *Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American*, pages 1–6.

Toffoli, T. (1984). CAM: A high-performance Cellular Automaton Machine. *Physica D*, 10:195–204.

Tsafrir, D., Etsion, Y., Feitelson, D. G., and Kirkpatrick, S. (2005). System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 303–312, New York, NY, USA. ACM.

Turanov, A. A., Lobanov, A. V., Fomenko, D. E., Morrison, H. G., Sogin, M. L., Klobutcher, L. A., Hatfield, D. L., and Gladyshev, V. N. (2009). Genetic Code Supports Targeted Insertion of Two Amino Acids by One Codon. *Science*, 323(5911):259–261.

Turing, A. (1950). Computing machinery and intelligence. *Mind*, 59:433–460.

Tveter, D. R. (2001). *The Backprop Algorithm*, chapter 2.

Vanag, V. K. (1999). Study of spatially extended dynamical systems using probabilistic cellular automata. *Physics-Uspekhi*, 42(5):413.

Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., and Borkar, S. (2008). An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE J. Solid-State Circuits*, 43(1):29–41.

Veen, A. H. (1986). Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396.

Von Neumann, J. (1945). First draft of a report on the EDVAC. *IEEE Annals Hist. Comput.*, 15(4):27–75. Exact copy of the original typescript draft – Contract No. W-670-ORD-4926, U.S. Army Ordnance Department, Philadelphia: University of Pennsylvania, Moore School of Electrical Engineering (30 June).

Von Neumann, J. (1966). *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA.

Vreeken, J. (2003). Spiking Neural Networks, an Introduction. Technical report, Adaptive Intelligence Laboratory, Intelligent Systems Group, Institute for Information and Computing Sciences, Utrecht University.

Wang, B. and Vachtsevanos, G. (1991). Storage capacity of bidirectional associative memories. In *Proc. IEEE International Joint Conference on Neural Networks*, pages 1831–1836 vol.2.

Wang, L., Jiang, M., Liu, R., and Tang, X. (2008). Comparison bam and discrete hopfield networks with cpn for processing of noisy data. In *Proceedings of the 9th International Conference on Signal Processing 2008 (ICSP2008)*, pages 1708 –1711.

Wang, L.-X. and Mendel, J. (1992). Generating fuzzy rules by learning from examples. *IEEE Transactions on Systems, Man and Cybernetics*, 22(6):1414–1427.

Watta, P. and Hassoun, M. H. (2001). Generalizations of the hamming associative memory. *Neural Process. Lett.*, 13(2):183–194.

Welberg, L. (2008). Synaptic plasticity: Learning through continuing potentiation. *Nat Rev Neurosci*, 9(2):82–83.

Wilkes, M. V. (1956). *Automatic Digital Computers.* John Wiley & Sons, New York.

Wolfram, S. (1984). Universality and complexity in cellular automata. *Physica D*, 10:1–35.

Wolpert, L. L. (1991). *The Triumph of the Embryo.* Oxford : Oxford University Press.

Wolpert, L. L. (2003). *Relationships between development and evolution*, chapter 2, pages 47–63. In Kumar and Bentley (2003b), 1st edition.

Wu, Y. and Pados, D. (2000). A feedforward bidirectional associative memory. *Neural Networks, IEEE Transactions on*, 11(4):859 –866.

Xilinx (1985). *XC2064/XC2018 Logic Cell^{TM}Array.* Xilinx Inc.

Xilinx (2002a). *Virtex 2.5V Field Programmable Gate Arrays Data Sheet Version 2.8.1.* Xilinx Inc.

Xilinx (2002b). *Virtex-E 1.8V Field Programmable Gate Arrays Version 2.3.* Xilinx Inc., v2.3 edition.

Xilinx (2008). *Spartan-3 FPGA Family Data Sheet.* Xilinx Inc., 2.4 edition.

Xilinx (2009). *Virtex-5 Family Overview Version 5.0.* Xilinx Inc., v5.0 edition.

Xilinx (2010a). *Virtex-5 FPGA Configuration User Guide Version 3.9.1.* Xilinx Inc., v3.9.1 edition.

Xilinx (2010b). *Virtex-5 FPGA User Guide Version 5.3.* Xilinx Inc., v5.1 edition.

Yang, C.-C., Prasher, S., Landry, J.-A., Ramaswamy, H., and Ditommaso, A. (2000). Application of artificial neural networks in image recognition and classification of crop and weeds. *Canadian Agricultural Engineering*, 42(3):147–152.

Zhan, S., Miller, J. F., and Tyrrell, A. M. (2008). A Developmental Gene Regulation Network for Constructing Electronic Circuits. In Hornby, G., Sekanina, L., and Haddow, P. C., editors, *ICES*, volume 5216 of *Lecture Notes in Computer Science*, pages 177–188. Springer.

Zhang, Y. (2003). Transcriptional regulation by histone ubiquitination and deubiquitination. *Genes & Development*, 17(22):2733–2740.

Zhang, Y. and Reinberg, D. (2001). Transcription regulation by histone methylation: interplay between different covalent modifications of the core histone tails. *Genes & Development*, 15(18):2343–2360.

Zheng, G., Givigi, S., and Zheng, W. (2005). A new strategy for designing bidirectional associative memories. In Wang, J., Liao, X., and Yi, Z., editors, *Advances in Neural Networks – ISNN 2005*, volume 3496 of *Lecture Notes in Computer Science*, pages 398–403. Springer Berlin / Heidelberg.

Zoutendyk, J. A., Schwartz, H. R., Watson, R. K., Hasnain, Z., and Nevill, L. R. (1987). Single-event upset (seu) in a dram with on-chip error correction. *Nuclear Science, IEEE Transactions on*, 34(6):1310–1315.