ARCHITECTURES FOR NANOSCALE DEVICES

by

Jaidev Patwardhan

Department of Computer Science
Duke University

Date: _____

Approved:

_____

Professor Alvin R. Lebeck, Advisor

_____

Professor Christopher Dwyer

_____

Professor John Reif

_____

Professor Eric Rotenberg

_____

Professor Daniel J. Sorin

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2006

ABSTRACT

ARCHITECTURES FOR NANOSCALE DEVICES

by

Jaidev Patwardhan

Department of Computer Science
Duke University

Date: _____

Approved:

_____

Professor Alvin R. Lebeck, Advisor

_____

Professor Christopher Dwyer

_____

Professor John Reif

_____

Professor Eric Rotenberg

_____

Professor Daniel J. Sorin

An abstract of a dissertation submitted in partial
fulfillment of the requirements for the degree
of Doctor of Philosophy in the Department of
Computer Science in the Graduate School of
Duke University

2006

Abstract

The semiconductor industry's roadmap identifies a "red brick wall" beyond which it is unknown how to extend the historical trend of ever-decreasing CMOS device size. While architectural innovations can provide short-term relief, there is a need to explore long-term alternatives to CMOS devices and fabrication techniques. A revolutionary technology change, such as replacing CMOS, is a potentially disruptive event in the design of computing systems. Emerging technologies for further miniaturization have capabilities and limitations that can significantly influence computer architecture and require re-examining or rebuilding abstractions originally tailored for CMOS.

DNA-based self-assembly of nanoscale components is a promising alternative to CMOS that holds the potential to usher in an era of tera- to peta-scale integration. Although much of this technology is in its infancy, by studying its potential uses for building computing systems, architects can better understand its opportunities and limitations while providing feedback to scientists developing the technologies. This thesis explores the architectural challenges introduced by bottom-up fabrication of nanoelectronic circuits. The goal is to design high-performance defect-tolerant architectures within technological constraints. While our designs assume one specific technology, they are compatible with other technologies with similar characteristics.

We make four primary contributions in this thesis. First, we propose a circuit architecture that enables the construction of circuits that balance three conflicting goals: 1) regularity for the DNA lattice, 2) complexity for the circuit, and 3) defect tolerance. This enables the creation of a large number of circuit elements (nodes) with basic compute and communication capabilities, connected in a random network. Second, we adapt an existing algorithm to isolate defective nodes and provide logical structure to the random network. Third, we design a general purpose architecture (Nanoscale Active Network Architecture or NANA) that exploits this logical structure to create execution and memory networks that can execute programs. Fourth, we design a data parallel architecture (Self-Organizing

SIMD Architecture or SOSA) that exploits hardware parallelism in the network to create a high-performance defect tolerant architecture. SOSA achieves the primary goal of this thesis by attaining performance equivalent to modern processors, while operating at a lower speed and consuming lesser power.

Acknowledgments

Throughout this Ph.D. thesis, I have benefited from the support, encouragement and guidance of family, mentors, colleagues and friends. This is where I get to express my gratitude towards them.

My wife, Lavanya, has been my pillar of strength through my time in graduate school. She has helped me in innumerable ways, from providing feedback on writing and presentations, to providing a sounding board for my ideas. I hope I can provide the same level of support to her in the future.

I am thankful to both our families for supporting us through these six years. I would especially like to thank my parents who instilled in me a sense of curiosity and wonder at a very young age, and they deserve much credit for my accomplishments. My sister and brother-in-law helped make my transition from life in India to the U.S as smooth as possible.

I could not have asked for a better advisor than Alvin Lebeck. Alvy has been an excellent mentor and guide, and has been instrumental in my development as a researcher. He always seemed to know how to motivate me to do better and struck the right balance between independence and supervision.

I have also been very fortunate to have gotten to work closely with Daniel Sorin and Chris Dwyer through my thesis work. Their ideas, insights and guidance have helped me become a better researcher. I would also like to thank Eric Rotenberg and John Reif for taking the time to serve on my committee and provide feedback.

Through graduate school, I have had the support of many friends and colleagues. I would like to thank all members of the Duke Computer Architecture group, past and present, for stimulating discussions, feedback and providing unique perspectives on problems. I would specifically like to thank three members of the group who were great role

models and sources of support and guidance - Srikanth Srinivasan, Mithuna Thottethodi and Tong Li.

I am thankful for the great officemates and roommates I have had during my time at Duke - Omer Asad, David Irwin, Rahul Lakhotia, Parag Palekar, Constantin Pistol, Abhijit Vijay, Mithuna Thottethodi, and Kashi Vishwanath. They have been the source of some extremely stimulating conversations. I would also like to thank Andrew Danner and Laura Grit for some great discussions over lunch and coffee. I have been fortunate to have made other good friends at Duke, including Padmaja Ayyagari, Juhi Juneja, Suparna Kanjilal, Pramod Kolar, Vijay Natarajan, Vamsee Pamula, Shobana Ravi, Vinay Singh, and Vijay Srinivasan.

I am extremely grateful to Diane Riggs for making life as a graduate student in this department as smooth as possible. Diane is one of the most hard-working and helpful people I have ever met and I could not imagine what life would be like in the department without Diane's steady presence.

Last, but not least, I would like to thank my late grandfather, Upendra Patwardhan, for being a tremendous source of inspiration and for convincing me to attend Duke. This thesis is dedicated to his memory.

Table of Contents

List of Tables

List of Figures

1 Introduction

The development and continued scaling of CMOS technology has enabled the tremendous growth of the computer and electronics industry over the past three decades. The semiconductor industry continues to meet and even exceed the pace dictated by Moore's law [95], which states that the number of transistors that can be packed on a chip doubles every 18 months. The decrease in device size has enabled a reduction in the size and power consumption of microprocessors, while providing designers with the flexibility to implement greater functionality to match the needs of a wide range of target applications. The increase in computational capabilities of microprocessors has also been matched by a corresponding increase in the computational demands of application software that runs on them. This has been complemented by the development of software applications that depend on the new features implemented in microprocessors, resulting in a positive feedback loop between hardware functionality and software requirements. The demand for increasing computational power in microprocessors is unlikely to diminish in the near future as we rely on computers to develop new drugs [30,118], understand and process genomes [146], predict the weather [32], study natural phenomena like earthquakes [79], create highly realistic virtual environments [25,96,132] and to design and test computers [154].

However, CMOS scaling is soon expected to reach physical limits that will make it difficult, if not impossible, to build smaller transistors with the required electronic properties [66,67,68]. Researchers have developed several new devices that could replace CMOS based transistors, including carbon nanotube transistors [10,138], silicon nanorod based transistors [26,63,93], single electron transistors [8] and even transistors made using organic molecules [22,24,141]. Preliminary studies have shown that these devices could be used in building circuits that can operate at higher speeds and consume lesser power, while

being packed at higher densities than CMOS based devices. This would allow us to maintain Moore's law beyond CMOS.

Also, as CMOS technology is scaled into the nanometer range, some assumptions about circuit properties that were true for larger device sizes are invalidated. For example, until recently, it was safe to assume that dynamic (switching) power was the only significant source of power consumption and it was safe to neglect static or leakage power. It has also been safe to assume that the circuits will largely function reliably, without frequent faults or defects. The scaling of CMOS and the use of emerging technologies to build circuits could invalidate some of these assumptions. For example, leakage power is now a significant fraction of the power being dissipated in modern CMOS circuits and can no longer be neglected. As circuits get smaller, reliable operation is no longer guaranteed due to manufacturing defects or faults during operation. The invalidation of these fundamental assumptions necessitates a re-examination of the process of circuit and architecture design when using emerging technologies.

The challenge to scaling CMOS extends to the top-down manufacturing process of photolithography used to build CMOS integrated circuits (ICs). Photolithography uses a combination of light sensitive chemicals and special 'masks' that define circuit patterns, to etch the circuits on a silicon wafer. This process is extremely sensitive to impurities, and needs a clean environment to manufacture reliable devices. As the size of devices reduces, the sensitivity to impurities increases and the tolerance to variations in manufacturing steps reduces. This has led to rising manufacturing costs [3] and increasing difficulty in achieving a high level of device reliability [15].

The increasing cost of optical lithography has led to an increased interest in bottom-up manufacturing techniques like self-assembly, that require less control during manufacturing. DNA-based self-assembly [126], one specific type of self-assembly, uses the well-known assembly properties of DNA to build a scaffold-like framework in which electronic devices can be assembled. The ability to control the placement of electronic devices at specific points on the DNA scaffold is a critical requirement for the development of DNA-based self-assembly as a viable manufacturing technique. Researchers have recently made

significant progress in achieving this goal by demonstrating the placement of aperiodic patterns on a DNA lattice [37,116,123,104] and the DNA-based self-assembly of nanowire transistors [129]. DNA-based self-assembly has the potential to significantly reduce manufacturing costs and opens up possibilities of constructing large scale systems with more than 10^{12} active elements. This scale is three orders of magnitude greater than the near term projections for CMOS and is made possible by the parallel nature of self-assembly.

In this thesis, we explore the effect of one emerging manufacturing and device technology on computer architecture. We assume the use of DNA-based self-assembly of carbon nanotube based devices as the underlying manufacturing technology. Despite this assumption, the design and analysis of the architectures presented in this thesis is applicable to other technologies with similar characteristics. The rest of this chapter is organized as follows. We start with the main statement and primary contributions of this thesis (Section 1.1). Next, we briefly describe the challenges faced due to the use of DNA-based self-assembly of carbon nanotube based devices (Section 1.2), and present a short description of a defect tolerance mechanism (Section 1.3). We then present brief descriptions of two architecture designs (Section 1.4). Next, we describe our analysis of the trade-off between node complexity and control over self-assembly to improve system connectivity (Section 1.5). We conclude this chapter with an outline of the structure of the thesis (Section 1.6).

1.1 Thesis Statement and Contributions

The main goal of this thesis is to establish the validity of the following hypothesis: “It is possible to design a high-performance defect-tolerant architecture that can match or even outperform existing architectures while operating at a lower speed, consuming less power and using at most the same area, despite the assumed limitations of DNA-based self-assembly of nanoelectronic components.”

This thesis makes four primary contributions:

1. We develop a circuit architecture for DNA-based self-assembly of nanoelectronic devices that requires system designers to balance the use of simple, regular building blocks to build complex circuits, while tolerating defects,

2. we adapt an existing mechanism to provide logical structure and tolerate defects in a random network of computing blocks,

3. we design and evaluate NANA, a proof-of-concept general purpose architecture built on top of a random network of heterogeneous self-assembled nodes, and

4. we use the insight gained from NANA to design and evaluate SOSA, a SIMD architecture built on a random network of identical self-assembled nodes. The power and area estimates for SOSA obtained through circuit design, combined with its performance evaluation through simulation help establish the validity of the primary hypothesis.

1.2 Impact of DNA-based Self-Assembly of Nanoelectronic Components on Architecture Design

DNA-based self-assembly of nanoelectronic devices is a promising technology that may be used in constructing circuits by placing aperiodic patterns on a DNA scaffold structure, and assembling electronic devices at certain locations on the scaffold [109]. However, the assumed capabilities of self-assembly impose certain constraints on the size and complexity of the circuits that can be self-assembled. This limitation in size is unlikely to change without significant yield improvements in building DNA scaffolds. Another limitation of self-assembly is the limited or lack of control over the placement and orientation of these self-assembled circuit blocks ("nodes"). However, one of the primary advantages of self-assembly is the ability to manufacture a large number of these computational blocks in parallel. However, a lack of control over this parallel self-assembly can result in a random network of computational blocks once they have been connected through a second self-assembly step.

To design a computer system using these random networks of self-assembled nodes, computer architects must:

(i) understand the characteristics of the random networks,

(ii) devise a technique to impose logical structure on the random network,

(iii) implement a mechanism to achieve defect tolerance,

(iv) design an architecture that can exploit the large number of nodes, including developing an instruction set and execution model and,

(v) determine the functionality that must be implemented in the limited sized nodes.

At minimum, these nodes must have the ability to communicate with each other, perform some computation, and store some state. Step (v) is critical since the capabilities of the nodes determine the capability of any computer system built using them, and an efficient node design can help maximize this capability. This thesis explores each of these steps and presents the design and evaluation of two different architectures built on a random network of self-assembled nodes.

1.3 Defect Tolerance

As discussed in the previous section, defect tolerance is one of the primary requirements of any architecture built using emerging nanotechnologies. We design a mechanism for achieving defect tolerance by modifying an existing broadcast algorithm to isolate defective nodes and to impose a logical structure on the random network of nodes [110]. This allows us to connect all functional nodes that can be reached from the node where the broadcast is initiated. The defect tolerance mechanism requires very simple hardware in each node and is able to tolerate a large node defect rate (up to 30% defective nodes). Both the architectures developed in this thesis use this defect tolerance mechanism to isolate defective nodes as well as to impart logical structure on the random network of nodes. In the next section, we present a brief overview of both architectures.

1.4 Architectures for Self-Assembled Nanoscale Devices

The first architecture (Nanoscale Active Network Architecture [111] or "NANA") targets general purpose workloads and supports the traditional Von-Neumann programming model and a memory system. This proof-of-concept architecture divides a heterogeneous random network of nodes into smaller groups of nodes called "cells" and constructs logically disjoint execution and memory networks within the cells. The isolation of the two networks reduces the physical resources required in each node in the system. Once the two logical networks are constructed, execution packets consisting of instructions and data operands in a specific order are routed in the execution network searching for appropriate resources to perform the operations specified in the instructions. A thorough evaluation of NANA using simulation and modeling reveals that it is unable to achieve good performance because of two primary reasons: 1) low node utilization and 2) bottlenecks in the memory system. The evaluation of NANA provides insight into performance problems that arise due to the execution model and highlights the limitations of the node interconnection network. Thus, while NANA fails to prove the thesis statement, it provides valuable insight into possible strategies that will or will not work in building a high performance architecture. It also demonstrates that it is possible to build a functional architecture that can tolerate high node defect rates.

The second architecture (Self-Organizing SIMD Architecture [112] or "SOSA") aims to achieve high node utilization by targeting data parallel workloads and supports the data parallel programming model. The architecture divides a homogenous random network of nodes into "cells", but each cell is further divided into computational blocks called "processing elements" or PEs. All PEs execute the same instructions, but operate on different data and are connected in a logical ring. This simplifies the programmer's view of the set of PEs and allows simple communication between PEs. Since all PEs execute the same instruction, a large fraction of nodes are active at the same time, allowing SOSA to better exploit the large number of nodes available. We perform a thorough evaluation of SOSA using a detailed simulator and circuit models of the node. We use a variety of circuit design tools - off the shelf (VHDL, HSPICE), as well as custom layout tools developed specifi-

cally for the underlying technology [41] to build a circuit model of the node, and to estimate its size and power consumption. We demonstrate that SOSA can tolerate node defects with the RPF algorithm by implementing fail-stop behavior for critical logic blocks within each node [107]. By simulating the execution of various programs on SOSA, we demonstrate that SOSA supports the primary statement of this thesis by exceeding the performance of existing architectures while operating at a lower speed and consuming lesser power.

1.5 Improving Network Connectivity

The process of self-assembly can be modified to provide control over node placement, orientation and creation of inter-node links. This adds additional complexity to the manufacturing process, but can result in simpler and more structured networks. Alternatively, the communication logic within each node can be augmented to support more complex protocols over inter-node links, thus improving system connectivity. We explore the trade-off between control over self-assembly and the added complexity required within each node to achieve good system connectivity [108]. We find that control over node placement and orientation results in better connected networks. However, by allowing each node to treat an inter-node link as a shared medium (i.e., a bus), we can achieve nearly the same degree of connectivity in an unstructured network.

1.6 Thesis Outline

This thesis is organized as follows. Chapter 2 provides background on DNA-based self-assembly, as well as carbon nanotube and other emerging device technologies. Chapter 3 studies the impact of DNA-based self-assembly of nanoelectronic components on architectural design, and describes the self-assembled circuit architecture that is used in the rest of this thesis. Chapter 4 describes a mechanism for tolerating defects in a random network of self-assembled computing blocks. In Chapter 5, we present the design and evaluation of NANA, a general purpose architecture built using random networks of heterogeneous self-assembled nodes. We present the design and evaluation of SOSA, a SIMD architecture

built using random networks of homogenous self-assembled nodes in Chapter 6. We describe a modular design for fail-stop SOSA nodes and explore how such nodes enable the system to tolerate increased device failure probabilities in Chapter 7. In Chapter 8, we explore the trade-off between control over self-assembly and node complexity to maximize network connectivity. Chapter 9 discusses other research related to this thesis and Chapter 10 concludes this thesis.

2 DNA-Based Self-Assembly of Nanoelectronic Devices

Photolithography has been the primary manufacturing technique used to build microprocessors and other integrated circuits for the last three decades. The semiconductor industry has been able to improve performance of circuits by reducing the size of devices manufactured. However, as CMOS devices shrink into nanometer scales, further scaling is difficult and is approaching hard physical limits. This has led to a search for alternative nano-scale technologies that might replace CMOS based devices. In this chapter, we provide background information about DNA-based self-assembly and some promising nanoelectronic devices that have the potential to replace CMOS as the dominant technology for manufacturing microprocessors in the future. There are several promising candidates, including carbon nanotube based devices, single electron transistors, silicon nanowire transistors and organic molecules. Each of these devices have their advantages and disadvantages and are the subject of much research to improve their properties and make them suitable for use with manufacturing technologies of the future.

Improvements in the top-down manufacturing process of photolithography have allowed CMOS devices to be scaled into the nanometer range. However, as device sizes shrink, the costs associated with photolithography have been increasing rapidly. DNA-based self-assembly is a bottom-up manufacturing technique that has the potential to replace photolithography. Self-assembly techniques have the advantage of requiring low control over the manufacturing process and potentially enable the parallel assembly of a large number of devices at once resulting in lower manufacturing costs. For any combination of device and manufacturing technologies that are picked to replace CMOS, it is critical that they be compatible with each other. This makes the combination of DNA-based self-assembly and carbon nanotube based devices promising, since researchers have already demonstrated the ability to link DNA and carbon nanotubes [36].

We break our discussion of the underlying technologies into four parts, starting by discussing the reasons why alternatives to CMOS technology are required in the coming decades (Section 2.1). Next, we describe DNA-based Self-Assembly (Section 2.2), and various nano-electronic devices that could be viable candidates to be used with DNA-based self-assembly (Section 2.3). Finally, we describe the use of DNA-based self-assembly and carbon nanotubes (Section 2.4) to build circuits.

2.1 End of Silicon Based CMOS

Silicon based CMOS devices have provided a stable platform for manufacturing complex microprocessors for over two decades. This has been achieved through advances in lithography, solid-state physics, and chemistry that have enabled a steady scaling down of device sizes. The semiconductor industry continues to meet and even exceed the pace dictated by Moore's law [95], which states that the number of transistors that can be packed on a chip doubles every 18 months. However, CMOS is nearing a point where further scaling is difficult if not impossible because of physical limits to building smaller transistors [66,67,68]. CMOS scaling faces additional hurdles due to the top-down manufacturing process of photolithography used to build CMOS integrated circuits (ICs). Photolithography uses a series of special 'masks' that define the patterns of circuits at various levels on the silicon wafer. ICs are manufactured using a combination of light sensitive chemicals, specific frequencies of light, metal interconnect and the masks. This process requires a very clean environment (typically less than 100 impurity particles per cubic meter of air). As the size of devices reduces, the sensitivity to impurities increases and the tolerance to variations in manufacturing steps reduces. As we progress further down on the nanometer scale, it is very hard to maintain the precision required during lithography. At nanometer scales, there is also increased vulnerability to electron tunneling, stray inductances and capacitances, transient faults caused by radiation, and defects. Setting up a manufacturing utility for 300mm wafers using 90nm technology costs more than one billion dollars [3]. This price will increase rapidly as the precision required during manufacturing increases. In addition, each chip requires many masks during the manufacturing process. As we shrink

device sizes, the masks need to be manufactured with greater precision. These mask sets already cost over one million dollars each to manufacture [3] and as device size shrink, these costs will increase.

The combination of exponentially increasing costs, reduced reliability, decreasing tolerance to manufacturing variations, and the rapidly approaching physical limits of scaling have led researchers to identify technologies that could replace photolithography and CMOS in the future. In the next section, we describe one bottom-up manufacturing technology (DNA-based self-assembly) that has the potential to replace photolithography and reduce costs.

2.2 DNA-based Self-Assembly

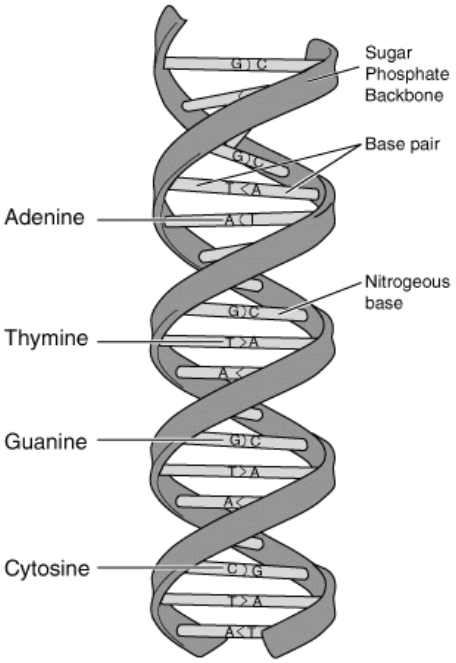


Figure 2-1. DNA double helix

DNA-based self-assembly is a bottom-up manufacturing process that uses the well-known assembly properties of DNA to build a lattice-like scaffold. Deoxyribonucleic acid or DNA is the primary carrier of genetic information in biological organisms. DNA consists of a chemically linked chain of molecules known as nucleotides, each of which consists of a sugar, a phosphate and one of four 'bases': adenine (A), thymine (T), cytosine (C) and guanine (G). A single nucleotide chain is also known as single-stranded DNA or ssDNA and can be defined by the sequences of bases present along the chain. The most stable form of DNA is a pair of nucleotide chains that link to form the well known double helix structure shown in Figure 2-1. The nucleotide chains pair through hydrogen bonding of the bases, where adenine pairs with thymine and cytosine pairs with guanine. While other base pairings are possible, they are not as stable as the A-T and C-G pairings.

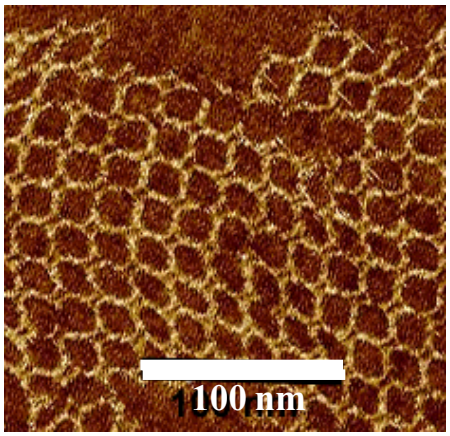


Figure 2-2. DNA lattice with repeating cavities [105]

The precise binding rules of DNA make this a promising technique to use with nanoscale devices. By specifying a particular sequence of base pairs on a single strand of DNA, we can exploit the base-pair rules as organizational instructions [120,126]. A region of ssDNA and its complement can act as 'tags' (T and T') for orienting objects in 3-space. The sequence of bases on the ssDNA must be carefully designed to minimize the probability of 'partial matches' where some non-complementary bases are forced to match due to the structure of other proximal base pairs [39]. Such carefully designed DNA tags can be used to create 2D patterned nanostructures [149] by combining the right fractions of synthetic ssDNA tags and annealing them by heating beyond their melting point and cooling slowly. Although the resulting structure can be used to perform computation [6, 119], we are interested in DNA's ability to self-assemble into large-scale nanostructures. Of particular interest to this thesis, is a structure that creates a 'waffle'-like lattice with repeating cavities [81,151,152] (see Figure 2-2). This type of lattice has been experimentally demonstrated and can achieve sizes that extend beyond 3 microns on each side (i.e., > 150 cavities on a side). This scaffold can be used to place and interconnect devices by forming tags at specific lattice points [150] and using a technique for attaching the appropriate complementary ssDNA tags [36]. The DNA self-assembly technique is independent of the specific nanoelectronic device used, however the limited size of each lattice (node) presents challenges for creating large sophisticated circuitry. Before we describe methods for building circuits,

we discuss some promising emerging devices. We focus our discussion on one specific device technology that is used in the rest of this thesis.

2.3 Emerging Nanoelectronic Devices

As the scaling of CMOS devices faces technological hurdles, researchers have been searching for new nano-scale devices that could potentially replace CMOS in the long term. There are a number of choices for building nanoelectronic devices and wires [10,26,63,93,138,141]. These include nanocells [141], silicon nano-rods [93], carbon nanotubes [10,138], and silicon nanowires [26,63], most of which have the potential for building transistors that are smaller than conventional CMOS transistors. Carbon nanotubes (CNTs) [64,87] are cylindrical molecules of carbon that resemble rolled sheets of graphite, and can be single-walled (SWNT) or multi-walled (MWNT). Single-walled nanotubes can be metallic or semiconducting depending on a property known as their 'chirality', which describes the atomic structure of the CNT.

One promising device is a field effect transistor constructed using carbon nanotubes (CNFET) [70,75,138] in which application of a gate voltage [49, 138, 148] modulates the conductivity of a semiconducting nanotube. Recent advances enable separating metallic nanotubes from semiconducting nanotubes, precisely controlling the length of individual nanotubes [89,135,155] and self-assembly of carbon nanotube based electronic devices [57]. Therefore, we could use both types of carbon nanotubes to construct logic gates, memory (e.g., with cross-coupled NOR gates), and circuit interconnect. The fact that CNFETs are amenable to self-assembly makes this an attractive alternative, or supplement, to silicon device technology. CNFETs are naturally p-type, but research has demonstrated the ability to electrostatically dope them to be n-type [10].

It is useful to compare the estimated latency of CMOS devices and CNFET based devices. We compare the delay of NAND gates in CMOS against CNFET NAND gates. The data for the CNFET NAND gates are based on empirical SPICE simulations [40,17, 94]. The CMOS data is obtained from the ITRS roadmap [14] and from standard industry processes [4,1,2]. The ITRS data represents target delays for specific CMOS technology

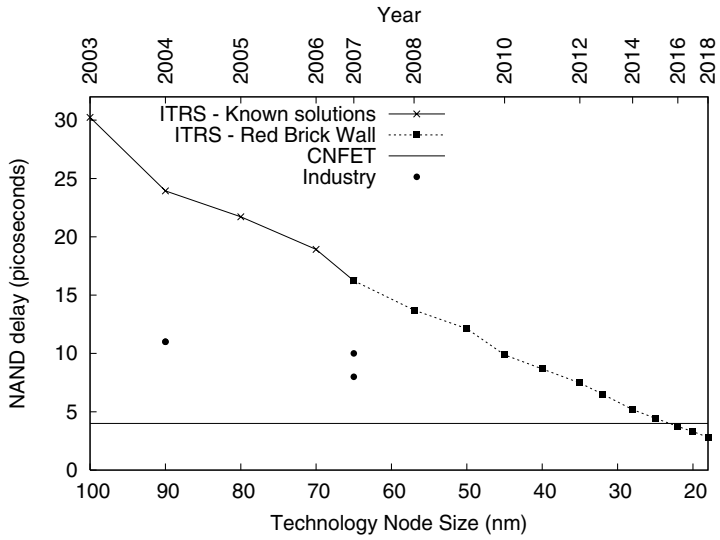


Figure 2-3. CMOS vs. CNFET

sizes and is not measured data from devices. In fact, there are no know solutions for manufacturing CMOS devices smaller than 65nm in bulk (Intel is currently developing a 45 nm technology, details of which are not publicly available). Figure 2-3 compares the estimated latency of CMOS NAND gates against the latency of CNFET based NAND gates. The solid line represents CMOS delay estimates that have demonstrated solutions. The dashed line represents 'desired' CMOS delay estimates that are part of the 'red brick wall' (no known solution exists). The dotted line at the bottom represents the CNFET delay. We can see that even with current CNFET based devices, the delay is lower than most CMOS technology nodes. This demonstrates the potential of CNFET devices to provide an alternative to CMOS based transistors in the future.

For CNFETs to replace CMOS based transistors, they must be able to achieve comparable or higher switching speeds. One important property that determines maximum switching speed in a particular technology is the charge carrier mobility. Silicon and germanium based semiconductors have mobilities that are less than 2000 cm^2/V-s [88]. Recent work in measuring the mobility of charge carriers in carbon nanotubes [33] indicates that the mobility is likely to be over 100,000 cm^2/V-s. Thus, the delay estimate obtained from the SPICE model is probably pessimistic. It is likely that CNFET based devices will operate at frequencies as high as 1 THz [18], and have already been tested at frequencies of over

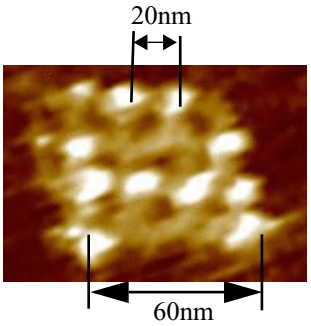


Figure 2-4. A DNA scaffold for nano-electronic circuits with patterned letter A (from [104])

10 GHz [121]. Next, we describe how DNA-based self-assembly could be used to manufacture CNFET based devices and build circuits.

2.4 DNA-based Self-Assembly of Carbon Nanotube Electronics

Researchers have demonstrated the ability to connect DNA to carbon nanotubes [36], enabling an assembly process that allows the placement of carbon nanotube based electronic circuits on a DNA lattice. Other potential materials (e.g., nanorods, silicon nanowires) could be substituted for the carbon nanotubes without loss of generality. A key requirement of this self-assembly process is the ability to control the placement of the electronic devices at specific points on the DNA scaffold to form a circuit. Researchers have recently taken two significant steps towards this by demonstrating the placement of aperiodic patterns on a DNA lattice [104,116,123] and the DNA-based self-assembly of nanowire transistors [129]. Figure 2-4 shows an atomic force microscope image of a DNA lattice with the letter "A" patterned on it. This is a critical step towards building DNA scaffolded electronic circuits (nodes). Current limitations of the self-assembly process place restrictions on the size of the DNA lattice that can be constructed, which in turn limits circuit size. While the size of individual nodes is small, the parallel nature of self-assembly enables the construction of a large number ($\sim 10^9$-10^{12}) of nodes. These characteristics impose significant implications on any circuit architecture that is built using this assembly process. In the

next chapter, we discuss the implications of DNA-based self-assembly of carbon nanotube electronic devices on circuit architecture and develop a nanoelectronic circuit architecture.

3 Implications for Architecture Design

The previous chapter provided background on DNA-based self-assembly, emerging nanoelectronic devices and how the two techniques could potentially be combined to build circuits. Before we build circuits, it is critical that we gain a thorough understanding of the capabilities and limitations of self-assembly due to its fundamentally different nature (as compared to photolithography). We expect that limitations of self-assembly in the near future will restrict our ability to place and route logic devices and interconnect on a DNA lattice, resulting in space overhead that is not typically found in CMOS based devices. For example, while modern CMOS processes rely on more than ten layers of metal interconnect, DNA-based self-assembly is likely to be restricted to two layers in the near future. Any circuit design methodology must account for the routing overheads imposed by the limited metal layers. In this chapter, we present the implications of DNA-based self-assembly of carbon nanotube devices on circuit and systems architecture and develop a nanoelectronic circuit architecture that could be used to build computational circuit blocks. The circuit architecture uses aperiodic patterns on a DNA lattice to place nanoelectronic devices. This enables the construction of small circuits (nodes) that can perform computation or communicate with other nodes. We can then interconnect these nodes using wires. We propose the use of metallized DNA links, grown between nodes to create an interconnection network.

We analyze the implications of this circuit architecture on the design of system architectures. While our analysis assumes the use of DNA-based self-assembly of carbon nanotube based devices, it is applicable to other technologies with high defect rates and a loss of precise control over parts of the fabrication process (e.g., process variability and defects in scaled-CMOS). By using small, replicated building blocks to create larger systems, scaled-

CMOS based designs can mitigate the effect of increasing defect rates and process variability. We make the following contributions in this chapter:

1. We propose a method for building circuits by placing carbon-nanotube based electronic devices in the cavities of DNA-lattices based on an analysis of the implications of DNA-based self-assembly on circuit architectures, and
2. We determine that the assumed characteristics of DNA-based self-assembly limit us building small computational nodes with ~10,000 transistors, which is significantly smaller than conventional CMOS designs. The design of each node must balance communication, computation and defect tolerance capabilities within technological limits.

The rest of this chapter is organized as follows. We start with the implications of using DNA-based self-assembly of carbon nanotube electronics on circuit architecture design (Section 3.1). We then describe the basic circuit building blocks assumed in the rest of the thesis (Section 3.2). Next, we describe the architectural implications of using these self-assembled circuit building blocks (Section 3.3) and follow that with a list of challenges that must be overcome in the design of an architecture using this technology (Section 3.4). We conclude the chapter with a summary of the key concepts presented (Section 3.5).

3.1 Implications for Nanoelectronic Circuit Architecture

To use DNA-based self-assembly of carbon nanotube electronics as the manufacturing and device technology, a nanoelectronic circuit architecture must strike a balance between 1) the *regularity* of DNA self-assembly patterning capabilities, 2) the *complexity* required for sophisticated system designs and 3) *tolerance* to the inevitable defects present in nanoscale systems. The remainder of this section elaborates on each of these issues, focusing on the fundamental differences between this nanoarchitecture and current CMOS based architectures.

3.1.1 Regularity

While the design of CMOS based circuits can be simplified by the use of regularity (e.g., standard cell VLSI), regularity is not a fundamental requirement. However, only periodic arrays of identical unit cells have been demonstrated on a large scale using DNA self-assembly technology. DNA self-assembly has a potential limitation in that the probability of incorrect tag matches increases as the number of unique tags increases. For each type of connection, we need a unique pair of complementary ssDNA tags. With more types of connections and a fixed number of base-pairs per tag, the tags become more similar (i.e., differ in fewer base-pairs) and partial matches become more likely. For example, if a functionalized nanotube binds to a partially matched tag, then it is in the wrong position. This situation is analogous to the Hamming distance [54] between encodings of symbols; if we need to encode more symbols with the same number of bits, then the Hamming distance is smaller and the probability of an error is greater. Minimizing the number of tags reduces the chances of partial matches, which could cause positional defects, during annealing. Therefore, repetitive structures are desirable, and circuit and system designers should strive to use them as much as possible.

3.1.2 Complexity

Design complexity is a function of the number of different component types and the placement of these components. Current CMOS based circuits can arbitrarily place hundreds of millions of devices (both nFET and pFET) and wires with precision on the order of 0.10μm. This precision is achieved by using photolithography to specify exactly where each individual component belongs. With the combination of carbon nanotube devices and DNA self-assemblies, we are trying to develop circuits that can perform useful computation. The components required to build circuits can be limited to CNFETs (as active devices), nanotube wires and metal plating for connecting wires. However, with DNA self-assemblies, we cannot specify component placement at the micro-scale with nearly the same degree of accuracy as CMOS. Complexity must be introduced without requiring a large number of

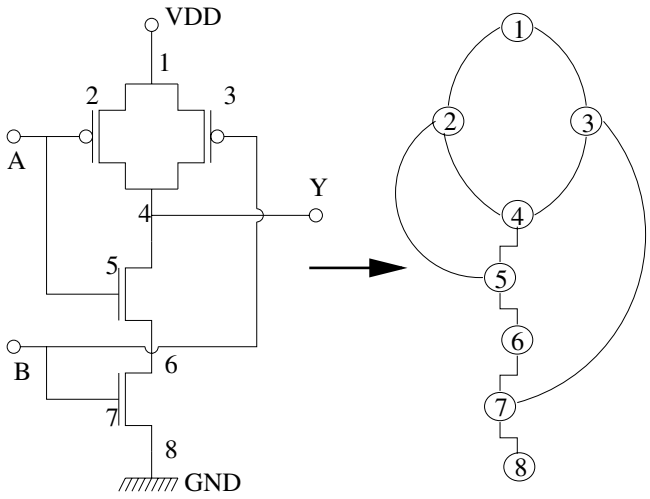


Figure 3-1. Two input CMOS NAND gate, and equivalent graph

tags. This mirrors the desire to use regular structures that minimize the number of tags. However, regular structures typically limit complexity.

Thus, the utility of self-assembled DNA arrays depends on the amount of complexity that we can introduce at various abstraction levels without causing an intractable number of partial matches. Consider a graph generated from the netlist of a transistor-level design of a combinational circuit (Figure 3-1). The vertices are transistor terminals and the edges are wires connecting the device terminals. A two-input CMOS NAND gate has eight vertices, and most combinational circuits require multiple NAND gates. Clearly, the naive approach of assuming a unique tag for each vertex in the graph requires a large number of unique tags (even ignoring fan-out issues). This will cause too many partial matches that create bridging faults (shorts), rendering the circuit mostly useless.

3.1.3 Defect Tolerance

A defect is a permanent physical fault introduced during fabrication. We consider two types of defects: functional and positional. A *functional defect* corresponds to a component that does not perform its specified function (e.g., a transistor that does not conduct when it should). A *positional defect* corresponds to a (functionally correct) component that is

placed incorrectly. Both CMOS and DNA self-assembled nanoelectronics can incur functional defects, but only self-assembly is likely to incur positional defects. Positional defects can be both defects of omission and commission. An omissive positional defect occurs when a component is not placed where it belongs. A commissive positional defect occurs when a component is placed where it does not belong (i.e., the partial match described above). Omissive defects behave similar to functional defects. Commissive defects are more dangerous, since they can behave like bridging faults. For example, a misplaced nanowire could cause a short between power and ground or it could change circuit functionality in unpredictable ways (e.g., by erroneously connecting the output of a gate to its input).

In CMOS based circuits, there is limited support for defect tolerance. Photolithographic placement of components is a mature technology that incurs few defects. However, in architectures with hundreds of millions of devices and wires, defects will still occur with some probability (i.e., yield is less than 100%). CMOS microchips are thus tested for defects. If a defect is uncovered and it cannot be tolerated, the chip is discarded. However, some limited number of defects can be tolerated. For example, a defect in a cache or memory cell can be tolerated by systems that provide redundant cells and allow for re-mapping. Tests on the self-assembled circuits must be simple to allow the testing of a very large number of components. Ideally, each circuit must include basic self-test circuitry that can be triggered by external inputs.

Functional defect rates for carbon nanotube devices and positional defect rates for DNA assembled nanoelectronics are currently unknown due to the relative immaturity of the technologies. Functional defect tolerance could be achieved with the same techniques used in CMOS, since the problem is not fundamentally different. Tolerance of commissive positional defects, however, is a new challenge. Because of the unknown positional defect rates, the assembly approach used in this thesis is to first strive to minimize positional defects by exploiting regularity in DNA self-assemblies. However, as complexity increases and regularity decreases, the probability of positional defects increases, thus more sophisticated circuitry will require more defect tolerance.

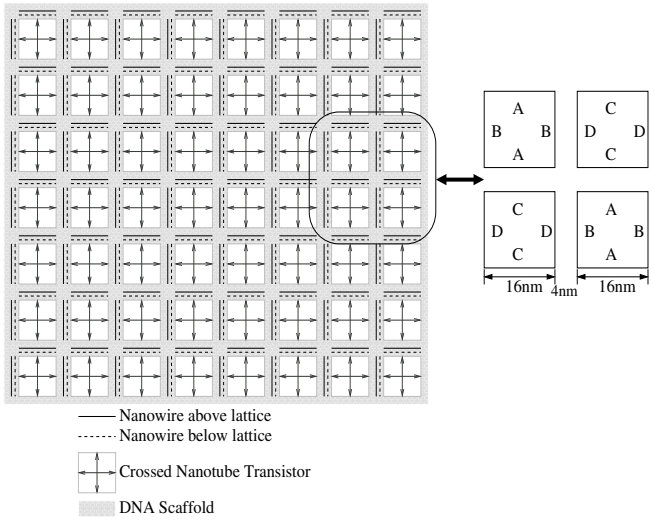


Figure 3-2. DNA Scaffold for Nanoelectronics

Any circuit architecture that uses DNA-based self-assembly of nanoelectronic devices as the manufacturing process must strike a balance between the three conflicting goals of regularity, complexity and defect tolerance, as described in this section. Next, we develop one such nanoelectronic circuit architecture that could be used to build the computing blocks assumed in the rest of the thesis.

3.2 Nanoelectronic Circuit Building Blocks

This section describes a nanoelectronic circuit architecture (shown in Figure 3-2) with structures based on a grid of CNFETs interconnected with conducting carbon nanotubes. At a high level, our proposed design addresses the conflicting goals of regularity and complexity by placing identical unit cells in the cavities of an aperiodic patterned DNA lattice. The lattice is regular in structure, but it has aperiodic binding points which can be used to connect the unit cells in complex patterns. This highlights a key difference between the proposed assembly process and existing approaches. Current nanoelectronic architectural approaches assume regularity in both the structure and the interconnect. We first present our initial proposed unit cell and then the proposed lattice. We then discuss how multiple building blocks could be self-assembled into a larger system. Finally, we describe how this system could interface with external circuitry.

3.2.1 Exploiting Regularity: A Replicated Unit Cell

The proposed unit cell in our design is a three terminal CNFET sitting in the cavity of a DNA lattice. To place the CNFET in the cavity, we need to functionalize one semiconducting and one conducting nanotube such that they bind to the complementary ssDNA tags on the cavity edges and form a cross. We assume one of the nanotubes is wrapped in a thin insulating layer, such as SiO_2 [48]. The conducting nanotube functions as the gate of the CNFET.

Using carbon nanotubes of a short length (~16nm) precludes commissive positional defects in which a carbon nanotube binds in two different cavities. By using two sets of tags in alternating cavities in each dimension (see Figure 3-2) and by using carbon nanotubes of a precise length, a nanotube cannot span across the DNA lattice to another cavity with the same tags. The distance between adjacent cavities is only 4nm, so if the same tag is used in adjacent cavities, then a nanotube may bind across the lattice arm rather than within a cavity. Using a checkerboard pattern of alternating tags with sufficient Hamming distance eliminates positional commissive defects. This approach requires carbon nanotubes of a precise length, which may be possible using a sonochemical method [87] to cut the originally long nanotubes into short segments and then using size-exclusion chromatography to separate the nanotubes by their length. This technique must be applied to both the semiconducting and conducting nanotubes.

We can augment this unit cell with short conducting carbon nanotubes that lie adjacent to the cavity on both the top and bottom of the DNA lattice. The short nanotubes are far enough apart to avoid cross-talk and may also be wrapped with an insulating polymer if necessary. The nanotubes initially would not intersect to form complete circuits. Instead, an electrical connection between nanotubes must be explicitly created by specifying an appropriate tag on the DNA lattice to which a gold nanosphere could bind. The nanosphere nucleates metal ions to form the connection with the help of an electroless plating process [16, 74]. Similarly, connecting transistors may require specifying whether the device connects to the top or bottom conducting nanotube. Forming these connections is where we add

complexity to our design, and we explain how to introduce this non-regular patterning in Section 3.2.2.

The unit cell design fosters regular, repetitive structures. All nanotubes are the same length (16nm) and we require five sets of nanotubes that are functionalized with different tags. Four sets of nanotubes are used for the CNFETs; two semiconducting sets and two conducting sets. This corresponds to the two tag sets of the checkerboard pattern of cavity tags. A nanotube from one set can bind to any cavity with a complementary tag. Similarly, the interconnect nanotubes (the fifth set) can bind adjacent to any cavity directly on either the top or bottom of the DNA lattice in either the vertical or horizontal direction. This approach enables the use of a regular pattern for the base DNA lattice scaffolding.

3.2.2 Introducing Complexity: An Aperiodic Pattern for Interconnecting Unit Cells

Our building block, while regular in structure, has aperiodic binding points for connecting together the nanowires of the unit cell. This aperiodic pattern could be achieved through either sequential assembly of tiles, extending recent work on one-dimensional aperiodicity [150] to two dimensions or through careful design of the DNA strands that comprise the unit cell to minimize assembly steps [104].

We could now potentially construct complex circuits by specifying the electroless plating points in the DNA lattice. For each of the top and bottom of the lattice, the plating point options include: the three transistor terminals to nanowire, interconnect nanowire in the vertical directions North and South, and interconnect nanowire in the horizontal directions East and West. We assume that to create a straight-through connection in the vertical direction requires both the North and South connections; similarly, both the East and West connections are required for a straight connection in the horizontal direction. We could build pass-throughs from the top-level interconnect to the bottom-level by connecting a transistor terminal to both interconnects.

Only a single tag on the DNA lattice is required to specify the plating points where the gold nanospheres can bind on the lattice. It is this tag that has the aperiodic pattern, and gold

will bind only where the tag appears. We note that this approach minimizes positional defects since the nanotubes are of specific lengths that can only bind in the appropriate positions of the lattice. In contrast, if we used long nanowires to connect distant points, then the number of tags to which they could potentially incorrectly bind is the number of tags on the circumference of a circle with radius equal to the nanowire's length.

3.2.3 Large-scale Interconnection of Circuit Nodes

The computational capabilities of the proposed building block (node) is limited by the size of the DNA lattice. Increasing the computing capacity requires interconnecting multiple building blocks. Using inexpensive laboratory equipment we could simultaneously self-assemble as many as 10^{12} identical, but small, nodes. This number of nodes would cover an area larger than two hundred 300 mm wafers. Although the size of an individual node is well above the minimum feature size of photolithography, the number of nodes fabricated through self-assembly limits how heavily the overall process can rely on silicon fabrication processes. Self-assembling nodes onto a substrate at well-defined places is also difficult without "naming" each placement site (pick and place methods would be difficult to scale to this number of components). Even with DNA tags on the substrate, the nodes are not guaranteed to fall into place precisely.

Most conventional architectures require precise placement and interconnection between circuits. Therefore, even if we could use a conventional photolithographically patterned network to interconnect nodes, the result would be a random interconnection due to the random placement of nodes on the substrate. This is the sacrifice a self-assembly process imposes: precision and control exist only at small length scales (e.g., < 3 micron, for now). One solution to this problem involves a large scale self-assembling process that can potentially interconnect nodes on a substrate using another form of DNA-based self-assembly. Individual DNA strands self-assemble between node edges, providing a scaffold for metal that forms an electrical connection. Researchers have previously demonstrated highly conducting wires made by coating DNA with metal [85,86,152]. This larger scale process is not expected to deliver the precise control found in the earlier process used to

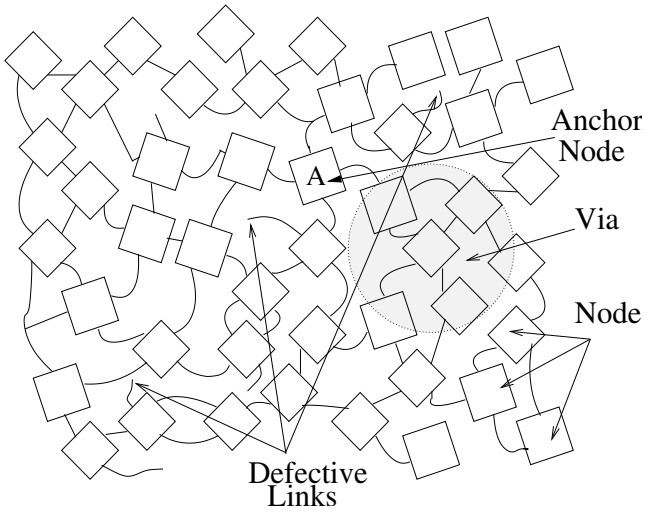


Figure 3-3. Schematic of self-assembled network of nodes

assemble the nodes, but it has the potential to fabricate single wire interconnections between the edges of the nodes, as illustrated in Figure 3-3. The lack of precise control during this process results in the formation of a random network of nodes that contains defective nodes and links. Next, we describe how this random network of nodes can be interfaced with external circuitry.

3.2.4 External Interface

The random network of nodes requires an interface with the external world in order to connect to an external power supply (V_{dd} and Gnd), as well as for communication with external circuitry. To simplify power and ground connections and to reduce routing overhead, we propose the use of two conducting planes that are parallel to the DNA-lattices. These planes provide power and ground, and are electrically insulated from the conducting wires around the DNA-lattice by a plane of insulating material (see Figure 3-4). The existence of the power and ground plane reduces routing overhead by allowing circuits to connect to power and ground using a vertical conductor that breaks through the insulating plane. To create these external connections, first the vertical conductors would be attached at appropriate places on the node using self-assembly. Next, we can cover each node with a thin insulating layer, either by self-assembling the layer, or by depositing it. Finally, the metal layer can

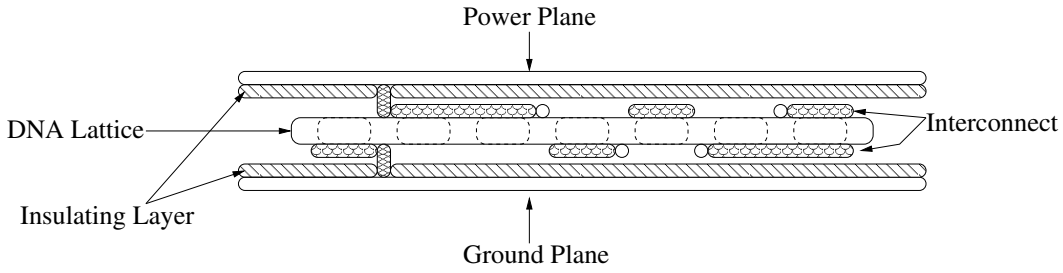


Figure 3-4. External power and ground planes

be deposited to complete the power (or ground) plane. The insulating layer must be thin enough for the vertical conductor to pass through and make contact with the metal layer.

Communication with external circuitry (CMOS or other) occurs through a metal junction ("*via*") that overlaps several nodes but interfaces with the network of nodes through a single "*anchor node*". There may be several via/anchor node pairs in large networks. Figure 3-3 shows a diagram of a small network of nodes, including regions with defective links, and a via/anchor. In the rest of the thesis we use the term "anchor" to refer to an anchor node and its corresponding via.

3.2.5 Summary

In this section, we have proposed a circuit architecture that could be used with DNA-based self-assembly of carbon nanotube electronics. This architecture relies on a simple replicated unit cell that consists of a pair of nanotubes placed in the cavity of a DNA-lattice. We then use aperiodic patterns to introduce the complexity required to build larger circuit blocks (nodes). Finally, we propose the use of self-assembly at a larger scale to connect nodes, resulting in the formation of a random network of nodes. Next, we explore the impact of our proposed circuit architecture on the design of system architectures.

3.3 Architectural Implications

The process of using DNA-based self-assembly to create nanoelectronic circuits presents several challenges that must be addressed when designing a system. The three primary aspects of the process are 1) small-scale control of placement and connectivity within a

single node, 2) large-scale randomness in node placement and interconnection, and 3) high defect rate. These three aspects significantly impact architectural decisions, particularly since conventional architectures assume precise control at both the small and large-scale.

3.3.1 Small-scale Control

The ability of DNA-based self-assembly to achieve only small-scale control impacts architectural decisions in several ways. Three of the most significant are: limited space, limited coordination, and limited communication.

Limited space. Each limited sized node can fit about 22,500 CNFETs (~3μmX3μm), however, on-node interconnect will reduce efficiency (since routing is limited to two layers) and it is unlikely that the usable number will reach even 50% of this. Furthermore, a portion of each node must be allocated as a "pad" for the DNA interconnect to other nodes. These two factors can dramatically reduce the usable area on a node. This limited node size presents a trade-off in node design. At one extreme, we could design just a single node type that contains both computation and storage capabilities. However, since the storage and computation circuits must share the node, each may be severely limited in capability. Alternatively, we could design a few specialized node types, some devoted to computation and others devoted to storage. Even when designing a specialized node, the limited space impacts architectural decisions. For example, large state machines are not an option since there is insufficient space for state storage. Similarly, the number of bits available in a storage node may be limited, thus affecting an architecture's word size.

Limited communication. Without large-scale control, there is limited communication among nodes. Each node has four neighbors and there is no long haul communication. Furthermore, the connections from a node to each of its neighbors is limited to a single wire. Although the degree of each node or the number of connections between neighbors could be increased, each connection occupies precious edge space. Conventional designs exploit multiple metal layers for long-haul communication and large-scale control to create multi-wire connections between components. Therefore, the architecture must avoid relying on sophisticated communication hardware.

Limited coordination. Conventional CMOS designs rely on precise control during fabrication to create sophisticated circuits (e.g., a 64-bit adder with carry lookahead). For our technology, if the most sophisticated node is a full-adder, then it is unlikely that 64 such nodes can be coordinated during self-assembly to implement a 64-bit carry lookahead adder. Coordination among nodes is very limited and it is difficult to *apriori* configure a group of nodes to operate in a coordinated manner. Each node can perform only limited coordination with its immediate neighbors. Thus, the architecture cannot rely on static configuration of the nodes into complex structures.

3.3.2 Large-Scale Randomness

Our proposed self-assembly process provides excellent control at the small-scale, however it cannot easily achieve such control at large scales, resulting in an unstructured network of nodes described in the previous section. The architecture and machine organization must accommodate arbitrary placement of nodes, and cannot make a priori assumptions about their location, orientation or connectivity.

3.3.3 High Defect Rates

An inherent aspect of any self-assembly process is defects. These fabrication defects can influence node functionality and connectivity. Some of the interconnect defects cause the above problems with connectivity. While some aspects of fabrication could reduce the likelihood of defects (e.g., purification steps or over design of DNA tags), there will always be a significant number of defects and any architecture using these technologies must tolerate these defects.

3.4 Architectural Challenges

The above discussion exposes several aspects of this fabrication technique for nano-scale circuits that must be addressed by any architecture and its corresponding implementations. In this section, we enumerate several challenges that must be overcome to build a computing system using the random network of nodes. A computing system built using this

random network must: a) tolerate node and interconnect defects, b) not rely on underlying network structure, c) balance node size limitations and functionality, d) compose more powerful computational blocks from simple nodes, e) minimize communication overheads, and f) achieve performance that is at least comparable to future CMOS based systems. Several research projects examined building computing systems with a subset of these goals, including self configuration [5,125], routing and resiliency in the face of defects [1,19,71,65] and the ability to compose complex computational units from simpler blocks [92], but we face added challenges because of the extremely limited computational capabilities available in nodes. We now elaborate on some of the challenges faced during system design.

Node Design. We must decide what additional functionality to place in each node. How does node design affect connectivity/communication within a node and with other nodes, and what primitives should be provided?

Utilizing Multiple Nodes. Since individual nodes do not contain sufficient computation and storage to perform much useful work in isolation, we must determine how to exploit multiple nodes. This must be achieved given the above limitations on coordination, communication, placement, orientation, and connectivity.

Routing with Limited Connectivity. Traditional routing techniques may not apply since there is limited space for the complexity of dynamic routing and there are insufficient guarantees on node placement and connectivity to use conventional static routing.

Developing an Execution Model. The execution model embodies the software visible aspects of the architecture and can be influenced by implementation constraints or instruction set requirements. For the envisioned fabrication technique, the execution model must overcome the severe implementation constraints outlined above while enabling a reasonable instruction set.

Developing an Instruction Set. Programmable systems require an interface that enables software to specify operations. Typically this is achieved by the instruction set architecture (ISA). The ISA may be influenced by the underlying capabilities of the technology. Given

our fabrication technique, the architect must design an appropriate ISA that supports the above execution model.

Providing a Memory System. Storage is a crucial component of most computing systems regardless of the execution model. The ability to store values for future use and to name and find particular values is a necessary aspect of most computing paradigms.

Interfacing to the Micro-scale. An important aspect of any nano-scale system is the interconnection to larger-scale components (e.g., micro-scale). This connection is necessary for at least providing an I/O interface for communication with the outside world. It may be possible for the architecture to exploit this interface in other ways.

In designing a high-performance system architecture, we must address these challenges within the constraints of the underlying technology.

3.5 Summary

In this chapter, we have presented the implications of DNA-based self-assembly of carbon nanotube electronics on circuit architecture, and used them to develop a design that could be used to build circuits. The proposed circuit architecture uses the regular structure of a DNA-lattice and introduces complexity through aperiodic patterning. The unit cell and the carbon nanotubes to be placed within each cell are given unique DNA tags in order to minimize positional defects. Self-assembly enables the construction of up to 10^{12} circuit blocks in parallel, but does not provide an easy mechanism to control the placement and orientation of those blocks. These blocks can then be connected using metallized DNA to form a large random network. Finally, we presented an analysis of the implications of the underlying circuit architecture and random node topology on architecture design and enumerated some of the challenges that must be faced in the design of a system. While we assume DNA-based self-assembly of nanoelectronic components as the underlying manufacturing technology, the challenges we describe are likely to arise with other technologies with high defect rates and a lower degree of control over the fabrication process. In the next

chapter, we describe our solution for tolerating defective nodes in the network, which is one of the primary challenges that we face.

4 Logical Structure and Defect Isolation in Random Networks of Nodes

In the previous chapter, we described the challenges that must be overcome in designing an architecture using random networks of self-assembled computing nodes. One of the critical challenges is to organize the nodes in some logical structure. This is especially important with self-assembly, since we have lesser control over each step of the manufacturing process than with CMOS. In this chapter, we describe and evaluate a mechanism for organizing nodes and isolating regions of defective nodes in a random network of self-assembled nodes. This approach does not require an external defect map, nor does it require redundancy of complex computational circuits, either of which would limit the scalability of the system. We use the reverse path forwarding (RPF) broadcast routing [28] algorithm, commonly used in wide-area networks, to map out defective nodes at startup. The algorithm guarantees two things: (a) the broadcast eventually terminates and (b) all functional nodes that have a path to the broadcast source will receive it. Thus, all functional and reachable nodes are organized in a broadcast tree, resulting in defect isolation. Simulations show that, for a fail-stop model of node failure, the broadcast connects all nodes that are reachable from the source. If the fraction of defective nodes is less than 10%, the broadcast reaches more than 97% of non-defective nodes. This chapter makes the following contributions:

1. We adapt RPF to impart logical structure to a random network of nodes, while isolating defective nodes and,

2. We evaluate the efficiency of the broadcast mechanism by computing the latency and "coverage" (the fraction of the non-defective nodes that the broadcast reaches) of the broadcast for different network sizes.

The rest of this chapter is organized as follows. We start with a brief description of the functionality required in each node and the node defect model (Section 4.1). We then describe (Section 4.2) and evaluate (Section 4.3) our defect isolation mechanism. Next, we discuss the weaknesses of the defect isolation mechanism and ways in which it could be improved (Section 4.4) and conclude the chapter with a short summary (Section 4.5).

4.1 Node Functionality and Defect Model

At minimum, each node must have the ability to store some configuration state and communicate with its neighbors. To build a useful computing system, a node should also have some compute logic. Each node is equipped with four transceivers that control communication with other nodes. Each transceiver controls data transfer on one link between the node and some neighbor. The assumed limitations of self-assembly restrict us to a single-wire link between two nodes, and all communication must occur on that wire. Each node has some storage space for global and local state and circuitry to control the flow of data. This includes control over the routing and actual decisions about performing operations in the ALU. As mentioned in Chapter 3, we assume the existence of anchors scattered across the random network of nodes. We assume a simple fail-stop defect model for the node - if a node is defective, it is completely isolated from its neighbors, i.e, it cannot perform any processing or communication. Fail-stop behavior can be achieved by augmenting node logic with simple test circuitry. We describe the design of a modular, fail-stop node in detail in Chapter 7. The defect tolerance mechanism does not require the extraction of a defect map from the random network, nor do we assume any knowledge of the location or nature of defects within the random network. Requiring the extraction of a defect map from the random network would not scale easily to networks with 10^9 or more nodes.

4.2 Reverse Path Forwarding

The RPF algorithm [28] forms the basis of our defect isolation mechanism. The key idea is to connect all operational nodes into a logical tree structure, while isolating all

defective nodes. Chapter 3 introduced our concept of an anchor which is an interface between the system and the micro-scale world. We use an anchor to insert a special broadcast packet into the network. Each node then forwards the packet using the RPF algorithm, which specifies that a node receiving this packet (called a gradient packet) broadcasts it on all its links, except the link that it received the packet on. Each gradient packet can be augmented with a simple test vector that tests basic functionality of the node. If the execution of the test vector results in invalid output, the node shuts down, otherwise it forwards the packet. Before forwarding the packet, the node stores the id of the link it received the packet on. Once a node processes a gradient packet, it does not forward any other gradient broadcast packets it receives. This ensures that the broadcast eventually terminates. Once all broadcast activity stops, we have effectively established a "gradient" [71,65] broadcast tree rooted at the via where we inserted the broadcast packet. Each node that received a gradient packet knows how to get a packet to this anchor.

We can use anchors located at four ends of the system to broadcast four "gradients" across the system. The idea is to set up a general routing framework with the ability to route in four directions (corresponding to each of the gradients). This routing framework can be used by a higher level architecture to route instructions and data across the system. To allow multiple gradient broadcasts in the network, we add a gradient ID (GID) field to each packet, such that each node runs the RPF algorithm once per gradient. By examining the GID in the packets, the nodes can decide whether to propagate the broadcast (in case of a GID not seen before), or to squash the broadcast (in case of a repeated GID).

The gradient broadcast mechanism achieves defect isolation. Since defective nodes cannot participate in the gradient forwarding process, no node ever receives a gradient packet from a defective node or link. This implies that we can never route data into a defective node, thus achieving defect isolation. The gradient broadcast mechanism is robust as the fraction of defective nodes increases. As long as there are large connected components in the random network, the gradient mechanism will connect all nodes within that region if the gradient source is also included in that region.

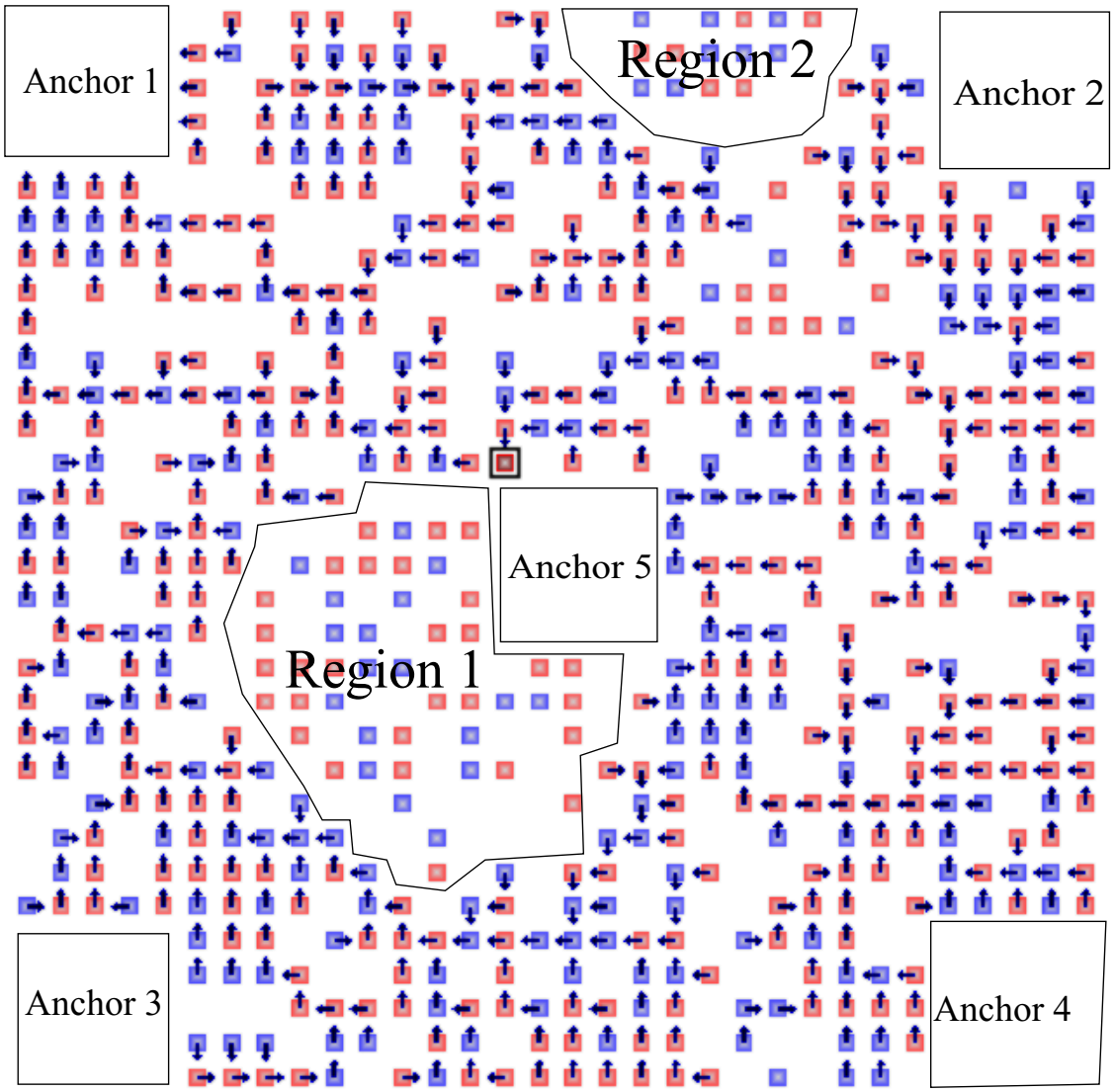


Figure 4-1. Gradient directions in a small network of nodes

We illustrate gradients in a network in Figure 4-1. The figure shows a small network, with each node having an arrow pointing in the direction that it received the gradient from (the gradient that originated from anchor 1). The absence of nodes (i.e., white spaces in place of nodes) corresponds to defects. The network in the figure has five anchors, one in each corner four anchors and one in the center (anchor 5). The figure illustrates how the gradient broadcast covers a large part of the network. It also shows how defects can cause regions of non-defective nodes to get isolated (region 1 and 2). In the next section, we eval-

uate the connectivity of our network of nodes equipped with the defect isolation mechanism.

4.3 Evaluation

We begin with a brief description of our experimental setup (Section 4.3.1). Through our evaluation, we seek to answer the following questions.

1. What is the coverage of the broadcast?

Ideally, the broadcast should reach all non-defective nodes. However, there could be cases where some nodes are cut-off due to the presence of surrounding defects. (Section 4.3.2)

2. What is the latency of a gradient broadcast as a function of network size?

The best case latency in a network with NxN nodes would be O(N). This would be obtained in the absence of all defects. In the worst case, the gradient needs to traverse the entire network linearly, giving a worst case latency of $O(N^2)$. (Section 4.3.3)

3. What is the effect of changing the location of the gradient insertion point in the network?

The location of the source of the gradients should make a difference in the coverage and latency of the broadcast mechanism. Conceptually, the source should be placed in a region that minimizes the chances of it being cut-off from a majority of the network. (Section 4.3.4)

4. What are the properties of the broadcast trees?

Ideally, we want to minimize the distance between the source and leaves of the tree. This will minimize the time spent in moving the data around the network. The minimum distance can be achieved if the broadcast follows the shortest path from the source to any other node. (Section 4.3.5)

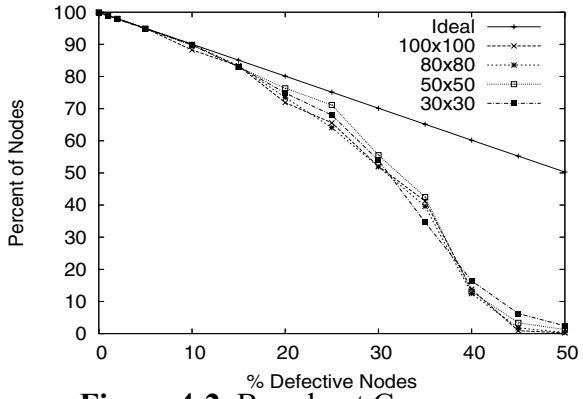


Figure 4-2. Broadcast Coverage

4.3.1 Experimental Setup

We use a custom event driven simulator to evaluate the defect isolation mechanism. We only consider networks of nodes that form a mesh since we are concerned with the coverage of the broadcast and not actual physical connectivity (if a node is not physically connected to the rest of the network, the broadcast cannot reach it). The simulator accepts various system parameters including the fraction of defective nodes as inputs. It uses a random number generator to mark certain nodes defective. Once a node has been marked defective, it ceases to be part of the network.

In our experiments, we vary the fraction of defective nodes from 0% to 50%. We vary network size from 30x30 nodes to 100x100 nodes. For each configuration, we present the average of 50 runs with random seeds used to generate distinct node topologies. All experiments use a single gradient source on the side of a square grid (except in Section 4.3.4).

4.3.2 Broadcast Coverage

The broadcast mechanism can get packets to all nodes that are “connected” to the gradient source. This means that any functional node that has a path to the gradient source, will receive a gradient packet. However, as the fraction of defective nodes increases, there is an increasing probability that regions of non-defective nodes will be cut-off from the gradient source because of a wall of defective nodes (see Figure 4-1). Figure 4-2 plots the percentage of non-defective nodes receiving the broadcast as we increase the fraction of defective

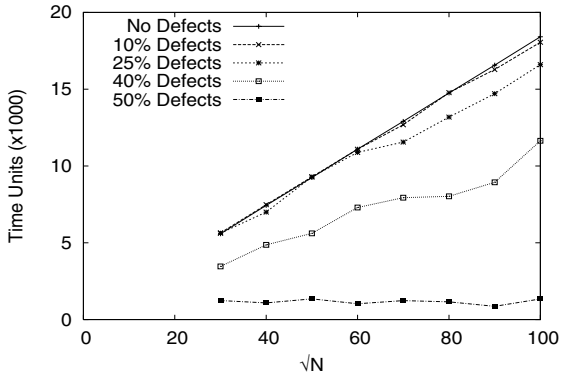


Figure 4-3. Broadcast latency as a function of $\sqrt{NetworkSize}$

nodes. Each line corresponds to a different network size. Data for limited (10) runs each for networks of 400x400, 500x500 and 800x800 nodes show trends similar to those observed for smaller networks.

As expected, we see that as the fraction of defective nodes increases, the percentage of nodes receiving the broadcast drops because of regions of non-defective nodes being cut-off. In addition, we see that for up to 20% defective nodes, the broadcast mechanism typically reaches 90% of the non-defective nodes in the network. This shows that for small fractions of defective nodes ($\leq 20\%$), the gradient broadcast is a good mechanism for isolating defective nodes and connecting non-defective nodes.

4.3.3 Broadcast Latency

One of the reasons we choose to use a self-configuring system is to eliminate the time overhead of obtaining an external defect map of the system. However, the gradient broadcast itself takes a non-zero time to complete. If a node can process and forward a gradient packet in unit time, we would expect that it would take at most 2N time units to finish broadcasting in an NxN system (corresponding to the manhattan distance between the nodes in opposite corners). In Figure 4-3 we plot the time taken to broadcast the gradients as a function of the square root of the number of nodes in the system, for different fractions of defective nodes. For a system with no defects, we see that the time taken to complete a gradient broadcast is a linear function of the square root of the number of nodes in the system (it is proportional to the maximum distance the broadcast packet has to cover, which

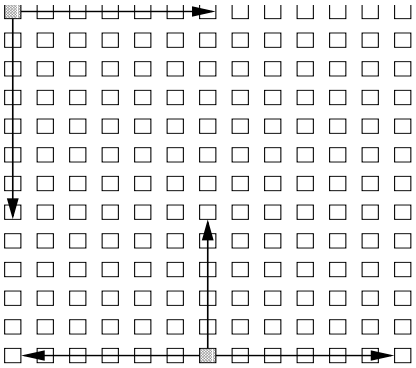


Figure 4-4. Two possible options for gradient sources

for a square network of NxN nodes is N). We see similar trends for larger networks (up to 800x800). As the fraction of defective nodes increases, we see that the time taken to complete the gradient broadcast decreases. This happens due to the fact that as defect probabilities increase, the probability of isolating a region of non-defective nodes increases. Thus, there are fewer "reachable" nodes in the system, reducing the time taken to complete the broadcast. Indeed, for a system with 50% defects, the time taken to complete the broadcast is almost independent of the number of nodes. This is because, as we see in Figure 4-2, the broadcast reaches very few nodes.

Our analysis shows that, in general, the latency of the broadcast is directly proportional to the maximum distance a broadcast packet has to cover in the network. This allows us to scale to very large systems and still have a broadcast latency low enough for practical use. In addition, we could divide large systems into logical regions by broadcasting multiple gradients.

4.3.4 Changing Broadcast Source

Intuitively, the placement of the gradient source vias in the random network will have an effect on how many non-defective nodes successfully receive a gradient. We run two configurations, one with gradients injected from the corner, and another configuration with the gradient injected from one of the sides of the network grid as shown schematically in Figure 4-4. The result of this analysis helps in choosing between the corners and the side midpoints as the source of the four planar gradients.

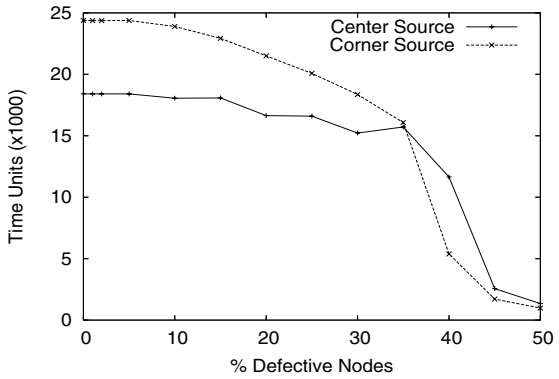


Figure 4-5. Broadcast latency as a function of the fraction of defective nodes and broadcast source

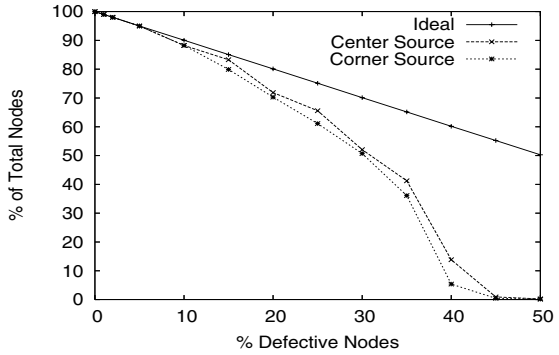


Figure 4-6. Varying Gradient Source:% Reachable Nodes

Figure 4-5 shows a graph where we compare the two schemes in terms of the time taken to complete a broadcast for a network with 10,000 nodes. From the figure we see that if we have less than 35% defective nodes, having a source in the corner takes longer to complete a gradient broadcast than having a source at the midpoint of a side of the grid. This is expected since a broadcast from a corner needs to travel a longer distance to get to all parts of the grid. However, once we have more than 35% defects, the probability of a corner source being cut off is higher than a source on the side being cut off. If a source is cut off from a large part of the network, it will "complete" the broadcast faster. In Figure 4-6 we compare the two schemes in terms of the number of non-defective nodes reached by a gradient. If the system has less than 10% defective nodes, the two schemes perform equally well, reaching most non-defective nodes. However, as we increase the fraction of defective nodes beyond 10%, the corner source reaches fewer nodes on average, since it has a higher

Fraction of Defective Nodes (%)	# of Nodes	Number of Children				Tree Height	
		0	1	2	3	Max	Avg
0	10000	2430	5192	2329	49	149	75
10	8822	1872	5167	1696	87	146	74
20	7186	1708	3926	1397	155	135	70
30	5203	1409	2553	1075	166	123	65
40	1382	394	641	301	46	93.6	50
50	23.22	6.22	11.8	4.6	0.6	9.2	4.8

Table 4-1. Properties of Broadcast Trees (100x100 network)

probability of being cut off due to defects. Our analysis shows that, as expected, the midpoint of a side of the grid is a better choice for the gradient source. A broadcast originating at this source is able to reach a larger fraction of nodes, with lower latency than one originating at a corner.

4.3.5 Broadcast Tree Properties

The gradient broadcast builds a spanning tree over the graph of all non-defective nodes that are reachable from the source. In most cases, there exist several spanning trees that can be built using the gradient source as a root. In the ideal case, we want a balanced 3-ary tree. However, given our grid-like topology, it is not possible to build a perfectly balanced 3-ary tree. An alternative to a balanced tree would be a tree that minimizes the number of hops between the source of the gradient and any other node in the network (i.e., minimizes the manhattan distance).

We analyze the broadcast trees generated by the gradient broadcast to determine their characteristics. Table 4-1 shows the results from this analysis on a network with 10,000 nodes. The source of the gradient is the midpoint of a side of the 100x100 square. The average manhattan distance from the source to any other point in the network is 74.5 hops, while the maximum distance is 149 hops. For the case with no defects, we see that the maximum and average height of the tree correspond exactly to the maximum and average manhattan distance between the gradient source and other nodes in the network. This implies that in

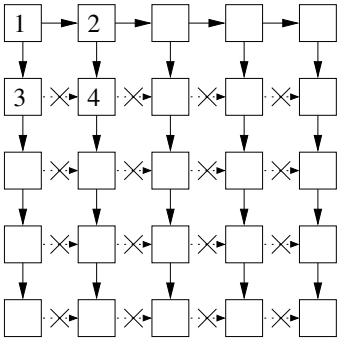


Figure 4-7. Gradient Broadcast: Cause of low branching factor

case of a defect free system, the broadcast finds a minimum-manhattan distance path between the gradient source and any other node. As we increase the fraction of defective nodes, the efficiency of a path from the gradient source to another node decreases. For example, in a network with 20% defects, there are 7,186 nodes in the broadcast tree with an average manhattan distance of 70 hops between the gradient source and other nodes. If we had a square grid with 7,186 nodes (~85x85), the average manhattan distance between the gradient source and other nodes would be 63 hops. This shows that the broadcast can no longer pick the ideal path because of defects, but picks the shortest path that avoids defects.

Another interesting property of the broadcast tree is the branching factor of nodes in the broadcast tree. It is preferable to have nodes with three children as that reduces the distance between the root and the leaves. If we have a large number of nodes with only one child, a failure in the link connecting this node to its child could potentially cut off a large section of nodes. From the table we see that as the fraction of defective nodes increases, the number of nodes with three children actually increases, which is desirable. At first, this seems counter-intuitive, but is the result of a peculiarity of the broadcast mechanism. As a broadcast packet spreads through the network, it often follows a “preferred” path. We illustrate this phenomenon in Figure 4-7. As the packet reaches node 1, it is sent to nodes 2 and 3. Now, nodes 2 and 3 both try to send the packet to node 4, however, only one of them (node 2) succeeds in this. All the crossed arrows show broadcasts that are not accepted. This “selection” of one direction over the other has a cascading effect and most nodes end up receiving a particular gradient from the same general direction. However, the gradient

broadcast always follows the fastest path, which is useful. In case some nodes are slower than others, the path of the gradient is more likely to go through the faster nodes (since they will broadcast faster). In addition, in the presence of defects, this phenomenon (linear paths) gets disrupted, creating more opportunities for the broadcast tree to branch out. This is also sensitive to the timing of the communication between nodes. If two nodes are not identical, one node will broadcast faster, reducing this problem in systems with low defects.

From our analysis of the properties of broadcast trees presented in this subsection, we conclude the following: a) the broadcast mechanism picks the shortest path consisting of non-defective nodes, but defects often cause the length of this path to deviate from the manhattan distance in a grid, b) defects in the network could improve the average out-degree of nodes in the broadcast tree.

4.4 Extending Gradient Broadcast

Our evaluation shows that gradient broadcast using the RPF algorithm should be an efficient way of achieving defect isolation in large scale systems of self-assembled nodes. Even with a large fraction of defective nodes, the gradient broadcast scheme can still be used on smaller scales using vias distributed across the network of nodes. By broadcasting a gradient per via, we can establish small “cells” of connected nodes.

The gradient broadcast mechanism presented here has no provision for handling transient faults or permanent faults that occur during system operation. One simple extension to the current system to handle runtime faults would be to maintain redundant path information at each node. Nodes often get the same broadcast packet on multiple links. The original scheme discards all but the first packet. If we use information from subsequent gradient packets to maintain multiple paths, the system could possibly handle transient faults. In addition, this redundant path information could also be used by higher level protocols for load-balanced routing. There is a trade-off to be made in maintaining multiple paths. Each additional path that needs to be stored requires extra storage at each node. There is also no guarantee that a node will actually receive multiple paths. In addition, path information will need to be periodically refreshed to keep it up to date. This will add to the overhead of gra-

dient broadcast. Permanent faults that occur during system operation can also be handled by running the configuration algorithm again.

The use of the RPF algorithm for defect isolation requires fail-stop nodes. This requires the ability to test the functionality of each node using built-in-self-test (BIST) or external test circuitry. A variation of the BIST would be the ability to inject a test vector packet into the network and have it propagate. Each node would execute the packet and get disabled if it fails the test. This extra functionality in the node must fit within manufacturing constraints. We describe one system specific implementation of fail-stop nodes in Chapter 7. So far, we have assumed a node to be fully operational or defective. In a real system, it is far more likely that only a part of a node is defective. However, except in the case of byzantine failures, partially defective nodes will not reduce the effectiveness of gradient broadcasts.

4.5 Conclusion

In this chapter, we presented one mechanism to impose logical structure on a network of self-assembled nodes while isolating regions of defective nodes. We adapt the reverse path forwarding broadcast routing algorithm to create a broadcast that connects all functional nodes that are reachable from the source of the broadcast. We have also presented an analysis of the connectivity of such a network of self-assembled nodes. This mechanism could potentially be extended to include multiple paths, thus providing robustness in the face of runtime faults. This extension involves a trade-off in terms of the storage required at each node, and the desired path redundancy. In the next chapter, we use multiple gradient broadcasts to organize a network of nodes in the design of a proof-of-concept architecture that addresses the challenges presented in Chapter 3.

5 Nano-Scale Active Network Architecture

In this chapter, we develop a proof-of-concept general purpose architecture built using a random network of self-assembled nodes. This architecture, called the Nano-scale Active Network Architecture or "NANA", supports the Von-Neumann programming model and a fully addressable memory system. The goal is to create a high-performance defect tolerant within the assumed constraints of the manufacturing technology. NANA uses the adapted RPF algorithm described in the previous chapter to isolate defective nodes and impart logical structure to the random network of heterogeneous nodes. It reduces routing resource requirements in each node by dividing the node network into distinct execution and memory networks. Since NANA represents our first attempt at designing a system architecture using DNA-based self-assembly of nanoelectronic devices, the guiding principle is to first design a working system before applying optimizations to improve performance.

NANA is similar to an active network [139] in that it uses "packets" that consist of instructions and data that are routed in the network in search of appropriate execution resources. When a node receives a packet with an instruction that can be executed at the node, it extracts data operands from the packet and performs the specified operation. NANA allows the execution of instructions to overlap by interleaving data operands and exploiting bit-level parallelism. The NANA memory system supports direct and indirect addressing, as well as the ability to fetch instructions and execute code from a specified address. The design and evaluation of the architecture provides insight into various design and performance trade-offs and highlights aspects of the design that prevent the system from achieving peak performance. The lessons learned from NANA are invaluable in the design of the data parallel architecture presented in Chapter 6. We make the following contributions in this chapter:

1. We develop NANA, a first attempt at designing a general purpose architecture using a random network of self-assembled heterogeneous nodes,

2. We develop an execution model to exploit bit-level parallelism for a stream of instructions, and

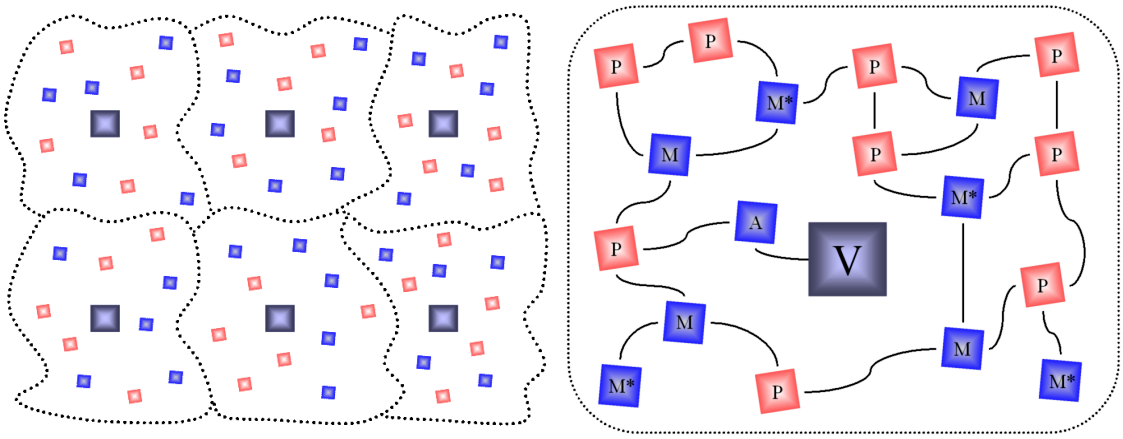
3. We use modeling and simulation to evaluate NANA and gain insights into its strengths and weaknesses, which can help guide future designs.

The rest of this chapter is organized as follows: Section 5.1 presents a brief overview of NANA. Section 5.2 describes the execution model and instruction set. Section 5.3 presents the configuration and operation of the memory system. The architectures of the different node types are described in Section 5.4. We describe program execution in Section 5.5, and evaluate the performance of NANA in Section 5.6. Section 5.7 contains a discussion of the strengths and weaknesses of the design, and Section 5.8 enumerates the lessons learned through the design and evaluation of NANA. Finally, Section 5.9 concludes the chapter with a summary.

5.1 NANA Overview

NANA is similar to an active network [139] in that execution packets that contain instructions and operands search through a logical network of processing and memory nodes for the functionality that they need at each step of execution. This architecture matches the underlying technology characteristics since it 1) supports a random interconnection of nodes, and 2) tolerates node and interconnect fabrication defects.

The system model is a random interconnection of various node types, in which all nodes contain circuitry for communication and each node has some specialized circuitry (e.g., processing, memory, etc.). A node communicates with a neighboring node via a single link that is asynchronous and bidirectional (time-multiplexed on a single physical wire). Groups of nodes are organized into *cells*. Each cell has an anchor that acts as its connection to the micro-scale. Inter-cell communication can be achieved through a micro-scale interconnection network. The memory nodes in each cell comprise a portion of the global memory



a) 2D mesh of cells b) Nodes within a cell

Figure 5-1. a) System Model. b) Processing nodes (P), memory nodes (M), memory port nodes (M*), anchor node (A), and via (V). This schematic is not to scale (w.r.t. nodes per cell).

space. Some fraction of nodes are configured as *memory ports* to provide an interface between execution packets and memory storage. Figure 5-1 illustrates our system model. To impose structure on the interconnection network and the memory system, there is a *configuration phase* that occurs before any execution. Reconfigurable architectures [27, 31, 53, 59] have demonstrated that this approach is important to achieve high performance in the context of highly focused (i.e., aggressive) or highly defective technologies, including nanotechnology.

While node functionality is heterogeneous, all nodes have some common responsibilities. Each node generates its own local clock (we choose a clock frequency of 10 GHz, which is likely to be a pessimistic value for carbon nanotube based devices [18,121]) and communicates asynchronously with its neighboring nodes using signaling techniques similar to push-style pipeline systems. High level communication between two devices over a single wire can be managed using four-phase single wire techniques [144]. Each node must also contain routing functionality for determining the outgoing link for an incoming packet (or the result of an operation). This circuitry maintains node state (e.g., currently processing a packet) and handles link contention. In the next four sections, we describe the execution model, the memory system, node architectures and program execution in more detail.

5.2 Execution Model and Instruction Set

This section provides a detailed description of the execution subsystem in NANA. We start with a description of the execution model (Section 5.2.1). We then describe the format of an execution packet (Section 5.2.2), and the instruction set used by NANA (Section 5.2.3). Finally, we discuss execution network specific configuration and routing (Section 5.2.4).

5.2.1 Execution Model

The execution model relies on an accumulator-based ISA. Conceptually, the accumulator is initialized and then a sequence of operations are performed on the corresponding series of operands. The accumulator-based ISA reduces the need for widespread *a priori* coordination and communication among many components (e.g., associative lookup in issue queues), since instructions are processed in order [76] and the only data dependence involves the accumulator. We support accumulator-based execution by forming an *execution packet* that contains the operations, the accumulator, and all operands in appropriate order (see Figure 5-2). Instructions are executed in the order specified in the packet, as they are routed through the network and find the requisite functional units (or memory ports). Logically, each functional unit performs its specified operation, removes the input operand(s) and forwards the new accumulator and the remaining operands to the subsequent functional units. Each subsequent functional unit performs a similar sequence until all operations in the packet are completed. Memory operations generate *memory packets* that are handled by the memory ports, as discussed in Section 5.3.2. Packet sequencing is achieved using a process called *chaining*, discussed in Section 5.5.

The system and execution model enable significant parallelism by allowing the instantiation of multiple execution packets within a cell and in multiple cells. While this parallelism is important to fully exploit the capabilities of the underlying technology, this thesis focuses on the operation of a single cell and sequentially instantiating execution packets.

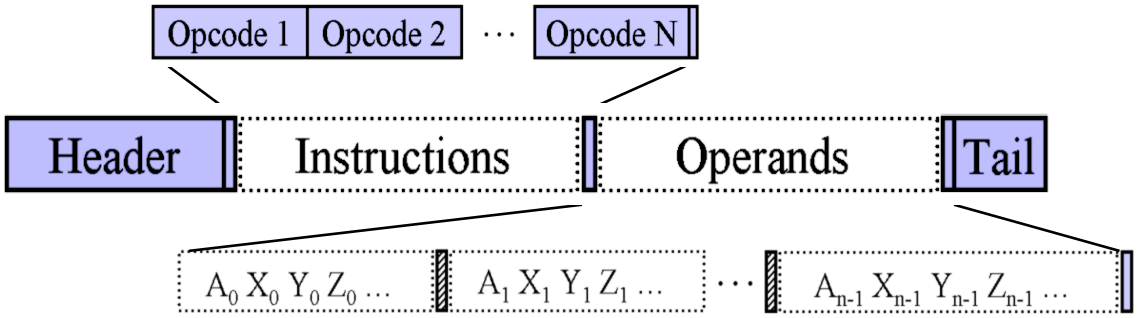


Figure 5-2. Execution Packet Format.

5.2.2 Execution Packet

The format of an execution packet is: header, instructions, operands, tail. The header is a fixed length field that includes packet type and routing information. The instructions field is a variable length list of opcodes in program order. The operands field is a variable length list of operand values. Specific bit patterns delineate field boundaries.

To accommodate the limited node size, we use a bit-serial implementation. The active network architecture and accumulator ISA are independent of this choice and provide an architecture that can scale with improvements in node capabilities (i.e., multi-bit operations). Figure 5-2 shows the execution packet format for our bit-serial implementation. The operands field is divided into bit-slices from least significant bit to most significant bit (from packet head to tail). Each bit slice starts with a bit from the accumulator and is followed by each bit (for the particular bit- slice) of the operands. Each logical bit is encoded as two physical bits (0=00, 1=01). A '11' in the operand field indicates a separator between operand bit-slices. The final field in the packet is the tail, which indicates the end of the execution packet, and also includes a flag bit used by conditional instructions.

5.2.3 Instruction Set

The instructions that this architecture supports is bit-serial in nature and requires little communication between bit slices. Many instructions are simple to implement with limited circuitry (e.g., ADD, SUB, OR, AND, XOR, NOR, NAND, compare, move) and require only small additions to a bit-serial full adder circuit. Each operation requires only a small

Instruction Type	Instructions
Arithmetic	ADD, INC, SUB, DEC, SHL, SHR
Comparison	COMPEQ, COMPGT, COMPLT, SETEQ, SETGT, SETLT, SETZ
Operand Stream Control	LDCONST0, LDCONST1, CPACC, MOV, DELOP, OPFLUSH, SWAP
Logical	AND, NAND, NOR, NOT, OR, XOR, XNOR, NOP
Load	LD [Mem], LDI [Mem]
Store	ST [Mem], STI [Mem]
Conditional Store	CST [Mem], CST_RST [Mem], CRST [Mem], CSTI [Mem], CSTI_RST [Mem], CRSTI [Mem]
Unconditional Control Transfer	JMP [Mem], CALL [Mem],JMPI [Mem],CALLI [Mem]
Conditional Control Transfer	CALLNZ [Mem], CALLZ [Mem], CALLNZI [Mem], CALLZI [Mem]

Table 5-1. NANA Instruction Set

amount of information (e.g., carry-out bit) to be communicated to subsequent bit slices. This simplifies the implementation details of the circuits so that they will fit within the node size limits of the underlying technology. Although each instruction is bit-serial, the bit interleaving enables parallel execution of consecutive operations in a pipelined manner. Instructions supported by NANA can be divided into nine categories and are listed in Table 5-1 (Appendix A describes the NANA instruction set in detail).

The serial nature of this architecture and the limited node complexity of the technology makes certain operations difficult. Table 5-2 lists several instructions specially designed to help overcome these difficulties. For example, right shifts (moving bits from the tail toward the head) are difficult because they require bits to be forwarded ahead of other bits unless entire operands are stored at the functional node. Since we assume that both operand storage and ALU-type functionality in a single node requires too much area for our limited node size, we exploit the stack-like nature of the operand stream to support right shifts. When a right shift is executed, it also places the result at the end of the operand stream. Thus, to execute a right shift, we buffer the field separator between bit slices and emit the next observed data bit before re-inserting the field separator into the packet bit stream.

Instruction	Operation
MOV	Move accumulator to end of operand stream
SWAP	Swap first and second operand
SHR	Shift accumulator right by 1 bit, move accumulator to end of operand stream
DELOP/OPFLUSH	Remove one/all operands from operand stream
CPACC	Create copy of accumulator at end of operand stream
SET (EQ/GT/LT/Z)	Set flag bit in tail if condition satisfied, consume accumulator
COMP (EQ/GT/LT)	Set flag bit in tail if condition satisfied, consume first two operands
LDI [Mem]/ STI [Mem]	Load/store indirect through constant address [Mem]
CST [Mem]/CSTI [Mem]	Conditional store direct (CST) or indirect (CSTI) to [Mem] (status bit in tail must be set)
CST_RST [Mem]	Conditional store to [Mem], reset status bit after performing store
JMP [Mem]/JMPI [Mem]	Fetch instructions into existing packet from direct (JMP) or indirect (JMPI) address [Mem]
CALL [Mem]/CALLI [Mem]	Create new packet using instructions from direct (CALL) or indirect (CALLI) address [Mem]
CALLNZ [Mem]/CALLNZI [Mem]	Fetch instructions into new packet if status bit is set (not zero) (direct/indirect)
CALLZ [Mem]/CALLZI [Mem]	Fetch instructions into new packet if status bit is not set (zero) (direct/indirect)

Table 5-2. Definitions of a selected subset of instructions

The bit-slice packet encoding also complicates memory operations. For example, a load (or a store) requires all of its address bits to generate a request. If the address is in the operand stream, then it is impossible for the load to interleave the resulting data in the same operand stream since all the low order bits are ahead in the packet flow before the entire address is obtained. Therefore a packet cannot both calculate an address and use it in the same packet. To address these limitations, we provide three specific types of memory addressing: immediate, constant address and indirect address. Constant addressing requires the address to appear in the instruction field of the packet. Indirect addressing supports indi-

Address	Instruction	NextPC	Address	Instruction	NextPC
0x10	LD y	0x14	0x40	LD x	0x44
0x14	LD a	0x18	0x44	LDI z	0x48
0x18	ADD	0x1A	0x48	ADD	0x4A
0x1A	ST z	0x1E	0x4A	ST x	0x0
0x1E	CALL (0x40)	0x0			

Table 5-3. Memory layout for two packets that compute x=x+ *(y+a)

rection through a memory location that is specified as a constant in the instruction field of the packet. We also provide special load instructions (JMP & CALL) for instruction sequencing (discussed in Section 5.5). Conditional execution is supported through status bits (e.g., condition codes) in the packet tail. Currently we support conditional store and CALL instructions that must wait to execute until the packet tail arrives so that they can examine the appropriate status bit.

Programming NANA is similar to programming other accumulator based ISAs [20,76,77,82], however, care must be taken to account for system capabilities and constraints. For example, the 'shift right' instruction (SHR) is constrained by node resources to shift the accumulator and move it to the end of the operand stream, while the 'shift left' instruction (SHL) operates as expected (i.e., it shifts the accumulator left by one bit). Another constraint arises from the structure of the memory system - all loads must precede stores in a packet. Consider a simple code fragment (x=x+ *(y+a)) that computes a memory address (y+a) and then adds the contents of that location to another variable stored in memory. Due to the load-store ordering constraint, instructions must be divided into two packets. Table 5-3 shows the two packets needed to implement the code segment, and how their fragments are arranged in memory. The first packet, starting at address 0x10, performs an address calculation (y+a) and stores the result in a third location, z. The last instruction, at address 0x20, chains this packet to the next packet, which starts at address 0x40. The second packet performs the addition of x with the value stored at the memory location pointed to by z, and stores the result into x (i.e., x=x+*z). This packet executes by first loading the value of x, then performing an indirect load on z (instruction at 0x44). Next, it exe-

cutes the add and stores the result into x. This example illustrates some constraints that must be faced in programming NANA. We expect that, as the underlying technology matures, a richer ISA with more complex instructions will become possible, including efficient variable bit shifts, bit-serial multiplication and division. Until then, we compose these more sophisticated operations in software using simpler primitives. Next, we look at how the network of nodes is configured to support instruction routing and execution.

5.2.4 Configuration and Routing

NANA must enable execution packets to find what they need without deadlocking or livelocking, despite high defect rates and traveling through a random network of nodes. To avoid request/response deadlock (i.e., fetch deadlock), the minimum requirement is three logical networks: one for execution packets, one for memory request packets and one for memory response packets. Each of these logical networks is irregular and must provide deadlock- and livelock-free routing. While we could implement these three networks using three virtual channels [29] per unidirectional link, this increases the amount of buffering required on a single node. We reduce the requirement to two virtual channels per unidirectional link by creating distinct physical networks for execution and memory; we explain how this is implemented in Section 5.3.1. We also use wormhole routing [98] since it requires the least buffering on each node (1 bit per channel).

Virtual networks avoid fetch deadlock, yet each network must still provide deadlock- and livelock-free routing. Given our irregular networks, we exploit the spanning tree created by our configuration algorithm and then employ a variant of up*/down* routing [125], a degenerate case of turn-model routing [52], and back pressure flow control. The challenge is implementing these techniques with limited node functionality.

To meet this challenge, each node must support two forms of communication: 1) broadcast and 2) routing along a *gradient* (see Chapter 4). Packet headers include information on the type of communication to use. Broadcast requires minimal state per node and is used during configuration only. Gradients reduce per-node resources while still enabling deadlock- and livelock-free routing.

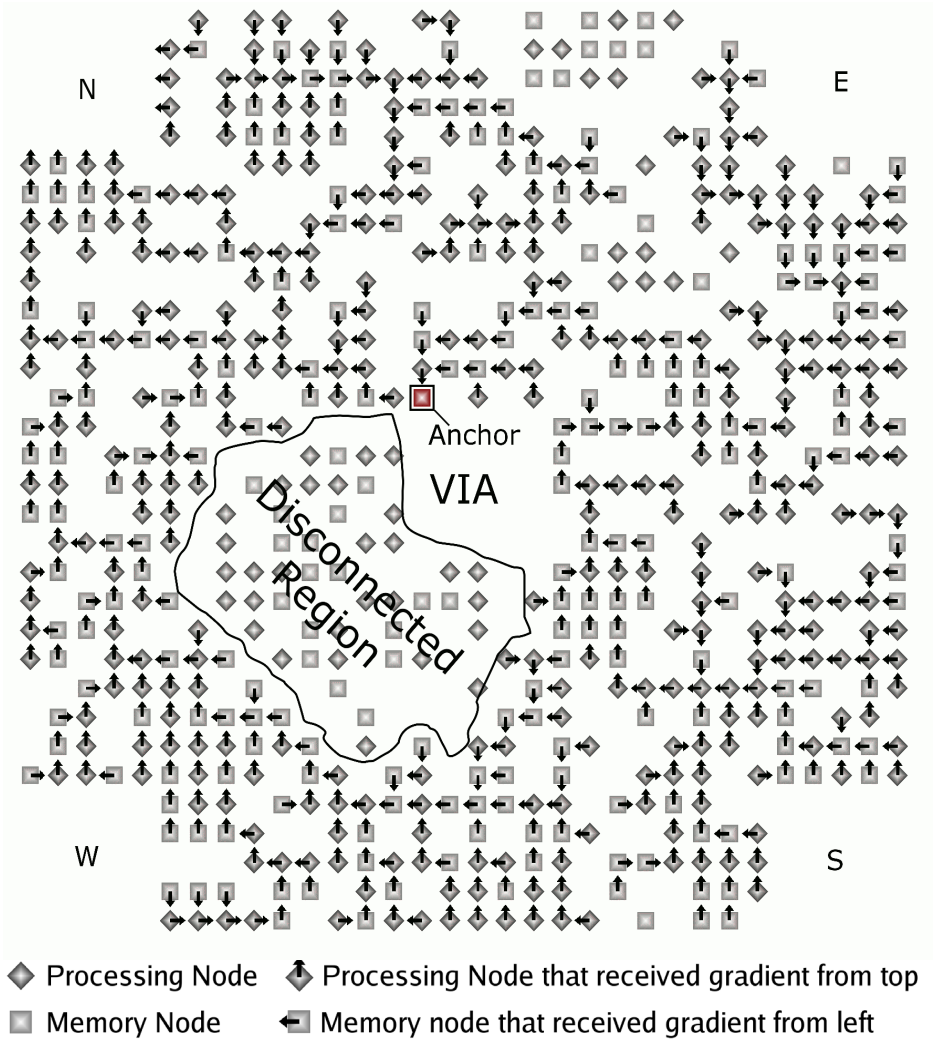


Figure 5-3. A 32x32 grid of memory and processing nodes with one established gradient (North)

We use five gradients: one for each planar direction (north, south, east, and west) and an additional gradient that establishes cell boundaries and the direction toward the via in each cell (called the *cell gradient*). The planar gradients are established by starting the broadcast at the north, south, east, and west edges (or corners) of the system, respectively. Figure 5-3 illustrates a gradient established from the upper left corner (north) in a 32x32 grid with a 30% defect rate. Defective nodes, not drawn in this figure, can cause islands of disconnected nodes such as the region near the via. Due to defects, some vias may not have a path to any of the four planar gradient destinations. This can be detected by monitoring

the via at the micro-scale during the broadcast of each of the planar gradients. If the via fails to receive any of the gradient broadcast packets, it should be marked as defective and not participate in cell configuration.

Cell configuration is initiated at each via in parallel by broadcasting a cell ownership packet that includes a cell identifier. The cell gradient broadcast stops when its wave front collides with the wave front from an adjacent via. Nodes that receive two (or more) distinct cell identifiers mark themselves as boundary nodes, creating a boundary layer between cells. Next, we describe how execution packets can be routed on the execution network.

5.2.4.1 Routing Execution Packets

The spanning tree structure imposed by gradients provides the framework for packet routing. Execution packets and memory packets never share physical links and thus cannot block each other. Up*/down* routing on the spanning trees prevents routing deadlock and livelock. However, execution packets must be able to find the necessary resources for execution, and memory packets must successfully find the appropriate memory location, which responds if necessary. To avoid deadlocking execution packets, we simply follow a single gradient (up* on one spanning tree) on one virtual channel until we reach a cell boundary, then reflect the packet back into the cell on the opposite planar gradient but on the other virtual channel. Reflection only occurs if there are remaining instructions in the packet, otherwise a special packet is sent to the anchor node to indicate completion. We note that the header can run ahead of the operand stream allocating nodes for instructions (due to execution delay in a node). This approach can indefinitely bounce a packet between cell edges. The only constraint is that packet length be less than the total number of nodes in the round trip traversal. Since execution packets only traverse in the up* direction of the spanning tree, each node must only store a single pointer per spanning tree (the gradient direction). An execution packet's ability to find the appropriate resources depends on several fabrication variables, including defect rates and the distribution of node types. An exploration of this space is beyond the scope of this thesis.

5.2.4.2 Improving Node Utilization

While the four planar gradients allow us to route execution packets in the cell, we find that only a small fraction of all execution resources in a cell are used. This is because the route taken by the execution packet depends on its insertion point in the cell, and the gradient that is being used to route. The execution network within the cell does not have a well defined structure if we use planar gradients for routing. To improve the number of nodes reachable by execution packets, we need to modify the structure of the execution network within a cell.

We add a second via and anchor node ("execution anchor") to the cell. This via is used only by the execution network. Once the memory system has been created, we broadcast an "execution" gradient in the cell. This gradient reaches nodes that have not been included in the memory network and any ports on the memory network. This allows us to create a single execution network by performing a depth-first traversal on the spanning tree created during the broadcast of the execution gradient. All execution packets follow this depth-first order ensuring high execution node utilization. The memory and execution networks now include most of the nodes in the cell, potentially allowing the use of about 97% of the cell (some nodes can become isolated during the creation of the memory network). However, as we discuss in Section 5.6, there are other aspects of NANA that limit node utilization. Next, we describe how we can exploit the packet routing infrastructure to configure a fully addressable memory system in each cell.

5.3 Memory System

Each cell represents a local namespace for memory and includes both data and instructions. The memory system must be able to (a) allocate (number or name) its locations, (b) provide an interface to execution packets, and (c) route memory packets (both requests to specified locations and responses back to requestors).

5.3.1 Memory Allocation

The memory network is a spanning tree rooted at an anchor. To configure memory, allocation packets are injected through the anchor node, initially routed on virtual channel zero using any planar gradient. When an unallocated memory node receives an allocation packet, it records the address in the packet, marks itself as allocated, and sinks the packet. The second allocation packet received by this node is forwarded along the specified gradient, forming a branch in the network. For the third allocation packet, the node modifies the header to route the packet on virtual channel one along a planar gradient that creates a second branch in the network. Three fourths of the subsequent allocation packets arriving on virtual channel zero are forwarded along the first branch while the remaining packets use the other branch and switch to virtual channel one. Packets on virtual channel one are never modified. Cycles in the memory network are prevented by having an allocated node only accept configuration packets on the same physical link as its original allocation packets[1].

Memory ports are allocated after memory nodes and must have three good links (excluding the link used by the incoming packet) with three distinct planar gradients. Ports, which are unnamed, never change an allocation packet gradient, thus keeping the remaining two links free for the execution network. Non-memory nodes between memory nodes route allocation packets according to the specified gradient and reserve the corresponding links only for memory operations. A second planar gradient configuration creates new spanning trees that do not include any of the memory network links, thus creating two separate networks. Figure 5-4 illustrates the allocation of 64 memory locations and 64 ports on a 32 x 32 grid with a 3% defect rate. For illustration only, we include only the West planar gradient on the execution network and use a low defect rate on a small grid. Clearly, in this memory system the anchor could be a bottleneck.

Now that the memory network has been created, the execution network can be created. Depending on the routing scheme being used, we initiate a broadcast of the planar gradients

1. This is implemented by signaling the appropriate neighbors to not forward along the specified physical link.

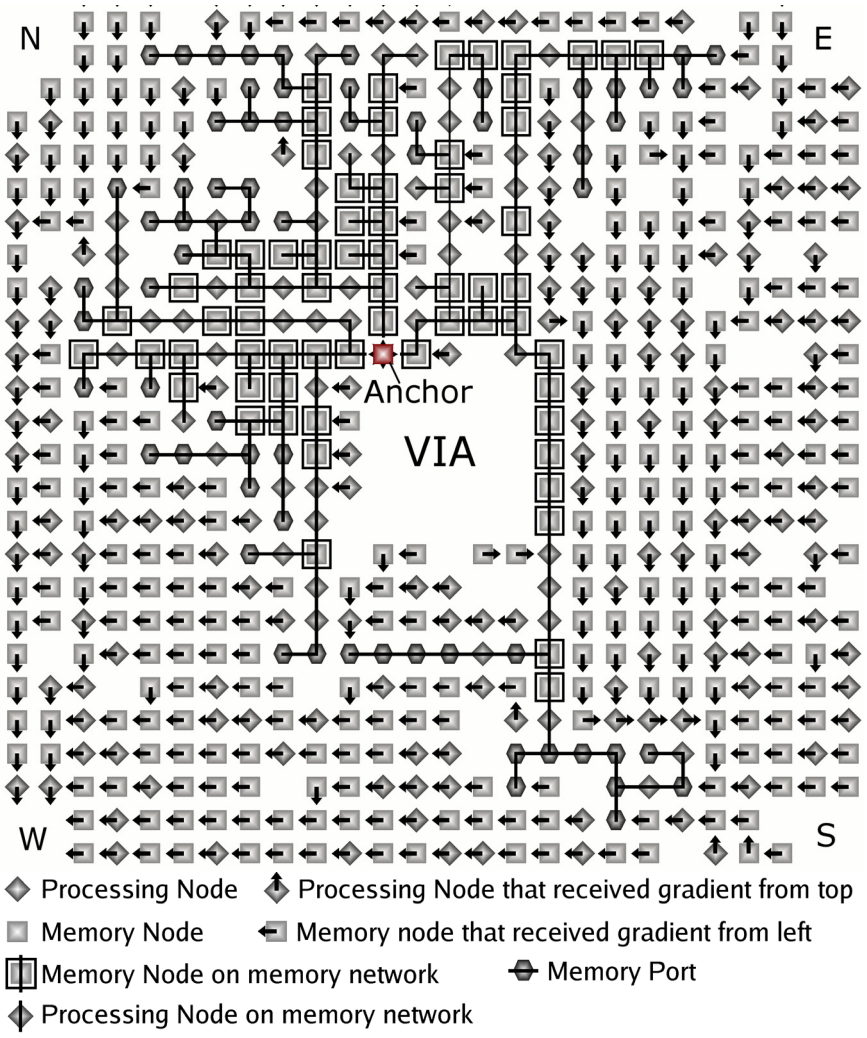


Figure 5-4. Memory Network. 32x32 grid with a fully configured memory network, showing one gradient (west)

(scheme 1), or the broadcast from the second anchor in the cell (scheme 2). In either case, nodes on the memory network do not propagate the broadcast, except for memory ports. This ensures that the two networks do not share physical links. We discuss the execution-memory interface formed by memory ports in the next subsection.

5.3.2 Interfacing Execution and Memory

The interface between the execution network and the memory network is controlled by memory ports that assume responsibility for handling all memory operations, including the JMP/CALL instructions for packet instantiation (see Section 5.5). When an execution

packet needs to perform a memory operation, it must encounter a memory port to execute the operation. A memory port servicing an execution packet stalls the execution packet, but at different points for loads and stores. Since load addresses are contained in the opcode field, the load can immediately issue and only stall the packet when the first bit of the operand stream arrives. Thus, the header continues searching for resources for subsequent instructions. When the memory port that initiated the load receives the response, it interleaves the memory contents into the execution packet's operand stream, enabling the operand stream to continue forward. A store must see the entire operand stream to extract the data, and after issuing stalls the packet until the store is acknowledged. This acknowledgment ensures intra-packet memory disambiguation. Memory ports also support indirect memory operations which require back-to-back memory operations: one to load the address and the other to access the contents at that address. We implement this by first issuing a constant load, to obtain the address, then using the result to generate another address for the load or store.

5.3.3 Routing Memory Packets

Memory packets are routed on either a request or response virtual network (two virtual channels per unidirectional link) that each obey up*/down* routing. Routing in the up direction follows the cell gradient up the spanning tree to the anchor node where the packet is broadcast in the down direction. Broadcasting is necessary since the destination memory node or port could be anywhere in the memory network. Loads require two full traversals of the memory network. However, since the anchor node is a serialization point for memory operations, it can acknowledge a store by broadcasting down the response network. Memory operations for addresses outside the originating cell are passed by the anchor onto the microscale network.

Once the memory and execution networks have been created, program execution can begin. Before we describe program execution, we present a detailed description of the different node types in NANA.

5.4 Node Architecture

There are three different node types in NANA: a) processing/ALU, b) memory and c) memory port. While all node types share some common functionality, the operational logic within each type differs. In this section, we describe the internal logic within each node type, starting with a description of the functionality common to all node types.

5.4.1 Common Functionality

As mentioned before, all three node types share some common functionality. This is primarily communication and routing logic. Each node type has four transceivers that support two virtual channels and asynchronous communication on single-bit links. The virtual channels are supported by single-bit buffers at the input and output for each transceiver. Data arriving at a transceiver from outside a node can be routed to one of the other three transceivers or to the internal logic of the node. Each virtual channel is handled independently, and a packet can switch virtual channels only in the internal logic of the node. Each transceiver has a dedicated point-to-point link per virtual channel to all possible destinations (3 other transceivers and the internal logic). In addition to the communication logic, the nodes also share test and configuration logic, which includes gradient configuration, as well as logic for configuring the execution and memory networks.

The node defect model assumes that the communication logic is either fully functional or not operational at all (the Byzantine defect model, in which defective nodes can produce arbitrary behavior, has been considered in the internet literature, but tolerating such defects requires a great deal of complexity at each node [21]). We can tolerate shorts in the node interconnect, and we call such defects *broadcast defects* because they represent the unintentional broadcast (to more than one link) of packet bits. Such defects are difficult to avoid during fabrication without control over self-assembly and require an arbitration scheme to control access to the link. The asynchronous link controllers in each node can be designed to assert a *link-good* signal after a random interval of time after power up. The randomness can be introduced during the self-assembling process [35]. Every node monitors its links

for the link-good signal and marks any link that has received more than one signal as defective. When the node's internal random interval has elapsed, if the link is not already marked defective it asserts its own link-good signal on all links. This arbitration scheme identifies both shorts and opens on links between nodes. The nodes connected to the via essentially share a single link (the via) that appears as a broadcast defect. The result of this arbitration scheme is for a single node to remain actively connected to the via, thus acting as the cell anchor.

5.4.2 Processing/ALU Node

Each processing node is equipped with a bit-serial ALU that can perform several simple arithmetic and logic operations. Each processing node can extract the first instruction in the execution packet and decode it. If the instruction cannot be executed by the processing node (for example, a memory operation), the instruction is treated as a NOP. If the instruction can be executed by the node, it pulls it out of the execution packet, and then bypasses all remaining instructions. Once the operand stream reaches the node, it extracts the first two operands per bit-slice and performs the decoded operation on that pair of operands. The result of the operation is then re-inserted in place of the two operands. Exceptions to this are shift and move instructions, which are explained in detail in Appendix A.

The processing node also carries single-bit state through the execution of an instruction (for example, a carry or borrow). This state is used by certain instructions to set the condition flag in the tail. Processing nodes that lie on the memory network do not need their internal logic block. The only requirement of these nodes is to route packets along the memory tree, which can be handled by the transceivers.

5.4.3 Memory Node

Memory nodes do not participate in instruction execution, and treat all instructions as NOPs. On the memory network, memory nodes have an associated address, and are used to store a 16-bit word. Memory nodes must be able to decode memory requests (load/store), decode the address in a memory request, compare it with their internal address, and take

action based on a match/mismatch. If the address from the incoming packet matches the address of the memory node, the memory node must take the appropriate action.

In case of a load request, the memory node creates a response packet consisting of its data contents and inserts it into the memory response network. In case of a store request, the memory node must replace its contents with data from the packet. In case of a mismatch, the memory node simply passes the request down the request network. The memory node is also required to forward requests moving up along the memory response network. This is handled by the routing logic.

5.4.4 Memory Port Node

A memory port node, or port forms the interface between execution and memory networks. Ports are responsible for executing all memory operations by creating requests for the memory network based on load and store instructions and inserting memory responses into execution packets. Ports are "active" only if they lie on both networks. If a port is only on the execution network, or only on the memory network, it acts as a null node and treats all instructions as NOPs.

When a port executes one of the special load instructions to fetch a packet, it must examine each word returned to determine the action to be taken. The port inserts the first half of the response into the execution network and this forms part of the execution packet. The second half of the response is the address of the next segment to be fetched. If this value is zero, the load terminates, if not, the port must generate a new load request with the new address.

The port must have enough storage space to hold the address of a load or store instruction that is executing. When it gets a load response or a store acknowledgement, it must compare the address received in the packet with the address from the executing instruction. In addition to being able to store the address, the store must be able to stall execution packets and move data between the two logical networks.

In the next section, we describe how programs can be loaded from memory, executed, and used to initiate execution of other programs.

5.5 Executing Programs

Execution packets (from header through tail) can be stored in memory by fragmenting them across memory locations. Each fragment contains a portion of the execution packet and the memory address of the next sequential fragment (zero indicates termination). The fragments are written into memory by using the micro-scale interface to inject store requests into the memory network. Packets are reassembled and instantiated on the execution network at a memory port using special sequencing instructions. Initial execution starts by using the micro-scale interface to inject one of these instructions on the memory response network for the named memory port. Once the JMP/CALL instruction is inserted, it executes at a port, and assembles the execution packet. Execution can begin as soon as the first fragment of the packet has been retrieved from memory. This allows us to overlap execution of the packet with packet reassembly.

The load instruction that reassembles execution packets can also be embedded in an assembled execution packet. This allows us to create an execution packet during the execution of another packet. This process of sequencing instructions or packets under software control by including a special load as the last operation is called “chaining”. We implement two forms of the sequencing instruction: 1) CALL creates an entirely new packet, but stalls until all previous instructions are complete (i.e., it sees the packet tail), and 2) JMP injects new operations into the existing packet by stalling the operand stream, thus enabling accumulator forwarding. Conditional CALL is easily supported since the instruction waits for the packet tail. Execution of one packet can overlap with its dependent packet’s search for functional and memory nodes. The ability to chain allows us to support different types of loops and conditional execution. It also enables the execution of one packet while the micro-scale circuitry stores another packet into the memory system.

Figure 5-5 illustrates a snapshot of the execution of the packet from Table 5-3 that starts at address 0x10 taken before the first load operation completes. In the figure we highlight the relevant nodes that correspond to each operation in the packet. On a 32x32 cell, the

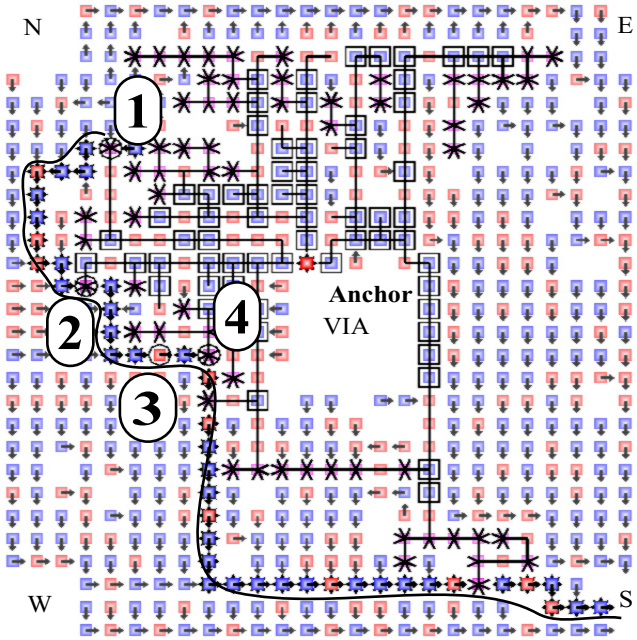


Figure 5-5. The path of a simple code fragment. 1: load Y, 2: load A, 3: Add, 4: store Z. Note that the Add instruction after second load hops through 5 nodes before finding an execution unit. The solid black line traces the path of the packet.

system takes about 70,000 simulated "time units" to configure gradients and memory and only around 10,000 are spent in gradient broadcast. We provide more information about the time units used to measure running time in the next section, where we evaluate NANA using simple programs to determine its peak performance, and to determine how much of this performance can be achieved in practice.

5.6 Evaluation

In this section, we evaluate the performance of NANA using simulation and modeling. We begin with a brief description of our evaluation framework in Section 5.6.1. In Section 5.6.2, we measure NANA's peak arithmetic performance, and explore the relationship between performance, instruction execution latency and execution packet length in Section 5.6.3. Next, we evaluate NANA's performance on two simple programs: 1) fibonacci in Section 5.6.4, and 2) string search in Section 5.6.5. This is followed by an analysis of memory system bottlenecks using a simple analytical model in Section 5.6.6. We

conclude this section with a discussion of two system optimizations and their effect on performance in Section 5.6.7.

5.6.1 Evaluation Framework

We evaluate NANA using a detailed custom event driven simulator written in C++. The simulator models the system at all stages, including gradient broadcast, memory configuration, execution configuration and run-time and also activity within all node types down to bit exchanges between components. It allows the user to vary a number of system parameters including the size of the network, node type distribution, event latencies, defect rate, and number of cells being simulated. Each cell holds a different part of the global address space and can execute different programs that are provided as input to the simulator. All events in the simulator are assumed to be a multiple of the clock cycle time (0.1 ns). The simulator accepts user-defined network topologies, or it can generate regular grid based topologies. For simplicity, we use a grid-based topology with a single 1024 node cell and a 3% node defect rate in our evaluation. As long as the defect rate is low (about 15% or lower), the network topology has little effect on performance.

5.6.2 Peak Performance

To measure the theoretical peak performance of NANA, we first compute the maximum performance that could be achieved by a single node. To execute an arithmetic operation like an ADD, a node must first receive the instruction (8 bits), then await the operands. If we assume 32-bit operands, the node must receive and send two 32-bit operands, and the bit-slice separators, which is a total of 128 bits. Thus, the execution time is dominated by the time to receive the data. To execute a 32-bit ADD, the node must receive 144 bits, and if we assume a latency of 1 ns to receive a bit (0.4 ns at the transceiver, 0.6 ns through rest of node), this translates to a node being able to execute 6.94 million ADD instructions per second. Now, if we have 10^9 nodes, and each node executes instructions all the time, we get a theoretical peak of 6.94×10^{15} instructions per second.

It is unlikely that all nodes will execute an instruction every cycle, so next we examine the effective latency per ADD instruction if we have an execution packet with a variable number of instructions. An execution packet with multiple instructions allows us to overlap the execution over multiple nodes, thus amortizing some of the communication overhead. The effective execution latency reduces from about 150 ns to around 100 ns if only execution/processing nodes are encountered during execution. If we take on average 10 ns to find a processing node, it would take about 110 ns per 32-bit ADD. It is useful to compare the execution time per instruction with the execution time of a similar instruction on an existing architecture. A 32-bit add takes half a cycle to execute on a Pentium 4, running at 3.2GHz [60] which is far faster than NANA. However, this comparison is biased in favor of the Pentium 4 since the execution time on the Pentium 4 does not include time to send data to registers, while the execution time on NANA includes creating the new accumulator. Despite this, NANA makes up for the lost speed by executing a large number of instructions on multiple cells, at the same time. If we make an equal-area comparison, we can fit ~3600 (60x60) cells in the 144 mm^2 area of a Pentium 4 (Northwood core). We could then potentially achieve a throughput of about 32 ADDs/ns. This gives NANA better throughput than the Pentium 4, which can issue at the most six instructions per cycle (~18 ADD/ns). These numbers would correspond to the peak performance of both machines.

5.6.3 Estimating Instruction Execution Time

To get an estimate of instruction execution time, including the effect of the bit level parallelism, we simulate arithmetic instructions in detail. We do not model memory instructions here, since they are covered by the queuing network and simulations in Section 5.6.6. The aim is to get a quick estimate of how long arithmetic instructions take to execute, and see the effect of overlapping instruction execution at the bit level (to exploit parallelism). As we add instructions to a packet, its length increases but the average time to complete an instruction in the packet decreases. Each packet has extra bits that provide control information and help in fault tolerance. The cost of this overhead is amortized over the number of instructions in the packet.

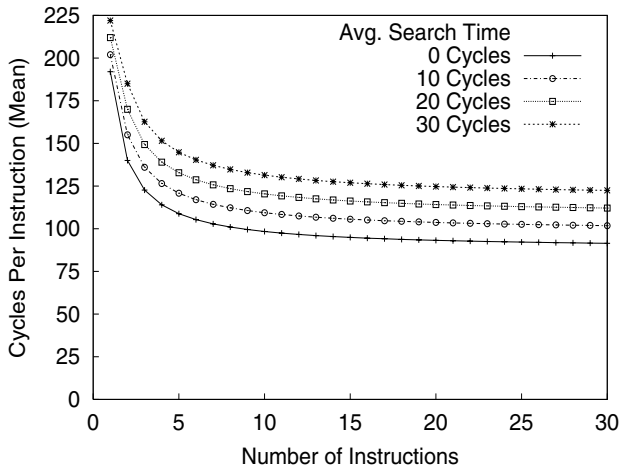


Figure 5-6. Avg. Instruction Latency vs. # Instructions (varying search time)

We use two steps to calculate the time an instruction takes to execute. First, we calculate the decode time for each instruction. On the execution network, each instruction can be decoded as soon as the entire opcode has been received by an appropriate processing element. Next, we update the execution time of each instruction with the time taken to operate on all operand bits, taking into account the time taken by prior instructions to execute and send the result bits over the execution network. We use this simple model of instruction execution to calculate the time it would take to execute multiple consecutive add instructions. In Figure 5-6, we plot the average time taken to execute an instruction as a function of the number of instructions executed. The graph shows the effects of the bit-level parallelism. In case of a completely sequential execution, the cycles per instruction would remain constant. Here, the effect of the overlapped execution is the reduction of the per-instruction execution time which is observed in packets with ten instructions or less. Beyond ten instructions, each new instruction adds enough data overhead to counter the effect of overhead amortization and execution time per instruction remains almost constant. Next, we evaluate NANA's performance on a simple program that includes both arithmetic and memory operations.

Address	Op	Next	Address	Op	Next
0x10	LD (0x02)	0x14	0x26	CPACC	0x28
0x14	LD (0x04)	0x18	0x28	ADD	0x2A
0x18	LD (0x06)	0x1A	0x2A	CST (0x08)	0x2E
0x1A	DEC	0x1C	0x2E	ST (0x06)	0x32
0x1C	CMPZ	0x20	0x32	ST (0x04)	0x36
0x20	ST (0x02)	0x24	0x36	CALLZ (0x10)	0x0
0x24	SWAP	0x26			

Table 5-4. Packet Layout

5.6.4 Fibonacci

In this section we consider the simple code that computes the N^{th} Fibonacci number. Table 5-4 shows the packet needed to implement Fibonacci for N >= 1 (N is stored at address 0x02), and how the fragments are arranged in memory. For simplicity, each instruction is a separate fragment. The first packet, starting at address 0x10, loads the value N (counter), which specifies which Fibonacci number to compute, and the constants 1 and 0 (pre-loaded into 0x04 and 0x06 to begin with). The fourth instruction decrements the counter and sets the condition bit in the tail if the counter is zero. The counter is then stored back at address 0x02. The seventh instruction swaps the first two operands in the operand stream. The eighth instruction creates a copy of the accumulator at the end of the operand stream. The ninth instruction (ADD) computes the next Fibonacci number. If the condition flag in the tail is set, this new computed value is stored at address 0x08. The two remaining operands are then stored at locations 0x06 and 0x04. Finally, if the condition flag is not set, we loop back to the beginning using a CALLZ instruction, creating a new packet. If the condition flag is set, the instruction is not executed, terminating the program. Figure 5-7 illustrates the creation of this packet with a bootstrapping JMP. In Figure 5-7a, we show the bootstrapping packet inserted at the via in the execution network. This packet is routed along the execution network until it finds a memory port. The JMP instruction in the packet executes at the port and starts fetching data from location 0x10 (where the Fibonacci code is stored). The data returned from location 0x10 (Figure 5-7b) is divided into two parts: 1)

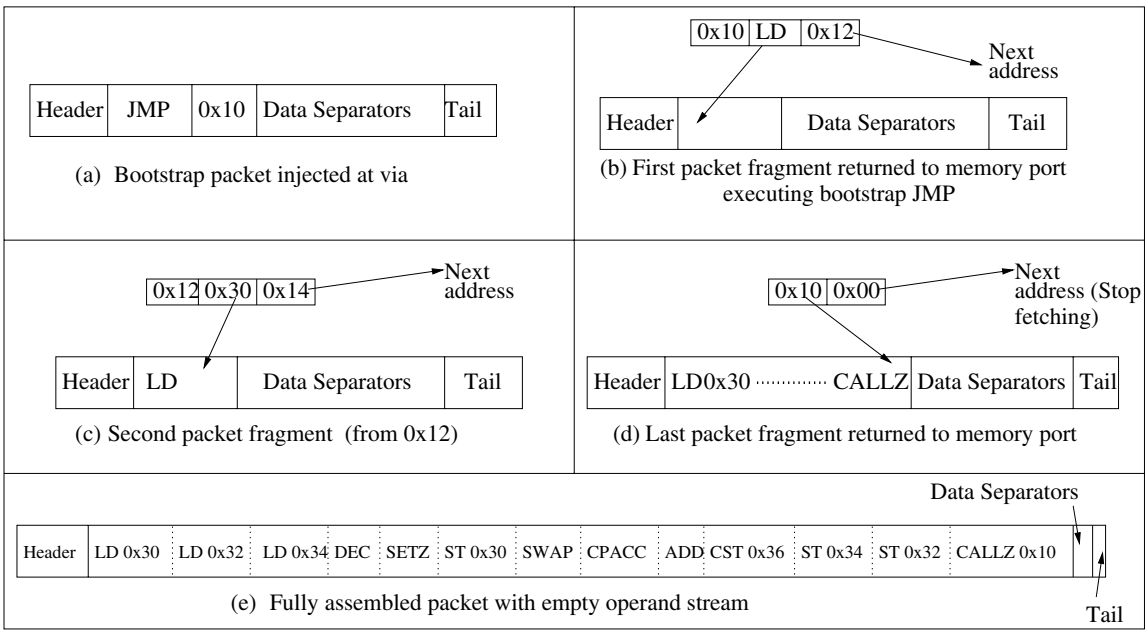


Figure 5-7. Bootstrapping the fibonacci execution packet with a JMP

data for packet and 2) next address. The data for the packet (in this case, a LD opcode) is inserted into the packet and sent out on the execution network. The next address is used to fetch the next fragment of code (in this case, from address 0x12). The data returned from location 0x12 (Figure 5-7c) provides the address for the LD instruction and the address of the next fragment of code. This process is repeated until we get a data fragment back with 0x00 as the next address (Figure 5-7d). This indicates that we have finished executing the JMP instruction. The final packet before execution begins is shown in Figure 5-7e. It is important to note that execution can begin while the JMP instruction is still executing.

To demonstrate our system operation, we simulate its behavior at the bit serial link level executing the above packets. We model a single 32x32 cell with 25% ALU nodes and four corner vias for planar gradients. We assume a random distribution of defective nodes, with 3% of all nodes being defective. The memory system in the cell includes 64 16-bit memory nodes and 80 ports. A system using a depth-first execution network would achieve similar performance (depth-first execution only increases the number of nodes reachable on the execution network). The average time per loop iteration (0x10 to 0x36) is 22,300 cycles and it might be possible to reduce this through loop unrolling. However, only 2,000 of the

22,300 cycles are spent in performing the actual computation. More than 20,000 cycles are spent in accessing the memory system. Figure 5-8 illustrates the execution of the program. We take a snapshot of execution before the first load operation completes. While the absolute performance of this example does not surpass even current CMOS, it serves to demonstrate the operation of a single cell. The greatest advantage of this technology arises from the scale of the system.

5.6.5 String Match

String searching is a common operation in many applications (e.g., searching for particular DNA sequences within a genome). Our string match program loads a 16-bit key and compares it to all data elements within the cell, and a conditional store indicates if a match was found. This implementation requires 48 memory locations for instructions and 16 for data. Therefore, we can search a 32GB database by using all 10^9 cells. The execution time within one cell is 35 ns per comparison, for a total of 28.5×10^6 comparisons/sec. The potential for massive parallelism would be exposed by having each of the 10^9 cells perform a unique comparison, yielding an overall rate of 2.85×10^{16} comparisons/sec.

While string matching helps us demonstrate the high performance that could be achieved with NANA, it serves to illustrate one potential problem: as the size of programs being executed increases, the size of the local memory system within the cell must be made larger. In addition, there is contention for memory locations between instructions and data.

5.6.6 Memory System: Queuing Network Model

As we saw with Fibonacci, the memory system in NANA lowers the potential performance that could be achieved. In this section, we use a simple queueing network model with mean value analysis (MVA) to estimate the throughput of the memory system, and identify potential bottlenecks. Model input parameters and outputs are listed in Table 5-5.

We build a queuing network to model the memory system. It consists of three servers, the anchor point, memory and the “gap” server. The anchor point is a queuing server that queues up memory requests (it acts as a serialization point in the real system). The other

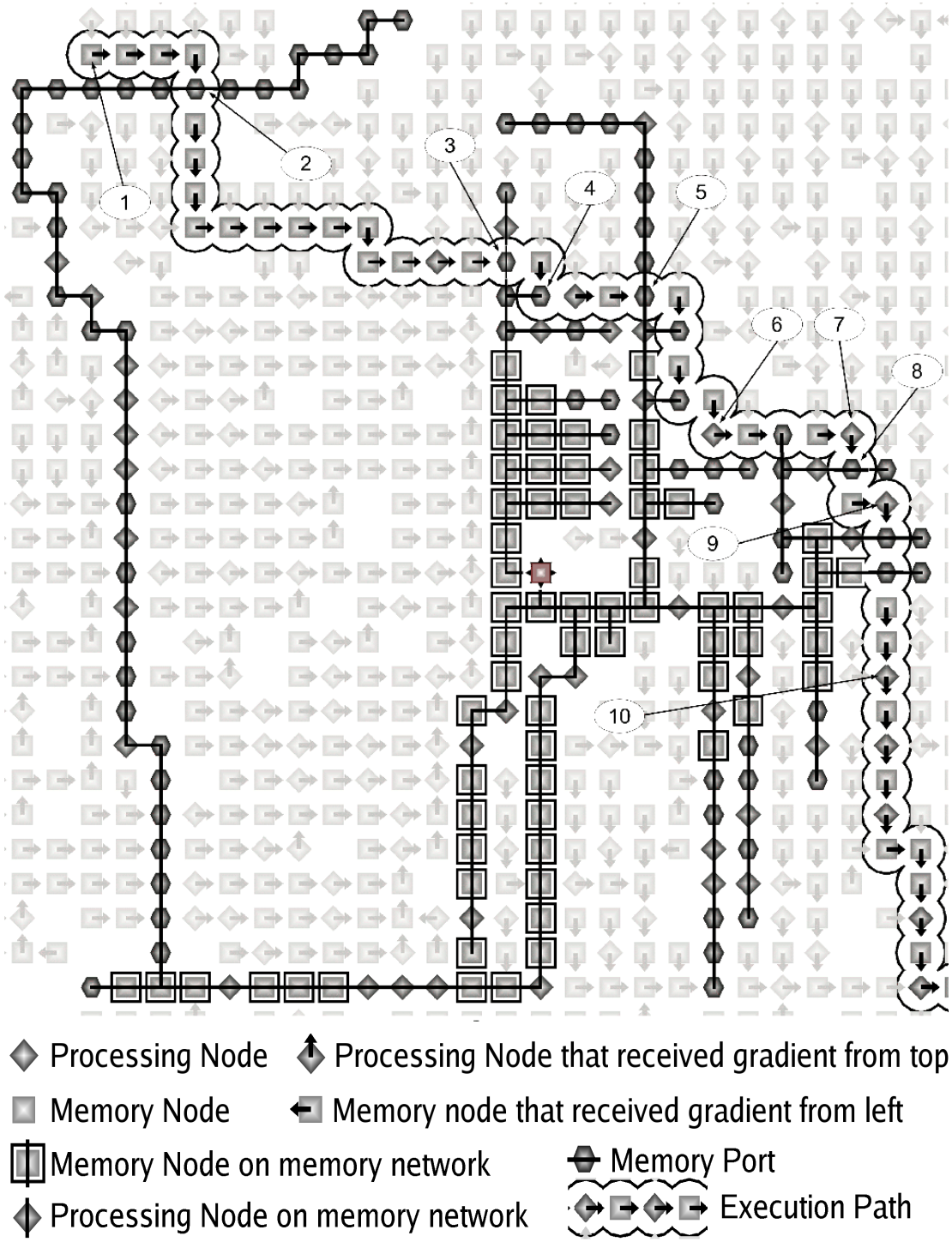


Figure 5-8. The path of Fibonacci code in one direction through configured network with 1024 nodes. Unused nodes in the execution network appear faded. 1: Bootstrap packet injected at via, 2: JMP executes at port, 3: LD 0x02 executes at port, 4: LD 0x04 executes at port, 5: LD 0x06 executes at port, 6: DEC at processing node, 7: CMPZ executes at processing node, 8: ST 0x02 executes at port, 9: SWAP executes at processing node, 10: ADD executes at processing node.

Input	Description	Output	Description
S_k	Service time at server k	U	System utilization
N	# requests in the system	R	System Response Time
		R_k	Response time at server k
		X	System throughput

Table 5-5. Model Parameters

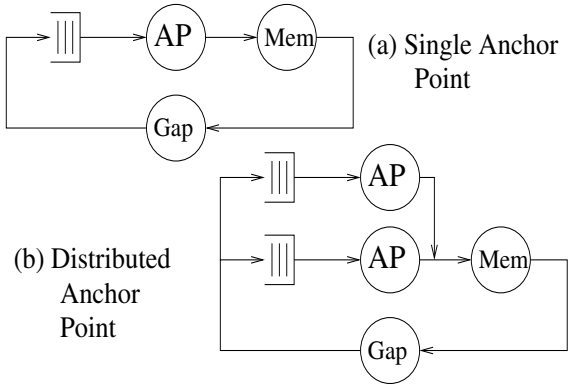


Figure 5-9. Memory Queuing Model

servers simply delay packets by a fixed amount. The gap server corresponds to the gap between consecutive memory requests. Figure 5-9a shows a block diagram of this model. For each experiment, we vary the service time per request at the anchor point, from 32 to 256 cycles. For all the results presented here, we use a gap time of 128 cycles, and a memory service time of 64 cycles (time taken by a memory node to decode the address and respond with data). Each experiment measures the throughput and response time for one to twenty memory requests.

Figure 5-10 shows a plot of the latency vs. the throughput of the system. There are two types of curves in the graph, one corresponds to different anchor point latencies, and the other corresponds to a particular number of memory requests in the system. The anchor point latency curves show that as we increase the anchor point latency, the maximum throughput of the memory network is lower. This could mean that the anchor point is a potential bottleneck. The utilization at the anchor point as obtained from the model also

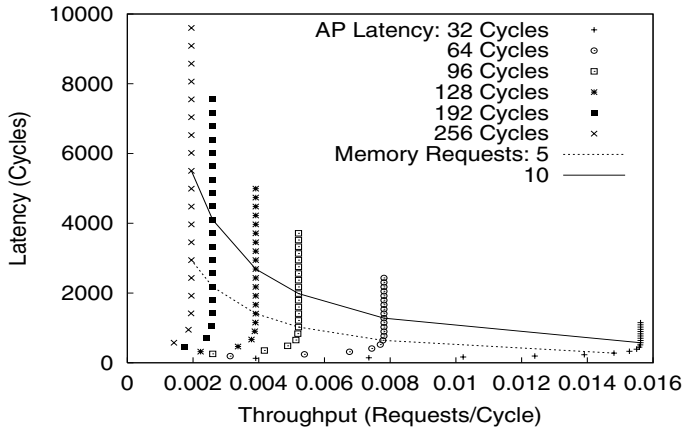


Figure 5-10. Latency vs. Throughput (varying AP latency)

seems to indicate the potential for a bottleneck. It is important to see if the anchor point would indeed be a bottleneck when the system is handling a typical number of memory requests. We estimate the number of requests in the memory network to be between five and ten (for example, 1-2 packets per cell, 4 loads, 1 store per packet). From Figure 5-10, we can see that for all anchor point latency values in the operating range, the latency of the system is increasing without any corresponding increase in the throughput. Utilization of the anchor point in the operating range is 100% for five or more active memory requests. Even with a single packet, executing three memory operations (for example, two loads, one store), utilization is 100%. This indicates that the anchor point will be a bottleneck.

If we model a system with replicated anchor points, where each anchor point can handle requests independently, we would expect an improvement in system throughput. Figure 5-9b shows a block diagram of such a system. Figure 5-11 plots the latency vs. throughput for a system with one to five anchor points. As before, there are two types of curves. The first type of curve represents the memory latency for a varying number of anchor points. The second type of curve represents the memory latency for a given number of memory requests in the network. We keep the latency of the anchor point fixed at 64 cycles. From the curve it is clear that having a set of replicated anchor points improves system throughput and postpones system saturation. The region bounded by the two curves representing five and ten active memory requests represents the expected operating region for the system.

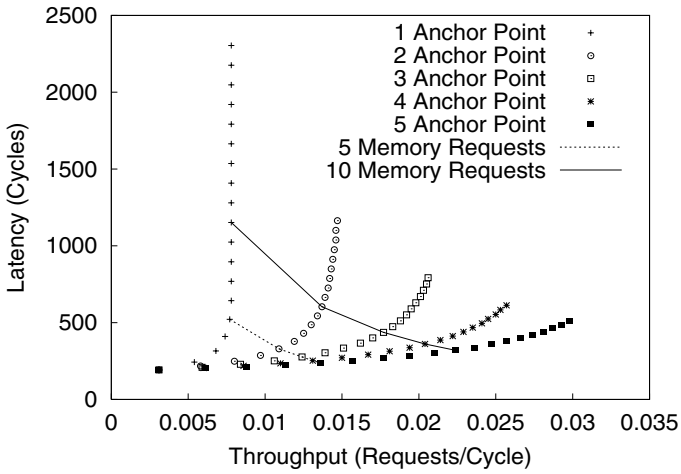


Figure 5-11. Multiple Anchor Points: Throughput vs. Latency

Our analysis of the memory system using the analytical model shows that the anchor node is likely to be a bottleneck. This bottleneck could be reduced by creating replicas of the memory anchor, or by having a distributed root for the memory. However, both these schemes require an increase in the functionality implemented in memory and port nodes. This is expected to be infeasible given the functionality already required in these node types and the limitations imposed by self-assembly. Thus, it is not currently possible to mitigate the anchor point bottleneck in the memory system. Future improvements in the fabrication technology might enable such improvements.

5.6.7 Effect of System Optimizations

There are numerous optimizations that can be implemented to improve system performance. However, most of these optimizations do little to affect the primary system bottlenecks. We briefly discuss two optimizations that show promise, but have little effect on system performance.

5.6.7.1 Routing in the Execution Network

There are two options for routing execution packets within a cell: 1) using planar gradients and 2) using a local execution gradient. The two schemes differ in the number of nodes on the execution network they provide access to. Using depth first routing on a local execution gradient within the cell makes more nodes available for use during execution. However,

this does not directly affect program performance since it does not increase the number of nodes that can be utilized at a time.

5.6.7.2 Memory System Optimizations

In the default system, memory operations (load/store) do not issue into the memory network until they have their address (for loads), or address and data (for stores). However, we can overlap some data movement latency by issuing the load/store requests early. For example, load requests can be inserted into the memory system as soon as the load instruction is decoded, since the address is guaranteed to follow. Similarly, for a store, the store request can be inserted as soon as the store instruction is decoded. While such early issue of requests can improve performance by overlapping data movement, it has little effect on system performance due to other system bottlenecks. In the next section, we explore the bottlenecks that prevent NANA from achieving peak performance in practice.

5.7 Performance Discussion

NANA can theoretically achieve a peak performance of 4.12×10^{21} bitops/sec (assuming 50% of nodes compute), which is significantly larger than today's supercomputers (the IBM Blue Gene can achieve a peak of 4.6×10^{16} bitops/sec, and the NEC Earth Simulator can achieve a peak of 5.2×10^{15} bitops/sec). However, it will be a challenge to realize this performance in practice. The two test programs expose two key limitations of this architecture: 1) under-utilization of nodes and network connectivity, and 2) the memory system is a bottleneck.

5.7.1 Under-utilization of Nodes

One of the key limiting factors to achieving good performance is the fact that nodes spend only a small fraction of their time doing useful work. For example, if we are executing 10 arithmetic instructions, the node that executes the first instruction is doing useful work only when a) it is receiving the first instruction and b) it is receiving its operands for execution. Since there are 10 instructions being executed, which will require 11 operands

(assuming data is pre-loaded), the packet will contain 868 bits (including header, instructions, operands, field separators and tail). Out of these, only 220 bits (header, instruction to be executed, separators, two operands, the operand separators and tail) are relevant to the execution of the instruction. Thus, the node is doing "useful" work only when it is dealing with ~25% of the bits in the packet. No useful computation is performed by the node in the remaining time.

The depth first execution network increases the number of nodes usable during execution, but does not reduce node idle time. The execution network can be thought of as a pipeline of nodes. The pipeline is most efficient only when it is full. Similarly, the execution network is fully utilized only when all nodes are actively executing instructions. This would require the creation of extremely long packets. However, the longer the packet, the longer it takes for a node to execute instructions because longer packets typically have longer instruction and data fields and the a node needs to forward the entire packet before it can handle the next packet. This limits the peak performance of NANA.

5.7.2 Memory System Bottleneck

The memory system in NANA has multiple bottlenecks. For example, to limit design complexity, it is not possible to execute ST instructions from a packet before any LD instructions. This limits the size and content of execution packets that can be created. In addition, all memory requests are serialized through the anchor node. This creates a substantial bottleneck at the anchor node. There is no easy way of alleviating this bottleneck, without significantly adding to the complexity of the system. Finally, our limited routing capability in the random network limits our ability to build a balanced memory network. This often results in unbalanced networks with long latencies.

5.8 Insights and Lessons

While NANA is unable to achieve high performance, we discuss some of the lessons and insights it provides that help us make informed decisions with future designs.

5.8.1 Configuration, Logical Structure and Defect Isolation

Our evaluation of the configuration mechanism shows that it is very effective in dealing with a large fraction of defective nodes (up to 30%). However, the planar gradients are not as useful in providing a general routing framework. Depth first routing on the local cell gradient provides a more effective mechanism for accessing all the nodes in a cell. In addition, the broadcast tree effectively connects nodes in a logical ring created by performing a depth first traversal of the tree. This opens up the possibility of organizing nodes into logical groups to achieve coordinated actions.

5.8.2 Heterogeneous Nodes

Heterogeneous nodes were needed in NANA to keep node complexity within technological constraints, while implementing all the required functionality in the system. However, a system that relies on heterogeneous nodes is likely to be highly dependent on the distribution of the different node types (we observe this during the configuration of the memory system in NANA). We thus would like to minimize the number of different node types in the system, ideally having homogenous nodes.

5.8.3 Bit-level parallelism

NANA is able to do a good job exploiting bit-level parallelism in the program. Since each node is likely to implement a small bit-slice of the total data word, exploiting bit-level parallelism can help overcome some of the performance penalty associated with small ALUs.

5.8.4 Exploiting Node Parallelism

NANA does a poor job of exploiting the massive computational parallelism that exists in a system with 10^9-10^{12} nodes. Any architecture built using such a large number of nodes must make efficient use of this parallelism.

The design and evaluation of NANA provided valuable insights for the design of the data parallel architecture that we present in the next chapter.

5.9 Conclusions

In this chapter, we presented an architecture that addresses the challenges posed by DNA-based self-assembly of carbon nanotubes and other nanotechnologies with similar characteristics (possibly even scaled CMOS). We developed an active-network architecture with an accumulator-based ISA to overcome (1) limited node size, (2) random interconnection of nodes, and (3) a high defect rate. This architecture enables execution packets to search through a sea of heterogeneous nodes for the functionality they need, while avoiding defective nodes. We use an initial configuration phase to impose some limited structure on the computing substrate, particularly for routing and memory allocation. We simulate this architecture running simple programs to demonstrate its viability, and provide preliminary performance numbers. While this architecture is only a relatively unoptimized first step, it addresses some of the key challenges in this class of nanotechnology and it highlights the technology's architectural implications. Despite its limitations, NANA demonstrates that it is possible to build a computing system within the severe technological constraints. In the next chapter, we present a different design that incorporates the lessons learned during the design of NANA to build a high performance data parallel architecture.

6 A Self-Organizing SIMD Architecture

In this chapter, we describe the design of a data parallel architecture called the "Self-Organizing SIMD Architecture", or "SOSA". The goal is to design a high-performance defect tolerant architecture within the assumed constraints of DNA-based self-assembly of carbon nanotube electronic devices. To achieve this goal, SOSA builds on the lessons and experiences gained from NANA. The architecture is designed to exploit a large number of identical nodes, each with limited compute power, by allowing groups of nodes to self-organize to create more powerful computational entities. The fundamental building block in SOSA is a relatively small node (e.g., 1-bit ALU, 32 bits of storage, and communication support for four neighbors) that operates asynchronously. The design of the node is aimed at maximizing the implemented functionality, while meeting the assumed constraints (e.g., limited size, etc.) of the underlying manufacturing technology. While the small node size leads to increased overhead per bit of data processed, some of this overhead can be mitigated by exploiting bit-level parallelism as demonstrated by NANA. We adapt the configuration mechanism used in NANA to group nodes into larger SIMD style "processing elements" (PEs) that are connected in a logical ring and can perform computation on multi-byte data words in parallel. The architecture simplifies the programmer's view of the system by supporting the data parallel programming model that enables the orchestration of data and computation on the ring of PEs.

SOSA overcomes two key problems that are encountered in NANA: 1) low node utilization and 2) the need for heterogeneous nodes. Nodes automatically synchronize with each other through program execution, without the need for explicit synchronization. Since SOSA composes PEs using several identical nodes, the performance of the architecture is no longer dependent on the distribution of different node types. In addition, we can focus on optimizing the design of a single node, rather than having to design multiple node types.

We make conservative assumptions about node size and operating speeds to avoid overestimating the performance of our design due to aggressive technological parameters. SOSA can utilize the higher device densities and parallelism enabled by a large number of nodes better than NANA, allowing it to achieve good performance while keeping operating speeds and power density low. While the design presented here assumes DNA-based self-assembly of carbon nanotube electronics as the underlying technology, it is applicable to other technologies with high defect rates and a loss of precise control over parts of the fabrication process. Further improvements are possible as the technology scales to permit more complex nodes, better inter-node connectivity, and faster devices.

We make the following contributions in this chapter:

1. We design a data parallel architecture ("SOSA") that requires only a single node type, and makes efficient use of the nodes, thus solving two critical problems encountered with NANA,

2. We develop a mechanism to allow a large number of nodes connected in a random network to self-organize to create SIMD style processing elements connected in a logical ring, and

3. We demonstrate through our evaluation that SOSA matches or exceeds the performance of existing architectures on data parallel workloads, while consuming less power and operating at a lower speed.

In Section 6.1, we present an overview of SOSA. In section Section 6.2, we describe the microarchitecture of a node used in SOSA. While our overall configuration approach has been described in Chapter 4, we describe the SOSA specific steps in Section 6.3. In Section 6.4, we present the system-level operation of SOSA. We evaluate the performance of SOSA in Section 6.5, and describe its limitations in Section 6.6. We discuss extensions to the architecture that could be made possible by technological advances in Section 6.7 and conclude the chapter with a summary in Section 6.8.

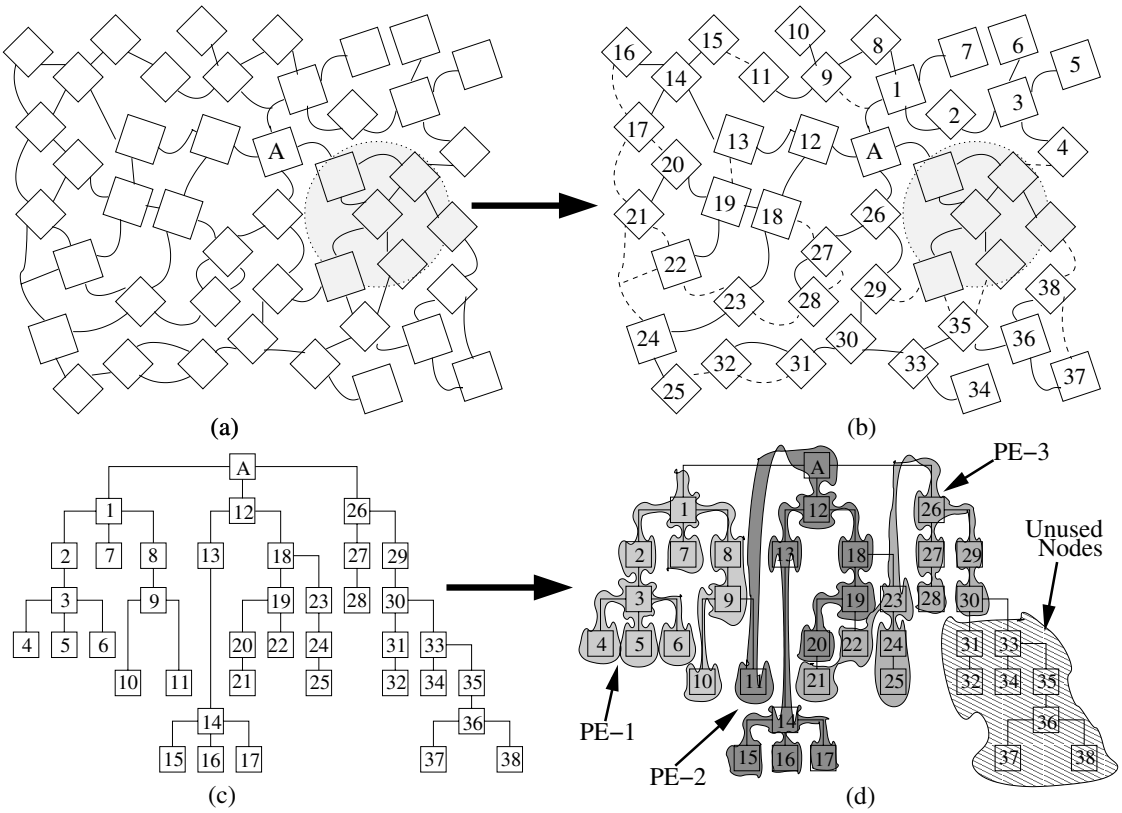


Figure 6-1. Random Node Network (a) before configuration (node A is the anchor), (b) nodes after gradient broadcast with their depth first order specified (dotted lines are links that are not part of depth first traversal), (c) broadcast tree and depth first node order and (d) nodes grouped into three 8-bit PEs (each PE has 8 data nodes and two control nodes).

6.1 System Overview

The goal of SOSA is to build a high-performance, defect tolerant, data parallel computing system. SOSA must efficiently use the large number (~10^9-10^{12}) of nodes connected with a random interconnect. SOSA supports a three operand register-based ISA with predicated execution and explicit PE-Shift instructions to move data between PEs and communicate with an external controller. We assume that the external controller has access to a conventional memory system and orchestrates the flow of instructions and data into SOSA.

Each self-assembled node is a fully asynchronous circuit and there is no global clock to synchronize data transfers between or within nodes. Each node has a 1-bit ALU with a small register file, and nodes are connected to each other by single wire links. Each link

supports very low bandwidth asynchronous communication that transfers 1 bit of data per handshake. To support deadlock-free routing, we add support for three virtual channels (1 bit each). Unlike NANA, we cannot reduce the number of virtual channels required since the virtual networks cannot be made disjoint. The random network of nodes is organized at two levels during a configuration phase. First, since a node is too small to hold a PE, we group sets of nodes to form a PE. Second, PEs are linked in a logical ring providing programmers a simplified system view to reason about inter-PE communication. Figure 6-1 shows a small random network of nodes as it is configured to create three 8-bit processing elements.

The configuration phase maps out defective nodes and connects functional nodes in a broadcast tree. The system can be configured in two ways: 1) as a monolithic system, all nodes on a single logical ring (one "cell"), or 2) as multiple, independent logical rings (multiple "cells"). For a monolithic system, anchors can be used to speed up PE configuration and data input/output by serving as "taps" into the logical ring. The only constraint enforced during configuration is that an anchor cannot partition a PE. In case of multiple cells, we achieve space partitioning by running the configuration algorithm from multiple anchors to create independent cells. Space partitioning is a common technique used in highly parallel systems to increase resource utilization by enabling the execution of multiple instances of one workload, or running multiple workloads.

6.2 Node Microarchitecture

Careful node design is critical in maximizing system performance. Due to limited node size, designing the node architecture involves a trade-off between maximizing functionality (compute, communicate, and self-organize) and performance while minimizing circuit size. To avoid the area and power overhead of routing clock signals and to mitigate the effects of device parameter variation, instruction execution and sequencing within a node are asynchronous. The rest of this section describes the node microarchitecture and circuit design. We split the discussion of the microarchitecture into the data path (Section 6.2.1), control (Section 6.2.2), and inter-node communication (Section 6.2.3), highlighting the

trade-offs between functionality, performance and circuit size. We then estimate the size and power consumption of a node and the entire system based on the node design (Section 6.2.4). We conclude this section with a summary of the node microarchitecture (Section 6.2.5).

6.2.1 Data Path

Each node has a simple data path that consists of a 1-bit ALU, a 32-bit register file, and a data buffer that stores incoming and outgoing data. The register file and data buffer can act as sources and/or sinks for the ALU. The data buffer cannot be written to unless the current instruction is waiting for data, and once written, cannot be overwritten until the data is used by the ALU. All internal node communication occurs on dedicated point to point links. Where possible, we overlap the latency of moving a bit between two parts of the node with other operations.

Nodes can be designed to partition the 32-bit register file into N-bit wide registers that require an N-bit ALU or repeated use of a single-bit ALU. For example, a 32-bit PE could be created with 32 one-bit registers, requiring 32 nodes for the PE, or with 16 two-bit registers, requiring 16 nodes to form the PE. Increasing register width increases the work done per instruction in a node, reduces the number of nodes required to form a PE, and reduces inter-PE communication overheads (since PE length reduces). However, for a fixed sized node, wider registers reduce the number of registers available to a programmer. Simulations reveal that 2-bit wide registers achieve the best trade-off in terms of maximizing the benefit of wider registers and the number of registers available to programmers (we evaluate this in detail in Section 6.5). We also find that program performance is not sensitive to ALU execution latencies shorter than the time taken to send/receive a bit between nodes.

6.2.2 Control

The control logic in the node can be divided into two parts. The first part (configuration logic) is used only during configuration and has two control registers used for defect testing and isolation, and PE configuration.

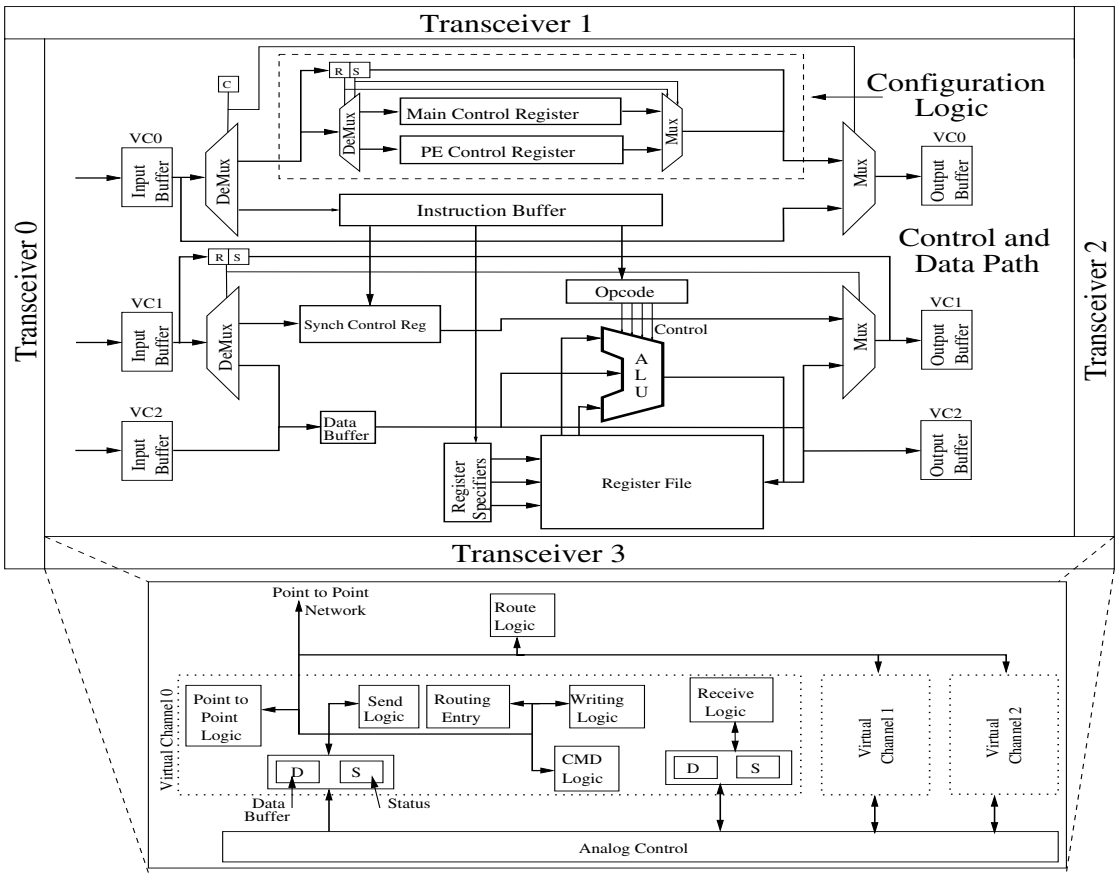


Figure 6-2. Node Floorplan, showing one transceiver, compute and configuration logic

The second part is the run-time control logic used to decode and execute instructions. Figure 6-2 shows a floorplan of the node with the configuration logic enclosed in a dashed-rectangle within the data and control logic block and the expanded view of one transceiver at the bottom. To reduce design complexity we sacrifice latency and use microcoded control logic with each instruction divided into multiple microinstructions. The run-time control logic has three control registers (buffers) to hold each of three micro-instructions that comprise an instruction: 1) opcode, 2) register specifier and 3) synchronization (synch). The synch microinstruction holds an optional counter value (“repeat counter”) to enable the repeated execution of one instruction and avoid broadcasting the same instruction consecutively. The register specifier also includes fields that allow simple increment or decrement operations on source and destination registers in conjunction with their reuse (for striding

through registers). We add a shared circuit that is used to increment/decrement register specifiers and the repeat counter. Because of high instruction execution latencies, the increment/decrement operations can be overlapped with other operations, effectively hiding their latency.

All arriving microinstructions are first sent to an instruction buffer before they are moved to the control registers, creating a simple two-stage pipeline (buffer, execute). Each entry in the instruction buffer can hold all three micro-instructions that form a full instruction. The instruction opcode is fully decoded and copying the instruction into the control registers enables all control signals required to execute the instruction and detect its completion so that the next instruction can begin to execute. Increasing the instruction buffer size can improve performance by overlapping instruction broadcast with execution, but can also lead to greater contention (and reduced performance) on the network since instructions and data must share link bandwidth. Simulations reveal that having a single-entry instruction buffer offers the best trade-off between improving performance and minimizing design complexity.

6.2.3 Inter-Node Communication

Nodes communicate with each other on single-bit asynchronous links. Each end of a link terminates in a transceiver that can handle three virtual channels (using 1-bit buffers per virtual channel). The transceiver can route each virtual channel (VC) independently and requires three bits of state per VC to store the destination address. To support self-organization, nodes include logic to configure static routes (see Section 6.3.1). Virtual channel 0 (VC0) is used to broadcast instructions. Virtual channel 1 (VC1) and virtual channel 2 (VC2) are used to route data in opposite directions on the logical ring. Each asynchronous transaction on a link is controlled through a four-phase handshake. The links support bidirectional full-duplex transfers. To simplify transceiver circuit size and complexity we transfer 1 bit per handshake (which severely limits link bandwidth). Next, we use our node design to estimate the expected size of the node and its power consumption.

6.2.4 Circuit Size and Power Estimates

To estimate the size of a node and its power consumption, we have implemented the different components of a node in VHDL (node design is discussed in detail in Chapter 7). Our simulator (discussed in Section 6.5) models the system in sufficient detail to make it relatively easy to extract a circuit model for most components. Figure 6-2 shows a floorplan of a node, showing the approximate position (not to scale) of the datapath, control and transceivers. Based on our implementation in VHDL, we estimate that the entire node will require about 11,000 transistors. Since the proposed fabrication technology currently imposes limitations on the number of metal layers, we estimate the final area of the node to be the equivalent of 23,000 transistors (based on our experience in laying out circuits) which translates to a 3μm x 3μm node. Recent work [58,106,152] has shown that it should be possible to manufacture DNA grids of this size.

To estimate system power consumption, we use the energy*delay product for carbon nanotube field effect transistor (CNFET) circuits [40]. Based on a conservative switching speed of 1 ns (carbon nanotube based devices are expected to operate at significantly higher switching speeds [18]) and estimated node gate and latch counts, we calculate an upper bound on the per node power consumption. During execution, the configuration logic and a large part of the register file are inactive (at most 3 registers can be active). Accounting for these inactive elements yields a node activity factor of 0.88, which corresponds to a power consumption of 0.775μW per node. To obtain an upper bound on the power density of this system, we assume that nodes are packed with no space between them. Using our estimated node area ($9\mu m^2$) and power (0.775μW), we get a maximum power density of $6.5W/cm^2$, with a node activity factor of 0.88. This is much less than the power densities of current processors, which are greater than 75 W/cm^2. This estimate is pessimistic since the activity factor is a conservative estimate, we cannot pack nodes perfectly, and defective nodes will further reduce power density.

6.2.5 Summary

Each node in SOSA is a small device with the ability to communicate with up to four neighbors, store small amounts of state and perform simple computation. To minimize area and power overheads the nodes use asynchronous logic. We can exploit the high device density and the parallelism enabled by the large number of nodes to achieve good performance without operating at high speeds, thus reducing system power density. Similar approaches have been used to reduce power consumption [44]. In the next section, we describe how we coordinate the operation of these nodes connected through an unstructured network to execute programs.

6.3 System Configuration

To use the random network of nodes to perform useful computation we impose a logical structure on the network and isolate defective nodes from the rest of the system. The ability to isolate defective nodes avoids the need for an external defect map, which would be impractical to obtain given the size and bandwidth limitations of the system. Once defective nodes are isolated, the functional nodes are grouped to form PEs. In the rest of this section, we describe the mechanism that configures nodes into PEs (Section 6.3.1) and optimizations to this configuration mechanism (Section 6.3.2).

6.3.1 Configuring Processing Elements

We use the configuration algorithm described in Chapter 4 to impose logical structure on the random network and isolate defective nodes. Once the configuration algorithm terminates, all reachable functional nodes are connected on a broadcast tree. The configuration algorithm also helps in setting up depth first routing on the broadcast tree. For forward depth first traversals of the tree (which use VC1), each node picks outgoing links in a predefined order relative to the gradient link. This order is reversed for reverse depth first traversals of the tree (which use VC2). If we use a single anchor to initiate the configuration algorithm, all the nodes in the random network are on the same broadcast tree. Alterna-

tively, we can use regularly placed anchors to broadcast multiple gradients and create independent broadcast trees. The only requirement for configuring a system in such a way is the presence of vias distributed through the random network. For example, we could self-assemble the random network of nodes on a silicon wafer with a grid of vias.

A node is too small to hold an entire PE, so we logically group a set of nodes to form a PE. To create PEs with N bits (we assume N=32), we traverse the broadcast tree in depth-first order (on VC1) and group together (N/b)+2 consecutive unconfigured nodes, where N is the data word width, and 'b' is the register width per node. We use one configuration packet per PE. An unconfigured node receiving a configuration packet examines it to determine what node in the PE is to be configured next. The first node holds auxiliary control bits for the PE and is called the "head" node. The next N nodes serve as compute nodes that form the N-bit PE. The last node ("tail") serves as the terminating point of the PE and is used to store the status bits (carry/borrow) resulting from an arithmetic operation. A newly configured tail node sinks the configuration packet. If the broadcast tree does not have sufficient nodes to form an integral number of PEs, the "incomplete" PE is deconfigured before execution begins by performing a reverse depth first traversal on VC2. PE deconfiguration uses a simple packet and starts with the last configured node of the partial PE, and deconfigures all intermediate nodes until it reaches (and terminates at) the head node. Figure 6-3 shows a network in a "configured" state with three 8-bit PEs ordered by the depth first traversal of the network. The links shown with solid lines correspond to edges on the broadcast tree. Links shown with dashed lines do not lie on the broadcast tree and are not used. The unlabeled nodes outside the via are part of an "incomplete" PE that has been deconfigured. The numbers within each node identify the PE that the node belongs to (first label) and the position of that node within the PE (second label). For example, the node marked '2.H' is the head node of the second PE. Figure 6-4 shows the logical order of nodes within a PE.

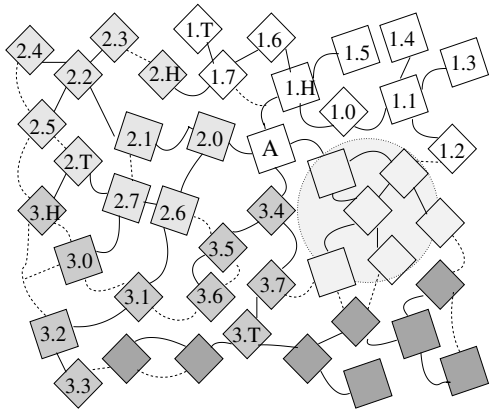


Figure 6-3. System Overview: configured system with 3 8-bit processing elements (PEs)

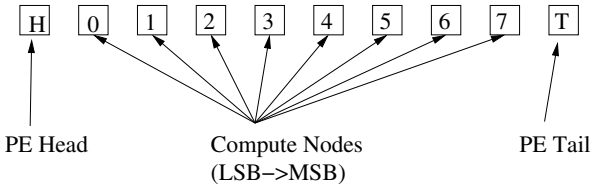


Figure 6-4. PE Layout

6.3.2 Optimizing PE Configuration

The configuration process creates PEs in a greedy manner. However, this can lead to very long PEs (in the number of hops taken to traverse the PE in depth first order), which in turn increases execution latencies of instructions by increasing the time taken to exchange data between nodes of a PE. For example, PE 3 in Figure 6-3 is 20 hops long, because it is allocated using a greedy strategy. For very large broadcast trees with many defective nodes, this length can be excessive and reduce performance. To improve performance, we modify the configuration process to reject any PE that is longer than a certain threshold. Since the post-configuration step deconfigures any partial PE (i.e., PEs with no tail), to reject a PE that crosses the length threshold, we simply start a new PE without creating a tail node. We experiment with different PE length thresholds (see Section 6.5) and find that a threshold of 4 times the minimum PE length achieves a good balance between extra nodes required in the system and the performance gained by limiting PE length. A configuration packet can be augmented to keep track of the number of hops within the PE using a unary encoding. If the maximum length threshold is exceeded, the packet is discarded. We further reduce the number of hops in a depth first traversal of the broadcast tree

by pruning a branch of the tree if there are no configured children on that branch. These optimizations reduce the runtime of our matrix multiply benchmark by 10-15%.

Once PEs are configured, all nodes set a "run" mode bit and each PE remains idle waiting for instructions to execute. Packets are no longer routed to the configuration control registers, until the node receives a global reset instruction. In the next section, we describe how SOSA uses the configured PEs to execute instructions.

6.4 System Architecture

In this section, we describe the architecture of our proposed system in detail. We begin by describing our instruction set (Section 6.4.1) and execution model (Section 6.4.2). Next, we present an example illustrating the execution of an instruction in the system (Section 6.4.3). We then describe techniques to reduce the number of instruction bits broadcast (Section 6.4.4). We conclude with a summary of the key ideas presented in this section (Section 6.4.5).

6.4.1 Instruction Set Architecture

SOSA uses a three register operand ISA, with microcoded instructions. Table 6-1 shows a subset of the instruction set supported by SOSA (Appendix B describes the instruction set in detail). A full instruction has between 39 and 44 bits and contains: a) a 16-bit fully-decoded opcode microinstruction, b) a 20-bit register specifier microinstruction (4 bits per register specifier for a 16-entry register file, and 2 extra bits per register specifier to allow increment/decrement/no change operations), and c) a 3-bit "synch" microinstruction with an optional 5-bit synch repeat counter. Each microinstruction type can be independently broadcast and includes 2 bits of control overhead to select a control register as a destination. Since opcodes are fully decoded, it is relatively straightforward to support fused instructions that include combinations of operations to increase the amount of work done per instruction. For example, a Copy-Shift first copies the source register to the destination register, and then performs a shift operation on the destination register. SOSA also

Instruction Type	Opcodes	Description
Arithmetic	ADD, SUB, INC, DEC, SETGT, SETLT, SETEQ, SETNEQ	Various arithmetic and conditional instructions "Set" instructions set the specified predicate register if the condition is satisfied
Logical	AND, XOR, OR, NOT	Various logical instructions
Shift	SHIFTML,SHIFTLM, PSHIFTML	Various SHIFT instructions. ML=>MSB to LSB, LM=>LSB to MSB. The prefix "p" indicates that the instruction modifies the specified predicate register (not a predicated instruction)
PE-Shift	SHIFTMLPE, SHIFTLMPE	PE-Shift instructions. Move register to adjacent PE
Register operations	CLEAR, CPREG, SWAP	Clear, Copy or Swap registers
Predicated	PR[OPCODE]	Any instruction with the prefix "Pr" is predicated. The predicate register corresponds to the first source register
Fused	CPSHIFTLM, CPSHIFTML	Copies source into destination, and performs a shift on the destination
Signal	SIG_CTRL	Send signal to external controller

Table 6-1. Instruction Set

supports predicated instruction execution (all instructions can be predicated) and has three types of instructions that can modify predicate bits: 1) conditional instructions, 2) unconditional predicate modifying instructions and 3) predicate-shift instructions.

Data exchange with the external controller and between PEs is handled through PE-Shift instructions. When PEs in a cell execute a PE-Shift instruction, each PE sends the contents of the specified register to one neighbor (left or right), and receives a new value for the register from the other neighbor (right or left). By controlling the internal routing configuration of the anchor node, the external controller can be made part of the logical ring of PEs, allowing the use of PE-shift instructions to insert and extract data from PEs. Since these instructions are critical for data communication, it is important to minimize their latency. We optimize PE-Shifts using the following observation: for a B-bit PE, every bit moves exactly (B+2) positions to the left or right, and a node only needs to store the $(B+2)^{th}$ bit in its register file and can "forward" the remaining bits without register access. We use the synch repeat counter to keep track of the number of bits being forwarded by the node. The node stops forwarding when it receives the $(B+2)^{th}$ bit. When a node is "forwarding" data, it copies the data bit directly from its input buffer to its output buffer. This reduces the critical path of a bit through the node.

6.4.2 Execution Model

Instructions are broadcast on VC0 to all nodes, thus PEs, in a cell. Nodes first place instructions in the instruction buffer and then forward them down the broadcast tree. The forwarding of instructions is synchronized with their placement in the instruction buffer. Instruction broadcast stalls when the instruction buffer is full. The arrival of the synchronization micro-instruction is a signal to the node that all parts of the instruction have been received. An instruction moves from the instruction buffer to the node's internal control registers only when the previous instruction finishes execution. Since nodes are bandwidth limited, we allow the partial broadcast of instructions to reduce the number of bits broadcast. If a microinstruction (except synch) is not broadcast, we reuse the previously latched value from the corresponding control register. The synch repeat counter also helps reduce the number of bits broadcast.

Non-predicated instructions can be executed independently by nodes of a PE, if there are no inter-bit data dependencies. For example, an OR instruction can be executed independently by each node, while an ADD instruction forces nodes to wait for a carry to ripple through the PE. The head and tail nodes act as PE delimiters, and ensure that intra-PE data packets do not cross PE boundaries. The tail node also stores the carry/borrow out from arithmetic operations. The head node stores predicate bits (one per physical register) that are used to conditionally execute predicated instructions. When executing a predicated instruction, the head of a PE reads the specified predicate bit and informs the remaining nodes in the PE whether the instruction is to be executed or squashed by sending a "synchronization" micro-instruction on the forward data channel (VC1). Since each node in a PE must await the arrival of the extra synchronization micro-instruction (which is consumed by the tail), execution of predicated instructions is serialized through a PE.

6.4.3 Instruction Execution Example

Figure 6-5 uses the small configured network with three 8-bit PEs from Figure 6-3 to illustrate the different steps involved in executing an ADD instruction. The anchor node

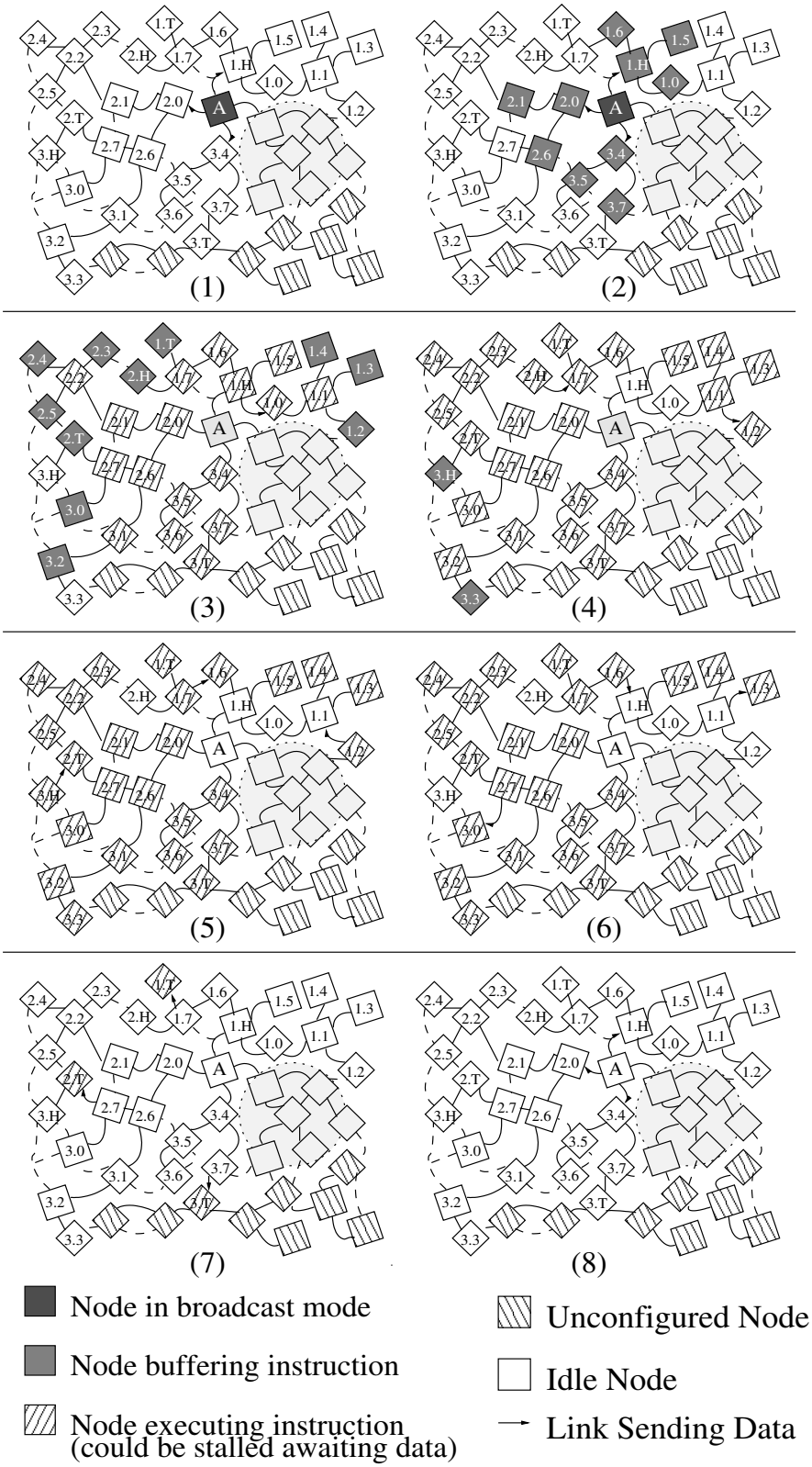


Figure 6-5. Instruction Execution

broadcasts three micro-instructions that form the ADD on VC0 (step 1). As each node receives the micro-instructions it buffers them (step 2) and waits for the synchronization micro-instruction to arrive. Once this microinstruction arrives (step 3), the node can start execution. Since we are executing an ADD, the head node of each PE must insert a carry-in for the first node (step 3). Each node then performs the ADD as it receives the carry-in (step 4, 5, 6), and sends the carry-out to the next neighbor. When a node finishes with the ADD, it clears any temporary internal state used by the instruction and goes back to waiting for instructions to arrive (step 7,8).

One important aspect of the execution model is that different nodes and PEs can be in different stages of execution at the same time. In step 3, nodes 3.H and 3.3 are still idle, while other nodes in PE-3 are receiving data (3.0, 3.2), and some have received the full instruction and are stalled waiting for the propagated carry (3.1, 3.4-3.T). This asynchronous execution within and between PEs allows them to make forward progress independently (as long as data dependencies are satisfied) and helps SOSA tolerate large inter-node communication latencies and achieve good performance. Next, we look at a series of optimizations implemented in each node to reduce instruction bandwidth by allowing reuse of microinstructions.

6.4.4 Microinstruction Reuse

Since our nodes are severely bandwidth limited, we try to minimize the number of instruction bits broadcast. We use the following mechanisms to reduce the number of bits broadcast: 1) allow the broadcast of partial instructions, 2) add an increment/decrement/unchanged field to the register specifier, and 3) add a new type of synchronization instruction that includes a “count” field, to allow repetition of instructions.

Figure 6-6 (a) and (b) show how we can reduce the number of micro-instructions through the broadcast of partial instructions. In part (a), we want to perform consecutive ADD instructions. Here, we take advantage of the fact that the two operations share a control word and broadcast only the register specifier and synchronization micro-instruction for the second instruction. In part (b), we need to shift R2 to the right by 3 bits. In this case,

Instructions	Micro-Instruction Stream
`ADD R3,R2,R1` ;R3=R2+R1	ADD
`ADD R5,R4,R3` ;R5=R4+R3	R3, R2, R1
	synch
	R5, R4, R3
	synch

(a) Partial Instruction Broadcast: Reusing Opcode

Instructions	Micro-Instruction Stream
`SHIFTML R2,3; R2>>=3;`	ADD
	R3 (11), R3 (01), R1(01)
	synch
	synch

(b) Partial Instruction Broadcast: Reusing Opcode + Register Specifier

Instructions	Micro-Instruction Stream
`ADD R3,R3,R1` ;R3=R3+R1	ADD
`ADD R2,R4,R2` ;R2=R4+R2	R3 (11), R3 (01), R1(01)
	synch
	synch

(c) Register Increment/Decrement

Instructions	Micro-Instruction Stream
`SHIFTML R2,3 ; R >>= 3;`	SHIFTML
	R2
	synch, 3

(d) Synch With Counter

Figure 6-6. Reducing Broadcast Bandwidth: Micro-instruction reuse

we just broadcast the synchronization micro-instruction three times after broadcasting the control word and the register specifier. Figure 6-6 (c) shows an example of the register specifier extension, where we add two bits per register specifier (total of 6 bits). This allows us to specify whether the register specifier is to be incremented by one (01), decremented by one (11), or left unchanged (00 or 10). In the example, we want to perform two ADDs, and the registers accessed by the instructions change only by one. For this example, we only have to broadcast the second synch microinstruction (saving one microinstruction broadcast). Figure 6-6 (d) shows an example of the case where we add a counter to the synch microinstruction. We use the same code segment from Figure 6-6 (b), where we had to perform three right shifts on R2. We now encode the repeat count with the synch microinstruction and avoid having to broadcast it three times. Since the width of the synch

microinstruction is 3 bits, if we repeat the instruction at least three times, we save on the number of bits broadcast (for a 5 bit synch counter).

6.4.5 Summary

We have presented a detailed description of our proposed architecture. SOSA is designed to achieve high-performance by exploiting data and bit parallelism in workloads. We organize a large number of simple nodes into SIMD style processing elements that are connected in a logical ring. The asynchronous design of the system enables nodes to overlap computation and communication and reduces synchronization overheads. It is important to note that, while we assume DNA-based self-assembly as the underlying fabrication process, the architecture does not require self-assembly. It is equally applicable to any manufacturing technique that results in high defect rates and a loss of precise control during parts of the fabrication process. Next, we evaluate the performance of SOSA using a variety of workloads to determine if it achieves its design goals.

6.5 Evaluation

In this section, we present a detailed evaluation of SOSA. We begin with a description of our simulation infrastructure, benchmarks and evaluation methodology (Section 6.5.1). We divide our performance evaluation of SOSA into four parts: a) peak arithmetic performance (Section 6.5.2), b) peak performance on data parallel workloads (Section 6.5.3), c) effect of various performance optimizations and sensitivity to operational and design parameters (Section 6.5.4), and d) performance in the presence of defective nodes (Section 6.5.5). Next, we perform an equal area comparison of SOSA and the Pentium 4 (Section 6.5.6). We conclude this section with a summary of our performance evaluation (Section 6.5.7).

Parameter	Value	Parameter	Value
Register File	16 entry, 2-bits per node	**Synch Repeat Counter Width**	5 bits
Time unit	1 ns	**PE Length Optimization**	Enabled
ALU Latency	1 time unit	**Register Increment/Decrement**	Enabled
Data Width	32 bits	**Instruction Buffer Size**	1 entry
Link Type	Full Duplex		

Table 6-2. SOSA System Parameters

Parameter	Value	Parameter	Value
Width	128 (Fetch/Decode/Issue/ Commit)	**Frequency**	10 GHz
ROB/LSQ	8192 entries, single cycle access	**Functional Units**	128 INT ADD, 128 INT MUL, 128 FP ADD 128, FP MUL
Instruction Fetch Queue	1024 Entries	**Branch Prediction**	Perfect
Memory Latency	1 cycle	**Memory Ports**	128

Table 6-3. Ideal Superscalar Parameters

6.5.1 Experimental Methodology

We evaluate SOSA using a custom, event-driven simulator and use results from simulating smaller systems to do an extrapolation to predict the behavior of larger systems. The simulator models the system in great detail, down to bit exchanges between nodes. An "event" in the simulator corresponds to data movement between components within a node (for example, transceiver to control register). Since the nodes do not use a clock, we define the time taken to perform one part of the inter-node asynchronous communication handshake as one "time unit". The latency of all activity in the node is a multiple of this time unit.

Experimental carbon nanotube based devices are expected to operate at frequencies exceeding 100 GHz [18] with demonstrated frequencies over 10GHz [121] (time unit of 0.1 ns), and asynchronous handshakes at high speeds have been previously demonstrated for high bandwidth crossbar networks [84]. We expect SOSA's performance to scale with increased device performance, but we assume a conservative value of 1 ns for the time unit

to avoid over-estimating performance due to aggressive technological parameters. We use the system parameters listed in Table 6-2 for all simulations. We use the custom tool to generate random network topologies. All our experiments in this section use a generated topology with no defective nodes unless explicitly stated otherwise. The running times of benchmarks do not include system configuration time (which is proportional to the number of nodes in the system).

We compare the performance of the benchmarks on SOSA with their performance on two uniprocessors (a Pentium 4 running at 3 GHz, 1MB L2 and 1 GB RAM and an ideal out-of-order superscalar), an ideal 16-way CMP (obtained by linearly scaling performance of the ideal superscalar processor) and an ideal implementation of SOSA (I-SOSA) that uses the same instruction set, but assumes unit execution latencies for all instructions and no communication overhead. The ideal superscalar (I-SS) and ideal CMP (16-CMP) models provide an upper bound on the performance that could be achieved by conventional architectures with aggressive technology scaling. I-SOSA provides an upper bound on how well SOSA would perform if all technological constraints were removed. Table 6-3 lists the microarchitectural parameters used for the I-SS, which we simulate using Simplescalar [9]. We use gcc to generate PISA binaries for use with Simplescalar. For the Pentium 4 (P4), we use optimized binaries generated by Intel's C Compiler (icc, flags: -O3 -fast -tpp7) since they outperform binaries generated by gcc (flags: -O3 -march=pentium4 -msse -msse2 -msse3 -mfpmath=sse -mmmx).

6.5.1.1 Benchmarks

Table 6-4 contains brief descriptions of the test programs, the broad application classes they fall under, and the number of PEs (as a function of N) required by SOSA to run one instance of the program. For all benchmarks other than the encryption algorithms, we configure the system as a single cell with the required number of PEs. For the encryption algorithms, we configure the system as a collection of cells, each of which operates as a pipelined encryption unit. We use C implementations of the best available algorithm for running on the I-SS and P4. For example, we use an implementation of quick sort, which

Application Class	PE Count	Benchmark	Description
Scientific	**N^2**	**Matrix Multiplication**	Multiply two NxN matrices
Image Processing	**N^2**	**Generic Filter - 3x3 mask**	Apply a generic 3x3 mask on an NxN image
		Separable Gaussian Filter	Apply a separable gaussian filter on an NxN image
		3x3 Median Filter	Apply a median filter to an NxN image to reduce noise
General Purpose	**N**	**Odd-Even Transposition Sort**	Parallel sort with nearest neighbor communication
Cryptography	**64**	**Tiny Encryption Algorithm (TEA)**	Simple encryption algorithm used on the XBox
		eXtended TEA(XTEA)	Extension to TEA, eliminates known vulnerabilities
Throughput or Pipelined	**N**	**Search**	Search a database for a match with an input 32 bit string
		Pipelined Binpacking	Pipelined version of bin packing with first-fit heuristic

Table 6-4. Benchmark Descriptions

has an average case running time of O(Nlog(N)) on 1 processor, as opposed to the parallel sort which requires $O(N^2)$ comparisons and O(N) time on N processors to sort N numbers. Each program is also implemented in SOSA assembly and hand-optimized to minimize execution time. The optimizations include loop unrolling and code re-organization to minimize the number of instruction bits broadcast. The SOSA code for matrix multiplication and the image filters assumes data is in place before execution begins and does not account for data input overheads. However, this overhead forms only a small fraction of total execution time and could be reduced by exploiting multiple anchors in the system. The other workloads explicitly account for I/O overheads.

6.5.1.2 Extrapolation

Long simulation times make it impractical to simulate systems with more than 16K PEs. To estimate the performance of SOSA for configurations with more than 16K PEs, we use simple linear extrapolation. We use the performance data obtained from simulating smaller systems to construct an equation for linear extrapolation. We verify the validity of the extrapolation by comparing the performance predicted by the formula against the perfor-

Figure 6-7. Effective Instruction Latency

mance predicted by larger simulated systems. For example, matrix multiply simulations show a uniform scaling of running time for matrix sizes beyond 16x16. The scaling factor between an NxN matrix multiply and a 2Nx2N matrix multiply is about 3.6. We use a conservative scaling factor of 3.8 for the matrix multiply extrapolations to account for overheads that might be encountered for larger systems. The recurrence relation used to calculate the extrapolated runtime is: $R(x)=3.8R(x/2)$. We compare extrapolated running times with simulated running times for two cases (128x128 and 256x256). In both cases, extrapolation always overestimates the run-time (due to the pessimistic scaling factor), making the extrapolated run-times conservative estimates.

6.5.2 Peak Performance

We measure the theoretical peak performance of SOSA by executing a series of integer 'ADD' operations. We use a throughput measure (ADDs/second) as a measure of performance, and compare the performance of SOSA with other high-performance architectures. We also measure the effective latency per ADD by dividing the total execution latency by the number of instructions executed (number of instructions times the number of PEs). As we increase the number of PEs, the effective latency per ADD decreases since we amortize the latency over a larger number of ADD instructions. Figure 6-6 shows the effective latency per ADD operation for SOSA for various network sizes. For a network with 29,411 PEs (~10^6 nodes), the effective latency per ADD is about 2 ps. This translates to 3.5 trillion

Architecture	Peak Performance (32-bit Integer Ops/second)
SOSA (10^{12} nodes)	$1.5x10^{18}$
IBM Blue Gene /L	$2.88x10^{15}$
NEC Earth Simulator	$3.28x10^{14}$
DAMP (10^{12} nodes)	$2.53x10^{18}$
NANA (10^{12} nodes)	$6.94x10^{18}$
SETI@Home	$2.4x10^{14}$

Table 6-5. Peak Performance Comparison

ADDs/second and this increases with the number of PEs in the system. Table 6-5 compares the peak theoretical performance of SOSA with other architectures. We see that the theoretical peak performance of SOSA is higher than all architectures except DAMP [35] and NANA. SOSA's performance is within an order of magnitude of NANA, while operating at one-tenth the speed (SOSA does better than NANA if both operate at the same speed). SOSA supports a wider variety of workloads compared to the DAMP, which is restricted to embarrassingly parallel workloads due to its limited inter-node connectivity. The peak arithmetic performance of SOSA provides us with an upperbound on system performance. Next, we evaluate SOSA's performance on nine benchmarks to determine its behavior on real applications.

6.5.3 Performance

SOSA provides users the flexibility to configure the system to minimize program running time (single cell, single program instance), or to maximize throughput (multiple cells, one program instance each). For many workloads (image filters, matrix multiplication, sorting), system performance is determined by program execution time since we are solving a single instance of each problem. To evaluate the performance of these programs on SOSA, we configure the system to create one cell with the required number of PEs. The latency of an individual instruction in SOSA is high due to the overheads caused by limited node capabilities. However, SOSA can amortize this overhead by executing the same instruction in

all PEs at the same time. Hence, we expect SOSA to perform poorly for small input sizes, where each instruction is executed in a small number of PEs. However, SOSA performance should improve as input size increases and eventually match (or exceed) the performance of the P4, I-SS and 16-CMP. The input size at which SOSA outperforms a particular architecture is application dependent. There are a large number of workloads where high system throughput is desirable. The parallel computational capabilities of SOSA can be used to achieve high system throughput by dividing the system into multiple cells, each having a set of PEs. While there are multiple ways to improve throughput, we focus on using multiple instances of a single application (operating on different data) running on different cells.

We find that SOSA achieves good performance on all the benchmarks that have data parallelism (except our implementation of sort). For a configuration with more than 64K PEs, SOSA matches the performance of the 16-CMP (with the exception of sort). Thus, despite SOSA's severe limits on node computational power, network bandwidth and connectivity, and low control over the fabrication process, it matches the performance of idealized conventional architectures, while running at a lower speed and with a lower power density. We now examine the performance of applications on SOSA in detail, starting with matrix multiplication.

6.5.3.1 Matrix Multiplication

Matrix multiplication is a common operation performed in scientific and other workloads. Our version of matrix multiply is a hand-optimized implementation of the N^3 matrix multiply algorithm for two NxN integer matrices. The assembly code for matrix multiply is presented in Figure 6-8(unrolling is not shown here). We evaluate the benefits of various optimization in Appendix B. We tested multiple versions of the matrix multiply algorithm obtained from the performance database server [114] on the P4 and on the I-SS and found that the naïve version with three nested loops performs the best on the P4 (icc vectorizes loops for the SSE units in the P4), while loop unrolling provides the maximum benefit for the I-SS.

```
; Initialize before Multiply
CPREG R4,R2          ; Copy R4->R2
CPREG R3,R1          ; Copy R3->R1
CLEAR R5             ; Clear R5
; Multiply (Loop Dw times) (Dw: Data Width)
SHIFTLM R1           ;  Shift LSB to MSB (multiply by 2)
PSHIFTML R2,R5      ;  Shift MSB to LSB, LSB to pred.reg R5
PRADD R5,R1,R5       ;   if predicate is set, R5=R5+R1
CLEAR R6            ;   Clear R6
; Accumulate partial products
;---Repeat log2(N) times---
ADD R6,R6,R5         ;  Accumulate partial sum
CPREG R6,R5          ;  Copy R6 to R5
SHIFTMLPE R5        ;  For iteration i, repeat (Dw+2)*i*2 times
; End Repeat
ADD R6,R6,R5         ;  Final add
; Align rows of matrix A for next set of multiplies
;(Repeat (Dw+2)*N times)
SHIFTMLPE R4      ; Move A 'N' PEs to the left
; Move Result
CPREG R8,R9       ; if R8==1, this PE holds the first
                  ; element of a row/column, move this to R9
PSHIFTML R9,R6  ; Move that bit into the predicate register R6
PRCPREG R6,R7     ; if predicate set, copy R6->R7
SHIFTMLPE R7      ; Move R7 one PE to the left (*(Dw+2))
```

Figure 6-8. Matrix Multiply: Assembly Code (no unrolling)

We plot the running time of matrix multiplication in Figure 6-9, The simulator can handle matrix sizes up to 128x128 with reasonable simulation time[1] and we use linear extrapolation to predict the performance for larger inputs. Because of the data communication and instruction broadcast overheads, we do not see good performance on SOSA for small data input sizes. However, as input sizes increase, the parallel processing capability of SOSA helps it amortize the cost of the various overheads, allowing SOSA to eventually catch up and do better than both uniprocessors. Matrix multiplication is able to match the 16-CMP at N=8K. I-SS is unable to fully exploit the data parallelism in the program and achieves an IPC of 9, despite a theoretical peak of 128. Since the binary generated by icc

1. We have successfully simulated the multiplication of two 256x256 matrices. This simulation took about 50 days on a 3 GHz P4 Xeon server with 32 GB RAM. While it is infeasible to run many simulations of this size, this result helps confirm the validity of our extrapolation.

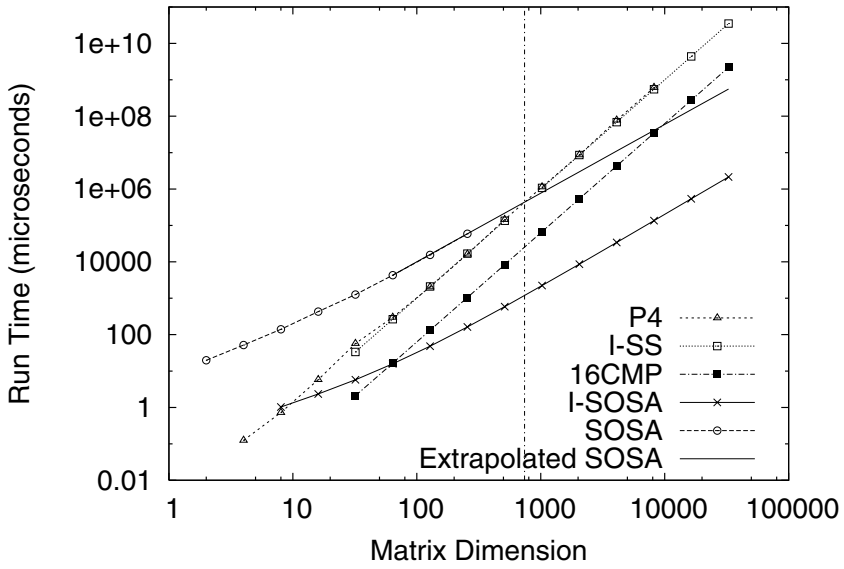


Figure 6-9. Matrix Multiply Run Time (# of PEs increases quadratically with matrix dimension)

is optimized to use the SSE units, the P4 is able to do as well as the I-SS. If we disable the use of SSE units, run times increase by an order of magnitude.

SOSA cannot match the performance of the P4 or I-SS for small inputs, but it also devotes a much smaller area to do computation for small inputs. We can improve the throughput of the system for small input sizes by configuring independent cells and running multiple instances of the workloads. For example, if we assume an area of 100mm^2 (approximately the area of a P4 in 90nm CMOS), we can configure over 5,000 cells, with 64 PEs per cell that each perform an 8x8 matrix multiplication (assuming an average inter-node gap of 1μm) and can achieve significantly higher throughput (~50 times) than either the P4 or the I-SS.

6.5.3.2 Image Filters

Image filters are widely used in almost all image processing software packages. We evaluate the performance of three filters (generic 3x3 filter, separable gaussian filter and median filter) on SOSA. The gaussian and generic 3x3 filter perform a convolution between a mask and the image pixels. The number of instructions used in the convolution is a function of the mask size only. Thus, increasing image size only increases the overhead

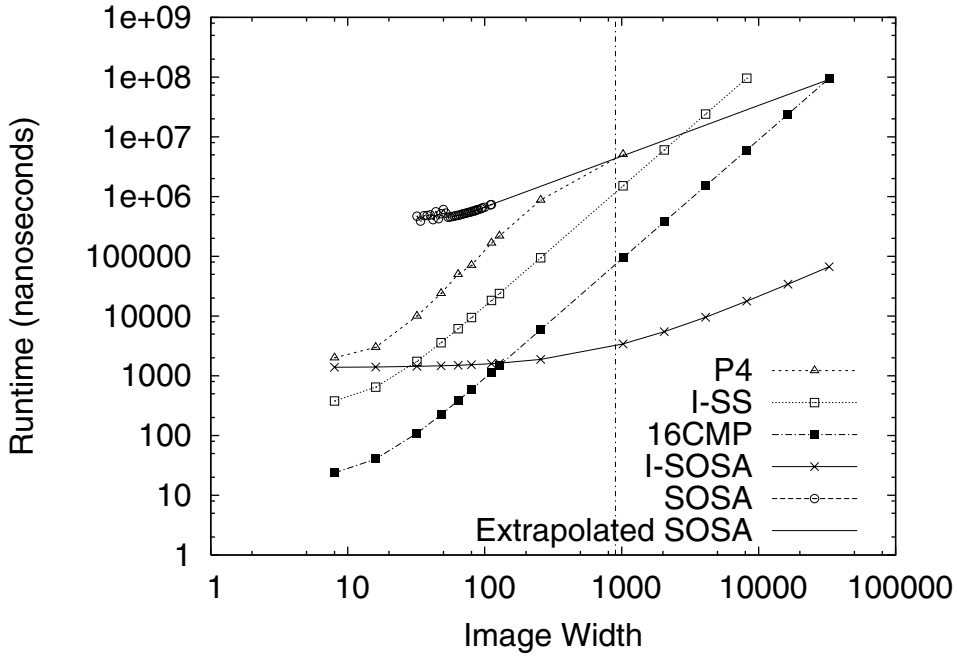


Figure 6-10. Gaussian Filter Runtime

in accumulating neighborhood pixel values in a PE. The median filter differs slightly in that it accumulates pixel values and then computes the median of those values.

We plot the running time for the three filters on different architectures in Figure 6-10, Figure 6-11 and Figure 6-12. The vertical line in each figure corresponds to the image size for which the running time of the filter is the same for the P4 and SOSA. The simulator can handle image sizes up to 128x128 with reasonable simulation time and we again use linear extrapolation to predict the performance for larger inputs. Because of the data communication and instruction broadcast overheads, none of the filters achieve good performance on SOSA for small data input sizes. However, as input sizes increase, the parallel processing capability of SOSA helps it amortize the cost of the various overheads, allowing SOSA to eventually catch up and do better than both uniprocessors. Due to differing overheads, the three filters outperform the P4 and I-SS at different data input sizes. The generic filter and separable gaussian filter on SOSA are able to match the 16-CMP at N=16K. Predicated instructions in the SOSA implementation of the median filter increase runtime overheads, and SOSA is unable to match the performance of the ideal CMP configuration for image sizes up to 16Kx16K pixels. As we saw for matrix multiplication, SOSA cannot match the

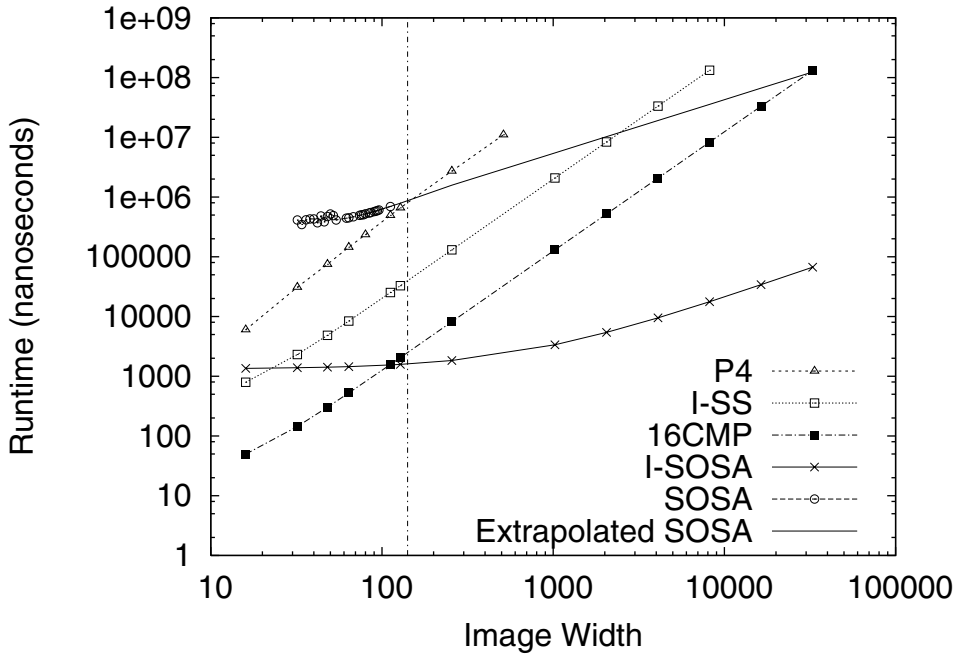


Figure 6-11. Generic Filter Runtime

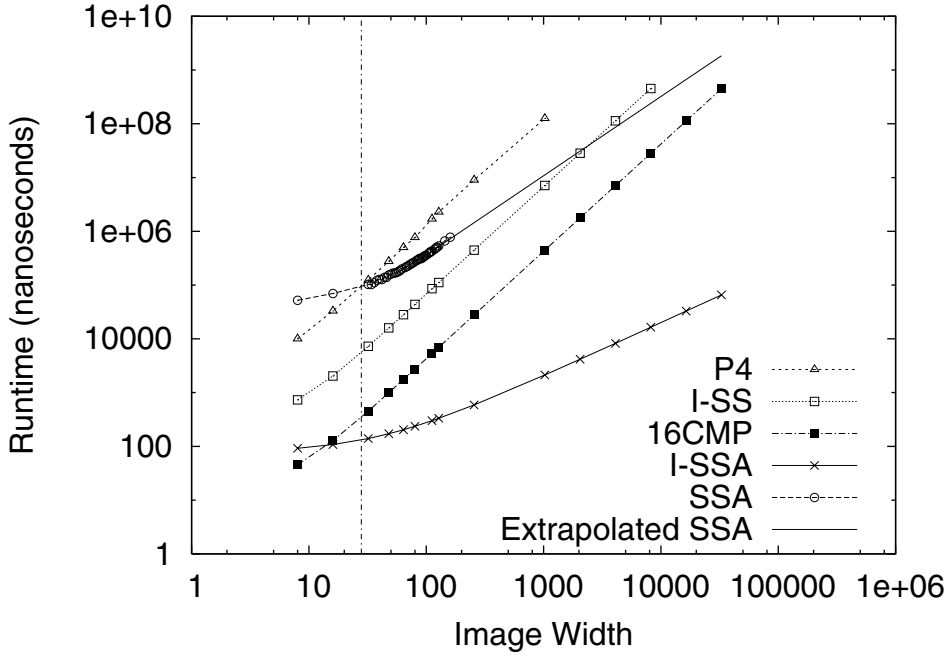


Figure 6-12. Median Filter Runtime

performance of the P4 or I-SS for small inputs for all three filters. However, we can achieve higher throughput by configuring multiple cells.

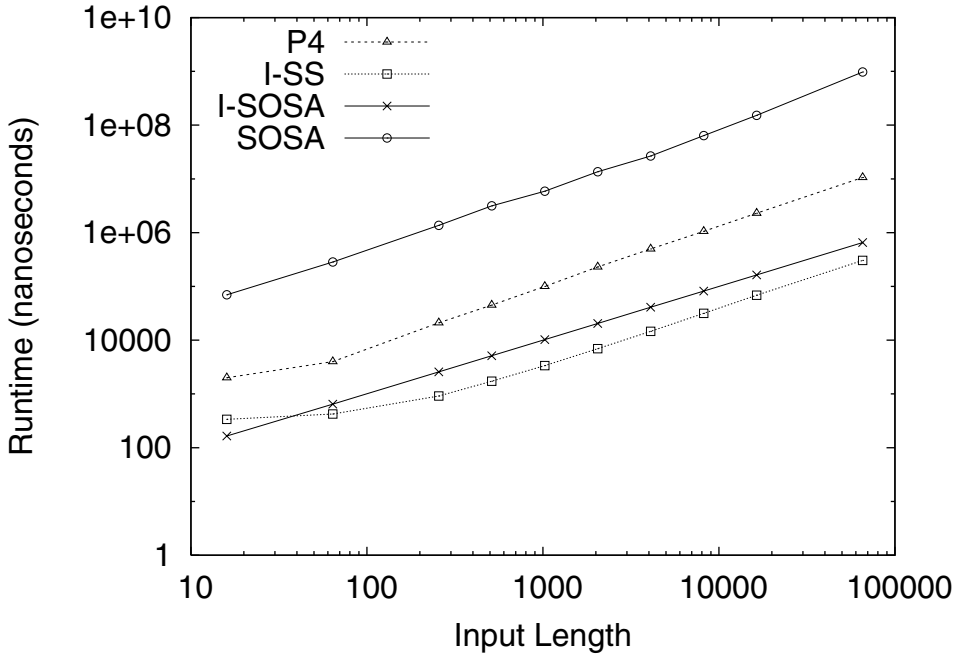


Figure 6-13. Sort Runtime

6.5.3.3 Sort

Sorting data is a common operation found in a wide variety of workloads. We implement a parallel sort algorithm known as the "odd-even transposition sort" [78]. We have examined other parallel versions of sort but do not implement them since they require complex and expensive data communication between PEs. Figure 6-13 compares the running time of sort on SOSA, and the other architectures. It is apparent from these results that this implementation of sort does not perform well on SOSA. Since this is an O(N) algorithm (N=>Input list size), the potential speedup over quicksort on a single processor (average case) is O(log(N)). However, as the number of PEs increases, the overhead of instruction broadcast increases (for N nodes, the height of the broadcast tree is approximately O(log(N)), thus increasing the running time. Combined with our high communication overhead, this makes it impossible for SOSA to match the performance of the I-SS or P4. Note that even I-SOSA cannot outperform the I-SS at sorting. This highlights one key limitation of SOSA: it is not a general purpose architecture and cannot match the performance of conventional processors on general purpose workloads.

Architecture	Throughput (Encryptions/sec)
P4 @ 3 GHz	3.9 M/sec
I-SS	73.62 M/sec
16-CMP	1180 M/sec
SOSA (1 cell)	0.175 M/sec
I-SIMD (1 cell)	27.7 M/sec
SOSA (100 mm^2, 5400 cells)	940 M/sec
I-SOSA(100mm^2)	72300 M/sec

Table 6-6. TEA Throughput

6.5.3.4 Tiny Encryption Algorithm (TEA) and eXtended TEA (XTEA)

TEA [147] and XTEA [97] are two simple encryption algorithms developed at the University of Cambridge that use a combination of shift, add and xor instructions to encrypt 64 bit blocks of data with a 128-bit key. XTEA and TEA use the 128-bit key in slightly different ways, with XTEA requiring more operations per iteration (to achieve better cryptographic security). We divide the random network of nodes into multiple independent cells. Each cell executes a pipelined version of the encryption algorithm and requires at least 64 PEs (corresponding to 64 encryption iterations).

Since each cell operates independently and can handle multiple data blocks in parallel, TEA and XTEA achieve better throughput on SOSA than on the I-SS or P4. A single cell (64 PEs) can perform 175,000 TEA encryptions per second and 170,000 XTEA encryptions per second. Table 6-6 compares the performance of TEA on SOSA with other architectures. The table shows that SOSA can achieve 79% of the throughput of the ideal 16-CMP, while using about the same area as a single core and running at a tenth of the speed (1ns vs. 0.1ns). The comparison with I-SOSA highlights the overheads due to simple nodes and limited bandwidth in SOSA.

6.5.3.5 Searching and Bin Packing

Searching and bin packing are two examples of applications that can be pipelined to achieve high throughput on SOSA. Bin packing with a first-fit heuristic is very similar to performing a linear search on a list (it requires a few more operations to adjust the weights

Benchmark	SOSA	P4	I-SS
Search (comparisons/ sec)	10^{10}	10^{9}	$2x10^{9}$
Bin Packing (bins/ sec)	10^{9}	$5x10^{8}$	$7.5x10^{8}$

Table 6-7. Search and bin packing throughput

of bins). We implement a pipelined version of search in assembly where a database of strings is distributed across the register files of all PEs. In every iteration, the input arrives in one of the registers in the PE, is compared to each entry in the PE register file and then passed on to the next PE. This gives rise to a large search pipeline, which leads to a very high throughput. The search terminates when a match is found by sending a signal to the external control processor. It is important to note that the algorithm does not use any knowledge of the string database to reduce the number of comparisons to find a match. The gap between SOSA and the superscalar processors is smaller for bin packing than for search since it requires more operations (which can happen in parallel on the P4 and I-SS) per iteration. Table 6-7 lists the throughput achieved by the different architectures for search and bin packing.

6.5.4 Performance Sensitivity to System Parameters and Optimizations

In this section, we quantify the effect of various optimizations and changes in system parameter values on the performance of SOSA. We start with the effect of the PE length optimizations (Section 6.5.4.1). Next, we examine the effects of various software optimizations (synch reuse and register specifier reuse) that reduce the number of instruction bits broadcast (Section 6.5.4.2). We then describe the effect of one- or two-bit wide registers on performance (Section 6.5.4.3). Next, we measure the effect of different compute and communication latencies on performance (Section 6.5.4.4). We then evaluate the impact of various instruction buffer sizes (Section 6.5.4.5), and finally, we examine the effect of various node operating speeds (Section 6.5.4.6).

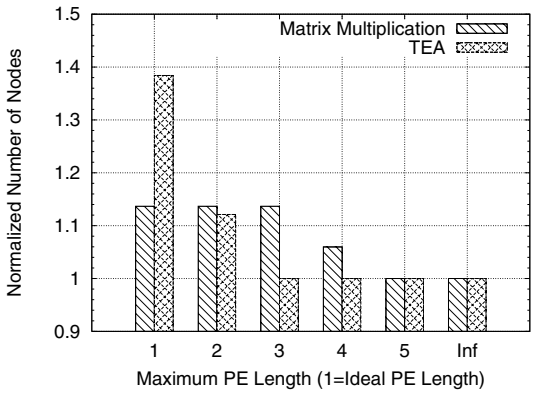


Figure 6-14. Maximum PE Length vs. Number of nodes required for 32x32 matrix multiplication and TEA

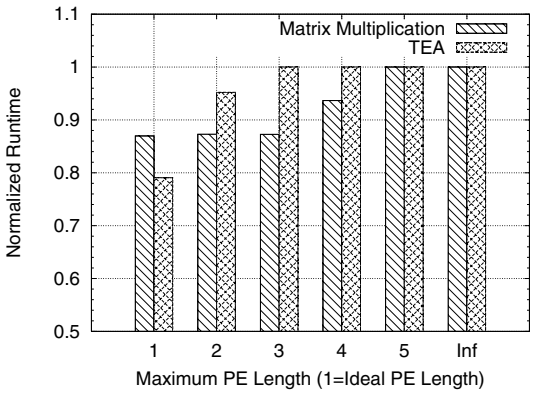


Figure 6-15. Maximum PE Length vs. running time of 32x32 matrix multiplication and TEA

6.5.4.1 PE Length Optimization

In Section 6.3.2, we described a mechanism to limit the length of PEs in order to improve system performance. We pick two representative benchmarks: 1) matrix multiplication for workloads that require monolithic cells and 2) TEA for workloads that require multiple cells. In Figure 6-14 we plot the number of nodes required for 32x32 matrix multiplication (1024 PEs) and TEA (64 PEs) as we vary the maximum permitted PE length in multiples of the ideal PE length (Ideal PE length = 2 + Data Width / Bits Per Register, Inf corresponds to no restriction on PE length). The results are normalized to the number of nodes required if there is no constraint on PE length. We see that as we restrict the PE length, the number of nodes required increases for both benchmarks (up to 14% for matrix multiplication, up to 38% for TEA). In Figure 6-15, we plot the running time for both benchmarks normalized to a configuration with no restrictions on PE length. As expected, limiting PE length reduces program running time (up to14% for matrix multiply, up to 22% for TEA). However, this increased performance comes at a cost of reduced node utilization as some nodes are now unused. For workloads that use multiple cells, this also implies a reduction in the number of available cells (since each cell is larger), which is likely to reduce system throughput. We can strike a balance between improved performance and extra nodes required by limiting PE length, as described in Section 6.3.2.

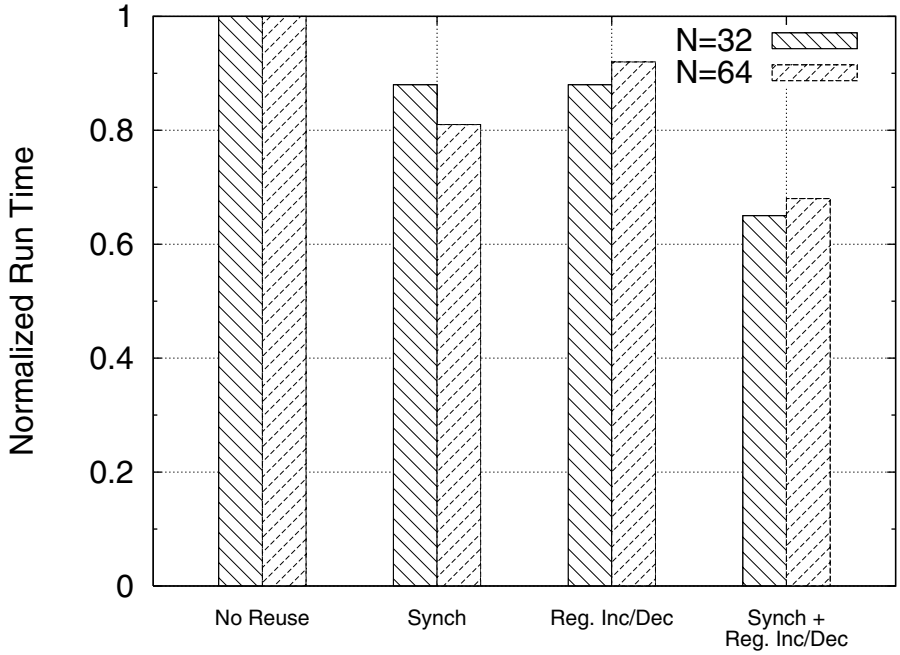
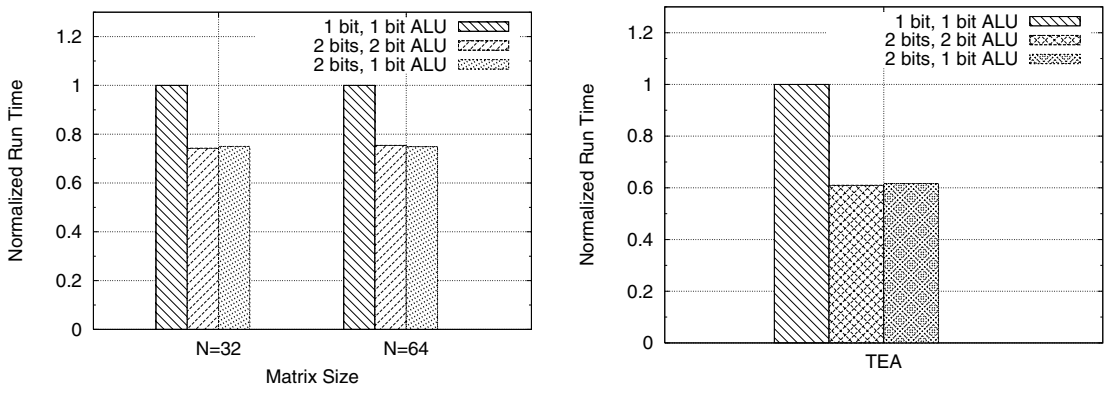


Figure 6-16. Effect of instruction reuse

6.5.4.2 Instruction Reuse

The results presented so far show the best performance of the SIMD architecture on matrix multiply, with instruction reuse allowed. In this section, we quantify the benefits of instruction reuse using matrix multiplication. Figure 6-16 plots the run time of matrix multiply normalized to a configuration without hardware support for instruction reuse. The base configuration includes hardware to optimize the PE-Shift and uses partial broadcast of instructions. We evaluate three cases in addition to the base case, the first with hardware support for 'synch' reuse, the second with hardware support for register increment/decrement, and the third with both. The two bars for each configuration represent the results for 32x32 and 64x64 matrices. Both reuse optimizations reduce the bandwidth requirement of the system by reducing the number of instruction bits broadcast. From our experiments, we see that program run time decreases by 12% and 19% for N=32 and N=64 respectively if the synch microinstruction is reused. Adding support for register increment/decrement decreases program run time by 12% for a 32x32 matrix, and by 8% for a 64x64 matrix. The larger matrix multiply is affected less because the run time of the program is dominated by



(a) Matrix Multiplication (b) TEA

Figure 6-17. Sensitivity to Register Widths

PE-Shifts, which do not benefit from the optimization. If we enable both optimizations, run time decreases by about 35%. A system with both optimizations presents more opportunities to reduce the number of instruction bits broadcast, and clearly benefits more than a system with any one of the optimizations.

6.5.4.3 Sensitivity to Register Width

Increasing the width of the register file increases the work done within a node per instruction. It also reduces the number of registers available to the programmer (since the total storage on the node is assumed to be fixed at 32 bits). To avoid having a very small register file, we only examine having 1-bit or 2-bit wide registers. Increasing the width of the register file requires time-multiplexing of a 1-bit ALU, or the use of a 2-bit wide ALU. We measure system performance under both cases. We plot the normalized running times for matrix multiplication and TEA in Figure 6-17. We see that in both cases, 2-bit wide registers reduce program running time. In addition to reducing running time, 2-bit wide registers also reduce the number of nodes required to create a 32-bit PE by 88% (from 34 down to 18). The reduction in running time occurs for a 2-bit wide ALU as well as for the reuse of a 1-bit wide ALU.

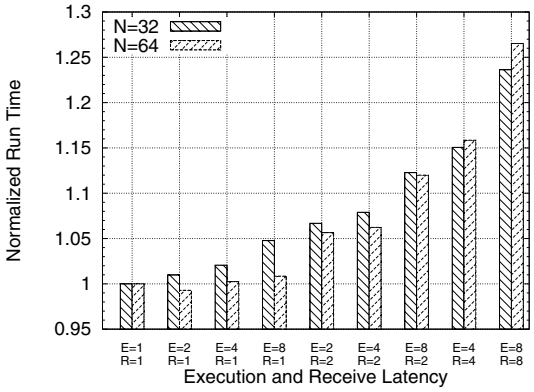


Figure 6-18. Matrix Multiplication: Varying execution and receive latency

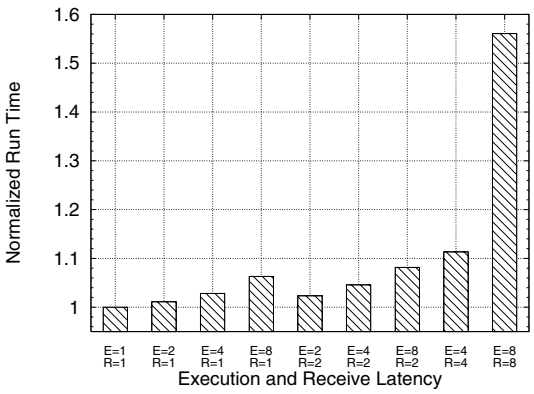


Figure 6-19. TEA: Varying execution and receive latency

6.5.4.4 Sensitivity to Compute and Communication Latencies

We measure the effect of increasing the latency of the control/compute logic of the node. So far, we have assumed that all activity within a node takes exactly one time unit. We use matrix multiplication and TEA to evaluate the effect of increasing the latency of the control/compute logic block as well as the communication latency between the compute logic and transceivers. We plot the normalized running time for matrix multiply and TEA for varying latencies in Figure 6-18 and Figure 6-19 respectively. For both benchmarks, we observe that system performance is fairly insensitive to increased latencies less than 4 time quanta. When the total latency of the two logic blocks is greater than the latency of a bit transfer, we see a significant drop in performance as the latencies of all instructions increase.

6.5.4.5 Impact of Instruction Buffer Size

The instruction buffer stores instructions before the node is ready to execute them. It also enables the instruction broadcast mechanism to propagate instructions down the broadcast tree. Increasing the size of the instruction buffer typically improves performance since it allows increased overlap of communication and computation. However, it can cause increased contention on the bandwidth constrained links, leading to a loss in performance. In addition, increasing instruction buffer size introduces additional complexity into the node. In Figure 6-20, we plot the normalized running time of matrix multiplication

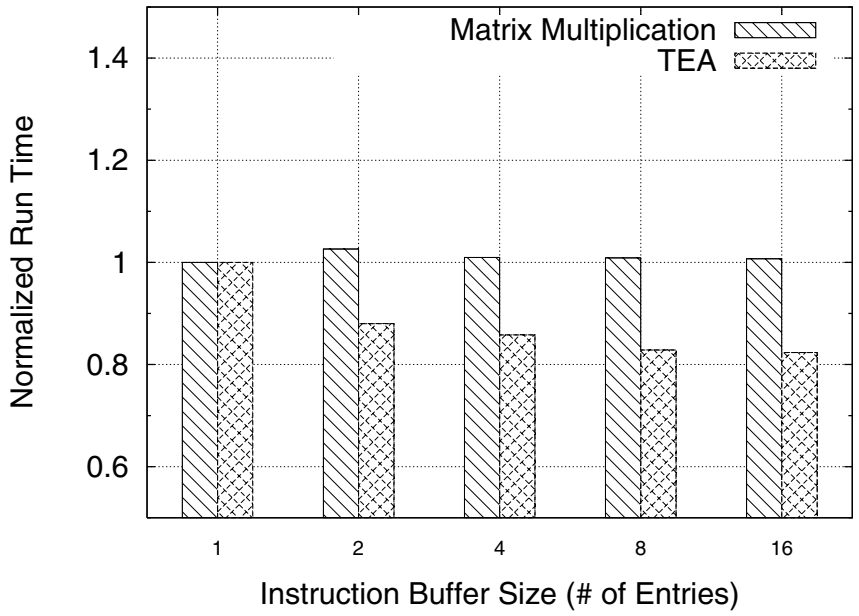


Figure 6-20. Performance sensitivity to instruction buffer size

(64x64) and TEA as we vary the number of entries in the instruction buffer from 1 to 16. For TEA, adding instruction buffer entries improves performance, but results in diminishing gains beyond four instruction buffer entries. For matrix multiplication, we actually see an increase in running time beyond one entry due to increased network contention. We use a single entry instruction buffer as a trade-off between node complexity and performance improvement over a node design without the instruction buffer.

6.5.4.6 Effect of Increasing Operating Speed

The results presented in the previous section assumed a conservative value of 1 nanosecond for the time unit. Recent measurements of carbon nanotubes indicate that it may be possible to operate devices based on nanotubes at very high frequencies (~1Terahertz) [34,122]. In Figure 6-21 we show the run time for the matrix multiply for two matrix sizes (N=128, N=512) for different time unit values. We also show the running time for the Pentium 4 running at 3 GHz as a point of comparison. The figure shows that if SOSA could operate with lower values for the time unit, it would achieve run-times closer to the Pentium 4 for smaller matrix sizes (N=128, with a time unit of ~100ps).

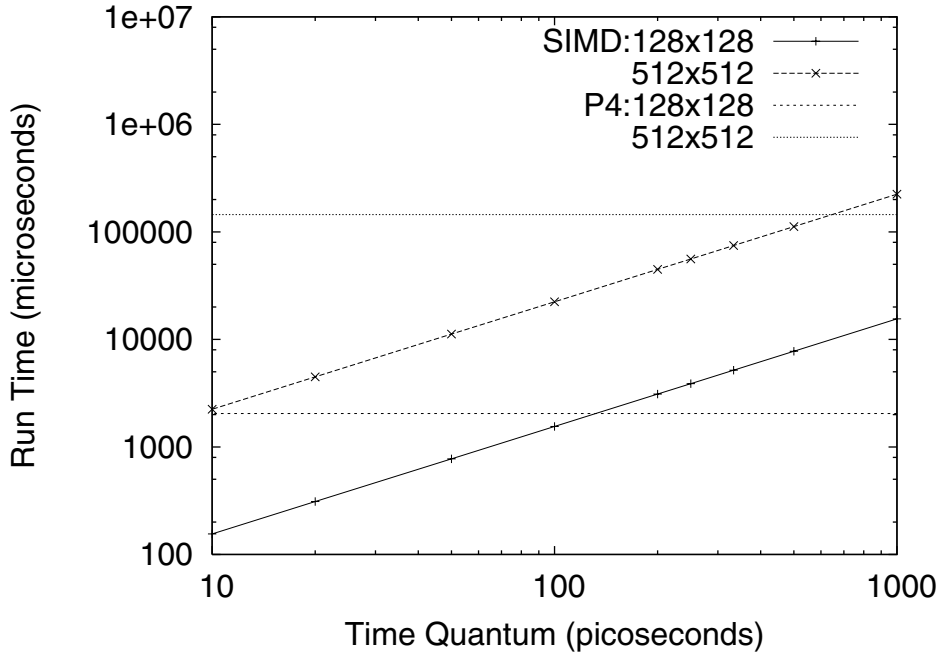


Figure 6-21. Running time of matrix multiply for different time unit values

6.5.4.7 Summary

In our sensitivity analysis, we find that SOSA's performance is not very sensitive to compute and internal communication latencies as long as these latencies are greater than inter-node communication latencies. We find that increasing the size of the instruction buffer can improve performance, but results in increased node complexity. SOSA's performance improves if we use wider registers, which also leads to a reduction in the number of nodes required to form a PE. However, due to node size limitations, there is a trade-off between wider registers and number of registers available. We also find that SOSA can benefit from running at faster speeds, limiting PE lengths and the instruction reuse mechanisms. Next, we evaluate a critical aspect of SOSA's design: its ability to tolerate defective nodes.

6.5.5 Defect Tolerance

SOSA tolerates high node defect rates using the RPF algorithm to isolate defective nodes. Critical logic within each node uses built-in self-test logic to implement fail-stop behavior (Chapter 7). For the encryption benchmarks, the performance of SOSA gracefully degrades

as we lose nodes to defects (up to 30% defective nodes). For the other benchmarks, by over provisioning the system, SOSA tolerates up to 20% defective nodes with a small (<10%) degradation in performance.

The ability to tolerate defects is one of the primary features of SOSA. To test the defect tolerance, and to measure the effect of defects on performance, we run a number of experiments varying the node defect rate. We break down our discussion of defect tolerance into two parts. First, we describe the effect of defects on the throughput of a system configured into multiple cells to run the encryption algorithms TEA and XTEA. Second, we describe the effect of defects on the performance of all the other workloads (which use a single cell).

For TEA and XTEA, if we keep the total area of the system constant ($100mm^2$), as node defect rates increase we are able to configure fewer cells, resulting in reduced throughput. Figure 6-22 plots the throughput as node defect rates increase from 0% to 30% revealing a graceful degradation in performance. The connectivity of the random network of nodes is severely affected by node defect rates greater than 30%. This partitions the network and results in most partitions having insufficient functioning nodes to configure a 64 PE cell.

For single cell applications, the entire system must be over provisioned to ensure that a sufficient number of PEs can be configured, thus defects indirectly impact performance by reducing network connectivity and bandwidth. In all experiments, SOSA had 30% more nodes (24,000 total nodes) than the theoretical value needed for a 32x32 matrix multiply. The 30% extra nodes correspond to the maximum fraction of defective nodes that can be handled by the RPF algorithm while achieving good network connectivity. In Figure 6-23 we plot the run time for matrix multiply for a 32x32 matrix, normalized to a base case with no defects. As can be seen from the figure, there is a slight increase in run time when defects are introduced into the system. This increase is primarily because the average length of PEs increases. We do not show results for the other workloads since they are qualitatively similar. If the system cannot configure sufficient PEs, the problem could potentially be divided into parts that can be solved with the available PEs. Such partitioning, if possible, is beyond the scope of this thesis. Though the defect tolerance capabilities of the RPF algorithm have been demonstrated before, our experiments show that the ability to tolerate high defect

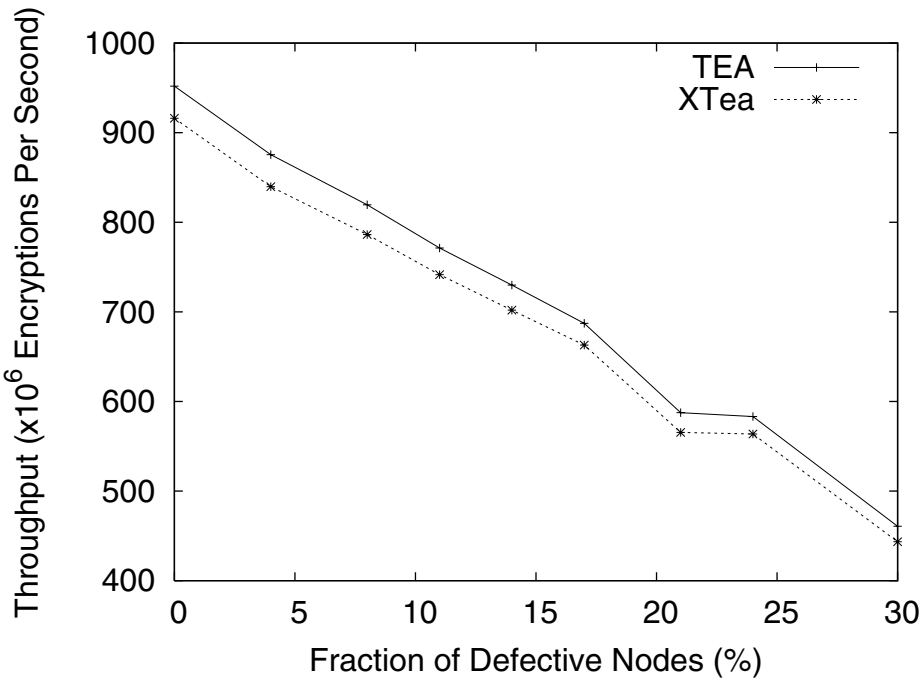


Figure 6-22. TEA/XTEA: Graceful degradation of throughput with increasing node defect rate

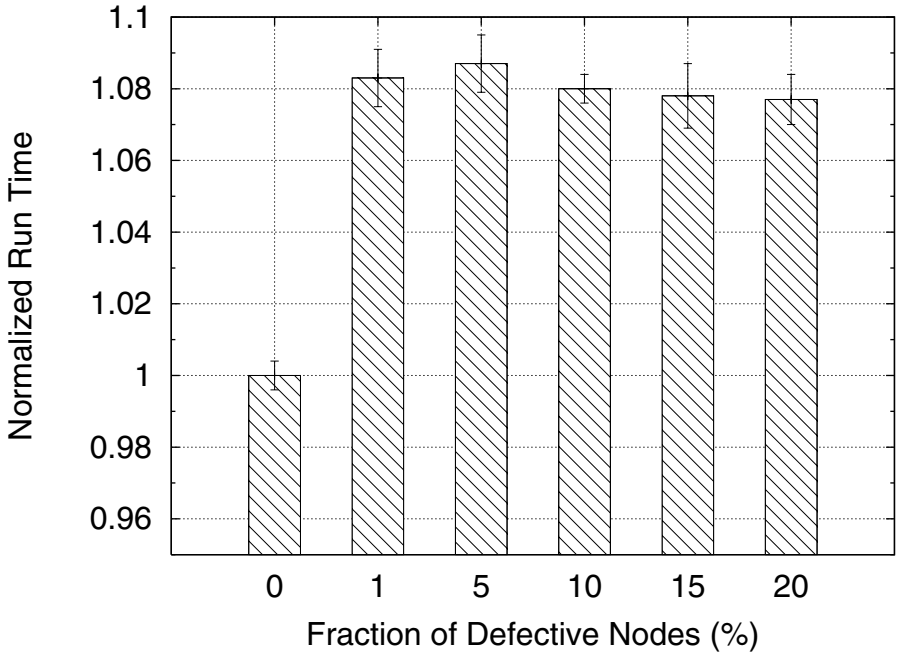


Figure 6-23. Matrix multiply performance with defects

rates incurs only a small performance penalty (~8% for N=32, 32-bit PEs), a characteristic of increasing importance for future systems.

6.5.6 Equal Area Comparison

While we showed that our architecture can do better than a Pentium 4, it is important to ensure that we perform a fair comparison. We compare the performance of SOSA with that of the P4, for the same hardware area. The Pentium 4 manufactured in the 90nm process takes up about 110mm^2 in die area. We estimate that each node occupies about 9µm^2. If we use an area equal to the Pentium 4 die area, we can fit a sufficient number of nodes to do a 512x512 matrix multiply. However, this occupies only a part of the die area (57mm^2). We cannot fit the number of nodes required to do a 1024x1024 matrix multiply (given our current matrix layout) in 110mm^2. Assuming that a time unit in our node is the same as the clock cycle time in a Pentium 4 (~333 picoseconds at 3GHz), we see that SOSA is about 83% faster than a Pentium 4. Note that we could have a network with a very high node defect rate and still achieve better performance than a Pentium 4 given equal area. Theoretically, we could have 47% defective nodes and still outperform the Pentium 4, but our ability to tolerate node defect rates beyond 30% would be limited by the RPF algorithm (see Chapter 4). However, even with the RPF algorithm, we could tolerate up to 30% defective nodes and still outperform the Pentium 4, while using less area.

6.5.7 Performance Summary

The results in this section show that a system built using a random network of simple nodes can outperform a Pentium 4 (P4) and an ideal superscalar processor (I-SS), despite being severely bandwidth limited and operating at a lower speed. A scaled version of the system can outperform an ideal 16-way CMP. The results also highlight the advantage of SOSA's flexibility in configuring independent cells to improve system utilization and throughput. SOSA provides higher throughput than the P4 and I-SS on most of our benchmarks while using the same area. Coupled with the ability to tolerate a significant defect rate, SOSA

shows potential in harnessing the higher device densities that emerging technologies will deliver.

6.6 SOSA Limitations

While SOSA is able to achieve good performance on most of our benchmarks, the design has limitations. Our performance evaluation reinforces the common knowledge that a high computation to communication ratio is critical for achieving good performance. This is especially true on SOSA due to its low bandwidth and high communication latencies. Programs that require little inter-PE communication, nearest neighbor communication, regular and unidirectional dataflow or pipelined implementations of programs that require high throughput are likely to achieve good performance. In contrast, SOSA is unlikely to achieve good performance for programs that require all-to-all communication because of the logical ring topology and limited network bandwidth. Although SOSA achieves good performance on most of the workloads we studied, it is not a general purpose architecture (as clearly demonstrated by the performance of sort). SOSA is unlikely to be able to match the performance of conventional processors on most general purpose workloads. SOSA is also limited by a lack of hardware support for floating point operations. We have software implementations of floating point operations, but performance is limited by the use of predicated instructions to handle control dependencies between different parts of the operations. This lack of support for floating point operations limits us to integer workloads. There are a large number of data parallel scientific workloads that would be well-suited for SOSA, but require floating point operations. As the underlying technology matures, it might be possible to incorporate floating point support and other features in each node. We discuss this in more detail in the next section.

6.7 Extending SOSA

In the near future, it might only be possible to self-assemble circuits that are smaller than the size required to implement a full SOSA node. It might still be possible to implement

SOSA given such a limited capability node but with a reduction in system performance. For example, it might be possible to reduce the number of transceivers, or even time-share a transceiver circuit over four links. Additional areas where functionality could be reduced include the ISA and the register file. However, the simpler node must still have support for the RPF algorithm, PE configuration, instruction execution and support for at least two virtual channels (one for data, one for instructions). It would be useful to study the minimum features required in a node to support SOSA.

As self-assembly technology matures, it is more likely that some of the severe fabrication limitations may be removed. The performance of I-SOSA provides an upper bound of SOSA performance, assuming a time unit of 1 ns. However, with fewer fabrication limitations, it might be possible to achieve better performance by revisiting design decisions that trade-off performance for reduced design complexity. For example, we might be able to use a more complex configuration mechanism to connect PEs in a mesh to increase network bisection bandwidth. If we can manufacture larger nodes, it might be possible to fit a full PE into one node. As emerging device technologies improve, it may be possible to operate them at higher speeds, leading to a potential increase in power consumption. However, even as technology scaling improves performance, the defect tolerance scheme used in SOSA would still be useful. It is important to note that while we assume DNA-based self-assembly as the underlying fabrication process, SOSA does not require self-assembly and is applicable to any manufacturing technique that results in high defect rates and a loss of precise control during parts of the fabrication process.

There are two primary areas in which SOSA can be extended: a) improving the architecture, and b) improving the evaluation infrastructure. SOSA can be improved by adding mechanisms to tolerate transient faults. This could be done by extending PEs to perform simple checksum/parity computations. SOSA's I/O bandwidth can be improved by exploiting multiple anchors. In terms of infrastructure, SOSA development is limited by a lack of software tools (compiler, libraries, debugger, power analysis tools, etc.). The simulator also needs to be extended to model interactions with the external control processor.

6.8 Conclusions

In this chapter, we have presented SOSA, a self-organizing SIMD architecture built from a random network of simple computational nodes. Despite high defect rates, low bandwidth and lack of underlying physical structure we show that, for data parallel workloads, SOSA is able to perform better than conventional superscalar processors, while operating at a lower speed and consuming much less power. A scaled version of SOSA can perform better than an ideal 16-way CMP. As the underlying technology matures, SOSA's performance can be further improved as fabrication limitations are removed. While SOSA does not solve all problems encountered with self-assembled architectures, it is a step towards realizing defect tolerant computing systems built using emerging technologies. In the next chapter, we present the design of fail-stop behavior in a SOSA node and explore how node modularity can help tolerate higher defect rates.

7 Design of a Fail-Stop SOSA Node

In the previous chapter, we presented the design and evaluation of a data parallel architecture (SOSA) built using a random network of simple self-assembled nodes. DNA-based self-assembly enables the construction of a large number of nodes (10^9-10^{12} nodes) in parallel, resulting in a large node network. SOSA uses the RPF algorithm presented in Chapter 4 to isolate defective nodes from functional nodes in the network. However, before defective nodes can be isolated, they must be identified as being defective. Since extracting a defect map to establish locations of individual defective nodes does not scale to node networks of this size, we require an alternative mechanism to identify defective nodes. In this chapter, we present the design and evaluation of fail-stop nodes for SOSA. We use modular node design to extend the defect isolation mechanism to operate within a node and use a combination of hardware and software test strategies to verify the operation of node components. If a node component fails or never completes the test, it is assumed to be defective and is not used, resulting in fail-stop behavior. This allows nodes to diagnose themselves and shut down in case of defects. We use hardware self-test mechanisms to verify critical node components, and software tests for non-critical components.

Distinct tests for different node components enable the use of nodes with some defective components, as long as the defects do not affect critical functionality. This allows the system to tolerate a higher device defect rate, and improves resource utilization. We explore different node failure modes that enable graceful node degradation in the presence of defective components. We find that the use of partially defective nodes increases the device defect probability that can be tolerated by the system by an order of magnitude to 1.5×10^{-4}. This is three orders of magnitude higher than the typical failure probability in current CMOS processes. We make the following contributions in this chapter:

1. We implement simple built-in self-test circuitry to achieve fail-stop behavior for critical logic blocks in SOSA nodes within assumed technological size constraints, and

2. We exploit modular node design to develop multiple modes of failure for a node to allow the node to be used even if some non-critical components are defective.

The rest of the chapter is organized as follows. We begin by describing the design of fail stop nodes (Section 7.1) and evaluate the design using a simple node model (Section 7.2). We conclude the chapter with a summary of the key ideas (Section 7.3).

7.1 Fail-Stop Node Design

The RPF algorithm used by SOSA to achieve defect isolation requires nodes to implement fail-stop behavior. In this section, we explore hardware and software test strategies that can help achieve fail-stop behavior. A node is composed of three main components: 1) communication logic, 2) configuration logic and 3) compute logic, and we develop independent test strategies for each. This simplifies test logic and enables the use of a partially functional node by isolating components that do not pass logic tests. The ability to use partially functional nodes allows us to develop different node failure modes that can better utilize the defect-free parts of a node. We assume a single stuck-at fault model for each component within a node. Thus each component must have the ability to detect a single line stuck at a zero or one.

We begin the section by identifying logic blocks that are critical to achieving fail-stop behavior (Section 7.1.1). We then examine different hardware/software design options for implementing fail-stop, and identify the benefits of each approach (Section 7.1.2). Next, we describe the test mechanisms we use for communication (Section 7.1.3), configuration (Section 7.1.4) and compute logic (Section 7.1.5). Finally, we develop various node failure modes that exploit node modularity to gracefully degrade node capabilities if some components are defective (Section 7.1.6).

7.1.1 Critical Node Logic

We designate a logic block that must be defect free for the node to function correctly as “critical”. These logic blocks must be tested before a node accepts any external input to

Component	Critical	Description
Configuration Logic	Yes	Input arbitration on VC-0, depth first route setup
Transceiver Logic - VC0	Yes	Send/Receive logic for VC-0
Transceiver Logic - VC1/VC2	No	Send/Receive logic for VC-1
Point-to-Point Interconnect	VC0-Yes, VC1/2-No	Data interconnect within Node
ALU	No	Arithmetic Logic Unit
Register File	No	Register File in Compute Block
Instruction Buffer	No	First pipeline stage
Execution Control Registers	No	Storage for microinstructions

Table 7-1. Node Component Classification

avoid the possibility of system misconfiguration. Logic for VC0 (communication logic) and route setup (configuration logic) is critical. All other logic in the node can be tested during the defect isolation phase since it does not affect the ability of a node to receive and send data. While this remaining logic is not critical, it must still be tested to ensure correctness. This can be performed with hardware or in software during defect isolation. Table 7-1 classifies various node logic blocks based on their criticality. The classification of logic blocks into critical/non-critical provides a simple way of determining what logic should be tested in hardware and what can be tested with software. Next, we explore different hardware and software test strategies.

7.1.2 Fail-Stop Node Design Options

Our goal is to achieve fail-stop behavior in nodes with minimal extra hardware. Critical logic must be tested before a node communicates with its neighbors, which implies the need for hardware test logic. For non-critical logic, we can choose between three options: 1) hardware test, 2) software test, and 3) hardware-software hybrid test.

Hardware Test. We can add logic to each node to test the functionality of all components. This is equivalent to built-in self-test (BIST) [7,115] that does not require external test vectors. The primary advantages of hardware testing are low latency and the ability to test the node independent of the rest of the system. However, a node that relies only on hardware test circuitry would not fit within technological size constraints due to the complexity of

the test circuits. This makes a pure hardware test strategy impractical. Note that critical logic still requires hardware testing.

Software (external) Test. For all non-critical logic, we could rely on software based testing using external test vectors. This can be combined with gradient broadcast to allow parallel testing of nodes, which would reduce test latency. This approach works well for instruction execution logic, but is not as useful for other components. For example, software testing of the transceiver circuitry for VC-1 requires hardware support to allow routing of test vectors to the transceiver logic. For small logic blocks, this extra hardware could be more expensive than implementing a hardware test scheme.

Hardware/Software Hybrid Test. The final option for testing is to use a hybrid approach of hardware testing for simple components, and software testing for more complex components. For example, transceiver logic is simple and requires identical testing for all three virtual channels. This can be done efficiently with simple test hardware. Furthermore, this test hardware can be shared between the three virtual channels. While this could increase test latency by a small amount, it results in reduced circuit size. Compute logic is fairly complex, and requires a large number of test vectors to ensure correct functionality. We can exploit existing hardware to test compute logic using external test vectors, with minimal extra hardware. This allows us to keep node size within technological constraints. The test vectors can be inserted after the RPF algorithm completes and before PE configuration begins. Using the same mechanisms as instruction broadcast in SOSA, the test vectors can be distributed to each node in an efficient manner using broadcast. As the test vectors are executed the result must be communicated to the node test logic. The number of test vectors inserted into the system depends on the test coverage required to achieve computation within reliability specifications. A detailed description of the test vectors is beyond the scope of this thesis.

In summary, we use hardware test strategies for node components that can be tested with simple logic. Where possible, we reuse test circuits to minimize overhead. In the next three subsections, we describe our test strategies for the three main components in a node.

7.1.3 Fail-Stop Communication Logic

Communication logic within a node supports three virtual channels and has two primary components: 1) four transceivers, and 2) point to point links. The circuits for VC0 are part of the node's critical logic since they are required during configuration. VC1 and VC2 are not part of critical logic, but can share test logic with VC0.

Each transceiver in a node must be tested to ensure correct functionality as defective logic in a transceiver can lead to incorrect system behavior. A node can be a useful part of a larger system even if it has only one functioning transceiver. However, if there are defective transceivers in a node, it is critical to isolate them from the rest of the system. To achieve this, we augment each transceiver with simple test logic and add a loopback path between the output and input logic of each transceiver. This path is enabled during test only. We exploit the simple four-phase handshake protocol used by the asynchronous logic in designing a test circuit that verifies the operation of the input/output logic. The transceiver is assumed to be defective by default. If the test verifies transceiver operation, the test circuit generates a signal to indicate that the transceiver is operational.

The largest component of the test logic is a two-bit state machine which inserts a test bit pattern into the transceiver output logic. The test pattern consists of two bits (0 followed by 1). The test logic inserts the 0, then waits until it loops back to the input logic. If the test logic successfully receives the 0 from the input logic, it inserts a 1 and waits for it to loop back. If both data bits (0 and 1) are received correctly, the test logic generates a "TEST_OK" signal, which indicates that the transceiver functions correctly. If the data is never received or incorrect data is received, this signal is not generated, isolating this transceiver from the rest of the node. To avoid errors due to a fault in the TEST_OK signal, the configuration test logic requires a transition on the TEST_OK line to indicate a valid test. Figure 7-1 shows the circuit for one virtual channel in a transceiver, along with test logic.

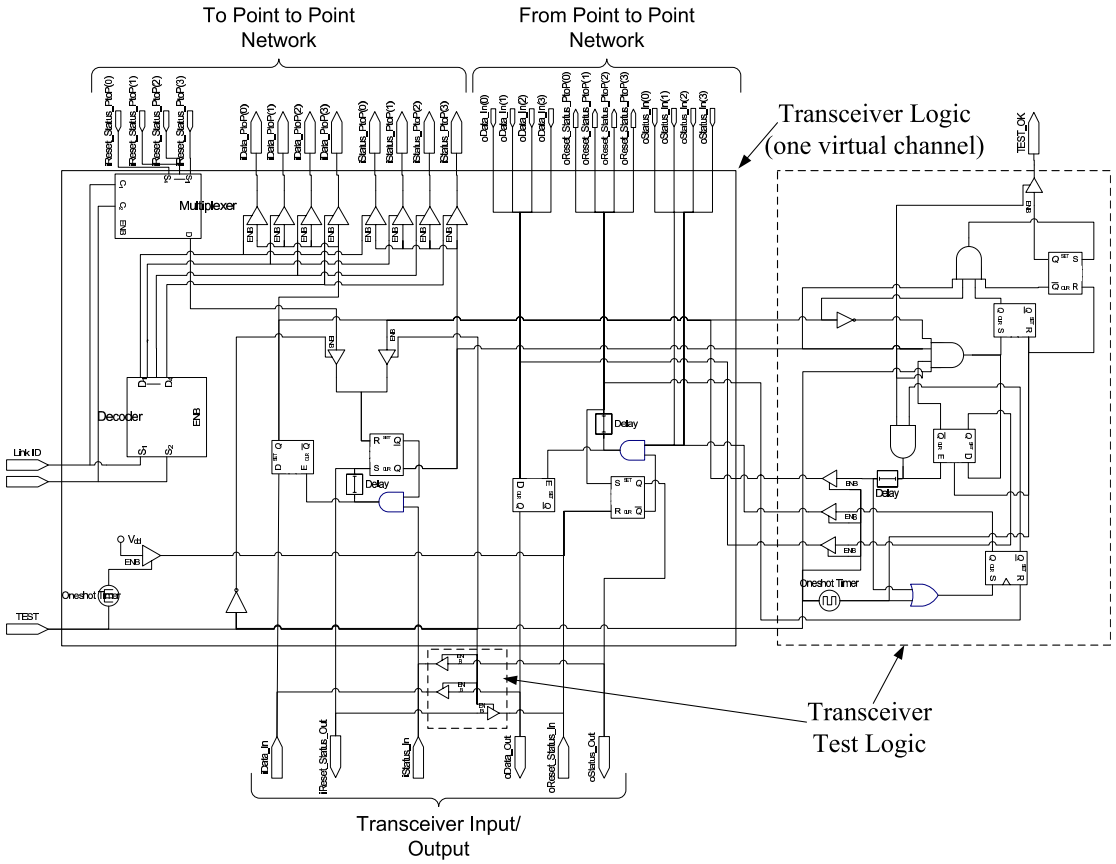


Figure 7-1. Transceiver logic for one virtual channel. (Test logic shown in the dotted rectangles is shared between virtual channels)

In addition to testing the transceiver logic, we need to test the point to point links that connect transceivers. However, routing on the point to point links depends on the result of the configuration process so we test point to point links when we test configuration logic.

7.1.4 Fail-Stop Configuration Logic

Configuration logic is responsible for determining the role of the node within the system, and for establishing communication routes (inter-node and intra-node). This makes the configuration logic an extremely critical component, and a node cannot operate correctly if it is defective. We use a hardware test mechanism that exploits transceiver logic to test the configuration block. Since it uses transceiver logic, the test occurs after the transceiver logic test. The test logic first configures the depth first traversal order of the transceivers within the node, skipping any transceivers that do not generate a “TEST_OK”

signal. Next, the test logic uses a two-bit state machine to circulate a pair of bits (0 and 1) on all virtual channels. If the bits are routed correctly, they arrive back at the insertion point due to the loopback path at the transceivers. If the bits are received correctly, the node generates a "CONFIGURATION_OK" signal. To avoid masking defects due to defective route setup, each transceiver must ensure that each bit passes through it only once per VC. To ensure that there is no stuck at fault in the "CONFIGURATION_OK" signal, we use redundancy and duplicate the signal to ensure correct operation. The configuration test fails if there is a routing error, the bits never return, or the test logic receives the wrong bit values. A failed configuration test causes the entire node to shut down and appear defective to the rest of the system.

7.1.5 Fail-Stop Compute Logic

Testing the compute logic in a node is not as critical as testing the communication and configuration logic. This is because compute logic does not affect system configuration and a node with defective compute logic can be used to improve network connectivity. However, to ensure that the system generates correct results, the compute logic of each node must be tested. This test can be performed at any point before nodes are organized into larger computational entities. This allows us the flexibility of implementing hardware or software test strategies. In either case, the principle is similar to our previous test strategies - a successful test connects the logic block with the rest of the node. If the test fails, or does not complete, the block remains disconnected from other parts of the node.

Hardware Test. We can exploit existing logic to allow repeated execution of test instructions to verify the compute logic. However, this test is unlikely to cover all the logic in the compute block without significant extra hardware. Node size constraints and limited test coverage make this test strategy impractical.

Software Test. Software testing can be performed with minimal additions to the existing node logic. Testing of the compute logic must happen before nodes are organized into larger computational entities. We can combine software testing of the compute block with defect isolation by including the test vectors along with the configuration packet. Another

advantage of software testing of the compute logic is the possibility of exhaustive testing to ensure correct operation.

Our choice of hardware testing for communication and configuration logic, and software testing for compute logic is driven by an analysis of the critical components of a node and technological constraints. As self-assembly technology matures, other test strategies could become more feasible. Next, we describe how we can exploit the modularity of the node to improve system connectivity and tolerate higher transistor defect rates.

7.1.6 Using Partially Functional Nodes

The test logic described earlier in this section opens up the possibility of using nodes with some defective components (if they do not affect system operation). For example, a node with a single defective transceiver can still communicate with up to three neighbors and perform computation. We explore four modes of failure that allow a node to operate with some defective components, defining each scheme based on the number of defects it can tolerate in the compute logic and transceivers. The failure modes are denoted C_xT_y, where x is the maximum number of defects that can be tolerated in compute logic (0 or 1), and y is the maximum number of defective transceivers that can be tolerated (0,1,2, or 3). The default scheme cannot tolerate any defects and is denoted C_0T_0. The four modes we add are: C_0T_2 (a node cannot tolerate defective compute logic, but can tolerate up to two defective transceivers), C_0T_3, C_1T_2 and a hybrid of C_0T_3 or C_1T_2. We list these failure modes in Table 7-2. Each failure mode tries to include nodes that could contribute to system operation. The difference is in the minimum operating components each node must have to be used by the system. Nodes are considered useful under C_0T_3 as long as they have one functional transceiver and can be used to compute. Under C_1T_2 a node is useful as long as it has the potential to improve system connectivity by providing an extra path between two parts of the system (i.e., two active transceivers). The hybrid scheme includes nodes that can either perform computation, or provide an extra path between two parts of the system. As transistor failure probability increases, the number of nodes marked “defective” by

Name	Description
C_0T_0	Node can tolerate no failures
C_0T_2	A node can tolerate up to two defective transceivers (compute logic must work)
C_0T_3	A node can tolerate up to three defective transceivers (compute logic must work)
C_1T_2	A node can tolerate defective compute logic as well as two defective transceivers
Hybrid	A node can tolerate C_0T_3 or C_1T_2

Table 7-2. Node Failure Modes. C_xT_y defines the number of compute logic (x) and transceiver (y) failures that can be tolerated

each scheme increases. Simulations reveal that this increase is fastest for C_0T_0, and slowest for the hybrid failure mode.

Each node requires extra logic to operate with some defective components. This logic keeps track of defective components in the node and disables the node if the defects cross the failure threshold. For example, the C_1T_2 scheme requires six bits to keep track of the 6 primary node components (four transceivers, configuration logic, compute logic). In addition, it requires logic that determines if more than two transceivers have failed. While this adds to the size of the node, it allows us to better utilize each node. Next, we evaluate the effect of different node failure modes on the transistor defect probability that can be tolerated by the system.

7.2 Evaluation

We evaluate three aspects of the design. First, we verify that the test logic for communication and configuration detects defects and measure the overhead of adding the test logic in terms of extra transistors required (Section 7.2.1). Next, we explore the relationship between device failure probability and the expected number of defective nodes in the system, in the context of different node failure modes (Section 7.2.2). Finally, we evaluate the benefit of our testing mechanisms by comparing how well the defect isolation mechanisms perform for different node failure modes (Section 7.2.3).

7.2.1 Test Logic

We implement the test logic described in Section 7.1.3 and Section 7.1.4 in VHDL and simulate it using the synopsys VHDL debugger. We first verify that the test circuit generates the "TEST_OK" signal in the absence of defects in the circuit within a deterministic delay. Next, we check the response of the test circuit when each signal within the circuit under test is forced to exhibit stuck-at behavior (i.e., forced to 0 or 1). In each case, we verify that in the presence of a stuck-at fault, the test logic does not return a "TEST_OK" signal. Since the test logic circulates a 0 and 1, we can detect single stuck at faults on data lines. Since most data exchanges use handshake signalling, stuck at faults prevent the circuit from making forward progress (the handshakes require changes in the logic level). To avoid incorrect test results due to defects in the test logic, we use a combination of two strategies. First, for the transceivers, we use an asynchronous handshake for the TEST_OK signal. This forces the signal to undergo a transition from 0 to 1 before being recognized by the node. Second, we replicate the test signal for logic that cannot be forced to make a transition and require both replicas to match before using the signal. We can detect single stuck at faults in all the communication logic as well as the configuration logic. For some logic blocks, we can also detect double stuck-at faults, and in some cases bridging faults (e.g., when a signal on the input path is bridged to a signal on the output path). However, we do not exhaustively test the ability of our test logic to detect all double stuck-at faults, or bridging faults. The test circuits increase the size of the communication and configuration logic by 18% (736 transistors) and 35% (248 transistors) respectively. The overhead for the configuration logic is higher since the original circuit is not very large.

7.2.2 Node Failure Modes

In this subsection, we explore the relationship between the transistor failure probability and defective nodes for different node failure modes (see Table 7-2). In Chapter 4, we showed that our defect isolation mechanism could tolerate up to 30% defective nodes. In that analysis, we assumed the C_0T_0 failure mode for a node, where 30% defective nodes corresponds to a transistor failure probability of less than $4x10^{-5}$. It is unclear if self-assem-

bly can guarantee such low transistor failure probabilities. We can tolerate a higher transistor failure probability by allowing nodes to operate with some defective components. We compute the expected number of defective nodes over a range of transistor failure probabilities, for different failure modes.

To study the relationship between per-transistor reliability and the fraction of defective nodes, we analyze a system with 10^6 nodes. Each node is assumed to have 10,000 transistors, with a uniform device failure probability (P_f). We use a uniform random number generator to generate random numbers (RND) in the interval [0,1]. Each random number corresponds to one transistor in a node. If RND<P_f, the transistor is defective. Each transistor is mapped to a component, and a defective transistor renders the entire component defective (the hardware test logic detects single stuck at faults in the configuration and communication logic, and we assume that software test vectors can detect defects in the compute logic). For each node, we compute whether it is defective for each failure mode. For each value of P_f, we run 500 experiments with different random seeds. This analysis ignores defective interconnect (within and between nodes). To get an estimate of the effect of defects in interconnect, we can use the number of unit cells in a node to approximate defects in both transistors and interconnect.

In Figure 7-2, we plot the percentage of defective nodes in a system with 1 million nodes, as a function of the transistor failure probability. Each curve corresponds to one failure mode. As expected, the number of defective nodes in the system decreases as device reliability increases. However, we also see that the ability to test components within a node and allow graceful degradation allows us to reduce the number of defective nodes without increasing device reliability. It is important to note that for the hybrid failure mode, while a smaller number of nodes are designated defective compared to other failure modes, a large number of nodes have some defective components. While nodes with defective compute logic cannot be used to perform computation, they are useful in improving the connectivity of the network. Next, we use two baseline network topologies to evaluate the benefit of using partially defective nodes.

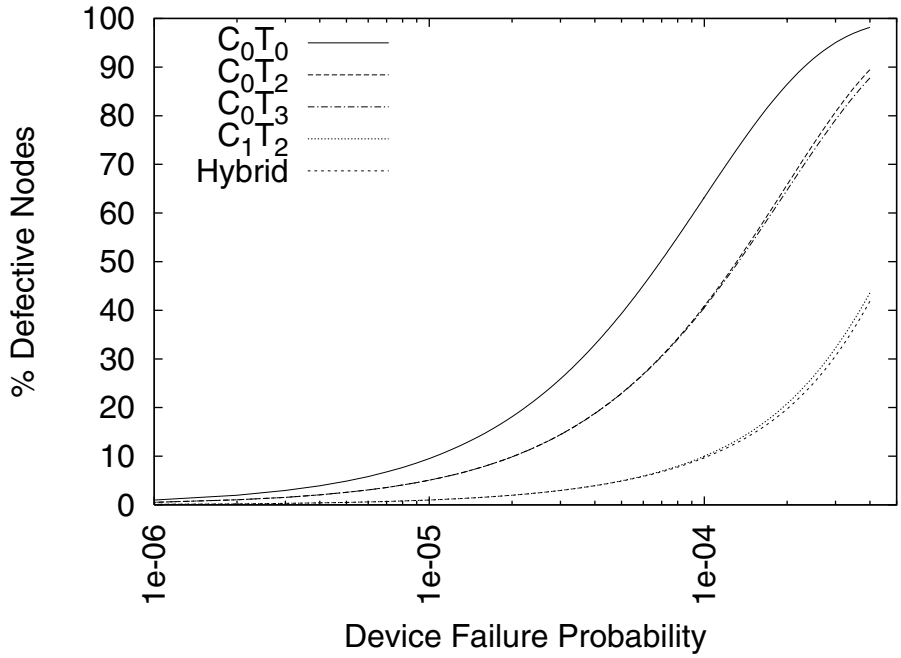


Figure 7-2. Percentage defective nodes vs. device failure probability for different node failure modes

7.2.3 Defect Isolation with Partially Defective Nodes

In the previous subsection, we examined the effect of different node failure modes on the relationship between transistor failure probability and node defect rate. This analysis did not examine the effect of the location of defective nodes on the system. We now explore two different node topologies (random and grid) to determine the effectiveness of the defect isolation mechanism with partially defective nodes.

First, we compute the number of non-defective nodes that are reachable by the broadcast as a function of device failure probability, for three node failure modes (C_0T_0, C_0T_3 and Hybrid). We expect C_0T_0 to have the lowest number of reachable nodes, followed by C_0T_3, with the hybrid mode having the highest number of reachable nodes. However, a large number of nodes that are reachable with the hybrid mode have defective compute logic and only act towards improving system connectivity. To account for this difference, we also plot the number of reachable nodes with operational compute logic (denoted as Hybrid-Compute). Figure 7-3 plots the average number of nodes (as a percentage of total

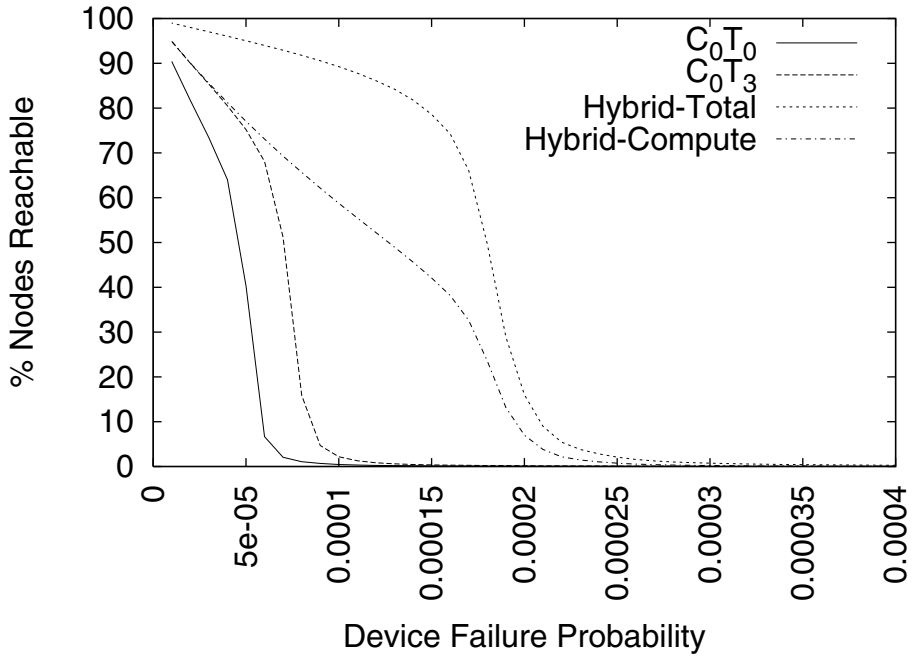


Figure 7-3. Percentage Nodes Reachable vs. Device Failure Probability for a grid with different node failure modes

nodes) that can be reached for the three failure modes as a function of device failure rate, when nodes are connected in a 100x100 grid. For each device failure rate, we use 100 seed values for the random number generator to create different defect distributions, and compute the average of these 100 runs. From Figure 7-3, we see that the Hybrid failure mode delivers a significant advantage over C_0T_0 and C_0T_3 (even if we look at nodes with functioning compute logic only). While there is a sharp decrease in the number of reachable nodes beyond a certain device defect probability, this threshold is higher with Hybrid failure than with C_0T_0 failure.

We also compute the number of non-defective nodes that are reachable in a random network with 10,000 nodes. This random network is meant to be representative of self-assembled networks of nodes. The random network has inherently lower connectivity than a regular grid and some nodes might be disconnected from the rest of the network. This implies that we should see a reduction in the failure probability threshold for all schemes. For the random networks, we generate 100 random topologies, and then use 100 seed values per topology to create distinct defect distributions and get statistically accurate results. Figure 7-4 plots the average number of nodes (as a percentage of total nodes) that

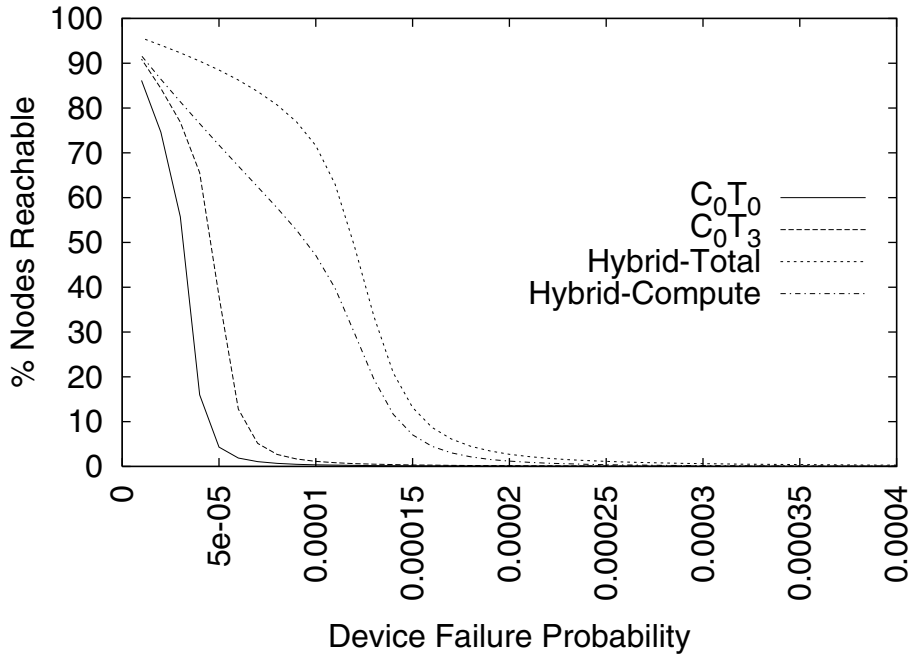


Figure 7-4. Percentage Reachable Nodes vs. Device Failure Probability for a random network with different node failure modes

can be reached for the three failure modes. As expected, we see the knees in the curves have shifted left, but the general shapes are similar to those seen for a regular grid.

Finally, we evaluate the benefit of using the hybrid failure mode over C_0T_0. In Figure 7-5, we plot the average fraction of all nodes that are reachable as a function of the percentage of defective nodes as defined by C_0T_0 (i.e., single defect renders node unusable). We plot two curves each for two types of network topologies (grid and random). The two curves correspond to the number of nodes reachable using C_0T_0, and the number of nodes with functioning compute logic reachable when using the hybrid failure mode. Note that the total number of nodes reachable by the hybrid mode is greater than those reachable with functioning compute logic, since the hybrid mode uses nodes with defective compute logic and two (or more) functioning transceivers. We see that the hybrid failure mode allows us to use nodes that would be unusable with C_0T_0.

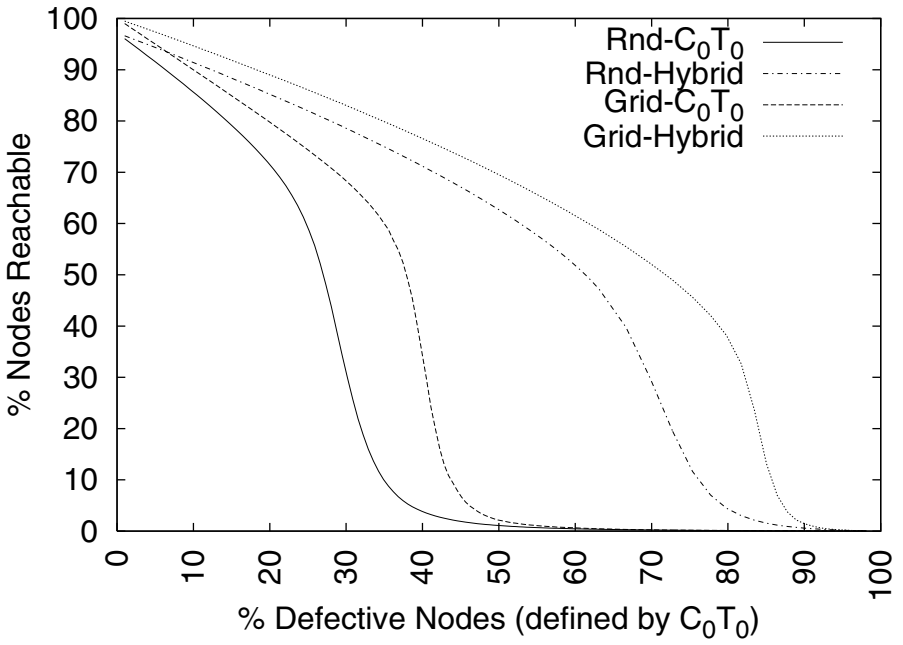


Figure 7-5. Effect of using nodes with some defective components

7.2.4 Result Summary

We have presented a scheme for achieving fail-stop behavior in limited size nodes by dividing them into modular components. We analyze the trade-offs in implementing hardware/software test schemes for the components, and use hardware testing for critical node logic, and software testing for other logic. Our results show that allowing partially defective nodes to participate in system operation increases the transistor failure probability that can be tolerated by the system. We have shown that allowing nodes with defective compute logic, but functional communication logic to remain in the system improves network connectivity.

7.3 Conclusions

The use of partially functional nodes improves network connectivity, and helps the system tolerate devices with higher failure probabilities (increased from 4×10^{-5} to 1.5×10^{-4}). The improved network connectivity allows the system to use a larger fraction of available compute resources, which can lead to an improvement in system performance. As self-assem-

bly matures as a technology, node size restrictions could reduce, allowing the use of faster, and more comprehensive hardware test schemes. In the next chapter, we explore the trade-off between node complexity and the control required over self-assembly to create well-connected networks of nodes.

8 Self-Assembled Networks: Control vs. Complexity

In this thesis, we have assumed that the nodes created by DNA-based self-assembly are connected either in a mesh, or in a random network. However, these two topologies lie at opposite ends of a spectrum defined by the amount of control exercised over the self-assembly process. In Chapter 3, we described one possible method of connecting self-assembled nodes by growing DNA links between them and metallizing the links. However, there is no easy method to control node placement and orientation during self-assembly. Linking the nodes without any control over self-assembly is likely to create a random network of nodes. However, self-assembly can potentially be augmented to allow control over node placement and control.

In this chapter, we study the properties of node networks created as we exercise varying degrees of control over how self-assembled circuit nodes are placed and oriented, and how inter-node links are created during self-assembly. We examine a range of networks, from a mesh (full control) to a random network of nodes (no control). For each network type, we determine the connectivity of the network, and the need for any additional hardware in each node's communication logic to maximize the number of connected nodes. In particular, we examine the trade-off between node complexity and control required during self-assembly to maximize the number of connected nodes in the network. As the level of control decreases, we find that node communication hardware should be augmented to allow sharing of links between several transceivers. This also results in better network connectivity in the presence of defective nodes and links. We evaluate the performance of SOSA on these networks using matrix multiplication as our benchmark and find that system performance is independent of the underlying network, as long as sufficient nodes are available for computation. Finally, we show that the introduction of defects in nodes and links can exacerbate the poor connectivity found in networks with low control during self-assembly.

Recently, researchers have been actively developing nanoelectronic devices and architectures that could potentially replace CMOS in the future. Most designs either assume the ability to create regular structures [31,53], or unstructured interconnect [141] (within a computing block). Since the networks we study are highly dependent on physical node locations, we cannot leverage the large body of work on generating [137] and analyzing internet topologies [91]. Future developments that allow embedding radio transceivers in nodes could potentially allow researchers to leverage this work. We make the following contributions in this chapter.

1. We show that system connectivity improves if we allow links to be shared between more than two transceivers, even for networks where we have low control during self-assembly, and

2. If technology constraints limit the extra functionality that can be implemented in a node, we need to control node placement and orientation during self-assembly to achieve good system connectivity.

The rest of this chapter is organized as follows. We begin our discussion with a brief review of the communication functionality within a node (Section 8.1). Next, we describe how three specific aspects of self-assembly could potentially be controlled (Section 8.2). We then describe our experimental methodology and present an analysis of network characteristics (Section 8.3). We conclude with a summary of the results presented in this chapter (Section 8.4).

8.1 Node Communication Logic

A node's communication logic has four transceivers that allow it to communicate with other nodes over single wire links. In any non-mesh topology, more than two links (and transceivers) can potentially be connected. In this case a transceiver can implement an infinite backoff mechanism that permits only two active transceivers on that link. If a transceiver detects more than one transceiver signal over the link, it shuts down. This can potentially affect network connectivity in cases where the transceiver that shuts down pro-

vides the only access to a region of the network. A more complex solution would be to allow more than two transceivers to share links (i.e., a bus mechanism). This requires additional functionality in each transceiver to allow arbitration for the link, as well as the use of source/destination identifiers per data transfer. We evaluate the potential benefits of one method of link sharing in Section 8.3.5. More details of the node's communication infrastructure can be found in Chapter 6 and Chapter 7. The choice between implementing infinite backoff or a bus mechanism is influenced by technological constraints, device defect rate, and the level of control exercised during self-assembly. As self-assembly matures, it might be easily possible to create larger nodes that incorporate this extra functionality. Next, we explore ways in which node placement, orientation and inter-node link creation can be controlled during self-assembly.

8.2 Controlling Placement, Orientation and Link Creation During Self-Assembly

The topology of the network of nodes depends on the level of control exercised during node self-assembly, and during the creation of inter-node links. As self-assembly technology matures, it might be possible to create three-dimensional topologies as well, but we limit ourselves to the analysis of two-dimensional topologies in this thesis. We explore topologies created as we vary control over three aspects of the manufacturing process:

- placement of nodes (P)
- orientation of nodes (O)
- creation of inter-node links (I)

In each case, we consider two alternatives: 1) full control and 2) no control to limit the parameter space to be explored. This results in eight network types, ranging from a random planar network to a mesh. Table 8-1 lists the networks by the type of control necessary to create them and Figure 8-1 shows examples of these networks. The goal is to identify the level of control necessary to maximize the number of connected nodes. Next, we describe

Name	Control			Example
	Placement (P)	Orientation (O)	Link (I)	
N0	No	No	No	Figure 8-1a
N1	No	No	Yes	Figure 8-1b
N2	No	Yes	No	Figure 8-1c
N3	No	Yes	Yes	Figure 8-1d
N4	Yes	No	No	Figure 8-1e
N5	Yes	No	Yes	Figure 8-1f
N6	Yes	Yes	No	Figure 8-1g
N7	Yes	Yes	Yes	Figure 8-1h

Table 8-1. Classification of network topologies based on control over P, O and I.

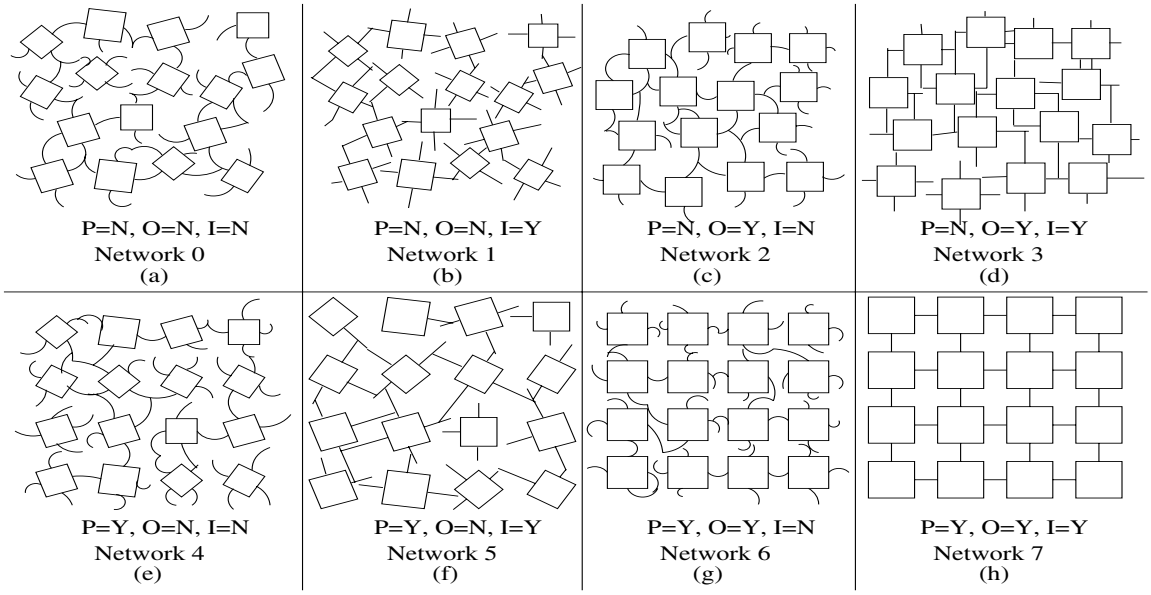


Figure 8-1. Examples of eight networks with varying control over placement (P), orientation (O), and inter-node link creation (I). (a-d): no control over P - nodes can get isolated due to large distances, control over O and I improve connectivity, (e-h): control over P improves connectivity, but can still result in isolated nodes without control over O and I.

how we could potentially control P, O or I, and the implications of that control on the number of connected nodes in the network.

Placement (P). Control over node placement enables uniformly spaced nodes. We expect the uniform spacing to improve network connectivity. Control over node placement can be achieved in two ways: 1) pick and place techniques, and 2) placing DNA tags on the under-

lying substrate to control the locations where nodes self-assemble. While each node is large enough to enable the use of pick and place strategies, they are not practical for systems with a large number of nodes. We can minimize external intervention by placing DNA tags on the substrate such that node growth is initiated at tag locations. The greater the number of tags per node, the greater the chance that nodes form at the right locations. However, increasing the number of tags, increases the effort required in preparing the substrate for self-assembly. Examples of the types of networks created with node placement can be found in Figure 8-1e-h. We expect that the uniform spacing of nodes decreases the chances of nodes being isolated.

Orientation (O). Control over orientation aligns node faces, which can increase the chances of links intersecting (as depicted in Figure 8-1c, Figure 8-1d, Figure 8-1g, and Figure 8-1h), potentially improving network connectivity. The techniques to control node placement could also be extended to control node orientation by increasing the number of tags per node. In addition to using multiple tags on the substrate, nodes could be aligned using an external electric field, or using fluid flow [62].

Inter-node Link Creation (I). Control over inter-node link creation implies control over the shape of links. Without creating a mesh network (and linear links), there is still a chance that more than two transceivers are connected by a link. Linear links cannot loop back on themselves and are useful in improving network connectivity. Researchers have demonstrated the creation of mostly linear wire structures [50,80]. Networks with linear links are shown in Figure 8-1b, Figure 8-1d, Figure 8-1f, and Figure 8-1h.

8.3 Experimental Setup and Evaluation

We begin with a description of our custom network topology generator (Section 8.3.1). Next, we discuss the methodology used to model infinite backoff (Section 8.3.2), and link sharing between transceivers (Section 8.3.3). We then describe our methodology and experiments (Section 8.3.4). We then analyze the characteristics of the networks (Section 8.3.5), and their sensitivity to input parameters (Section 8.3.6). Finally, we

explore the effect of network topologies on system performance (Section 8.3.7) and the effect of defects on system connectivity for different network topologies (Section 8.3.8).

8.3.1 Topology Generator

The topology generator's input parameters include the number of nodes, total area, type of control over placement (P), orientation (O), interconnect (I), and an optional parameter that decays interconnect growth with time (this reflects a potential reduction in the concentration of DNA material available for self-assembly of the links). It also accepts a random seed, which allows the creation of distinct topologies. For networks with no control over node placement (P=N in Figure 8-1), it generates a random location for the node and places it there if all constraints are met (no overlap, minimum distance, within area). The program attempts to place each node a maximum of 10^6 times. For networks with control over node placement (P=Y in Figure 8-1), a simple check of the area and number of nodes allows the program to determine if the nodes fit. If O=N, each node is rotated (about its center) through a random angle before being placed.

After placing all nodes, the simulator models link growth between nodes. For random growth (I=N in Figure 8-1), we use a random number generator and a probability distribution function (PDF) for the angle and distance by which the link grows to perform a directed random walk. When we model linear growth (I=Y in Figure 8-1), we grow the link by a random length (<=50nm). Each link is grown iteratively until one of two conditions is satisfied: 1) it collides with another node or link, or 2) the simulation terminates as a user-defined condition is satisfied. Once growth of all links terminates, the simulator generates a graph corresponding to the node network created by the links, and generates connectivity statistics for the graph.

8.3.2 Modeling Infinite Backoff

The graph generated by the topology generator can include multiple intersecting links, which may link more than two node transceivers. A transceiver can implement infinite backoff by attempting to signal on a link. If the signal attempt collides with another trans-

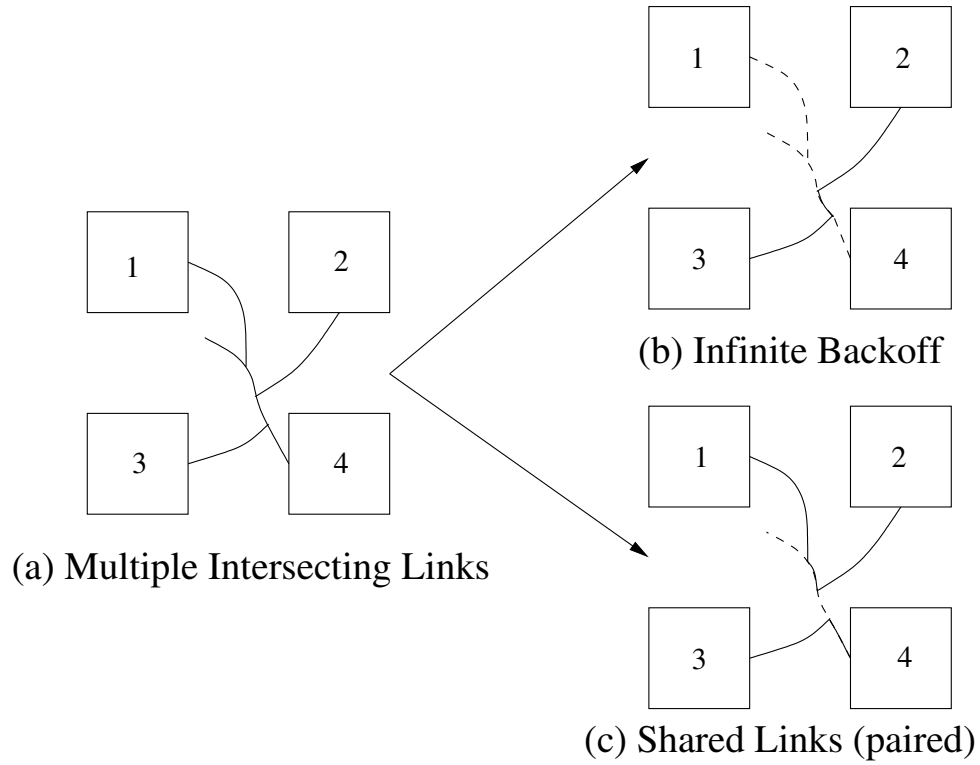


Figure 8-2. Multiple Intersecting links (a) Unmodified, (b) with infinite backoff, and (c) paired links on a bus

ceiver's attempt, the transceiver retries after a random interval. If a transceiver receives two successful signals on the link, it shuts down. To model infinite backoff, we identify links with more than two transceivers, randomly pick two transceivers to be active, and disconnect the rest. There are multiple ways of picking a pair of transceivers, and we generate multiple networks by randomly picking different pairs of transceivers. Figure 8-2b shows four nodes with intersecting links after two transceivers apply infinite backoff and shut down (Node 1 and Node 4). This leaves the links from Node 2 and Node 3 connected.

8.3.3 Modeling Links as Buses

We model one possible implementation of shared links, where the N transceivers connected by a single link are divided into pairs that communicate with each other. If N is odd, one transceiver is not used. Figure 8-2c shows four nodes that can share links. Node 1 and Node 2 form one pair, and Node 3 and Node 4 form a second pair. While each pair of nodes can

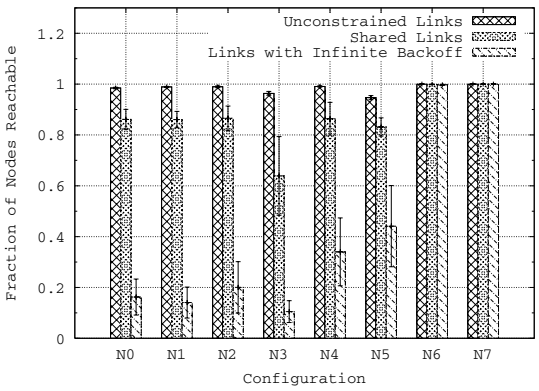
be viewed as being connected on distinct links, node communication hardware must deal with arbitration for the link.

8.3.4 Methodology and Experiments

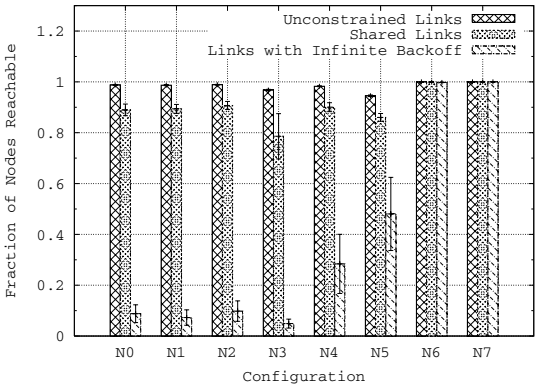
The goal of the experimental evaluation is to analyze the characteristics of the different network topologies. We use three metrics to assess network connectivity: 1) fraction of reachable nodes, 2) the number of transceivers connected per link, and 3) the number of connected links per node. The higher the fraction of connected nodes, the better the network connectivity. The number of transceivers connected per link captures the instances where multiple links intersect. Such links have more than two transceivers connected per link. The number of connected links per node is a measure of how well a node is connected to the rest of the network (higher is better). The highest value is 4, but due to boundary effects, the value is limited to about 3.9 for a mesh. We also measure the effect of defective nodes and links on network connectivity, and measure the impact of network topologies on system performance. Finally, we measure the sensitivity of network characteristics to various inter-node link growth decay rates. For each of the eight network types, we generate 100 topologies for different network sizes (1,296 nodes, 4,900 nodes, 10,000 nodes, 21,025 nodes). For each network size, we also vary the inter-node link growth decay rate (0%-2%). For each topology, we generate ten networks each for shared links, and links that model infinite backoff. We report the average value over all runs for each metric.

8.3.5 Network Connectivity

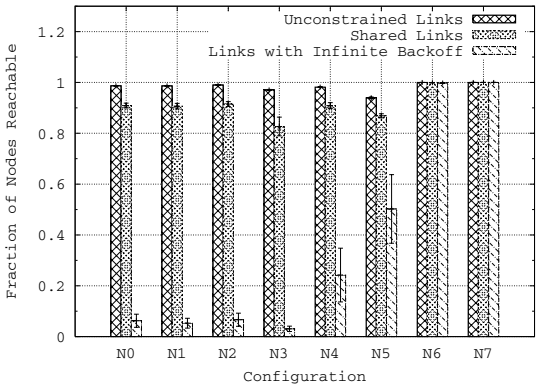
In Figure 8-3, we plot the size of the largest connected group of nodes as a fraction of total nodes for the four network sizes (1,296 nodes, 4,900 nodes, 10,000 nodes and 21,025 nodes). For each network type, we plot three bars, the first representing the unconstrained network, the second corresponding to links modelled as buses, and the third with transceivers implementing infinite backoff. From Figure 8-3, we see that if connectivity is unconstrained all networks are able to connect in excess of 95% of the nodes. However when modeling realistic hardware, the fraction of reachable nodes decreases. The decrease is not



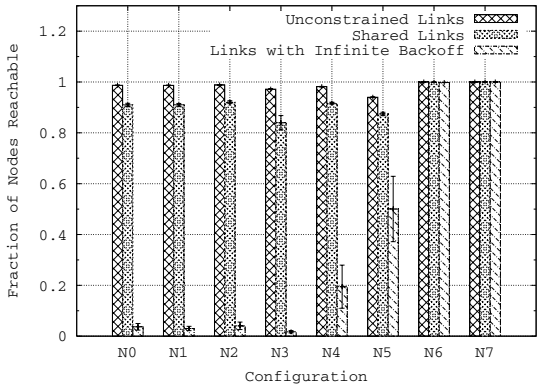
(a): 1,296 Nodes



(b): 4,900Nodes



(c): 10,000 Nodes

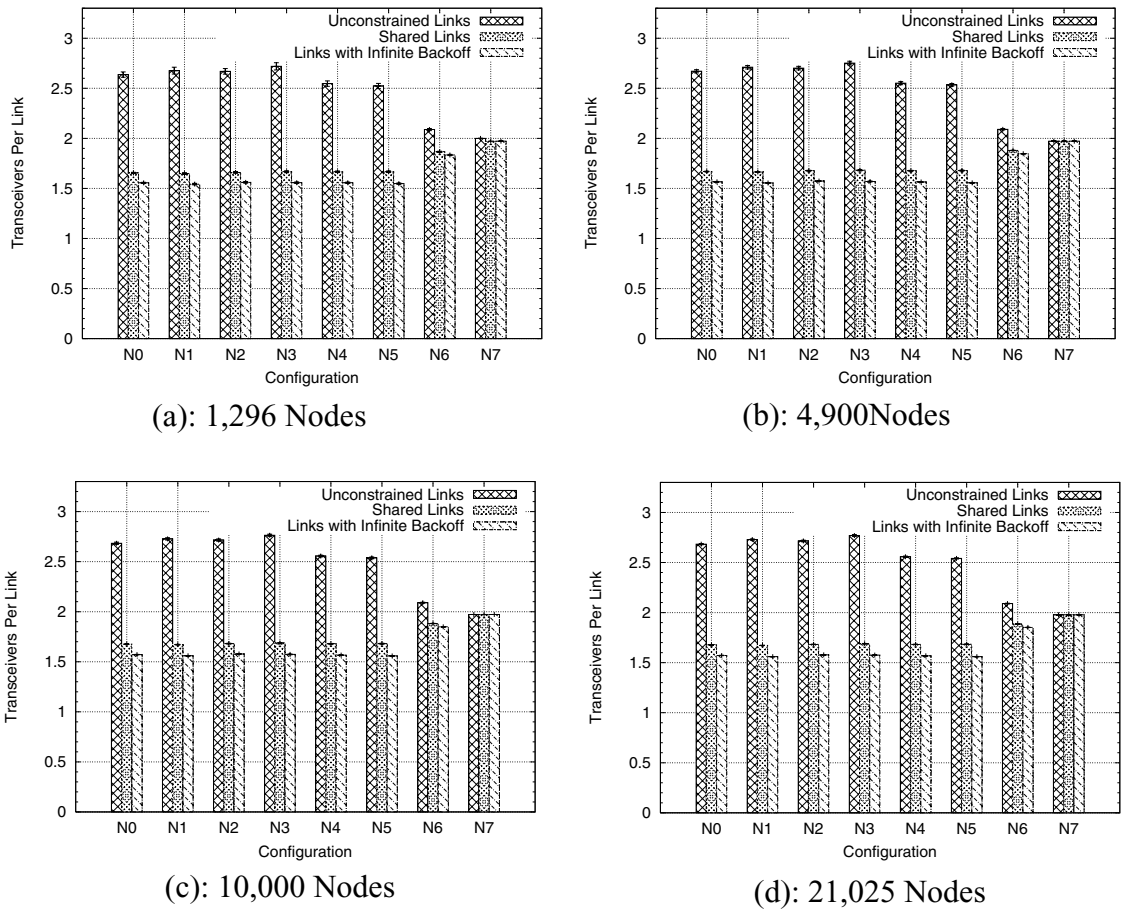


(d): 21,025 Nodes

Figure 8-3. Fraction of Reachable Nodes

very large when links are treated as shared media. However, if we model infinite backoff on links, for networks without control over placement and orientation, the fraction of reachable nodes is less than 50%. The results are consistent across network sizes.

In Figure 8-4 we plot the average number of transceivers connected per link. For a fully connected network of nodes there would be two transceivers per link and 1.97 for a mesh since the boundary transceivers are disconnected. For the unconstrained networks the value is over 2 indicating that multiple transceivers share links. The value drops to about 1.7 for shared links, and about 1.55 for links with infinite backoff. This implies that only 55% links are connected to a second transceiver if we implement infinite backoff, which explains the poor network connectivity. The poor network connectivity is also apparent if we examine the number of transceivers per node that are connected to other transceivers (even if they



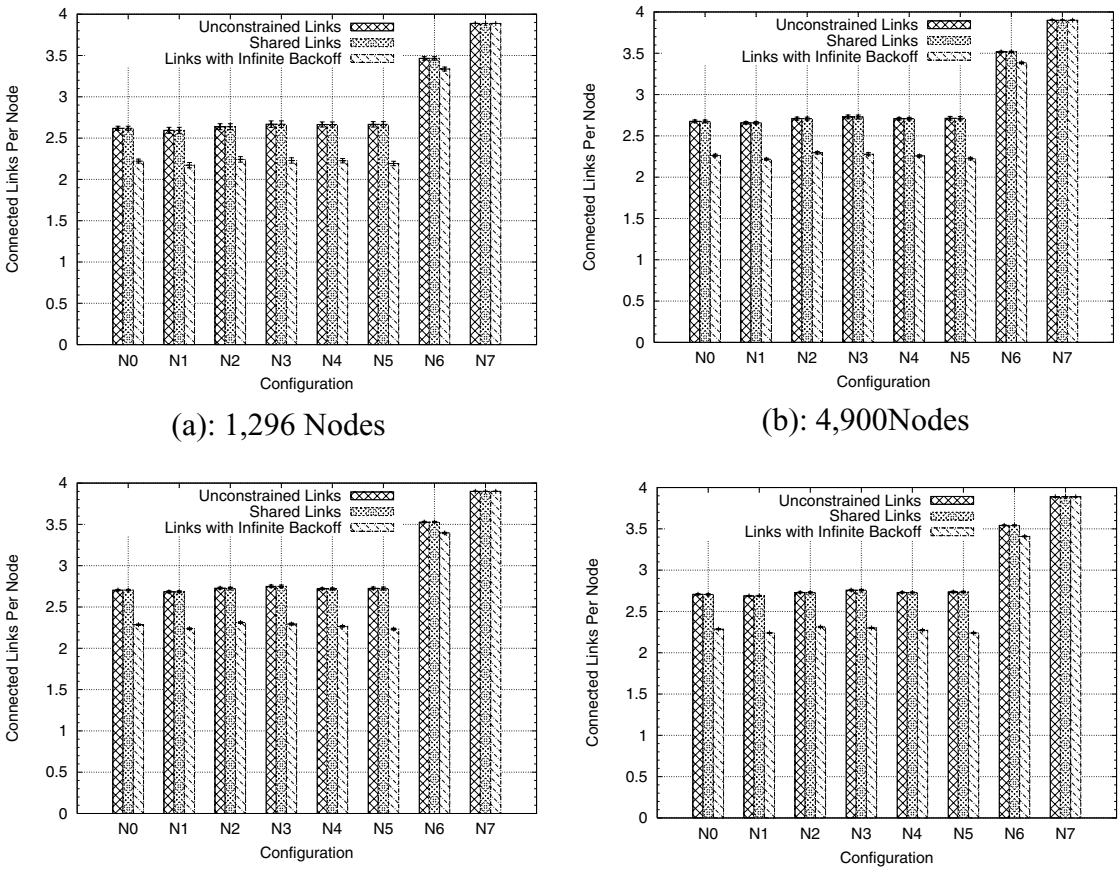
(a): 1,296 Nodes (b): 4,900Nodes

(c): 10,000 Nodes (d): 21,025 Nodes

Figure 8-4. Transceivers Per Link

loop back to the same node). We plot the average number of active links per node in Figure 8-5. For a mesh, the ideal value is 3.89 (due to disconnected boundary links), but we see that except for networks where we exercise control over placement and orientation, the value is about 2.7. If we implement infinite backoff, the average number of active links per node drops to 2.3 since some nodes are forced to disconnect from links.

This highlights the trade-off between simple nodes and the degree of control required during self-assembly to achieve good network connectivity. Simpler nodes require regular topologies to achieve good connectivity. If nodes can implement mechanisms to allow more than two transceivers to share a single link, the system can be well connected even if there is no control over the manufacturing process. Next, we examine the sensitivity of network connectivity as we vary the inter-node link growth decay rate.



(a): 1,296 Nodes

(b): 4,900Nodes

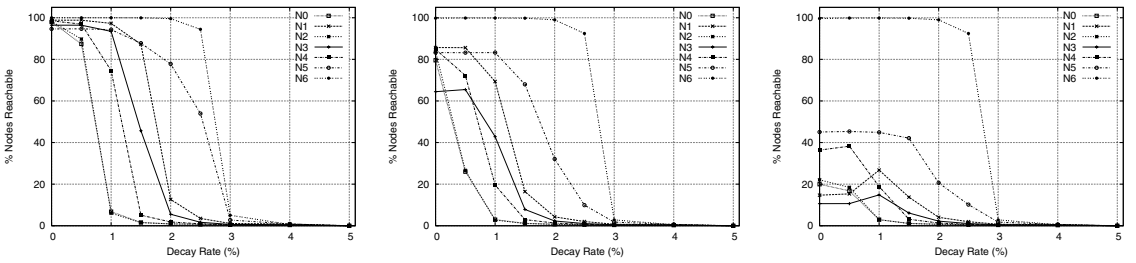
(c): 10,000 Nodes

(d): 21,025 Nodes

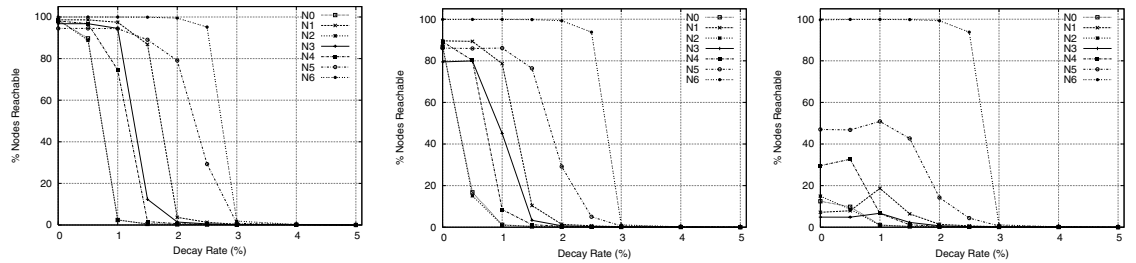
Figure 8-5. Average Active Links Per Node

8.3.6 Effect of Decaying Growth Rate

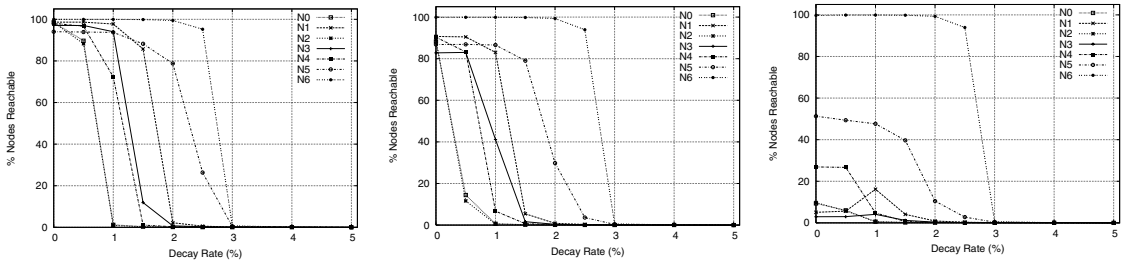
In this subsection, we evaluate the sensitivity of our results to the decay factor that accounts for the slowing of the rate of growth of inter-node links. The decay factor represents the slow reduction in the concentration of the raw chemical components as self-assembly proceeds. We model this decay by iteratively decreasing the length that a link can grow in each iteration. In Figure 8-6, we plot the average number of reachable nodes for four network sizes as we vary the decay rate from 0% to 5%. We see that the effect of the decay rate is qualitatively similar for different network sizes. The fraction of reachable nodes drops to nearly zero beyond a decay rate of 3%. This is because links are unable to grow far enough to actually reach other nodes. Since a mesh requires full control over self-



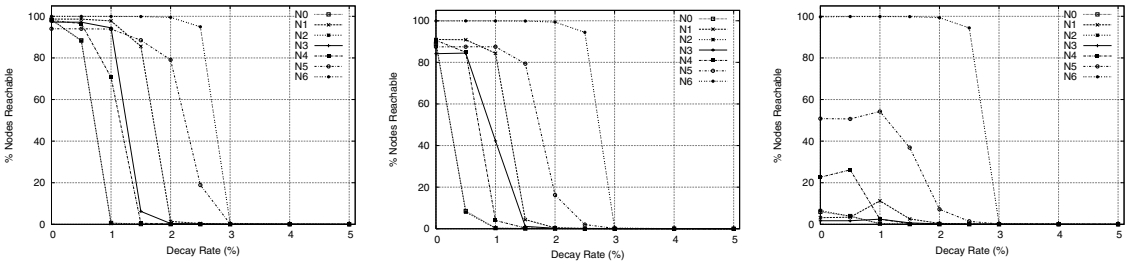
Unconstrained Links | Shared Links (a) 1296 Nodes | Links with Infinite Backoff



Unconstrained Links | Shared Links (b) 4900 Nodes | Links with Infinite Backoff



Unconstrained Links | Shared Links (c) 10000 Nodes | Links with Infinite Backoff



Unconstrained Links | Shared Links (d) 21,025 Nodes | Links with Infinite Backoff

Figure 8-6. Sensitivity to Decaying Growth Rate

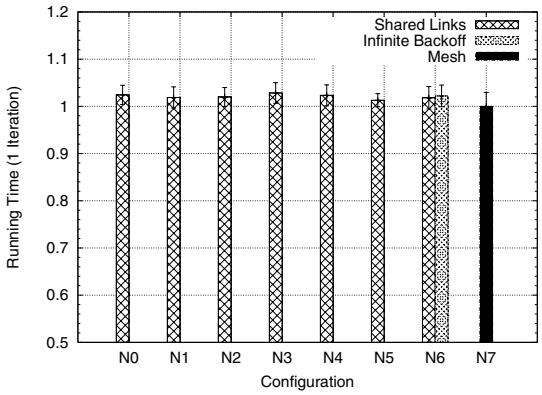
assembly, we do not plot the results for a mesh (no drop in connectivity up to a decay rate of 5%). Of the other networks, N6 (P=Y, O=Y, I=N) can tolerate a decay rate up to 2.5%, and N5 (P=Y, O=N, I=Y) can tolerate a decay rate up to 2%. The remaining configurations are more sensitive to the decay rate, primarily because they rely on longer link growth to achieve node connectivity. Networks N0 (P,O,I=N) and N2 (P=N, O=Y, I=N) are unable to tolerate decay rates greater than 0.5%. From these results, we can conclude that it will be important to maintain a sufficient concentration of raw materials during self-assembly to maximize system connectivity.

8.3.7 System Performance

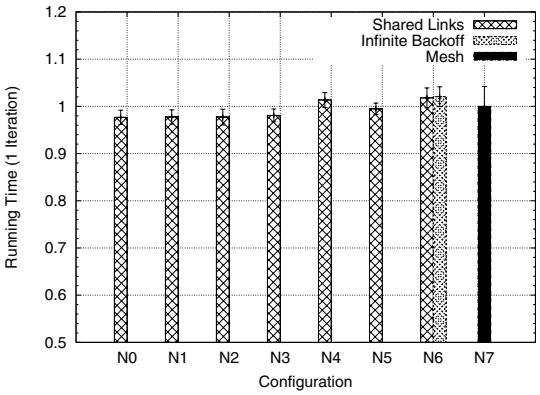
In an ideal system, performance would be independent of network topology. To quantify the effect of topology on system performance, we measure the running time of an application (matrix multiplication) on different networks using a simulator for our data parallel architecture. We measure program run time for networks with at most two transceivers sharing links (infinite backoff), or pairs of transceivers sharing links (links as buses). We simulate matrix multiplication for three matrix sizes - 8x8, 16x16 and 32x32. In Figure 8-7, we plot the running time of the three matrix sizes, normalized to the running time on a mesh. We find that as long as enough PEs can be configured in the network there is less than 5% variation in program running time. However, we also see that if we model infinite backoff, the system cannot configure sufficient PEs in networks without full control over placement and orientation.

8.3.8 Effect of Defects

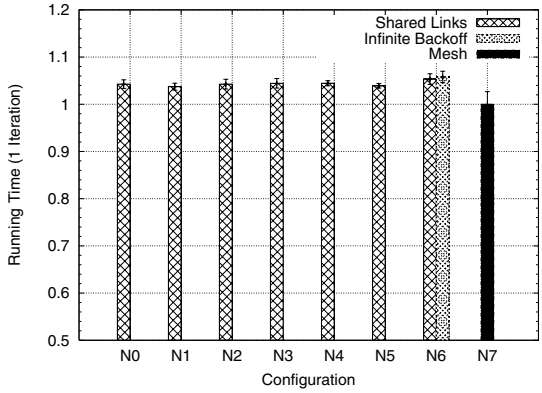
To study the effect of defects on network connectivity, we apply a node failure model [107] with a range of device reliabilities. Table 8-2 lists the percentage of nodes that are reachable in a network of 21,025 nodes for shared links. The numbers in parentheses are the percentage of reachable nodes when modeling infinite backoff. We do not show this number if it is less than 10%. We see that the percentage of reachable nodes drops rapidly as device reliability or control over self-assembly decreases. System connectivity decreases



(a) 1296 Nodes - 8x8 Matrix Multiply



(b) 4900 Nodes - 16x16 Matrix Multiply



(c) 21025 Nodes - 32x32 Matrix Multiply

Figure 8-7. SOSA performance sensitivity to different networks for 8x8, 16x16 and 32x32 matrix multiplication, normalized to performance of a mesh

% Device Reliability	Configuration							
	N0	**N1**	**N2**	**N3**	**N4**	**N5**	**N6**	**N7**
99.990	3.8	4.9	3.9	2.5	3.4	6.6	86 (84)	90 (90)
99.993	18	26	19	7.2	18	38	91 (91)	93 (93)
99.996	73	73	73	31	72	72	95 (95)	96 (96)
99.999	88	88	89	78	89 (12)	85 (36)	99 (99)	99 (99)
100.00	91	91	92	84	92 (20)	87 (50)	100 (100)	100 (100)

Table 8-2. Percentage of nodes reachable with varying device reliabilities when links are shared between multiple transceivers. The figure in parentheses is for nodes implementing infinite backoff.

since some regions get disconnected due to the loss of critical nodes/links to defects. This is reflected by a drop in the number of transceivers per link (between 22-50% drop) as the device reliability decreases from 100% to 99.99%.

The results highlight the benefit of link sharing over infinite backoff. Link sharing allows a larger number of nodes to remain connected as device reliability decreases. This is true even for configurations with low control during self-assembly (N0-N3). We can draw two conclusions from these results: 1) if device reliability is lower than 99.999%, we either need to control placement and orientation during self-assembly, or we need to implement link sharing, to maintain network connectivity, and 2) controlling placement and orientation has a greater effect on network connectivity than link sharing.

8.4 Conclusions

In this chapter, we evaluate the characteristics of a class of network topologies that could be created by exercising varying degrees of control during the self-assembly of simple nodes. The evaluation highlights the trade-off between node complexity and the amount of control required during self-assembly to maximize the number of connected nodes in the network. We also see that so long as the network has enough nodes, system performance is not affected by the type of configuration created by self-assembly. Finally, we see that introducing defects has a greater effect on networks with a lower degree of control during self-assembly. However, this can be mitigated to some extent by allowing more than two transceivers to share a link.

9 Related Work

In this chapter, we describe prior architecture research related to the work presented in this thesis. We divide our discussion into two categories, CMOS-based architectures (Section 9.1) and architectures based on technologies other than CMOS (Section 9.2).

9.1 CMOS-based Architectures

There has been extensive research on designing and building vector [23,45] and SIMD machines [73,83,13,143]. The CM-2 [143] data parallel architecture uses up to 65536 bit-serial processing elements to execute parallel programs. Workloads are split up into their parallel and sequential components. The processor array executes the parallel component, while an external unit handles the sequential parts of the code. Each processor has 64 kilo-bits of bit addressable memory and connects to its four neighbors. All processors execute the same instruction at a time. The result of the operation can either be stored or discarded. The recent "Cell" processor [61] has eight SIMD cores that can be programmed independently, unlike the PEs in SOSA. The primary differences between SOSA and past work is our focus on overcoming the challenges imposed by the fabrication technology and the need to tolerate defects.

The Teramac [27,59] architecture developed at HP Labs was one of the first attempts at building a high performance defect tolerant architecture. The Teramac achieves defect tolerance by creating a defect map for the system and then configuring the rest of the system to avoid defective regions. The Teramac uses an extremely long (~300 Mbit) configuration instruction to setup every FPGA in the system. This method would not scale if the number of devices that needed to be configured was very high. NANA and SOSA use a distributed configuration algorithm to provide logical structure to the node network without requiring

an external defect map. This is critical since we have little information about the topology of the network.

Researchers have proposed various voting and redundancy schemes [56] to deal with defects, including triple modular redundancy (TMR) [90], recursive TMR [140], N-modular redundancy [145], NAND multiplexing and hot/cold sparing [31] (particularly in the context of molecular electronic systems). The defect tolerance scheme presented in this paper does not rely on redundant computation but isolates defectives regions in the system.

The accumulator based design and ISA in NANA are similar to the design used by Kim et al [76]. Their aim was to exploit the fact that instructions can be grouped into independent sets of dependant instructions. Dependant instructions are linked through the use of an accumulator. The differences between the two designs lie in the bit-serial nature of our design, and the use of only a single accumulator, embedded in the instruction stream.

Both NANA and SOSA could potential leverage prior work to exploit thread level parallelism. This includes work done on multiscalar processors [131], slipstream [136], and the TRIPS architecture [124]. Multiscalar processors break programs into speculative threads that help speed up execution. Slipstream uses a run-ahead thread either for fault tolerance, or to speed up execution.

Several research projects have looked at building computing systems with a subset of the goals for NANA and SOSA, including self-organization [5,19,125], routing and resiliency in the face of defects [1,71,65] and the ability to compose complex computational units from simpler blocks [92], but we face added challenges because of the extremely limited computational capabilities available in nodes.

9.2 Architectures based on Emerging Technologies

The most related work to this thesis, is Dwyer's proposal to use a DNA guided self-assembly technique to build a massively parallel computer [35,38,42]. Two machines were proposed as part of that work, the Decoupled Array Multi Processor (DAMP), and the Oracle. The Oracle solves instances of NP-complete problems by storing all solutions in the assem-

bly process, and then performing a simple search for the solution. The DAMP attacks "embarrassingly parallel" problems by dedicating a vast number of bit serial processing elements to solving a single problem. The processing elements of the DAMP can be controlled only through a shared controller and cannot communicate with each other. In contrast, the nodes in NANA and SOSA can communicate with each other which enables both architectures to execute more complex workloads. In addition, the DAMP does not explicitly deal with defect or fault tolerance.

The Nanofabrics [53] work from CMU is similar to the Teramac in its use of reconfigurable devices as well as its approach to defect tolerance. Resonant tunneling diodes (two terminal devices) are configured into supernodes of appropriate functionality after a test phase maps out defective components. The logic blocks are similar to FPGAs and can implement 3-bit boolean functions and their complements. The devices use the Split-phase Abstract Machine (SAM) architecture, which implements active messages in the reconfigurable hardware. Nanofabrics requires the device to be reconfigured per application, while both NANA and SOSA can potentially be used without reconfiguration (However, SOSA might require reconfiguration to adjust cell sizes, not device function). Nanofabrics also relies on a very regular arrangement of devices to achieve defect tolerance. It is unclear whether devices can be manufactured with the precision required to make precise mesh structures.

DeHon presents an architecture that exploits three terminal devices (FETs) by self-assembling arrays of nanowires and FETs [31]. Sparing and remapping are used to provide defect tolerance. The architecture provides a mechanism to interface the nano-scale components to micro-scale components. The design assumes the ability to self-assemble nanowires and devices into arbitrary patterns, without providing details of how this could be done. Snider et al. [130] use CMOS like logic to build defect tolerant nanoscale fabrics out of crossbars of three terminal devices. The system can configure a regular crossbar of devices to implement a microprocessor, but assumes the ability to locate defects before mapping a circuit onto the crossbar.

Several other researchers propose various forms of array-based nanoarchitectures. Ancona proposes a systolic array architecture for single electron transistors [8], but it requires precise control over fabrication. Beckett et al propose a nanoarchitecture based on integrated processing and memory nodes with a local interconnection [12]. At the nanoscale they note that computation is cheap, while long distance communication is very expensive. SOSA makes use of the same fact, and attempts to minimize communication between processing elements. Fountain introduces the propagating instruction processor (PIP) that is a pipelined SIMD machine [47]. The PIP also uses bit-serial processing elements, where the data is held at each element, and instructions flow through the processor. SOSA is similar to the PIP, but has a substantially less structured interconnection network.

Han exploits NAND multiplexing and reconfiguration to support a defect tolerant architecture [55]. More generally, Nikolic et al. argue that reconfiguration is the best approach for handling fabrication defects, but that other redundancy techniques are necessary to handle transient faults [101]. They acknowledge that one of the keys to using reconfiguration is that defect isolation must be easy.

Bit serial architectures have been commonly used in digital signal processing (DSP) applications [11,72], and are also being used with new technologies like single flux quantum (SFQ) [153] to demonstrate their feasibility. Both designs presented in this thesis use a bit-serial approach for reasons similar to the DAMP [35], PIP [47] and CM-2 [143], and because of the limited size of a self-assembled circuit. The bit-serial approach helps in exploiting bit-level parallelism.

Current research is exploring the impact on architecture of emerging nanoelectronic technologies. This includes molecular electronics [43,141,51,142], quantum dots [99], cellular automata [113] and quantum computing [103,134]. Neural networks [102,117,128] have been used by researchers due to their inherent fault tolerance. However, they have limited applications due to the complexity of the learning algorithms required to train them. Hardware implementations of neural networks [69,127] also exist, but are limited by the complexity of the design. Researchers have tried using cellular automata [99,113] to build systems out of a large number of simple components. Peper et al [113] use asynchronous

arrays of delay insensitive circuits to build a specific type of cellular automata known as a self-timed cellular automaton. Cells are connected to their immediate neighbors and do not require global knowledge to process data. Each cell has interaction rules that control state transitions. State transitions occur when neighboring cell states and the state of the current cell match a predefined pattern. Boolean gates cannot be used as universal sets for self-timed logic, so a different set of logic primitives is designed and used. Cellular automata suffer from problems of limited fan-out (maximum of 3). Also, because of the different logic primitives used, there is no longer a direct correspondence between conventional programs and the underlying hardware primitives. This is likely to limit the use of asynchronous cellular automata to a narrow range of applications. Researchers have also tried combining neural networks with cellular automata, to produce cellular neural networks, where the network can learn, but requires limited connectivity. However, even this limited connectivity is hard to achieve, which limits the usefulness of this approach. Other researchers have focused on the challenges in designing computing systems using emerging technologies like molecular electronics and scaled CMOS, and suggest ways of overcoming these challenges [46,100,133].

10 Summary and Conclusions

Manufacturing defects, power density, process variability, transient faults, quantum effects, bulk semiconductor material limits, rising verification costs and multibillion dollar fabrication facilities are some of the challenges facing the continued scaling of CMOS. While architectural modifications (e.g., multicore) can provide some short-term relief, the semiconductor industry recognizes the need to explore long term alternatives to CMOS devices and fabrication techniques. Consequently, researchers have focused on identifying device and manufacturing technologies that could replace CMOS in the future. While these technologies are in their infancy, by studying their potential uses for building computing systems, architects can gain a deeper understanding of their limitations and opportunities while providing important feedback to the scientists developing the new technologies.

In this thesis, we study the impact of one class of emerging technologies, DNA-based self-assembly of nanoelectronic components, on architecture design. These technologies are characterized by their ability to manufacture a large number of simple devices in parallel, but suffer from increased defect rates and limited control over fabrication. A promising instance of such technologies is DNA-based self-assembly of nanoscale components that has the potential to achieve tera- to peta-scale integration. We start with the development of a circuit architecture for this technology, we design and evaluate an active-network based architecture (NANA) that incorporates an execution and memory system. Using the lessons learned through the design of NANA, we develop a data parallel architecture (SOSA) that makes efficient use of a large number of nodes to form a high-performance computing system.

The first contribution of this thesis is the design of a circuit architecture [109] that can be used to build small nodes with the ability to compute and communicate with up to four neighbors. We propose the use of aperiodic patterns to create circuits on a DNA-lattice.

These circuits are then interconnected to create a large random network of simple nodes that can then be organized to create system architectures.

The second contribution of this thesis is the adaptation of an existing routing algorithm to isolate defective nodes and provide logical structure in a random network of nodes. Self-assembly provides a lower degree of control than conventional manufacturing processes like lithography, and is expected to increase the fraction of defective components in a system. We adapt the reverse path forwarding algorithm [110] to isolate defective nodes and organize nodes in a broadcast tree. We can then use simple algorithms like depth-first traversal to organize nodes into more complex entities.

The third contribution of this thesis is the design of the nanoscale active network architecture (NANA) [111]. This architecture represents a first cut at the development of a high-performance architecture built using self-assembled computing blocks. NANA uses the logical structure provided by the RPF algorithm to organize a random network of nodes into disjoint execution and memory networks. The architecture exploits bit-level parallelism in the data stream to improve performance and can execute a variety of general purpose workloads.

The fourth contribution of this thesis is the design of a data parallel architecture (SOSA) [112] that incorporates lessons learned during the design and evaluation of NANA to build a high-performance computing system. SOSA organizes a random network of homogenous nodes to create SIMD style processing elements connected in a logical ring. By exploiting the large parallel computing capability of the node network, SOSA is able to match the performance of existing architectures while operating at a lower speed and consuming lower power. While this architecture has some limitations it is a positive step towards realizing defect tolerant computing systems built using emerging technologies that may provide inexpensive terascale integration. Future designs that use emerging technologies can benefit from the lessons learned through the design and evaluation of SOSA.

As advances are made in the development of emerging device and manufacturing technologies, some of the limitations assumed in this thesis may no longer hold. In the early stages of development of these technologies, it might be possible to create hybrid devices

that mix CMOS and self-assembled circuits. One possible use would be to build special-purpose high-performance processor cores that are added to conventional processors to improve system performance on a specific class of applications. However, the overall goal would be to develop defect-tolerant high-performance architectures that make efficient use of the available compute resources.

Appendix A: NANA Instruction Set

NANA supports a small general purpose instruction set, listed in Table A-1. Before we describe the operation of the instructions, establish some basic terminology. The operand stream is assumed to consist of a series of operands. The first operand (accumulator) is denoted ‘A’, the second operand is denoted as ‘B’, the third operand is ‘C’, and the last operand is denoted as ‘Z’. Bit-slice separators are denoted by [-]. We explain the operation of all instructions in terms of their effect on the operand stream.

Instruction Type	**Instructions**
Arithmetic	ADD, INC, SUB, DEC, SHL, SHR
Comparison	COMPEQ, COMPGT, COMPLT, SETEQ, SETGT, SETLT, SETZ
Operand Stream Control	LDCONST0, LDCONST1, CPACC, MOV, DELOP, OPFLUSH, SWAP
Logical	AND, NAND, NOR, NOT, OR, XOR, XNOR, NOP
Load	LD [Mem], LDI [Mem]
Store	ST [Mem], STI [Mem]
Conditional Store	CST [Mem], CST_RST [Mem], CRST [Mem], CSTI [Mem], CSTI_RST [Mem], CRSTI [Mem]
Unconditional Control Transfer	JMP [Mem], CALL [Mem],JMPI [Mem],CALLI [Mem]
Conditional Control Transfer	CALLNZ [Mem], CALLZ [Mem], CALLNZI [Mem], CALLZI [Mem]

Table A-1. NANA Instruction Set

A.1 Arithmetic Instructions

Opcode	Length	Input	Output
ADD	8 bits	A,B	A+B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A+B].[C]...[Z]		

Opcode	Length	Input	Output
SUB	8 bits	A,B	A-B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A-B].[C]...[Z]		

Opcode	Length	Input	Output
INC	8 bits	A	A+1
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A+1].[B].[C]...[Z]		

Opcode	Length	Input	Output
ADD	8 bits	A	A-1
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A-1].[B].[C]...[Z]		

Opcode	Length	Input	Output
SHL	8 bits	A	A<<1
Operand Stream			
Before	[A0].[B0][C0]...[Z0][-][A1][B1][C1]..[Z1][-]...		
After	[0].[B0][C0]...[Z0][-][A0][B1][C1]...[Z1][-][A1][B2][C2]...		

A.2 Logical Instructions

Opcode	Length	Input	Output
SHR	8 bits	A	A>>1
Operand Stream			
Before	[A0].[B0][C0]...[Z0][-][A1][B1][C1]..[Z1][-]...		
After	[B0][C0]...[Z0][A1][-][B1][C1]...[Z1][A2][-][B2][C2]...[Z2][A3]		

Opcode	Length	Input	Output
AND	8 bits	A,B	A.B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A.B].[C]...[Z]		

Opcode	Length	Input	Output
OR	8 bits	A,B	A U B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A U B].[C]...[Z]		

Opcode	Length	Input	Output
XOR	8 bits	A,B	A XOR B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A xor B].[C]...[Z]		

Opcode	Length	Input	Output
NOT	8 bits	A	A
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[$\overline{A}$][B].[C]...[Z]		

Opcode	Length	Input	Output
XNOR	8 bits	A,B	A xnor B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A xnor B].[C]...[Z]		

Opcode	Length	Input	Output
NOR	8 bits	A,B	A U B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[$\overline{\text{A U B}}$].[C]...[Z]		

Opcode	Length	Input	Output
NAND	8 bits	A,B	A.B
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[$\overline{\text{A.B}}$].[C]...[Z]		

A.3 Operand Stream Control Instructions

Opcode	Length	Input	Output
SWAP	8 bits	A,B	B,A
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[B].[A].[C]...[Z]		

Opcode	Length	Input	Output
CPACC	8 bits	A	A,Z=A
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A].[B].[C]...[Z][A]		

Opcode	Length	Input	Output
MOV	8 bits	A	See Below
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[B].[C]...[Z][A]		

Opcode	Length	Input	Output
LDCONST0	8 bits	-	Z=0
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[A].[B].[C]...[Z][0]		

Opcode	Length	Input	Output
LDCONST1	8 bits	-	Z=1
Operand Stream			
Before	[A].[B].[C]....[Z]		
After	[A].[B].[C]...[Z][1]		

Opcode	Length	Input	Output
DELOP	8 bits	A	-
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[B].[C]...[Z]		

Opcode	Length	Input	Output
OPFLUSH	8 bits	-	-
Operand Stream			
Before	[A].[B].[C]...[Z]		
After			

Opcode	Length	Input	Output
NOP	8 bits	-	-
Operand Stream (No change)			
Before	[A].[B].[C]...[Z]		
After	[A].[B].[C]...[Z]		

A.4 Comparison Instructions

The comparison instructions modify a flag that is stored in the tail of the execution packet. The flag is modified after the node has seen all operands and performed the comparison operation. There are two types of comparison operations that differ in their effect on the second operand in the comparison. The first type consumes the second operand of the comparison and can be identified by the prefix “COMP”. The second type does not affect the second operand and can be identified by the prefix “SET”. Neither type affects the first operand. In the rest of this appendix, we use [Flag] to denote the input value of the flag and [Updated Flag] to denote the new value after the comparison instruction has been performed.

Opcode	Length	Input	Output
COMPEQ	8 bits	A,B	A=B =>F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
COMPLT	8 bits	A,B	A<B=>F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
COMPGT	8 bits	A,B	A>B=>F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
SETZ	8 bits	A	A=0 => F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[B].[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
SETEQ	8 bits	A,B	A=B=>F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[B].[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
SETLT	8 bits	A,B	A<B=>F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[B].[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
SETGT	8 bits	A,B	A>B=>F=T
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[B].[C]...[Z][Updated Flag]		

A.5 Memory Instructions

The Memory instructions can be divided into four parts: 1) load instructions, 2) store instructions 3) conditional store instructions and 4) load PC instructions. We introduce additional notation to denote contents of a memory location. [Mem] denotes the contents of memory location with the address specified by the instruction. [[Mem]] denotes the contents of the memory location with the address [Mem]. For example, given an 8-bit address 0xAB containing 0x00DE, [Mem] evaluates to 0x00DE, while [[Mem]] evaluates to the contents of memory location 0x00DE.

A.5.1 Load Instructions

Opcode	Length	Input	Output
LD	16 bits	8-bit address	Z=[Mem]
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[A].[B].[C]...[Z][Mem]		

Opcode	Length	Input	Output
LDI	16 bits	8-bit address	Z=[[Mem]]
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[A].[B].[C]...[Z][[Mem]]		

A.5.2 Store Instructions

Opcode	Length	Input	Output
ST	16 bits	8-bit address, A	[Mem]=A
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[B].[C]...[Z]		

Opcode	Length	Input	Output
STI	16 bits	8-bit address,A	[[Mem]]=A
Operand Stream			
Before	[A].[B].[C]...[Z]		
After	[B].[C]...[Z][Mem]		

A.5.3 Conditional Store Instructions

Conditional store instructions do not remove A from the operand stream since there is no guarantee that the store has occurred. To remove A, an explicit 'DELOP' instruction must be placed after the conditional store to guarantee removal of the operand.

Opcode	Length	Input	Output
CST	16 bits	8-bit address,A	if [Flag]=True then [Mem]=A, else NOP
Operand Stream (No Change)			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Flag]		

Opcode	Length	Input	Output
CST_RST	16 bits	8-bit address,A	if [Flag]=True then [Mem]=A and [Flag]=False, else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
CRST	16 bits	8-bit address,A	if [Flag]=False then [Mem]=A, else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Flag]		

Opcode	Length	Input	Output
CSTI	16 bits	8-bit address,A	if [Flag]=True then [[Mem]]=A, else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Flag]		

Opcode	Length	Input	Output
CSTI_RST	16 bits	8-bit address,A	if [Flag]=True then [[Mem]]=A and [Flag]=False, else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Updated Flag]		

Opcode	Length	Input	Output
CRSTI	16 bits	8-bit address,A	if [Flag]=False then [[Mem]]=A, else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Flag]		

A.5.4 Control Transfer Instructions

There are two types of control transfer instructions: 1) those that create a new execution packet (‘CALL’ variants) and 2) those that do not result in the creation of a new packet (‘JMP’ variants). Each instruction operates in a similar manner. The data returned by the memory system is split into two parts: the first byte forms part of the execution packet, while the second part is used as an address for the next fragment of the execution packet. The instruction continues executing until the address in the second part is zero, at which point the instruction terminates. This behavior is shared by all the control transfer instructions.

Opcode	Length	Input	Output
CALL	16 bits	8-bit address	New packet starting from [Mem]
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	Empty		

Opcode	Length	Input	Output
CALLNZ	16 bits	8-bit address,A	If [Flag]=True, then fetch new packet starting from [Mem], else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	Empty		

Opcode	Length	Input	Output
CALLZ	16 bits	8-bit address,A	If [Flag]=False, then fetch new packet starting from [Mem], else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	Empty		

Opcode	Length	Input	Output
CALLI	16 bits	8-bit address,A	Fetch new packet starting from [[Mem]]
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	Empty		

Opcode	Length	Input	Output
CALLNZI	16 bits	8-bit address,A	If [Flag]=True, then fetch new packet starting from [[Mem]], else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	Empty		

Opcode	Length	Input	Output
CALLZI	16 bits	8-bit address,A	If [Flag]=False, then fetch new packet starting from [[Mem]], else NOP
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	Empty		

Opcode	Length	Input	Output
JMP	16 bits	8-bit address	Fetch instructions into current packet starting from [Mem]
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Flag]		

Opcode	Length	Input	Output
JMPI	16 bits	8-bit address	Fetch instructions into current packet starting from [[Mem]]
Operand Stream			
Before	[A].[B].[C]...[Z][Flag]		
After	[A].[B].[C]...[Z][Flag]		

Appendix B: SOSA Instruction Set

SOSA supports a small microcoded instruction set, listed in Table B-1. Instructions can be divided into six classes: a) arithmetic, b) logical, c) bit shift, d) predicate modifying, e) comparison, and f) miscellaneous and pseudo instructions,. Since the instructions are micro-coded, it is possible to create instructions that overlap two categories. With the exception of instructions that cause inter PE communication, all instructions can be predicated. All instructions are 16 bits long and the definitions of individual bits in the instruc-

Instruction	**Type**	**Instruction**	**Type**
ADD	Arithmetic	AND	Logical
BITSHIFTLMPE	Bit Shift/Inter PE	BITSHIFTMLPE	Bit Shift/Inter PE
CHREG	Pseudo Instruction	CLEAR	Logical
CPPRED	Logical	CPREG	Logical
CPSHIFTML	Bit Shift	CPSHIFTLM	Bit Shift
DEC	Arithmetic	INC	Arithmetic
MVSTCURRPE	Logical/Bit Shift	MVSTNEXTPE	Logical/Bit Shift
NOP	Miscellaneous	NOT	Logical
OR	Logical	PINV	Predicate Modifying
PSet	Predicate Modifying	PSetEven	Predicate Modifying
PSetOdd	Predicate Modifying	PSHIFTML	Bit Shift
PSHIFTLM	Bit Shift	REPEAT	Pseudo Instruction
SETEQ	Comparison	SETGT	Comparison
SETLT	Comparison	SHIFTLM	Bit Shift
SHIFTLMPE	Bit Shift/Inter PE	SHIFTML	Bit Shift
SHIFTMLPE	Bit Shift/Inter PE	SIG_CTRL	Miscellaneous
SUB	Arithmetic	SWAP	Miscellaneous
XOR	Logical		

Table B-1. SOSA Instruction Set

Bit	Description
0-1	uop register selector
2	Add
3	Sub
4	Constant Op 1 (Ignore second register specifier)
5	AND
6	OR
7	XOR
8	Move Status Bit
9	Not
10	Shift LSB to MSB
11	Shift MSB to LSB
12	Set Predicate Bit
13	Reset Predicate Bit
14	Predicated Instruction
15	Inter PE instruction

Table B-2. Instruction Bit Definitions

tion are given in Table B-2. Before we describe individual instructions, we establish notation that will be used through this appendix. Registers are specified by a register micro-instruction that allows the user to specify up to three instructions. We will denote these three as R_s, R_t and R_d. Since the PEs are connected in a logical ring, the current PE is denoted by PE_C, the PE to its left is denoted by PE_L and the PE to its right is denoted by PE_R. Each PE has one predicate and status bit per physical register. These are denoted by P_i (predicate bit i) and S_i (status bit i). We now describe instructions, breaking down the discussion into the different categories.

B.1 Arithmetic Instructions

The arithmetic instructions modify the status bit stored in the tail of the PE. All arithmetic instructions can be predicated. We present one example of a predicated arithmetic instruction (predicated ADD) to illustrate the operation of a predicated instruction.

Mnemonic	Operands	Value	Operation
ADD	R_s, R_t, R_d	0x0004	$R_d=R_s+R_t$
Notes	Sets status bit S_d if there is a carry out from the MSB		

Mnemonic	Operands	Value	Operation
INC	R_s	0x0014	$R_s=R_s+1$
Notes	Sets status bit S_s if there is a carry out from the MSB		

Mnemonic	Operands	Value	Operation
SUB	R_s, R_t, R_d	0x0008	$R_d=R_s-R_t$
Notes	Sets status bit S_d if there is a borrow out from the MSB		

Mnemonic	Operands	Value	Operation
DEC	R_s	0x0018	$R_s=R_s-1$
Notes	Sets status bit S_s if there is a borrow out from the MSB		

Mnemonic	Operands	Value	Operation
PRADD	R_s, R_t, R_d	0x4004	$R_d=R_s+R_t$ if $P_s=1$
Notes	If P_s is set, the add is performed. If P_s is 0, the instruction is treated as a NOP. Sets status bit S_d if there is a carry out from the MSB		

Mnemonic	Operands	Value	Operation
PRSUB	R_s, R_t, R_d	0x4008	$R_d=R_s-R_t$ if $P_s=1$
Notes	If P_s is set, the sub is performed. If P_s is 0, the instruction is treated as a NOP. Sets status bit S_d if there is a borrow out from the MSB		

Mnemonic	Operands	Value	Operation
PRINC	R_s	0x4014	$R_s=R_s+1$ if $P_s=1$
Notes	If P_s is set, the INC is performed. If P_s is 0, the instruction is treated as a NOP. Sets status bit S_d if there is a carry out from the MSB		

Mnemonic	Operands	Value	Operation
PRDEC	R_s	0x4018	$R_s=R_s-1$ if $P_s=1$
Notes	If P_s is set, the DEC is performed. If P_s is 0, the instruction is treated as a NOP. Sets status bit S_d if there is a borrow out from the MSB		

B.2 Logical Instructions

SOSA supports basic logical instructions. Each of these instructions can be predicated. Logical instructions do not modify status bits.

Mnemonic	Operands	Value	Operation
AND	R_s, R_t, R_d	0x0020	$R_d=R_s.R_t$
Notes	-		

Mnemonic	Operands	Value	Operation
CLEAR	R_s	0x0090	$R_s=0_t$
Notes	AND R_s with 0 to clear register		

Mnemonic	Operands	Value	Operation
CPPRED	P_s, P_d	0x1030	$P_d=R_s$
Notes	Copy predicate bit P_s into predicate bit P_d		

Mnemonic	Operands	Value	Operation
CPREG	R_s, R_d	0x0030	R_d=R_s
Notes	Copy R_s to R_d		

Mnemonic	Operands	Value	Operation
NOT	R_s	0x0200	R_s=$\overline{R}_s$
Notes	Invert bits of R_s		

Mnemonic	Operands	Value	Operation
OR	R_s, R_t, R_d	0x0040	R_d=R_s U R_t
Notes	-		

Mnemonic	Operands	Value	Operation
XOR	R_s, R_t, R_d	0x0080	R_d=R_s XOR R_t
Notes	-		

B.3 Bit Shift Instructions

SOSA supports instructions to shift bits within a register in a PE, and between PEs. Instructions that send bits between PEs cannot be predicated.

Mnemonic	Operands	Value	Operation
BITSHIFTLMPE	R_s	0x8410	R_s << 1
Notes	Shift from LSB to MSB, move across PE boundaries, $PE_C.S_s$ gets copied into $PE_R.P_s$, $PE_L.S_s$ gets copied into $PE_C.P_s$, $PE_C.P_s$ becomes the LSB		

Mnemonic	Operands	Value	Operation
MVSTCURRPE	S_s	0x0900	P_s=S_s
Notes	Copy the status bit S_s into the predicate bit P_s		

Mnemonic	Operands	Value	Operation
BITSHIFTMLPE	R_s	0x8810	$R_s >> 1$
Notes	Shift from MSB to LSB, move across PE boundaries, $PE_C.S_s$ gets copied into MSB, $PE_C.P_s$ gets copied into $PE_L.S_s$, $PE_R.P_s$ gets copied into $PE_C.S_s$		

Mnemonic	Operands	Value	Operation
CPSHIFTLM	R_s, R_t	0x0430	$R_t=R_s >> 1$
Notes	Copy Rs to Rt, and shift R_t from LSB to MSB 1 position. Does not cross PE boundaries, P_t, S_t not changed		

Mnemonic	Operands	Value	Operation
CPSHIFTML	R_s, R_t	0x0830	$R_t=R_s <<1$
Notes	Copy Rs to Rt, and shift R_t from MSB to LSB 1 position. Does not cross PE boundaries, P_t, S_t not changed		

Mnemonic	Operands	Value	Operation
MVSTNEXTPE	$PE_C.S_s$	0x8500	$PE_R.P_s=PE_C.S_s$
Notes	Copy the status bit S_s from the current PE into the predicate bit P_s of the next PE		

Mnemonic	Operands	Value	Operation
PSHIFTLM	R_s	0x1400	$R_s >> 1$
Notes	Shift R_s from LSB to MSB 1 position. Does not cross PE boundaries, LSB sent to P_s, S_s copied to P_s		

Mnemonic	Operands	Value	Operation
PSHIFTML	R_s	0x1800	$R_s <<1$
Notes	Shift R_s from MSB to LSB 1 position. Does not cross PE boundaries, P_s copied to LSB, S_s copied to MSB, $S_s=0$		

Mnemonic	Operands	Value	Operation
SHIFTLM	R_s	0x0400	$R_s >> 1$
Notes	Shift R_s from LSB to MSB 1 position. Does not cross PE boundaries. P_s and S_s unchanged		

Mnemonic	Operands	Value	Operation
SHIFTML	R_s	0x0800	$R_s <<1$
Notes	Shift R_s from MSB to LSB 1 position. Does not cross PE boundaries. P_s and S_s unchanged		

Mnemonic	Operands	Value	Operation
SHIFTLMPE	R_s	0x8400	$PE_C.R_s=PE_L.R_s$ $PE_R.R_s=PE_C.R_s$
Notes	Send register to next PE. Status and predicate bits also copied		

Mnemonic	Operands	Value	Operation
SHIFTMLPE	R_s	0x8800	$PE_C.R_s=PE_R.R_s$ $PE_L.R_s=PE_C.R_s$
Notes	Send register to previous PE. Status and predicate bits also copied		

B.4 Predicate Modifying Instructions

SOSA supports instructions that modify predicate bits. These instructions themselves can be predicated.

Mnemonic	Operands	Value	Operation
PINV	P_s	0x3200	$P_s=\overline{P_s}$
Notes	Invert predicate bit P_s		

Mnemonic	Operands	Value	Operation
PSet	P_s	0x1000	$P_s=\overline{1}$
Notes	Set predicate bit P_s		

Mnemonic	Operands	Value	Operation
PSetEven	P_s	0x2000	P_s=1 if Even PE
Notes	Set P_s if current PE has an even ID		

Mnemonic	Operands	Value	Operation
PSetOdd	P_s	0x1010	P_s=1 if Odd PE
Notes	Set P_s if current PE has an odd ID		

B.5 Comparison Instructions

SOSA supports three comparison instructions that perform a comparison and set a predicate bit based on the result of the comparison.

Mnemonic	Operands	Value	Operation
SETEQ	R_s,R_d,P_t	0x1084	P_t=1 if R_s=R_d
Notes	Set P_t if the two source registers are equal.		

Mnemonic	Operands	Value	Operation
SETGT	R_s,R_d,P_t	0x1008	P_t=1 if R_s>R_d
Notes	Set P_t if R_s is greater than R_d		

Mnemonic	Operands	Value	Operation
SETLT	R_s,R_d,P_t	0x1208	P_t=1 if R_s<R_d
Notes	Set P_t if R_s is less than R_d		

B.6 Miscellaneous and Pseudo-Instructions

Mnemonic	Operands	Value	Operation
NOP	-	0x0000	None
Notes	Do nothing		

Mnemonic	Operands	Value	Operation
SIG_CTRL	-	0x8C00	
Notes	Send a signal to the external controller		

Mnemonic	Operands	Value	Operation
SWAP	R_s,R_d	0x0370	Swap R_s and R_d
Notes	Swap the values of R_s and R_d		

Mnemonic	Operands	Value	Operation
CHREG	R_s,R_d,R_t	-	
Notes	This pseudo-instruction allows reuse of current opcode with new register specifiers		

Mnemonic	Operands	Value	Operation
REPEAT	5 bit Count	-	
Notes	This pseudo-instruction allows the repeated execution of an instruction		

```
for i=1 to N
 for j=1 to N
  for k=1 to N
   C[i][j]=C[i][j]+A[i][j]*B[j][k];
  end
 end
end
```

Figure B-1. Matrix Multiplication - N^3 algorithm

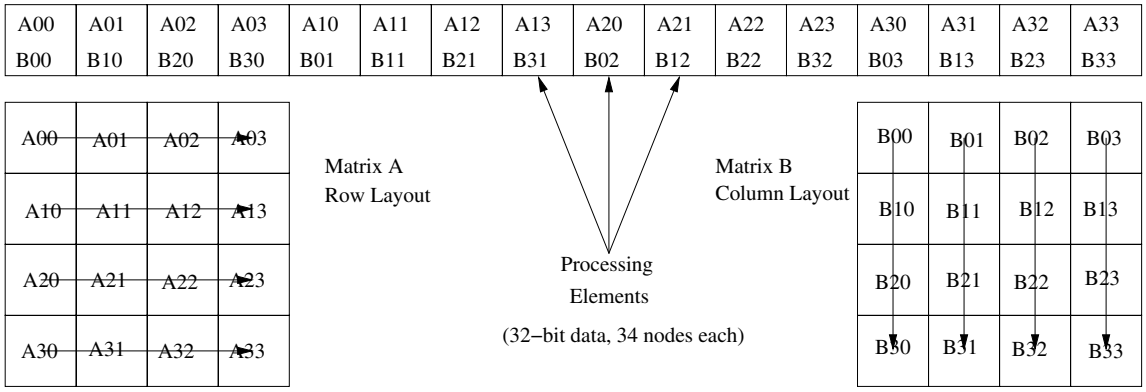


Figure B-2. Matrix Layout

B.7 Programming SOSA - Matrix Multiplication

In this section, we provide a brief overview of programming SOSA. We use matrix multiplication as a running example, and demonstrate how various optimizations can be applied to improve performance. We begin with the N^3 algorithm for multiplying two NxN matrices A and B, shown in Figure B-1. Now, since SOSA does not include memory addressable from within the PEs, we assume that data is distributed among the PEs. We choose a simple data layout - each PE holds one element each of the input matrices (depicted in Figure B-2, for two 4x4 matrices). We divide the algorithm into four parts, each of which is repeated N times. The first part computes the N^3 products, the second part accumulates sums to create elements of the result, the third part moves data within the PEs to set up the next iter-

```
; Initialize before Multiply
CPREG R4,R2            ; Copy R4->R2
CPREG R3,R1            ; Copy R3->R1
CLEAR R5               ; Clear R5
; Multiply (Loop Dw times) (Dw: Data Width)
SHIFTLM R1             ;  Shift LSB to MSB (multiply by 2)
PSHIFTML R2,R5      ;  Shift MSB to LSB, LSB to pred.reg R5
PRADD R5,R1,R5         ;   if predicate is set, R5=R5+R1
CLEAR R6               ;   Clear R6
; Accumulate partial products
;---Repeat N times---
ADD R6,R6,R5           ;  Accumulate partial sum
CPREG R6,R5            ;  Copy R6 to R5
SHIFTMLPE R5         ;  Send accumulated sum to previous PE
; Align rows of matrix A for next set of multiplies
;(Repeat (Dw+2)*N times)
SHIFTMLPE R4       ; Move A 'N' PEs to the left
; Move Result
CPREG R8,R9         ; if R8==1, this PE holds the first
                    ; element of a row/column, move this to R9
PSHIFTML R9,R6   ; Move that bit into the predicate register R6
PRCPREG R6,R7      ; if predicate set, copy R6->R7
SHIFTMLPE R7       ; Move R7 one PE to the left (*(Dw+2))
```

Figure B-3. Matrix Multiply: Assembly Code - No Optimizations

ation and the fourth part moves the each newly computed element of the result to its final location. Since SOSA does not have a native multiplication instruction, the first part is not trivial, and is implemented using a shift-add algorithm.

Figure B-3 shows the first version of the primary matrix multiply loop. There are four components as stated earlier: multiply, accumulate, align data, move result. The largest fraction of running time is spent in the first two parts of the algorithm, and we focus on optimizing those parts. The primary optimizations applied to the third and fourth part include the reuse of microinstructions where possible.

To optimize the accumulate, we observe that in each iteration, we want to accumulate N products into a single sum. However, we can exploit matrix sizes that are a power of two, to optimize this accumulation step. We replace the N add iterations by log(N) iterations, and in every k^{th} iteration, we move the sum 2^k PEs before performing the accumulate. This is depicted in Figure B-4 for N=16. This reduces the number of iterations, but does not reduce the amount of data that must be communicated. Note that we perform some extra

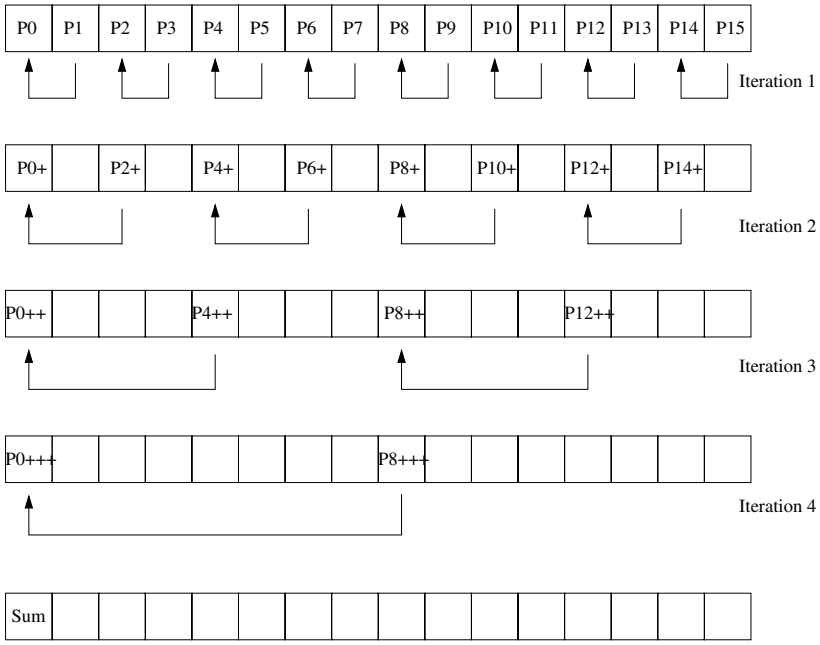


Figure B-4. Logarithmic Accumulate

```
; Accumulate partial products
;---Repeat log2(N) times---
ADD R6,R6,R5          ;  Accumulate partial sum
CPREG R6,R5           ;  Copy R6 to R5
SHIFTMLPE R5          ;  For iteration i, repeat (Dw+2)*i*2 times
; End Repeat
```

Figure B-5. Matrix Multiply: Assembly Code - No Optimizations

ADD instructions on data elements that do not contribute to the final result. We show the final accumulate code in Figure B-5.

To optimize the multiplication, we use loop unrolling, and maximize our use of the register file within each node. If we use 1-bit wide registers, we can unroll the multiply loop 16 times, and perform only two iterations of shift-add. In each unrolled iteration, we create a shifted version of the multiplicand, and generate predicate bits using the multiplier. We use a predicated add to control whether the shifted multiplicand gets added depending on the predicate bit created by the multiplier. The loop unrolling allows us to reuse microinstructions, which helps reduce instruction execution time.

Bibliography

[1] Fujitsu 65nm CMOS Technology. http://www.fujitsu.com/downloads/MICRO/fma/pdf/65nm.pdf.

[2] IBM Standard Cell Design Systems. http://www-03.ibm.com/chips/asics/products/stdcell.html.

[3] IC Knowledge Economics Articles. http://www.icknowledge.com/economics/economics_articles.html.

[4] Toshiba Semi-Custom ICs. http://www.semicon.toshiba.co.jp/eng/prd/common/list/pdf/03asic_200510e.pdf.

[5] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald J. Sussman, and Ron Weiss. Amorphous Computing. *Communications of the ACM*, 43(5):74–82, 2000.

[6] L. Adleman. Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266(5187):1021–1024, November 1994.

[7] V. C. Alves, F. M. G. Franca, and E. P. Granja. A BIST scheme for asynchronous logic. In *Proceedings of the Seventh Asian Test Symposium*, pages 27–32, December 1999.

[8] M. G. Ancona. Systolic Processor Designs Using Single-Electron Digital Circuits. *Superlattices and Microstructures*, 20(4), 1996.

[9] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[10] Adrian Bachtold, Peter Hadley, Takeshi Nakanishi, and Cees Dekker. Logic Circuits with Carbon Nanotube Transistors. *Science*, 294:1317–1320, November 2001.

[11] Gary L. Baldwin, Bernard L. Morris, David B. Fraser, and Angelo R. Tretola. A modular, high-speed serial pipeline multiplier for digital signal processing. *IEEE Journal of Solid-State Circuits*, 13:400–408, June 1978.

[12] Paul Beckett and Andrew Jennings. Toward Nanocomputer Architecture. In *Proceedings of the Seventh Asia-Pacific Computer Systems Architecture Conference*, pages 141–150, 2002.

[13] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif. BLITZEN: A Highly Integrated Massively Parallel Machine. *Journal of Parallel and Distributed Computing*, 8:150–160, February 1990.

[14] Richard Blish et al. Process Integration, Devices and Structures (International Technology Roadmap for Semiconductors). Technical report, International SEMATECH, March 2003.

[15] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), November-December 2005.

[16] Erez Braun, Yoav Eichen, Uri Sivan, and Gdalyahu Ben-Yoseph. DNA-Templated Assembly and Electrode Attachment of a Conducting Silver Wire. *Nature*, 391:775–778, 1998.

[17] P. J. Burke. An RF Circuit Model for Carbon Nanotubes. *IEEE Transactions on Nanotechnology*, 2(1):55–58, March 2003.

[18] Peter J. Burke. Carbon Nanotube Devices for GHz to THz Applications. *Proceedings of SPIE*, 5593:52–61, 2004.

[19] William J. Butera. *Programming a Paintable Computer*. PhD thesis, MIT Media Lab, February 2002.

[20] M. Campbell-Kelly. Programming the EDSAC: Early programming activity at the University of Cambridge. *IEEE Annals of the History of Computing*, 20(4):46–67, 1998.

[21] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[22] J. Chen, M. A. Reed, A. M. Rawlett, and J. M. Tour. Large On-Off Ratios and Negative Differential Resistance in a Molecular Electronic Device. *Science*, 286:1550, 1999.

[23] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The Reconfigurable Streaming Vector Processor (RSVP). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–150, December 2003.

[24] C. P. Collier, E. W. Wong, M. Belohradsky, F. M. Raymo, J. F. Stoddart, P. J. Kuekes, R. S. Williams, and J. R. Heath. Electronically Configurable Molecular-Based Logic Gates. *Science*, 285:391–394, July 1999.

[25] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142, New York, NY, USA, 1993. ACM Press.

[26] Yi Cui and Charles M. Lieber. Functional Nanoscale Electronic Devices Assembled Using Silicon Nanowire Building Blocks. *Science*, 291:851–853, February 2001.

[27] W. Bruce Culbertson, Rick Amerson, Richard J. Carter, Philip Kuekes, and Greg Snider. The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, November 1996.

[28] Yogen K. Dalal and Robert M. Metcalfe. Reverse Path Forwarding of Broadcast Packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

[29] William J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[30] Han Van de Waterbeemd (editor). *Advanced Computer-Assisted Techniques in Drug Discovery.* John Wiley and Sons, second edition, 1994.

[31] Andre DeHon. Array-Based Architecture for Molecular Electronics. In *Proceedings of the First Workshop on Non-Silicon Computation (NSC-1)*, February 2002.

[32] K. Droegemeier, D. Gannon, D. Reed, B. Plale, J. Alameda, T. Baltzer, K. Brewster, R. Clark, B.Domenico, S. Graves, E. Joseph, D. Murray, R. Ramachandran, M. Ramamurthy, L. Ramakrishnan, J. Rushing, D. Webeer, R. Wilhelmson, A. Wilson, M. Xue, and S. Yalda. Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather. *CiSE, Computing in Science & Engineering*, 7:12–29, November 2005.

[33] T. Durkop, S. A. Getty, Enrique Cobas, and M. S. Fuhrer. Extraordinary Mobility in Semiconducting Carbon Nanotubes. *Nano Letters*, 4:35–39, January 2004.

[34] T. Dürkop, S. A. Getty, Enrique Cobas, and M. S. Fuhrer. Extraordinary Mobility in Semiconducting Carbon Nanotubes. *Nano Letters*, 4(1):35–39, 2004.

[35] C. Dwyer. *Self-Assembled Computer Architecture: Design and Fabrication Theory.* PhD thesis, University of North Carolina, May 2003.

[36] C. Dwyer, M. Guthold, M. Falvo, S. Washburn, R. Superfine, and D. Erie. DNA Functionalized Single-Walled Carbon Nanotubes. *Nanotechnology*, 13:601–604, 2002.

[37] C. Dwyer, S. H. Park, T. LaBean, and A. Lebeck. The Design and Fabrication of a Fully Addressable 8-tile DNA Lattice. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, pages 187–191, April 2005.

[38] C. Dwyer, L. Vicci, J. Poulton, D. Erie, R. Superfine, S. Washburn, and R. M. Taylor. The Design of DNA Self-Assembled Computing Circuitry. *IEEE Transactions on VLSI*, 12:1214–1220, November 2004.

[39] Chris Dwyer. Computer-Aided Design for DNA Self-Assembly: Process and Applications. In *Proceedings of IEEE ICCAD*, pages 662–667, November 2005.

[40] Chris Dwyer, Moky Cheung, and Daniel J. Sorin. Semi-empirical SPICE Models for Carbon Nanotube FET Logic. In *Proceedings of the Fourth IEEE Conference on Nanotechnology*, August 2004.

[41] Chris Dwyer, Vijeta Johri, Jaidev P. Patwardhan, Alvin R. Lebeck, and Daniel J. Sorin. Design Tools for Self-assembling Nanoscale Technology. *Institute of Physics Nanotechnology*, 15(9), September 2004.

[42] Chris Dwyer, John Poulton, Russell Taylor, and Leandra Vicci. DNA self-assembled parallel computer architectures. *Nanotechnology*, pages 1688–1694, 2004.

[43] James C. Ellenbogen and J. Christopher Love. Architectures for Molecular Electronic Computers: Logic Structures and an Adder Designed from Molecular Electronic Diodes. *Proceedings of the IEEE*, 88(3):386–426, March 2000.

[44] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *IEEE Micro*, 24:10–20, November 2004.

[45] Roger Espasa et al. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 281–292, May 2002.

[46] Jose A. B. Fortes. Future challenges in VLSI system design. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 5–7, February 2003.

[47] T. J. Fountain, M. J. B. Duff, D. G. Crawley, C. D. Tomlinson, and C. D. Moffat. The Use of Nanoelectronic Devices in Highly-Parallel Computing Systems. *IEEE Transactions on VLSI Systems*, 6(1):31–38, 1998.

[48] Qiang Fu, Chenguang Lu, and Jie Liu. Selective Coating of Single Wall Carbon Nanotubes with Thin SiO2 Layer. *Nano Letters*, 2(4):329–332, 2002.

[49] M. S. Fuhrer, J. Nygard, L. Shih, M. Forero, Young-Gui Yoon, M. S. C. Mazzoni, Hyoung Joon Choi, Jisoon Ihm, Steven G. Louie, A. Zettle, and Paul L. McEuen. Crossed Nanotube Junctions. *Science*, 288:494–497, April 2001.

[50] Y. Fukunaka, M. Motoyama, Y. Konishi, and R. Ishii. Producing Shape-Controlled Metal Nanowires and Nanotubes by an Electrochemical Method. *Electrochemical and Solid-State Letters*, 3(9):C62–C64, 2006.

[51] Aman Gayasen, N. Vijaykrishnan, and Mary J. Irwin. Exploring Technology Alternatives for Nano-Scale FPGA Interconnects. In *Proceedings of the 42nd Annual Design Automation Conference (DAC-2005)*, June 2005.

[52] C. J. Glass and L. M. Ni. The Turn Model for Adaptive Routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 278–287, May 1992.

[53] Seth C. Goldstein and Mihai Budiu. NanoFabrics: Spatial Computing Using Molecular Electronics. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 178–191, July 2001.

[54] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29:147–160, 1950.

[55] J. Han and Pieter Jonker. A System Architecture Solution for Unreliable Nanoelectronic Devices. *IEEE Transactions on Nanotechnology*, 1(4):201–208, December 2002.

[56] Jie Han, Jianbo Gao, Yan Qi, Peter Jonker, and Jose A. B. Fortes. Toward Hardware-Redundant, Fault-Tolerant Logic for Nanoelectronics. *IEEE Design & Test of Computers*, 22(4):328– 339, April 2005.

[57] Miron Hazani, Frank Hennrich, Manfred Kappes, Ron Naaman Naaman, Dana Peled, Victor Sidorov, and Dmitry Shvarts. DNA-mediated self-assembly of carbon nanotube-based electronic devices. *Chemical Physics Letters*, 391:389–392, 2004.

[58] Yu He, Ye Tian, Yi Chen, Zhaoxiang Deng, Alexander E. Ribbe, and Chengde Mao. Sequence Symmetry as a Tool for Designing DNA Nanostructures. *Angewandte Chemie International Edition*, 44(41):6694–66966, 2005.

[59] James R. Heath, Philip J. Kuekes, Gregory S. Snider, and R. Stanley Williams. A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology. *Science*, 280:1716–1721, June 1998.

[60] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, February 2001.

[61] H.P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 258–262, February 2005.

[62] Y. Huang, X. Duan, Q. Wei, and C. M. Lieber. Directed Assembly of One-dimensional Nanostructures into Functional Networks. *Science*, 291:630–633, 2001.

[63] Yu Huang, Xiangfeng Duan, Yi Cui, Lincoln J. Lauhon, Kyoun-Ha Kim, and Charles M. Lieber. Logic Gates and Computation from Assembled Nanowire Building Blocks. *Science*, 294:1313–1317, November 2001.

[64] Sumio Ijima. Helical Microtubules of Graphitic Carbon. *Nature*, 354:56–58, 1991.

[65] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[66] International Technology Roadmap for Semiconductors, 2001.

[67] International Technology Roadmap for Semiconductors, 2003.

[68] International Technology Roadmap for Semiconductors, 2005.

[69] Hiroshi Ishi, Tadashi Shibata, Hideo Kosaka, and Tadahiro Ohmi. Hardware-Backpropagation Learning of Neuron MOS Neural Networks. In *International Electron Devices Meeting Technical Digest*, pages 435–438, December 1992.

[70] Ali Javey, Jing Guo, Damon B. Farmer and Qian Wang, and Dunwei Wang. Carbon Nanotube Field-Effect Transistors with Integrated Ohmic Contacts and High-K Gate Dielectrics. *Nano Letters*, 4(3):447–450, 2004.

[71] David B Johnson and David A Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[72] N. Kanopoulos. A bit-serial architecture for digital signal processing. *IEEE Transactions on Circuits and Systems*, 32:289–291, March 1985.

[73] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Bruce Khailany. The Imagine Stream Processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[74] K. Keren, M. Krueger, R. Gilad, G. Ben-Yoseph, U. Sivan, and E. Braun. Sequence-Specific Molecular Lithography on Single DNA Molecules. *Science*, 297:72, 2002.

[75] B. M. Kim, T. Brintlinger, E. Cobas, M. S. Fuhrer, Haimei Zheng, Z. Yu, R. Droopad, J. Ramdani, and K. Eisenbeiser. High-performance Carbon Nanotube Transistors on SrTiO3 Si Substrates. *Applied Physics Letters*, 84(11), March 2004.

[76] Ho-Seop Kim and James E. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[77] Ho-Seop Kim and James E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO) 2003*, pages 25–35, March 2003.

[78] Don E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[79] Dimitri Komatitsch, Seiji Tsuboi, Chen Ji, and Jeroen Tromp. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, November 2003.

[80] Hidetoshi Kudo and Masamichi Fujihira. DNA-Templated Copper Nanowire Fabrication by a Two-Step Process Involving Electroless Metallization. *IEEE Transactions on Nanotechnology*, 5(2):90–92, 2006.

[81] Thomas H. LaBean, Hao Yan, Jens Kopatsch, Furong Liu, Erik Winfree, John H. Reif, and Nadrian Seeman. Construction, Analysis, Ligation, and Self-Assembly of DNA Triple Crossover Complexes. *Journal of the American Chemistry Society*, 122:1848–1860, 2000.

[82] S. H. Lavington. The Manchester Mark I and atlas: a historical perspective. *Communications of the ACM*, 21(1):4–12, 1978.

[83] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[84] A. Lines. Asynchronous interconnect for synchronous SoC design. *IEEE Micro*, 24:32–41, Jan/Feb 2004.

[85] D. Liu, S-H. Park, J. H. Reif, and T.H. LaBean. DNA Nanotubes Self-assembled from TX Tiles as Templates for Conductive Nanowires. *Proceedings of the National Academy of Science*, 101(3):717–722, 2004.

[86] D. Liu, J.H. Reif, and T.H. LaBean. DNA Nanotubes: Construction and Characterization of Filaments. In *The 8th International Meeting on DNA Based Computers (DNA 8), Sapporo, Japan*, June 2002.

[87] Jie Liu, Andrew G. Rinzler, Hongjie Dai, Jason H. Hafner, R. Kelley Bradley, Peter J. Boul, Adrian Lu, Terry Iverson, Konstantin Shelimov, Chad B. Huffman, Fernando Rodriguez-Macias, Young-Seok Shon, T. Randall Lee, Daniel T. Colbert, and Richard E. Smalley. Fullerene Pipes. *Science*, 280:1253–1256, 1998.

[88] D. C. Look and J. R. Sizelove. Predicted Maximum Mobility in bulk GaN. *Applied Physics Letters*, 79(8):1133–1135, August 2001.

[89] S. R. Lustig, E. D. Boyes, R. H. French, T. D. Gierke, M. A. Harmer, P. B. Hietpas, A. Jagota, R. S. McLean, G. P. Mitchell, G. B. Onoa, and K. D. Sams. Lithographically Cut Single-walled Carbon Nanotubes: Controlling Length Distribution and Introducing End-group Functionality. *Nano Letters*, 3(8):1007–1012, August 2003.

[90] R.E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal*, pages 200–209, 1962.

[91] Priya Mahadevan, Dmitri Krioukov, Kevin Fall, and Amin Vahdat. A Basis for Systematic Analysis of Network Topologies. In *ACM SIGCOMM Conference*, 2006.

[92] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[93] Benjamin R. Martin, Daniel J. Dermody, Brian D. Reiss, Mingming Fang, L. Andrew Lyon, Michael J. Natan, and Thomas E. Mallouk. Orthogonal Self-Assembly on Colloidal Gold-Platinum Nanorods. *Advanced Materials*, 11(12):1021–1025, August 1999.

[94] Paul L. McEuen, Michael S. Fuhrer, and Hongkun Park. Single-Walled Carbon Nanotube Electronics. *IEEE Transactions on Nanotechnology*, 1(1):78–85, March 2002.

[95] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, April 1965.

[96] J. Mosegaard and T. S. Sorensen. GPU accelerated surgical simulators for complex morphology. pages 147–153, March 2005.

[97] Roger Needham and David Wheeler. Tea Extensions. Technical report, Computer Laboratory, University of Cambridge, October 1997.

[98] L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, February 1993.

[99] Michael T. Niemier and Peter M. Kogge. Exploring and Exploiting Wire-Level Pipelining in Emerging Technologies. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 166–177, July 2001.

[100] Michael T. Niemier, R. Ravichandran, and Peter M. Kogge. Using circuits and systems-level research to drive nanotechnology. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 302–309, October 2004.

[101] K. Nikolic, A. Sadek, and M. Forshaw. Fault-Tolerant Techniques for Nanocomputers. *Nanotechnology*, 13:357–362, 2002.

[102] N.Kamiura, Y. Taniguchi, T.Isokawa, and N.Matsui. Design of Fault Tolerant Multistage Interconnection Networks with Dilated Links. *IEIECE Transactions on Information and Systems*, E84-D:1500–1507, November 2001.

[103] M. Oskin, F. T. Chong, I. Chuang, and J. Kubiatowicz. Building Quantum Wires: The Long and the Short of it. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 374–385, June 2003.

[104] S. H. Park, C. Pistol, S. J. Ahn, J. H. Reif, A. R. Lebeck, C. L. Dwyer, and T. H. Labean. Finite-size, Fully-Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures. *Angewandte Chemie*, 45:735–739, January 2006.

[105] Sung Ha Park, Hao Yan, John H. Reif, Thomas H LaBean, and Gleb Finkelstein. Electronic nanostructures templated on self-assembled DNA scaffolds. *Institute of Physics Nanotechnology*, 15(8):S525–S527, July 2004.

[106] Sung Ha Park, Peng Yin, Yan Liu, John H. Reif, Thomas H. LaBean, and Hao Yan. Programmable DNA Self-Assemblies for Nanoscale Organization of Ligands and Proteins. *Nano Letters*, 5(4):729–733, 2005.

[107] Jaidev P. Patwardhan, Chris Dwyer, and Alvin R. Lebeck. Design and Evaluation of Fail-Stop Self-Assembled Nanoscale Processing Elements. In *IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '06)*, June 2006.

[108] Jaidev P. Patwardhan, Chris Dwyer, and Alvin R. Lebeck. Self-Assembled Networks: Control vs. Complexity. In *1st International Conference on Nano-Networks*, September 2006.

[109] Jaidev P. Patwardhan, Chris Dwyer, Alvin R. Lebeck, and Daniel J. Sorin. Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, pages 344–358, April 2004.

[110] Jaidev P. Patwardhan, Chris Dwyer, Alvin R. Lebeck, and Daniel J. Sorin. Evaluating the Connectivity of Self-Assembled Networks of Nano-scale Processing Elements. In *IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '05)*, pages 2.1–2.8, May 2005.

[111] Jaidev P. Patwardhan, Chris Dwyer, Alvin R. Lebeck, and Daniel J. Sorin. NANA: A Nano-scale Active Network Architecture. *ACM Journal on Emerging Technologies in Computing Systems*, 2(1):1–30, 2006.

[112] Jaidev P. Patwardhan, Vijeta Johri, Chris Dwyer, and Alvin R. Lebeck. A Defect Tolerant Self-Organizing Nanoscale SIMD Architecture. In *International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[113] F. Peper, J. Lee, S. Adachi, and S. Mashiko. Laying Out Circuits on Asynchronous Cellular Arrays: A Step Towards Feasible Nanocomputers. *Nanotechnology*, 14(4):469–485, 2003.

[114] Performance Database Server. http://www.netlib.org/performance/html/PDStop.html.

[115] O. A. Petlin and S. B. Furber. Built-in self-testing of micropipelines. In *Proceedings., Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 22–29, April 1997.

[116] Constantin Pistol, Chris Dwyer, and Alvin R. Lebeck. Design Automation for DNA Self-Assembled Nanostructures. In *Proceedings of the 43rd Design Automation Conference (DAC)*, July 2006.

[117] Vincenzo Piuri. Analysis of Fault Tolerance in Artificial Neural Networks. *Journal of Parallel and Distributed Computing*, pages 18–48, January 2001.

[118] Soumya S. Ray, Richard J. Nowak, Robert H. Brown Jr., and Peter T. Lansbury Jr. Small-molecule-mediated stabilization of familial amyotrophic lateral sclerosis-linked superoxide dismutase mutants against unfolding and aggregation. *PNAS*, 102(10):3639–3644, 2005.

[119] J.H. Reif, T.H. LaBean, and N.C. Seeman. Challenges and Applications for Self-Assembled DNA Nanostructures. A. Condon and G. Rozenberg, editors, In *Proceedings. Sixth International Workshop on DNA-Based Computers, Leiden, The Netherlands. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelber*, volume 2054, June 2000.

[120] B. H. Robinson and N. C. Seeman. The design of a biochip: a self-assembling molecular-scale memory device. *Protein Engineering*, 1(4):295–300, 1987.

[121] S. Rosenblatt, H. Lin, V. Sazonova, S. Tiwari, and P. L. McEuen. Mixing at 50GHz using a Single-Walled Carbon Nanotube Transistor. *Applied Physics Letters*, 87:153111, October 2005.

[122] Sami Rosenblatt, Yuval Yaish, Jiwoong Park, Jeff Gore, Vera Sazonova, and Paul L. McEuen. High Performance Electrolyte Gated Carbon Nanotube Transistors. *Nano Letters*, 2(8):869–872, 2002.

[123] Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440:297–302, 2006.

[124] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 422–433, June 2003.

[125] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A High-speed, Self-Configuring Local Area Network Using Point to Point Links. *IEEE Journal on Selected Areas in Communications*, 9(8), October 1991.

[126] N.C. Seeman. DNA Engineering and its Application to Nanotechnology. *Trends in Biotech*, 17:437–443, 1999.

[127] Tadashi Shibata and Tadahiro Ohmi. A Functional MOS transistor featuring gate-level weighted sum and threshold operations. *IEEE Transactions on Electron Devices*, 39:1444–1455, June 1992.

[128] Fadi N. Sibai. A Fault-Tolerant Digital Artificial Neuron. *IEEE Design and Test of Computers*, 10:76–82, December 1993.

[129] K. Skinner, R. L. Carroll, S. Washburn, and C. L. Dwyer. Nanowire Transistors, Gate Electrodes, and Their Directed Self-Assembly. In *The 72nd Southeastern Section of the American Physical Society (SESAPS)*, November 2005.

[130] Greg Snider, Philip Kuekes, and R Stanley Williams. CMOS-like logic in defective, nanoscale crossbars. *Nanotechnology*, (15):881–891, 2004.

[131] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[132] Mark A. Spicer and Michael Apuzzo. Virtual Reality Surgery: Neurosurgery and the Contemporary Landscape. *Neurosurgery*, 52(3):489–498, March 2003.

[133] M. R. Stan, Paul D. Franzon, Seth C. Goldstein, J. C. Lach, and M. M. Ziegler. Molecular electronics: from devices and interconnect to circuits and architecture. In *Proceedings of the IEEE*, volume 91, pages 1940–1957, November 2003.

[134] M. Steffen, L. M. K. Vandersypen, and I. L. Chuang. Toward Quantum Computation: A Five-qubit Quantum Processor. *IEEE Micro*, 21:24–34, March 2001.

[135] M. S. Strano, C. A. Dyke, M. L. Usrey, P. W. Barone, M. J. Allen, H. W. Shan, C. Kittrell, R. H. Hauge, J. M. Tour, and R. E. Smalley. Electronic Structure Control of Single-walled Carbon Nanotube Functionalization. *Science*, 301:1519–1522, September 2003.

[136] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, November 2000.

[137] Hongsuda Tangmunarunkit, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Network topology generators: degree-based vs. structural. *SIGCOMM Comput. Commun. Rev.*, 32(4):147–159, 2002.

[138] S.J. Tans, A.R.M. Verschueren, and C. Dekker. Room-temperature Transistor Based on a Single Carbon Nanotube. *Nature*, 393:49–52, 1998.

[139] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), 1996.

[140] D. D. Thaker, F. Impens, I. L. Chuang, R. Amirtharajah, and F. T. Chong. Recursive TMR: Scaling Fault Tolerance in the Nanoscale Era. *IEEE Design & Test of Computers*, 22(4):298– 305, April 2005.

[141] James M. Tour. Molecular Electronics. Synthesis and Testing of Components. *Accounts of Chemical Research*, 33(11):791–804, 2000.

[142] Greg Y. Tseng and James C. Ellenbogen. Toward Nanocomputers. *Science*, 294:1293–1294, November 2001.

[143] Lewis Tucker and George Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, 21:26–38, August 1988.

[144] K. van Berkel and A. Bink. Single-track Handshake Signaling with Application to Micropipelines and Handshake Circuits. In *Procceding of the Seconds International*

Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 122–133, March 1996.

[145] J. von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, Princeton, NJ, 1956.

[146] M.C. Wendl, I. Korf, A.T. Chinwalla, and L.W. Hillier. Automated processing of raw DNA sequence data. *Engineering in Medicine and Biology Magazine, IEEE*, 20:41–48, July-August 2001.

[147] David Wheeler and Roger Needham. TEA: A Tiny Encryption Algorithm. In *Fast Software Encryption: Second International Workshop*, December 1994.

[148] S. J. Wind, J. Appenzeller, R. Martel, V. Derycke, and Ph. Avouris. Vertical Scaling of Carbon Nanotube Field-Effect Transistors using Top Gate Electrodes. *Applied Physics Letters*, 80:3817–3819, May 2002.

[149] E. Winfree, F. Liu, L. A. Wenzler, and N.C. Seeman. Design and Self-Assembly of Two-Dimensional DNA Crystals. *Nature*, 394:539, 1998.

[150] Hao Yan, Thomas H. LaBean, Liping Feng, and John H. Reif. Directed Nucleation Assembly of Barcode Patterned DNA Lattices. *Proceedings of the National Academy of Sciences*, 100(14):8103–8108, July 2003.

[151] Hao Yan, Sung Ha Park, Liping Feng, Gleb Finkelstein, John H. Reif, and Thomas H. LaBean. 4x4 DNA Tile and Lattices: Characterization, Self-Assembly, and Metallization of a Novel DNA Nanostructure Motif. In *Proceedings of the Ninth International Meeting on DNA Based Computers (DNA9)*, June 2003.

[152] Hao Yan, Sung Ha Park, Gleb Finkelstein, John H. Reif, and Thomas H. LaBean. DNA Templated Self-Assembly of Protein Arrays and Highly Conductive Nanowires. *Science*, 301(5641):1882–1884, September 2003.

[153] N. Yoshikawa, F. Matsuzaki, K. Fujiwara, K. Yoda, and K. Kawasaki. Design and Component Test of a Tiny Processor Based on the SFQ Technology. *IEEE Transactions on Applied Superconductivity*, 13c:441–445, June 2003.

[154] G. Zheng, Kakulapati Gunavardhan, and L. V. Kale. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2004.

[155] Ming Zheng, Anand Jagota, Ellen D. Semke, Bruce A. Diner, Robert S. McLean, Steve R. Lustig, Raymond E. Richardson, and Nancy G. Tassi. DNA-assisted Dispersion and Separation of Carbon Nanotubes. *Nature Materials*, 2:338–342, May 2003.

Biography

Jaidev Patwardhan was born on January 20, 1979 in Pune, India. He received his B.E in Computer Engineering from Veermata Jijabai Technological Institute in Mumbai, India in 2000 and M.S in Computer Science from Duke University in 2002. He was a recipient of a Graduate Fellowship in 2000, and received an award for 'Outstanding Teaching Assistant' in 2002. His research interests include processor and platform architecture and performance analysis. His Ph.D thesis explores the impact of emerging technologies on computer architecture design, focusing on the design of two defect tolerant architectures that account for the limitations of the underlying technology.

Publications

[1] Jaidev P. Patwardhan, Vijeta Johri, Chris Dwyer, and Alvin R. Lebeck. A Defect Tolerant Self-Organizing Nanoscale SIMD Architecture. In *International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[2] Jaidev P. Patwardhan, Chris Dwyer, and Alvin R. Lebeck. Self-Assembled Networks: Control vs. Complexity. In *1st International Conference on Nano-Networks*, September 2006.

[3] Jaidev P. Patwardhan, Chris Dwyer, and Alvin R. Lebeck. Design and Evaluation of Fail-Stop Self-Assembled Nanoscale Processing Elements. In *IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '06)*, June 2006.

[4] Jaidev P. Patwardhan, Chris Dwyer, Alvin R. Lebeck, and Daniel J. Sorin. NANA: A Nano-scale Active Network Architecture. *ACM Journal on Emerging Technologies in Computing Systems*, 2(1):1–30, 2006.

[5] Jaidev P. Patwardhan, Chris Dwyer, Alvin R. Lebeck, and Daniel J. Sorin. Evaluating the Connectivity of Self-Assembled Networks of Nano-scale Processing Elements. In *IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '05)*, pages 2.1–2.8, May 2005.

[6] Chris Dwyer, Vijeta Johri, Jaidev P. Patwardhan, Alvin R. Lebeck, and Daniel J. Sorin. Design Tools for Self-assembling Nanoscale Technology. *Institute of Physics Nanotechnology*, 15(9), September 2004.

[7] Jaidev P. Patwardhan, Chris Dwyer, Alvin R. Lebeck, and Daniel J. Sorin. Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, pages 344–358, April 2004.

[8] Jaidev P. Patwardhan, Alvin R. Lebeck, and Daniel J. Sorin. Communication Breakdown: Analyzing CPU Usage in Commercial Web Workloads. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 12–19, March 2004.

[9] Annie P. Foong, Thomas Huff, Herbert J. Hum, Jaidev P. Patwardhan, and Greg Regnier. TCP Performance Re-Visited. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 70–79, March 2003.

[10] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, May 2002.

[11] Jagadeeswaran Rajendiran, Jaidev P. Patwardhan, Vijay Abhijit, Rahul Lakhotia, and Amin Vahdat. Exploring the Benefits of a Continuous Consistency Protocol for Wireless Web Portals. In *IEEE Workshop on Wireless Internet Applications (WIAPP)*, pages 65–73, July 2001.