A Defect Tolerant Self-organizing Nanoscale SIMD Architecture

Jaidev P. Patwardhan†, Vijeta Johri†, Chris Dwyer‡, and Alvin R. Lebeck†
{jaidev,vijeta,alvy}@cs.duke.edu, dwyer@ee.duke.edu

†Department of Computer Science
Duke University
Durham, NC 27708

‡Department of Electrical and Computer Engineering
Duke University
Durham, NC 27708

Abstract

The continual decrease in transistor size (through either scaled CMOS or emerging nano-technologies) promises to usher in an era of tera to peta-scale integration. However, this decrease in size is also likely to increase defect densities, contributing to the exponentially increasing cost of top-down lithography. Bottom-up manufacturing techniques, like self-assembly, may provide a viable lower-cost alternative to top-down lithography, but may also be prone to higher defects. Therefore, regardless of fabrication methodology, defect tolerant architectures are necessary to exploit the full potential of future increased device densities.

This paper explores a defect tolerant SIMD architecture. A key feature of our design is the ability of a large number of limited capability nodes with high defect rates (up to 30%) to self-organize into a set of SIMD processing elements. Despite node simplicity and high defect rates, we show that by supporting the familiar data parallel programming model the architecture can execute a variety of programs. The architecture efficiently exploits a large number of nodes and higher device densities to keep device switching speeds and power density low. On a medium sized system (~1cm^2 area), the performance of the proposed architecture on our data parallel programs matches or exceeds the performance of an aggressively scaled out-of-order processor (128-wide, 8k reorder buffer, perfect memory system). For larger systems (>1cm^2), the proposed architecture can match the performance of a chip multiprocessor with 16 aggressively scaled out-of-order cores.

Categories and Subject Descriptors B.4.3 [Input/Output and Data Communications]: Interconnections (Subsystems); B.6.1 [Logic Design]: Design Styles; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors).

General Terms Design, Performance, Reliability

Keywords self-organizing, SIMD, data parallel, bit-serial, defect tolerance, DNA, nanocomputing.

1 Introduction

Manufacturing defects, power density, process variability, transient faults, bulk silicon limits, rising test 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 importance of these issues and the need to explore long term alternatives to CMOS devices and fabrication techniques [17].

ASPLOS'06 October 21-25, 2006, San Jose, California, USA.

One promising alternative is DNA-based self-assembly of nanoscale components using inexpensive laboratory equipment to achieve tera to peta-scale integration. Although much of this technology is in its infancy (i.e., demonstrated in research lab experiments), by studying its potential uses for building computing systems, architects can gain a deeper understanding of its limitations and opportunities while providing important feedback to the scientists developing the new technologies.

DNA-based fabrication produces precise control within a small area (e.g., 9 μm^2) enabling the construction of a large number (~10^9-10^{12}) of small nodes (computational circuits with ~10^4 transistors) that can be linked together using self-assembly. This produces a random network of nodes, due to the lack of control over placement and orientation of nodes, that contains defective nodes and links. While our work is motivated by DNA-based self-assembly, it is applicable to any technology with similar characteristics (e.g., scaled CMOS with high process variability, high defect rates and point-to-point links between relatively small compute nodes). The challenge for computer architects is to efficiently exploit the computational power of the large number of nodes while overcoming two primary challenges: 1) loss of precise control over the entire fabrication process, and 2) high defect rates.

This paper presents a SIMD architecture designed to address these challenges. The fundamental building block in our architecture is a relatively small node (e.g., 1-bit ALU with 32 bits of storage and communication support for four neighbors) that operates asynchronously. A configuration phase at startup isolates defective nodes and allows groups of nodes to self-organize into SIMD processing elements (PEs) which are connected in a logical ring, thus simplifying the programmer's view of the system.

Simulations using conservative estimates for node size and device speed show that the proposed design can match the performance of aggressively scaled architectures for 8 out of 9 benchmarks tested. Furthermore, this performance is achieved with a very low power density of 6.5 W/cm^2 (vs. >75 W/cm^2 for modern cores) while conservatively assuming that about 90% of the devices in the system switch every nanosecond. Finally, we show that our system can tolerate up to 30% defective nodes. Our results demonstrate the potential of this technology for building high performance architectures despite high defect rates and loss of precise control during fabrication. Further improvements are possible as the technology scales to allow more complex nodes, better internode connectivity, and faster devices. The main contributions of this paper are:

- adapting self-organization methods to computer architectures,
- designing a node that balances fabrication constraints with functionality needed to communicate, compute, and self-organize, and,

- demonstrating the above capabilities by composing a high-performance, defect tolerant SIMD architecture from a random network of nodes.

The rest of this paper is organized as follows. Section 2 describes self-assembled nanoscale systems. Section 3 presents a brief overview of our system. Section 4 describes the node architecture in detail. We present node self-organization mechanisms in Section 5 and system architecture in Section 6. We evaluate system performance in Section 7, describe limitations and identify areas for improvement in Section 8, and discuss related work in Section 9.

2 DNA-based Self-Assembled Nanoscale Systems and the Architectural Implications

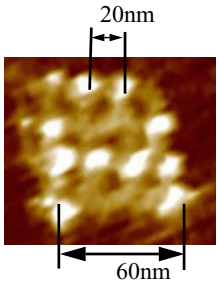

FIGURE 1. Patterned DNA [25]

Self-assembly of nano-electronic devices has the potential to emerge as a lower cost alternative to top-down manufacturing. DNA-based self-assembly [32] uses the precise binding rules of DNA with nanoscale devices to build computing systems. We assume a proposed assembly process [26] to place electronic circuits on a DNA grid [38,39]. The basic principle is to replicate a simple unit cell on a large scale to build a circuit. The unit cell consists of a transistor placed in the cavity of a DNA-lattice. A key requirement of this process is the ability to control the placement of electronic devices (e.g., carbon nanotubes [3,9] or silicon nanowires [15]) at specific points on the DNA scaffold to form a circuit. Recently, two critical steps towards this goal were demonstrated: 1) aperiodic patterns, with a 20nm pitch, on a DNA grid [25] and 2) DNA-guided self-assembly of nanowire transistors [35]. Figure 1 shows an atomic force microscope image of the letter "A" patterned on a DNA grid. Figure 2 shows a schematic of a small lattice with carbon nanotube based transistors. We currently assume only two layers of metal interconnect within a lattice, which limits our ability to place and route circuits. We propose the use of conducting metallic planes separated by insulating layers to provide power and ground to the circuit. Figure 3 depicts a cross-sectional view of the lattice, with two layers of interconnect and the power and ground planes.

Current self-assembly processes produce limited size DNA grids and thus limit circuit size. However, the parallel nature of self-assembly enables constructing many nodes (~10^9-10^{12}) that

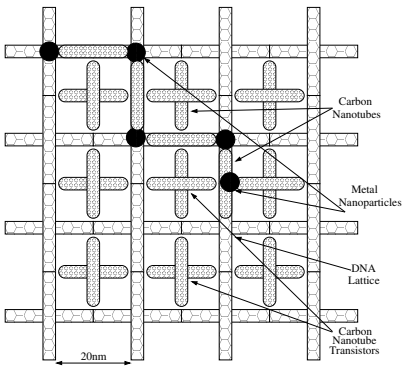

FIGURE 2. DNA Lattice with transistors and interconnect

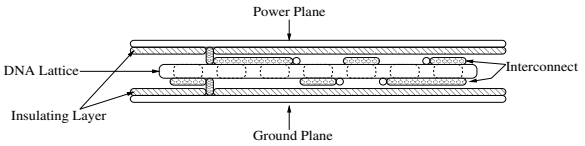

FIGURE 3. Lattice with two levels of interconnect, and connections to V_{dd} and Gnd

may be linked together by self-assembled conducting nanowires [39]. The proposed self-assembly method does not control the placement and orientation of nodes as they are interconnected, resulting in a random network of nodes that contains defective nodes and links. Communication with external CMOS circuitry 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 4 shows a small network of nodes, including regions with defective links, and a via/anchor. In the rest of the paper we use the term "anchor" to refer to an anchor node/via pair.

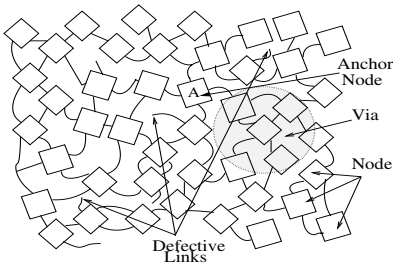

FIGURE 4. Self-assembled network of nodes

A computing system built from this random network must: a) tolerate node and interconnect defects, b) not rely on underlying network structure, c) compose more powerful computational blocks from simple nodes, d) minimize communication overheads, and e) achieve performance that is at least comparable to future CMOS based systems. Several research projects examine building computing systems with a subset of these goals, including self-organization [1,34], routing and resiliency in the face of defects [1,16] and the ability to compose complex computational units from simpler blocks [23], but we face added challenges because of the extremely limited computational capabilities available in nodes. Our previous work, the nanoscale active network architecture (NANA) [29] is a general purpose architecture designed with a similar set of goals, assuming similar underlying technology. However, it fails to match the performance of conventional CMOS systems since it is unable to efficiently utilize the computational capabilities of the nodes at the same time. The design of the SIMD architecture presented in this paper is guided by the lessons learned through the design and evaluation of NANA. We defer discussion of other closely related research to Section 9.

3 System Overview

The goal of this work is to build a defect tolerant computing system with a random network of nodes using a mix of new solutions and adaptations of known techniques and achieve performance comparable to future CMOS based systems. To efficiently utilize large numbers (>10^9-10^{12}) of nodes we implement a SIMD architecture and focus on data parallel workloads. Our proposed system - called the "Self-Organizing SIMD Architecture" (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.

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 connects to other nodes with (up to four) single wire links. Each link supports low bandwidth asynchronous communication that transfers 1 data bit per handshake. To support deadlock-free routing, we add support for three virtual channels (1 bit each). 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.

The configuration process, initiated from an anchor, maps out defective nodes and connects functional nodes in a broadcast tree. The system can be configured in two ways: a) as a monolithic system, all nodes on one logical ring (one "cell"), or b) 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 (b), 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. We discuss space partitioning for our benchmarks in Section 7.

In the next three sections, we describe SOSA in detail. Though we present a bottom-up view of the system, the actual design process was iterative and involved several passes through node and system design, requiring a balance between size constraints and adding hardware optimizations to improve performance.

4 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, splitting the discussion into the data path (Section 4.1), control (Section 4.2), and inter-node communication (Section 4.3), highlighting the trade-offs between functionality, performance and circuit size.

4.1 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.

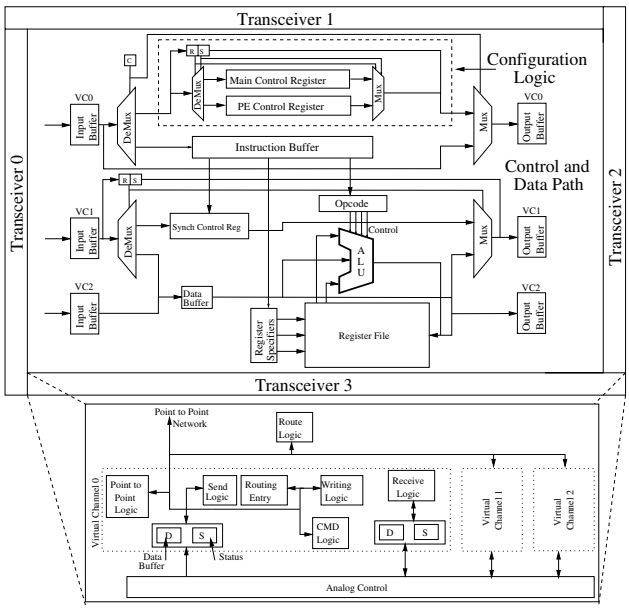

FIGURE 5. Node Floorplan

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 1-bit registers, requiring 32 nodes for the PE, or with 16 2-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 also find that program performance is not sensitive to ALU execution latencies shorter than the time taken to send/receive a bit between nodes.

4.2 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 control registers for defect testing/isolation (main control register), and PE configuration (PE control register). Figure 5 shows a floorplan of the node with the configuration logic demarcated by a dashed rectangle within the control and data block.

The second part is the run-time control logic used to decode and execute instructions. 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 to hold each of three micro-instructions that comprise an instruction: a) opcode, b) register specifier and c) 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 includes fields that allow simple increment/decrement operations on register specifiers 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 cause greater contention (and reduce performance) on the network since instructions and data must share link bandwidth. Simulations reveal that a single entry instruction buffer offers the best trade-off between improving performance and minimizing design complexity.

4.3 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 5.1). 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).

4.4 Circuit Size and Power Estimates

We have completed the circuit design for all node components. We use this design in conjunction with layouts of simple logic blocks to estimate node size and power consumption. Our simulator (discussed in Section 7) models the system in sufficient detail to make it relatively easy to extract a circuit model for most components. Figure 5 shows a floorplan of a node, showing the approximate position (not to scale) of the datapath, control and transceivers. We estimate that the entire node will require 10,000 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 22,000 transistors (based on our experience in laying out circuits) which translates to a 3µm x 3µm node. Recent work [39] 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 [11]. Based on a switching speed of 1 ns (see Section 7.1), 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 $6.5 W/cm^2$, 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.

4.5 Summary

Each node in SOSA is a small circuit that can 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, however like current processors we still dedicate significant area to control and communication circuitry. The challenge is to coordinate the operation of these nodes connected through an unstructured network to execute programs.

5 System Configuration

To use the random network of nodes to perform useful computation we use a configuration mechanism to impose logical structure on the network and isolate defective nodes and links from the rest of the system. This allows nodes to self-organize and avoids the need for an external defect map, which would be impractical to obtain given the scale and bandwidth limitations of the system. Once defective nodes are isolated, the functional nodes are grouped to form PEs. We now describe this configuration in detail.

5.1 Logical Structure and Defect Isolation

We use a variant of the "reverse path forwarding" (RPF) algorithm [7,27] to impose a logical tree structure on the network and isolate defects. When the system is powered up or reset, all nodes enter a "configuration mode", steer incoming packets to the configuration control registers and execute the distributed RPF algorithm. A small packet is inserted through an anchor and is broadcast on all its active links (the transceiver analog control circuitry tests the liveness of its physical link).

The RPF algorithm states that any node receiving the broadcast propagates it on all links except the receiving link if and only if the node has not seen the broadcast before. The node also stores the direction ("gradient") from which it received the broadcast and sets up internal routing information based on this direction. Following the gradient through a set of nodes leads to the broadcast source—the tree root. A depth first traversal is established by nodes locally selecting links in a predefined order relative to their gradient link. Opposite orderings are used for forward (VC1) and reverse (VC2) traversals. This method can be used to have all nodes in the system self-organize into a tree or it can be used to create multiple trees by initiating the broadcast through multiple anchors. For example, we could self-assemble the random network of nodes on a silicon wafer with a grid of vias to create a system with multiple anchors.

Defect isolation is achieved by 1) augmenting each node with built-in-self-test and assuming fail-stop behavior [28], and 2) including a simple test vector in each broadcast packet that each node must successfully execute before propagating the broadcast. Nodes failing the test are isolated since there is no path through the node. Simulations show that the gradient can reach a very large fraction of functional nodes (i.e., achieve good coverage) for node

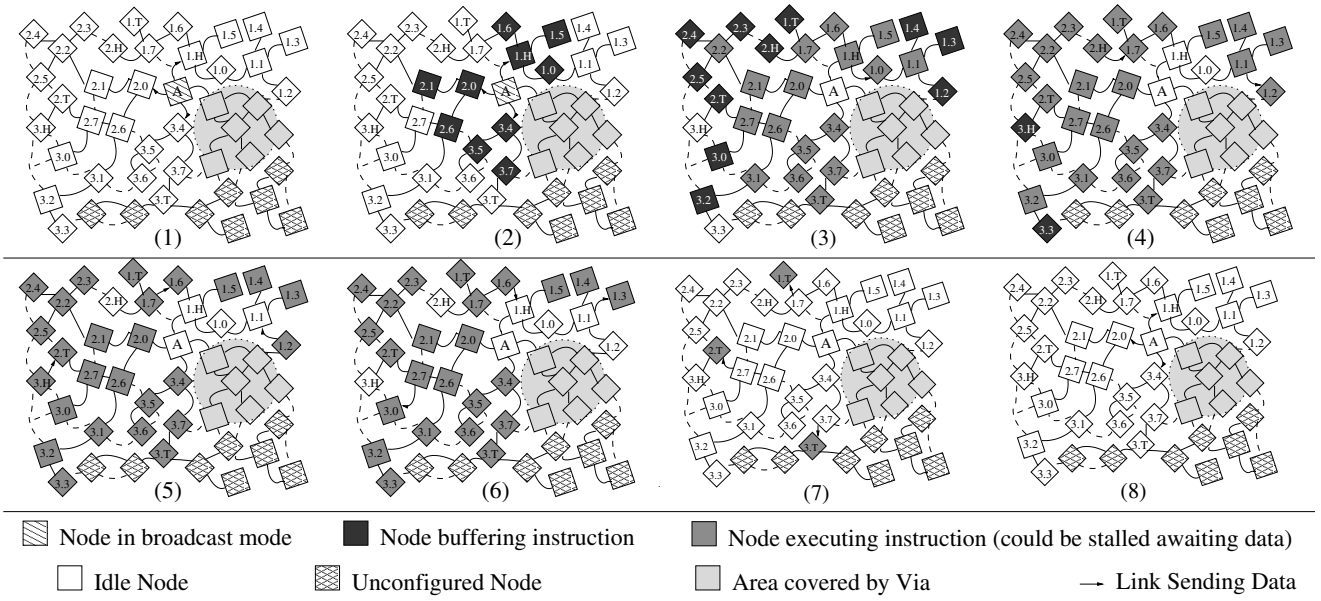

FIGURE 7. Instruction execution in a random network with three configured PEs. The via is shown to cover multiple nodes, which are rendered unusable. The via is connected to the PEs through the anchor node (A). I/O bandwidth into the system could be improved by adding more via/anchor pairs.

defect rates up to 30%. Handling more complex defects like Byzantine failures is beyond the scope of this work.

5.2 Configuring Processing Elements

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 N+2 consecutive unconfigured nodes. 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. To minimize PE setup time in large networks (>10^9 nodes), we could distribute configuration by exploiting multiple anchors.

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 (i.e., PEs with no tail), and deconfigures all intermediate nodes until it reaches (and terminates at) the head node. Figure 6 shows the logical order of nodes within a PE. Figure 7-(1) shows the network from Figure 4 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 that do not lie on the broadcast tree (dashed lines) are not used. The unlabeled nodes outside the via are part of a partial 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).

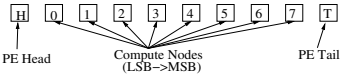

FIGURE 6. PE Layout

We extend PE configuration to optimize PE length (hops from head to tail). Very long PEs (e.g., a PE that spans the broadcast tree root) may reduce performance due to longer intra-PE communication latencies. Since the post-configuration step deconfigures partial PEs, a PE that crosses a length threshold can be rejected by starting a new PE without creating a tail node. We empirically find that a threshold of 4 times the minimum PE length (compute nodes + head + tail) achieves a good balance between extra nodes required and performance gained by reducing PE length.

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

6 System Architecture

In this section, we describe the architecture of SOSA. Careful node design coupled with the self-organizing capability of each node enables us to map a data parallel architecture onto the random network of nodes. We begin by describing the instruction set (Section 6.1) and execution model (Section 6.2). Then, we present an example illustrating the execution of an instruction in the system (Section 6.3).

6.1 Instruction Set Architecture

SOSA uses a three register operand ISA, with microcoded instructions (Table 1 shows a subset of the instruction set). 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

TABLE 1. Instruction Set

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

microinstruction 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 work done per instruction. For example, a Copy-Shift first copies the source to the destination register, and then performs a shift operation on the destination register. SOSA also supports predicated instruction execution (all instructions can be predicated) and has three types of instructions that can modify predicate bits: a) conditional instructions, b) unconditional predicate modifying instructions and c) 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 a neighbor (left or right), and receives a new value for the register from the other neighbor (right or left). 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 N-bit PE, each bit moves exactly (N+2) positions to the left or right, and a node only needs to store the $(N+2)^{th}$ bit in its register file and can "forward" the remaining bits without register access. We use the synch repeat counter to track the bits being forwarded by the node. The node stops forwarding when it receives the $(N+2)^{th}$ 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.

6.2 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. 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 an instruction broadcast skips a microinstruction (except synch), 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 (e.g., for an OR instruction). 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. The head node reads the specified predicate bit and informs the remaining nodes in the PE whether the predicated instruction is to be executed or squashed by sending a synch microinstruction on VC1. Since each node in a PE must wait for the extra synchronization microinstruction (which is consumed by the tail), execution of predicated instructions is serialized through a PE.

6.3 Instruction Execution Example

Figure 7 uses the small configured network with three 8-bit PEs to illustrate the different steps involved in executing an ADD instruction. The anchor node 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 (steps 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 (steps 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 data (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. In the next section we evaluate the performance of SOSA.

7 Evaluation

This section describes our evaluation methodology, simulation infrastructure and workloads (Section 7.1), then compares SOSA performance to four other architectures (Section 7.2). We find that SOSA achieves good performance on benchmarks that have data parallelism. For a configuration with more than 64K PEs, SOSA

TABLE 2. SOSA System Parameters

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

TABLE 3. Ideal Superscalar Parameters

Parameter	Value	Parameter	Value	Parameter	Value
Width	128 (Fetch/Decode/Issue/Commit)	Integer ALU	128 Add,128 Mul	Branch Prediction	Perfect
Instruction Fetch Queue	1024 Entries	FP ALU	128 Add,128 Mul	Memory Latency	1 cycle
ROB/LSQ	8192 entries, single cycle access	Frequency	10GHz	Memory Ports	128

matches the performance of an ideal 16-way CMP. *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, with lower device switching speeds and a lower power density.* We then show that SOSA can tolerate high node defect rates (Section 7.3). For the encryption benchmarks, performance gracefully degrades as the fraction of defective nodes increases to 30%. For the other benchmarks, by over-provisioning the system, SOSA tolerates up to 20% defective nodes with a small (<10%) degradation in performance. We also find that the instruction buffer and microinstruction reuse optimizations improve performance. Increasing ALU execution latency does not impact performance so long as it is lower than communication latencies.

7.1 Methodology

We evaluate SOSA using a custom, event-driven simulator and use results from simulating smaller systems to extrapolate the behavior of larger systems. 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 quantum". The latency of all activity in the node is a multiple of this time quantum. Experimental devices are expected to operate at frequencies exceeding 100 GHz [4] with demonstrated frequencies over 10GHz [33] (time quantum of 0.1 ns), and asynchronous handshakes at high speeds have been demonstrated for high bandwidth crossbar networks [21]. We expect SOSA's performance to scale with device performance, but assume a conservative time quantum of 1 nanosecond to avoid over-estimating performance due to aggressive technological parameters. We list our default simulation parameters in Table 2. We use a custom tool that models the growth of DNA nanotubes between nodes to generate network topologies.

We compare the performance of SOSA to a Pentium 4 (P4) (3 GHz, 1MB L2, 1 GB RAM), an ideal out-of-order superscalar (I-SS) (128-wide, 8k ROB, 1-cycle memory latency), an ideal 16-way CMP (16-CMP) (obtained by linearly scaling performance of the I-SS) and an ideal implementation of SOSA (I-SOSA) that uses the same instruction set, but assumes unit instruction execution latencies, and no communication overhead. Table 3 lists the parameters used to simulate the I-SS with SimpleScalar [2].

Table 4 contains brief descriptions of the test programs, the broad application classes they fall under, and the number of PEs required by SOSA to run one instance of a program. For all programs other than the encryption algorithms, we configure the system as a single cell with the necessary 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 gcc to generate PISA binaries for simplescalar (flags: -O3) and Intel's C Compiler (icc, flags: -O3 -fast -tpp7) for the P4 since optimized icc binaries outperform optimized gcc binaries. We test several versions of matrix multiplication from [31] and identify the best version for the P4 (naïve version with three nested loops, since icc vectorizes loops for the SSE units) and I-SS (static loop unrolling). For sorting, we use an implementation of quicksort. For SOSA each program is hand-optimized (e.g., loop unrolling, code re-organization). The SOSA code for matrix multiplication and the image filters assumes data is in place before execution begins. However, this overhead forms only a small fraction of total execution time and can be reduced by exploiting multiple anchors in the system. The other workloads explicitly account for I/O overheads. The running times of programs do not include system configuration time (which is proportional to the number of nodes in the system). To estimate SOSA performance for configurations with more than 16K PEs, we use simple linear extrapolation (simulating a 256x256 matrix multiplication on a 3 GHz P4 with 32 GB RAM takes ~50 days, which is impractical for data collection purposes). To validate the extrapolations we compare extrapolated run times to simulated run times for large configurations (8K-16K PEs).

TABLE 4. Benchmark Descriptions

Application Class	Benchmark Description
Scientific	*Multiply* integer *NxN matrices* (N^2 PEs)
Image Processing (Filters)	Apply a *generic 3x3 filter* on an NxN image (N^2 PEs)
	Apply a separable *gaussian filter* on an NxN image (N^2 PEs)
	Apply a *median filter* to an NxN image to reduce noise (N^2 PEs)
General Purpose	Odd-Even Transposition *Sort* [19]- Parallel sort with nearest neighbor communication (N PEs for sorting N numbers)
Cryptography	Tiny Encryption Algorithm *(TEA)* - Simple encryption algorithm used in the XBox (64 PEs)
	eXtended TEA (*XTEA*) - Eliminates known vulnerabilities in TEA (64 PEs)
Search	*Search* a database for a match with an input 32 bit string (O(N) PEs for N strings)
Bin-Packing	Pipelined version of *bin-packing* with first-fit heuristic (N PEs for N bins)

7.2 Results

We now examine the performance of applications on SOSA with no defects. 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). We divide our evaluation in two parts based on the performance metric being used (execution time or throughput).

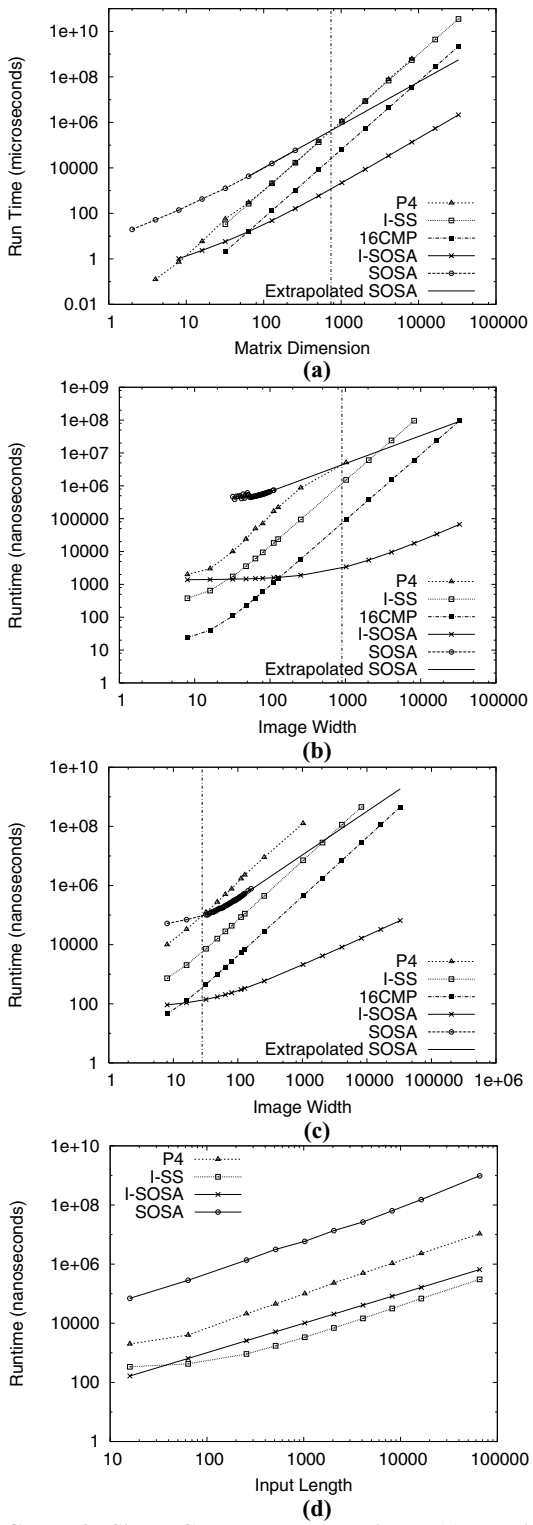

FIGURE 8. Single Cell Program Runtimes: (a) Matrix Multiplication, (b) Gaussian Filter, (c) Median Filter and (d) Sort. The vertical line denotes the input size beyond which SOSA does better than the Pentium 4

Execution Time. 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.

Inspecting the main loop body for matrix multiplication in fFigure 9 (optimizations are omitted to keep the code compact and readable), we see that the primary advantage for SOSA is the simultaneous computation of all products in the N^2 PEs. This allows SOSA to convert the $O(N^3)$ algorithm to $O(N^2)$. Image filters and sorting are reduced from $O(N^2)$ algorithms to O(N).

We plot the running time of matrix multiplication, gaussian filters, median filters and sorting on different architectures in Figure 8, marking the input size beyond which SOSA outperforms the P4 with a vertical line (results for the generic 3x3 filter are qualitatively similar to the gaussian filter, and are skipped due to space constraints). As expected, SOSA does worse than the conventional architectures for small input sizes, but matches and overtakes them as input size increases (except for median filter and sort). The P4 matches the I-SS on matrix multiplication for two reasons: a) the P4 makes use of its SSE units, and b) I-SS only achieves an IPC of 9. The P4 performs much worse without the SSE units.

The performance of the median filter and sort algorithms is limited by their dependence on predicated instructions which serialize execution in a PE. While the number of predicated instructions in the median filter is fixed (independent of input size), for sort it scales with input size. For the median filter, SOSA is able to match the performance of the uniprocessors, but not the ideal 16-CMP (for image sizes up to 16Kx16K). For sort, the potential speedup on SOSA over quicksort on a single processor (average case) is O(log(N)). However, the overhead introduced by predicated instructions makes it impossible for SOSA to match the performance of the I-SS or P4. Exploring techniques to reduce this overhead is future work. 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.

Throughput. 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. For example, if we assume an area of $100mm^2$ (approximately the area of a P4 in 90nm

```
  ; 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->LSB, LSB->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 (i is iteration count)
ADD R6,R6,R5    ;Accumulate partial sum
CPREG R6,R5     ;Copy R6 to R5
SHIFTMLPE R5    ;Repeat i*2 times
  ; End Repeat
ADD R6,R6,R5    ;Final add
  ; Align rows of matrix A for next set
  ; of multiplies (Repeat N times)
SHIFTMLPE R4    ;Move A 'N' PEs to the left
  ; Move Result
CPREG R8,R9     ;if R8==1, this PE holds the first
                ;row/column element, move 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
```

FIGURE 9. Matrix Multiply: Assembly Code (no unrolling)

TABLE 5. TEA Throughput for different architectures

Architecture	Encryptions/sec
P4 @ 3 GHz (100mm^2)	3.9 M/sec
I-SS	73.62 M/sec
16-CMP	1180 M/sec
SOSA (1 cell ~ 0.019mm^2)	0.175 M/sec
I-SOSA (1 cell)	27.7 M/sec
SOSA (5400 cells, 100 mm^2)	940 M/sec
I-SOSA(5400 cells)	72300 M/sec

CMOS), we can configure over 5,000 cells (assuming an average inter-node gap of 1µm) that each perform an 8x8 matrix multiplication and achieve much higher throughput than the P4 or the I-SS.

TEA [37] and XTEA [24] are two simple encryption algorithms developed at the University of Cambridge that use a combination of shift, add and xor operations to encrypt 64 bit blocks of data with a 128-bit key, with XTEA requiring more operations per iteration to achieve better cryptographic security. We implement pipelined versions of both algorithms that require 64 PEs (corresponding to 64 encryption iterations) in a cell. Due to their requirement of fixed sized cells, these algorithms are well suited for the high-throughput, multiple cell configuration.

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 can perform 175,000 TEA encryptions per second and 170,000 XTEA encryptions per second. Table 5 compares the performance of TEA on different 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 with devices switching 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.

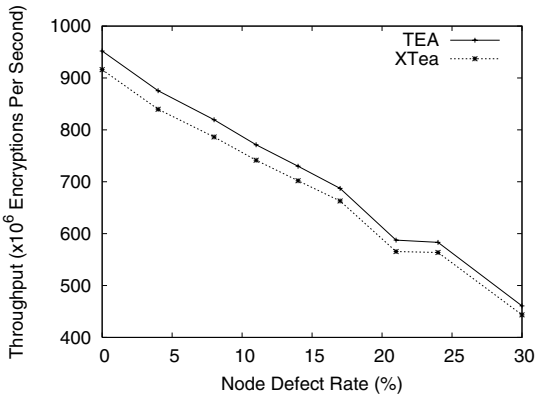

FIGURE 10. TEA/XTEA: Graceful degradation of throughput with increasing node defect rate

We have implemented pipelined versions of searching and bin-packing algorithms in SOSA to maximize throughput. Our implementation of search achieves about 10 billion comparisons per second on SOSA while using the same area as a P4 (the P4, I-SS and 16-CMP achieve about 0.5, 2 and 32 billion comparisons per second respectively). We see qualitatively (not quantitatively) similar results for bin-packing. SOSA's ability to exploit data parallelism in these workloads helps it outperform conventional architectures.

7.3 Defect Tolerance

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[1]. First, we examine the effect of defects on the throughput of a system configured into multiple cells. If we keep the total system area constant (100mm^2), as node defect rates increase we are able to configure fewer cells, resulting in reduced throughput. Figure 10 plots the throughput for TEA and XTEA, 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 results in network partitions with insufficient functioning nodes in each partition 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 has 30% more nodes (24,000 total nodes) than the minimum needed for a 32x32 matrix multiply. Figure 11 shows the running time for 32x32 matrix multiplication as we increase the number of defective nodes from 0% to 20%. We see that the running time increases by about 8% (compared to a case with no defects), primarily because the average length of PEs increases. We do not present 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 work. Though the defect tolerance capabilities of the RPF algorithm have been demonstrated before, our experiments show that the ability to

1. Our generated topologies include link defects but these only have an indirect (and minor) effect on performance. Performance is affected if the average number of links per node is less than 2. We find that nodes have 3.2 active links on average.

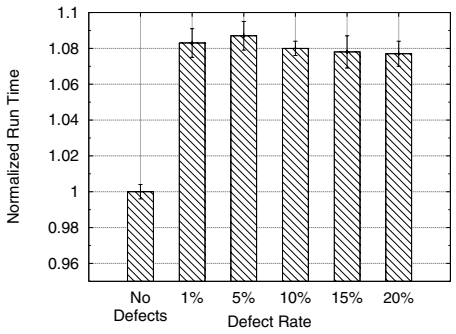

FIGURE 11. Matrix Multiplication: Effect of defects on run time

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

7.4 Result 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 devices at a lower switching speed. A scaled up version of the system can outperform an ideal 16-way CMP. The results also highlight SOSA's flexibility in configuring independent cells to improve system utilization and throughput. SOSA provides higher throughput than the P4 and I-SS 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 promise to deliver.

8 Limitations and Future Work

Our performance evaluation reinforces the common knowledge that a high computation to communication ratio is critical for achieving good performance, particularly on SOSA due to its low bandwidth and high communication latencies. SOSA is likely to achieve good performance on pipelined implementations of programs that require high throughput, or programs that require little inter-PE communication, nearest neighbor communication or regular and unidirectional dataflow. 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 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.

There are a number of avenues for further research. We plan to extend SOSA to speed up floating point operations, exploit multiple anchors to increase system I/O bandwidth, and to handle transient faults through redundant execution or by extending PEs to perform simple checksum/parity computations. We are also looking at extending the software toolchain to explore compiler optimizations. Other open research areas include modifications to the configuration mechanism to exploit unused links to improve I/O bandwidth, configuring nodes for specific functionality (e.g., floating point or storage), using SOSA as an add-on to a conventional core to improve performance on data parallel workloads, and creating hybrid cores that mix CMOS and self-assembled devices.

As self-assembly technology matures, some of the severe fabrication limitations may be removed. The performance of I-SOSA provides an upper bound of SOSA performance, assuming a time quantum 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, if we can manufacture larger nodes, it might be possible to fit a full PE in one node, or to build more complex transceivers to achieve better network connectivity [30]. As emerging device technologies improve, it may be possible to operate them at higher speeds (causing a potential increase in power consumption). It is important to note that while we assume DNA-based self-assembly as the fabrication process, SOSA is applicable to any manufacturing technique that results in high defect rates and a loss of precise control during parts of the fabrication process.

9 Related Work

There is a large body of research on building computing systems with similar goals, but differs primarily in the granularity of the basic computational blocks used to form the system. SOSA must use very simple computational nodes due to fabrication constraints. In this section, we focus on closely related work applicable to emerging technologies. The decoupled array multiprocessor (DAMP) [10] and the nano-scale active network architecture (NANA) [29] use DNA-based self-assembly of nano-electronic devices. The DAMP exploits data parallelism, but it is not capable of efficient data exchange between processing elements, limiting it to embarrassingly parallel problems. SOSA uses more sophisticated self-organization and achieves better performance than NANA since it has lower communication overheads, better node utilization and uses a single node type.

Researchers have developed FPGA-based reconfigurable architectures [6,13] that extract a system-level defect map, and use this external map to configure the system, while isolating defective regions. The key difference is that SOSA configures higher level logic blocks (nodes as opposed to gates in an FPGA) and does not require an external defect map. This is critical since we have little information about the physical network topology. Researchers have proposed various voting and redundancy schemes to deal with defects, including triple modular redundancy (TMR) [22], N-modular redundancy [36], NAND multiplexing and hot/cold sparing [8] (particularly in the context of molecular electronic systems). The defect tolerance scheme used in this paper does not rely on redundant computation but isolates defectives regions in the system. There has been extensive research on designing and building vector [5,12] and SIMD machines [18,20], including the "Cell" processor [14]. The cell processor 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.

10 Conclusions

With the expected rise in defect rates as device sizes shrink, defect tolerance will be a critical requirement for future system

architectures. These increasing defect rates will contribute directly to the exponentially increasing cost of top-down manufacturing. The use of bottom-up techniques like self-assembly will help lower costs but may also result in higher defect rates and a loss of precise control over the manufacturing process. This makes it imperative for architects to develop defect tolerant architectures to exploit the full potential of future nanoscale devices. This paper presents 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 that may provide inexpensive terascale integration.

Acknowledgements

We thank the anonymous reviewers and members of TROIKA for comments and suggestions that improved this work. This work is supported by an NSF ITR grant CCR-0326157, the Duke University Provost's Common Fund, AFRL contract FA8750-05-2-0018, and equipment donations from IBM and Intel.

References

[1] H. Abelson et al. Amorphous Computing. *Communications of the ACM*, 43(5):74–82, 2000.

[2] T. Austin et al. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] A. Bachtold et al. Logic Circuits with Carbon Nanotube Transistors. *Science*, 294:1317–1320, Nov. 2001.

[4] P. J. Burke. Carbon Nanotube Devices for GHz to THz Applications. *Proc. of SPIE*, 5593:52–61, 2004.

[5] S. Ciricescu et al. The Reconfigurable Streaming Vector Processor (RSVP). In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[6] W. B. Culbertson et al. The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In *Proc. of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov. 1996.

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

[8] A. DeHon. Array-Based Architecture for Molecular Electronics. In *Proc. of the First Workshop on Non-Silicon Computation (NSC-1)*, Feb. 2002.

[9] C. Dwyer et al. DNA Functionalized Single-Walled Carbon Nanotubes. *Nanotechnology*, 13:601–604, 2002.

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

[11] C. Dwyer et al. Semi-empirical SPICE Models for Carbon Nanotube FET Logic. In *Proc. of the Fourth IEEE Conference on Nanotechnology*, Aug. 2004.

[12] R. Espasa et al. Tarantula: A Vector Extension to the Alpha Architecture. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[13] S. C. Goldstein and M. Budiu. NanoFabrics: Spatial Computing Using Molecular Electronics. In *Proc. of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[14] H. Hofstee. Power Efficient Processor Architecture and the Cell Processor. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Feb. 2005.

[15] Y. Huang et al. Logic Gates and Computation from Assembled Nanowire Building Blocks. *Science*, 294:1313–1317, Nov. 2001.

[16] C. Intanagonwiwat et al. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[17] International Technology Roadmap for Semiconductors, 2005.

[18] U. Kapasi et al. The Imagine Stream Processor. In *Proc. 2002 IEEE International Conference on Computer Design*, pages 282–288, Sept. 2002.

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

[20] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proc. of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

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

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

[23] K. Mai et al. Smart Memories: A Modular Reconfigurable Architecture. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[24] R. Needham and D. Wheeler. Tea Extensions. Technical report, Computer Laboratory, University of Cambridge, Oct. 1997.

[25] S. H. Park et al. Finite-size, Fully-Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures. *Angewandte Chemie*, 45:735–739, Jan. 2006.

[26] J. P. Patwardhan et al. Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*, pages 344–358, Apr. 2004.

[27] J. P. Patwardhan et al. 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.

[28] J. P. Patwardhan et al. 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.

[29] J. P. Patwardhan et al. NANA: A Nano-scale Active Network Architecture. *ACM Journal on Emerging Technologies in Computing Systems*, 2(1):1–30, 2006.

[30] J. P. Patwardhan et al. Self-Assembled Networks: Control vs. Complexity. In *1st International Conference on Nano-Networks*, Sept. 2006.

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

[32] B. H. Robinson and N. C. Seeman. The design of a biochop: a self-assembling molecular-scale memory device. *Protein Engineering*, 1:295–300, Aug. 1987.

[33] S. Rosenblatt et al. Mixing at 50GHz using a Single-Walled Carbon Nanotube Transistor. *Applied Physics Letters*, 87:153111, Oct. 2005.

[34] M. D. Schroeder et al. Autonet: A High-speed, Self-Configuring Local Area Network Using Point to Point Links. *IEEE Journal on Selected Areas in Communications*, 9(8), Oct. 1991.

[35] K. Skinner et al. Nanowire Transistors, Gate Electrodes, and Their Directed Self-Assembly. In *The 72nd Southeastern Section of the American Physical Society (SESAPS)*, Nov. 2005.

[36] 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.

[37] D. Wheeler and R. Needham. TEA: A Tiny Encryption Algorithm. In *Fast Software Encryption: Second International Workshop*, Dec. 1994.

[38] E. Winfree et al. Design and Self-Assembly of Two-Dimensional DNA Crystals. *Nature*, 394:539, 1998.

[39] H. Yan et al. DNA Templated Self-Assembly of Protein Arrays and Highly Conductive Nanowires. *Science*, 301(5641):1882–1884, Sept. 2003.