A Self-Organizing Defect Tolerant SIMD Architecture

JAIDEV PATWARDHAN
MIPS Technologies
CHRIS DWYER
Department of Electrical and Computer Engineering, Duke University
and
ALVIN R. LEBECK
Department of Computer Science, Duke University

The continual decrease in transistor size (through either scaled CMOS or emerging nanotechnologies) promises to usher in an era of tera to peta-scale integration but with increasing defects. Regardless of fabrication methodology (top-down or bottom-up), defect-tolerant architectures are necessary to exploit the full potential of future increased device densities.

This article explores a defect-tolerant SIMD architecture (SOSA) that self-organizes a large number of limited capability nodes with high defect rates into SIMD processing elements. Simulation results show that SOSA matches or exceeds the performance of conventional systems for moderate to large problems, but with lower power density.

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 Archtitectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Self-organizing, SIMD, data parallel, bit-serial, defect tolerance, DNA, nanocomputing

ACM Reference Format:

Patwardhan, J., Dwyer, C., and Lebeck, A. R. 2007. Self-organizing defect tolerant SIMD architecture. ACM J. Emerg. Technol. Comput. Syst. 3, 2, Article 10 (July 2007), 33 pages. DOI = 10.1145/1265949.1265956 http://doi.acm.org/10.1145/1265949.1265956

This article is an extended version of a previous published paper in ASPLOS XII 2006.
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.
Author's address: A. R. Lebeck, Department of Electrical and Computer Engineering, Duke University, Durham, NC; email: alvy@cs.duke.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1550-4832/2007/07-ART10 $5.00. DOI 10.1145/1265949.1265956 http://doi.acm.org/10.1145/1265949.1265956

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 [ITRS 2005].

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 ($\sim 10^9 - 10^{12}$) of small nodes (computational circuits with $\sim 10^4$ transistors) that can be linked together using self-assembly. This produces a random network of nodes, due to the lack of control over the 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 article 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 article are:

—adapting self-organization methods to computer architectures,

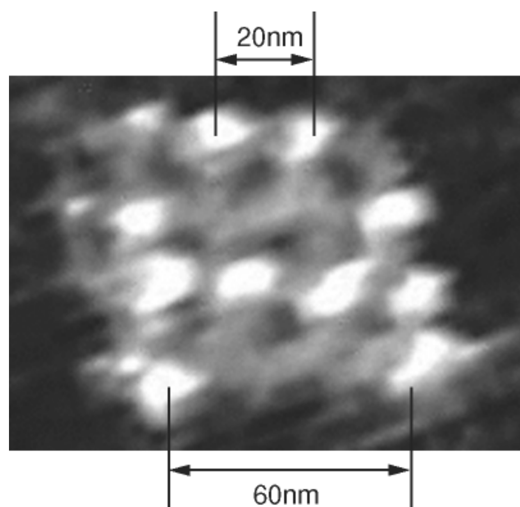


Fig. 1. Patterned DNA [Park et al. 2006].

— designing a node that balances fabrication constraints with functionality needed to communicate, compute, and self-organize and,
— demonstrating these capabilities by composing a high-performance, defect-tolerant SIMD architecture from a random network of nodes.

The rest of this article 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. We conclude in Section 10.

2. DNA-BASED SELF-ASSEMBLED NANOSCALE SYSTEMS AND THE ARCHITECTURAL IMPLICATIONS

Self-assembly of nano-electronic devices has the potential to emerge as a lower cost alternative to top-down manufacturing. DNA-based self-assembly [Robinson and Seeman 1987] uses the precise binding rules of DNA with nanoscale devices to build computing systems. We assume a proposed assembly process [Patwardhan et al. 2004] to place electronic circuits on a DNA grid [Winfree et al. 1998; Yan et al. 2003]. 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 [Bachtold et al. 2001; Dwyer et al. 2002] or silicon nanowires [Huang et al. 2001]) 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 20 nm pitch, on a DNA grid [Dwyer et al. 2005; Park et al. 2006] and (2) DNA-guided self-assembly of nanowire transistors [Skinner et al. 2005]. Figure 1 shows an atomic force microscope image of the letter A patterned on a

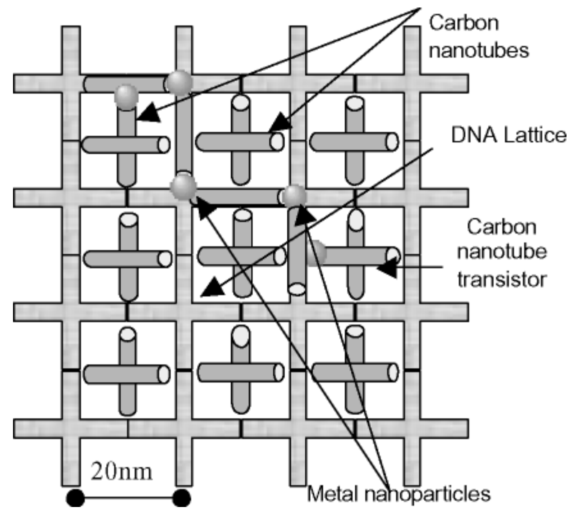


Fig. 2. DNA Lattice with transistors and interconnect.

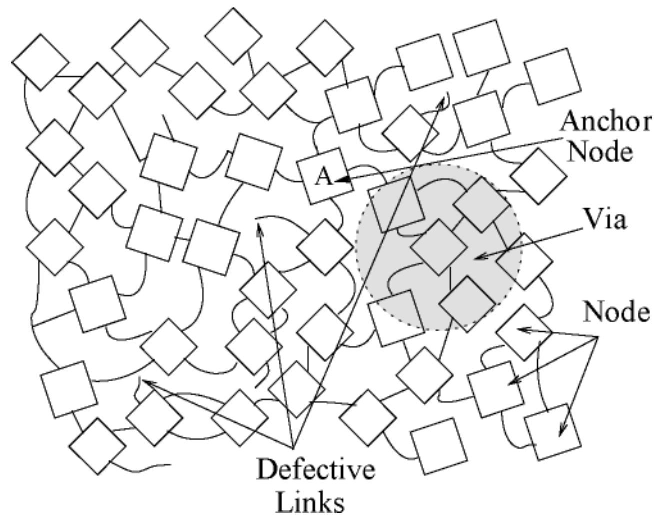


Fig. 3. Self-assembled network of nodes.

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 4 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 the construction of many nodes ($\sim 10^9 - 10^{12}$) that may be linked together by self-assembled conducting nanowires [Yan et al. 2003]. The proposed self-assembly

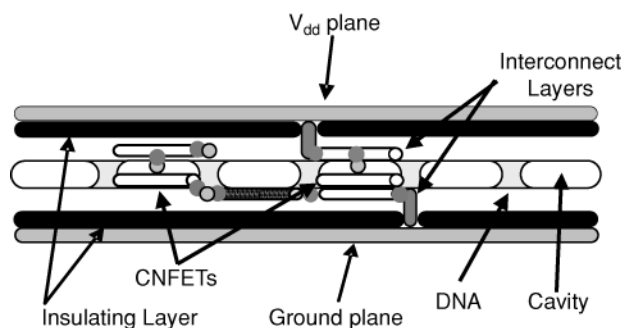


Fig. 4. Lattice with two levels of interconnect and connections to vdd and ground.

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 3 shows a small network of nodes, including regions with defective links, and a via/anchor. In the rest of the article, we use the term *anchor* to refer to an anchor node/via pair.

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 [Schroeder et al. 1991; Abelson et al. 2000], routing and resiliency in the face of defects [Abelson et al. 2000; Intanagonwiwat et al. 2000], and the ability to compose complex computational units from simpler blocks [Mai et al. 2000], but we face added challenges because of the extremely limited computational capabilities available in nodes. Our previous work, the nanoscale active network architecture (NANA) [Patwardhan et al. 2006], 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 article 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 and that it executes conditional branches, such as loops, that cannot be implemented with predication.

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 internode communication (Section 4.3), highlighting the trade-offs between functionality, performance, and circuit size (Section 4.4).

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 1-bit 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 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 (see Section 7.3.3). 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 (see Section 7.3.4).

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 runtime 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 runtime control logic has three control registers to hold each of three microinstructions 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 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

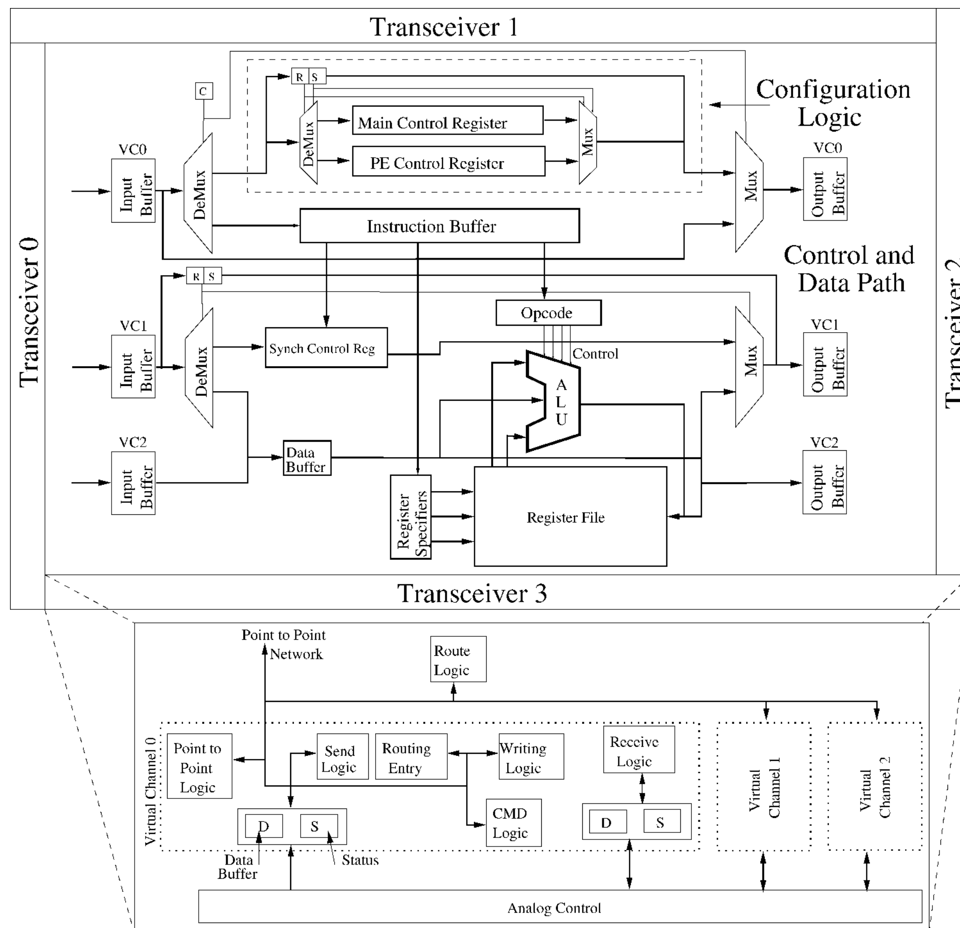


Fig. 5. Node floorplan.

microinstructions 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 Internode 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 except the transceivers. 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 depicts 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 $3\mu m \times 3\mu m$ node. Recent work [Yan et al. 2003] has shown that it should be possible to manufacture DNA grids of this size. The transistor overhead is large, but it enables support for defect tolerance.

To estimate system power consumption, we use the energy delay product for carbon nanotube field effect transistor (CNFET) circuits [Dwyer et al. 2004]. Based on a switching speed of 1ns (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 $0.775\mu W$ 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 ($0.775\mu W$), we get a maximum power density of $6.5 W/cm^2$, with a node activity factor of 0.88. As a point of comparison, this is much less than the power densities of current CMOS processors, which are greater than 75 W/cm^2. However, this comparison is between different technologies. Nonetheless, our estimate is pessimistic since the node 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 [Dalal and Metcalfe 1978; Patwardhan et al. 2005] 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 a built-in self-test to implement fail-stop behavior [Patwardhan et al. 2006] 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 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

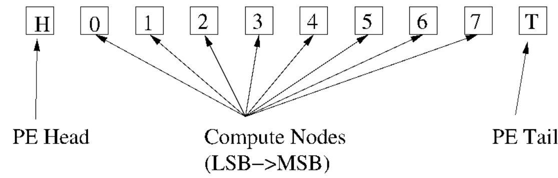


Fig. 6. PE layout.

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 the 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 3 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).

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 postconfiguration step deconfigures partial PEs, a PE that crosses a length threshold can be rejected by starting a new PE without creating a tail node. By varying the PE length threshold and determining the corresponding effect on performance, 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. Section 7.3 provides further details on this evaluation.

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).

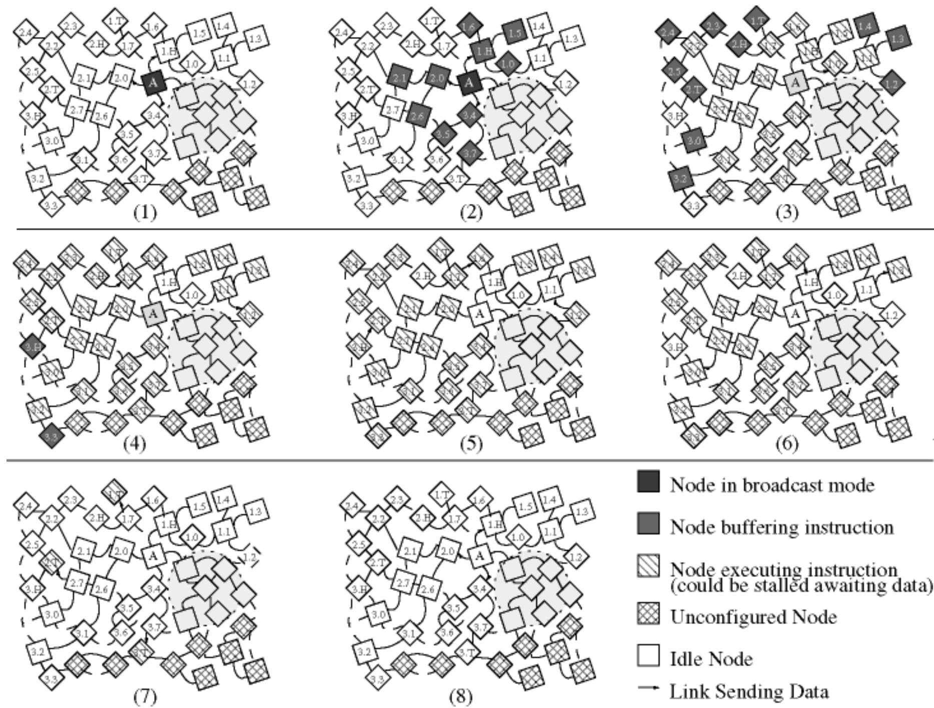


Fig. 7. Instruction execution in a random network with three configured PEs. The via is shown to cover multiple nodes which are rendered unusable. The via is connected to the PEs through the anchor node (A).

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 I shows 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 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.

Table I. 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, PSHIFTLM	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
Bit PE-Shift	BITSHIFTMLPE, BITSHIFTLMPE, MVSTCURRPE, MVSTNEXTPE	Shift single bits between PEs
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
Predicate Modifying	PSet, PSetEven, PSetOdd, PInv	Predicate modifying instructions
Fused	CPSHIFTML,CPSHIFTM	Copies source into destination, and performs a shift on the destination
Signal	SIG_CTRL	Send signal to external controller

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 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 microinstruction 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.

Nonpredicated instructions can be executed independently by nodes of a PE if there are no interbit 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 microinstructions that form the ADD on VC0 (step (1)). As each node receives the microinstructions, it buffers them (step (2)) and waits for the synchronization microinstruction 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 (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 internode 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

Table II. SOSA System Parameters

Parameter	Value	Parameter	Value	Parameter	Value
Register File	16 entry, 2-bits/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

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 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. Section 7.3 explores the sensitivity of SOSA to changes in various design parameters. We find that the instruction buffer and microinstruction reuse optimizations improve performance. Increasing ALU execution latency does not impact performance provided the total execution latency is less than communication latencies. We then show that SOSA can tolerate high node defect rates (Section 7.4). 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.

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 internode 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 100GHz [Burke 2004] with demonstrated frequencies over 10GHz [Rosenblatt et al. 2005] (time quantum of 0.1ns), and asynchronous handshakes at high speeds have been demonstrated for high-bandwidth crossbar networks [Lines 2004]. We expect SOSA's performance to scale with device performance but assume a conservative time quantum of 1 nanosecond to avoid overestimating performance due to aggressive technological parameters. We list our default simulation parameters in Table II. 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) (3GHz, 1MB L2, 1GB 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

Table III. 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, 1 cycle latency	Frequency	10 GHz	Memory Ports	128

Table IV. Benchmark Descriptions

Application Class	Benchmark Description
Scientific	*Multiply* integer *N* × *N matrices* (N^2 PEs)
Image Processing	Apply a *generic 3 × 3 filter* on an N × N image (N^2 PEs)
	Apply a *separable Gaussian filter* on an N × N image (N^2 PEs)
	Apply a *median filter* on an N × N image to reduce noise (N^2 PEs)
General Purpose	Odd-Even Transposition *Sort* [Knuth 1973]—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)

uses the same instruction set, but assumes unit instruction execution latencies, and no communication overhead. Table III lists the parameters used to simulate the I-SS with SimpleScalar [Austin et al. 2002].

Table IV 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 [Runnels and Scarlata 1995] 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 reorganization). 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 extrapolation (simulating a 256×256 matrix multiplication on a 3GHz P4 with 32GB RAM takes $\sim$50 days, which is impractical for data collection purposes). For matrix multiplication, we extrapolate running time using $R(N) = 3.8^*R(N/2)$, where R(N) is the running time for an $N \times N$ matrix. For generic and median filters, we use $R(N) = 1.85R(N/2)$, and for the separable Gaussian filter, we use $R(N) = 2.77R(N/2)$. To validate the extrapolations, we compare extrapolated runtimes to simulated runtimes for large configurations (8K–16K PEs). We do not need to extrapolate for sorting since we are able to simulate systems with up to 16K PEs (and hence, sort 16K numbers). The other workloads are throughput oriented, and do not require extrapolation.

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 into two parts based on the performance metric being used (execution time or throughput).

7.2.1 *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 overhead 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 Figure 9 (optimizations are omitted to keep the code compact and readable, see Appendix for details on the optimizations we performed to tune matrix multiplication), 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 3×3 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

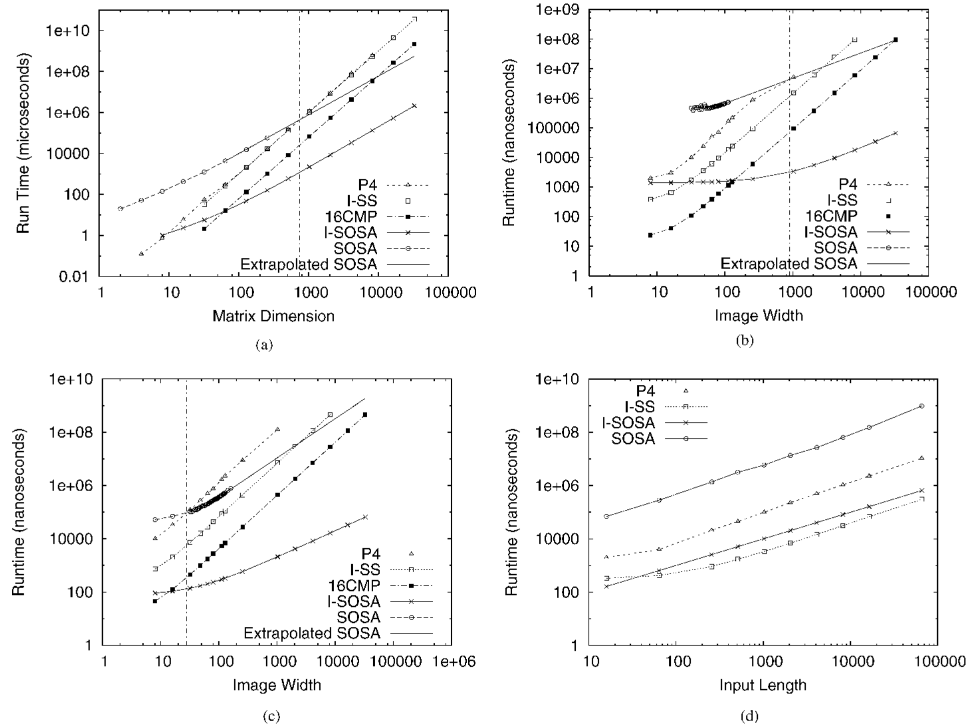


Fig. 8. Single Cell Program Runtimes: (a) Matrix Multiplication, (b) Gaussian Filter, (c) Median Filter and (d) Sort. The vertical line denotes the input size beyond which SOSA does better than the Pentium 4.

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 16K $\times$ 16K). 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.

7.2.2 *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

```
; 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 → MSB (multiply by 2)
PSHIFTML R2, R5   ; Shift MSB → LSB, LSB → Predicate register 5
PRADD R5, R1, R5  ; If predicate register 5 is set, R5=R1+R5
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 → 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 Pred. Reg 8 == 1, this PE holds the
                  ; first row/col element, move to R9
PSHIFTML R9, R6   ; Move that bit into the predicate register 6
PRCPREG R6, R7    ; if predicate register 6 ==1, copy R6 → R7
SHIFTMLPE R7      ; Move R7 one PE to the left
```

Fig. 9. Matrix multiply: assembly code (no unrolling) (see appendix A for more details).

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

TEA [Wheeler and Needham 1994] and XTEA [Needham and Wheeler 1997] 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

Table V. TEA Throughput for Different Architectures

Architecture	Encryptions/sec
P4 @ 3GHz ($100mm^2$)	3.9 M/sec
I-SS	73.62 M/sec
16-CMP	1180 M/sec
SOSA (1 cell $\sim 0.019mm^2$)	0.175 M/sec
I-SOSA (1 cell = 64 PEs)	27.7 M/sec
SOSA (5400 cells $\sim 100mm^2$)	940 M/sec
I-SOSA (5400 cells)	72300 M/sec

both algorithms that require 64 PEs (corresponding to 64 encryption iterations) in a cell (i.e., 1 cell = 64 PEs). 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 V 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 overhead due to simple nodes and limited bandwidth in SOSA.

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 Sensitivity Analysis

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. Next, we examine the effects of various software optimizations (synch reuse and register specifier reuse) that reduce the number of instruction bits broadcast. We then describe the effect of one- or two-bit wide registers on performance. Next, we measure the effect of different compute and communication latencies on performance. We then evaluate the impact of various instruction buffer sizes, and, finally, we examine the effect of various node operating speeds.

7.3.1 *PE Length Optimization.* In Section 5.2, 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.

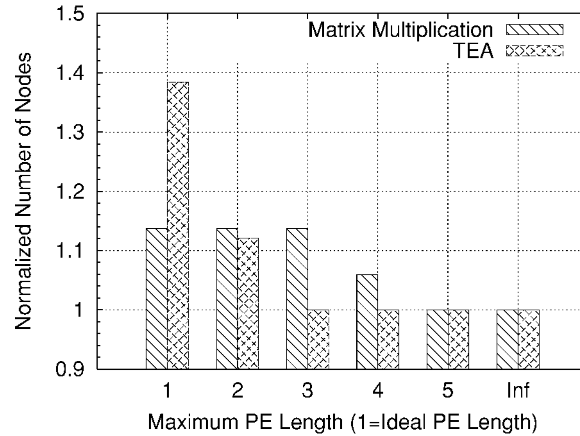


Fig. 10. PE length vs. number of nodes.

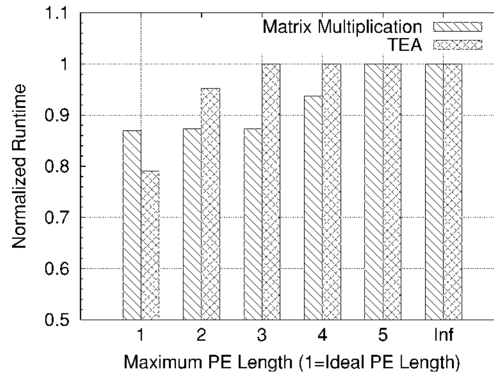


Fig. 11. Effect of PE length optimization on program running time.

In Figure 10, we plot the number of nodes required for 32×32 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 11, 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 5.2.

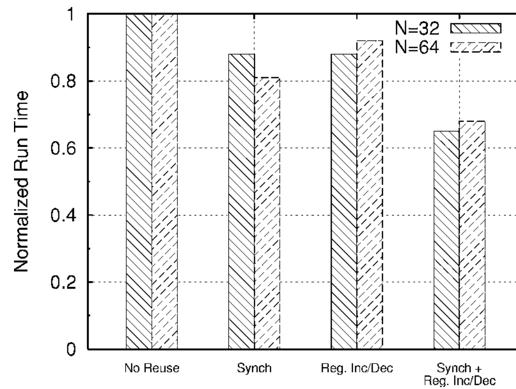
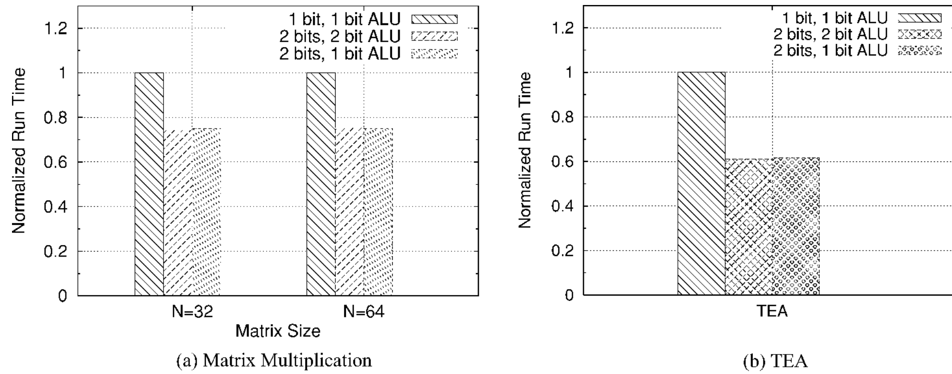


Fig. 12. Effect of instruction reuse.

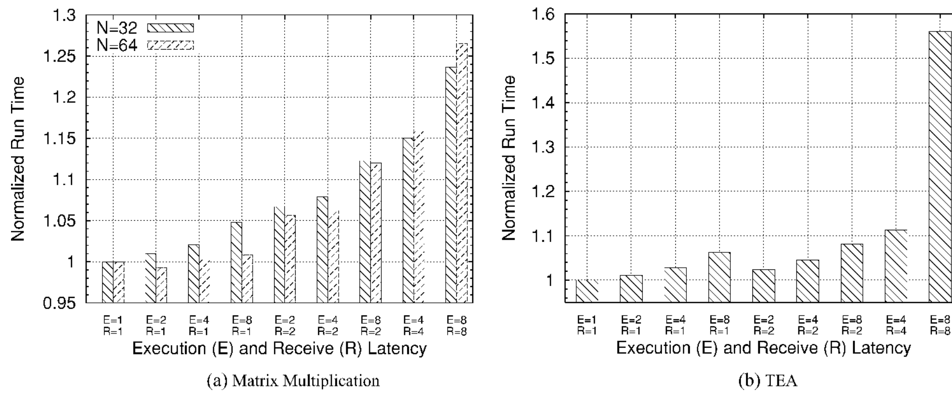
7.3.2 *Instruction Reuse.* The results presented so far show the best performance of the SIMD architecture on matrix multiply with instruction reuse allowed. In this section, we quantify the benefits of instruction reuse using matrix multiplication. Figure 12 plots the runtime of matrix multiply normalized 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 32×32 and 64×64 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 runtime 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 runtime by 12% for a 32×32 matrix, and by 8% for a 64×64 matrix. The larger matrix multiply is affected less because the runtime of the program is dominated by PE-Shifts which do not benefit from the optimization. If we enable both optimizations, runtime 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.

7.3.3 *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 for both cases. We plot the normalized running times for matrix multiplication and TEA in Figure 13. 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



(a) Matrix Multiplication (b) TEA

Fig. 13. Sensitivity to register width.



(a) Matrix Multiplication (b) TEA

Fig. 14. Sensitivity to execution and communication latencies.

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.

7.3.4 *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 14(a) and Figure 14(b), 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.

7.3.5 *Impact of Instruction Buffer Size.* The instruction buffer stores instructions before the node is ready to execute them. It also enables the

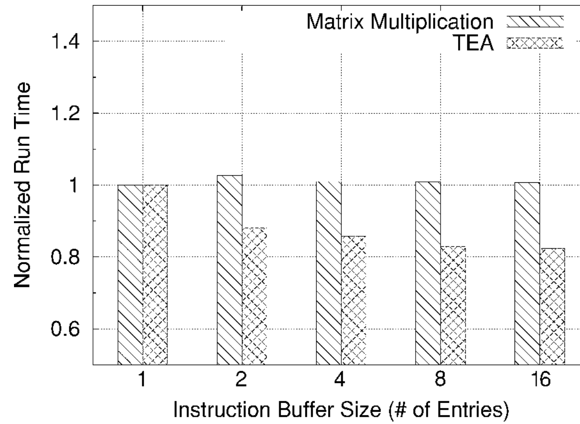


Fig. 15. Effect of instruction buffer size.

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 15, we plot the normalized running time of matrix multiplication(64×64) and TEA as we vary the number of entries in the instruction buffer from 1 to 16 (with instruction reuse optimizations enabled). 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.

7.3.6 *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 ($\sim$1Terahertz) [Burke 2004; Rosenblatt et al. 2005]. In Figure 16, 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 3GHz as a point of comparison. The figure shows that if SOSA could operate with lower values for the time unit, it would achieve runtimes closer to the Pentium 4 for smaller matrix sizes (N = 128, with a time unit of $\sim$100ps).

7.3.7 *Sensitivity 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 internode 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

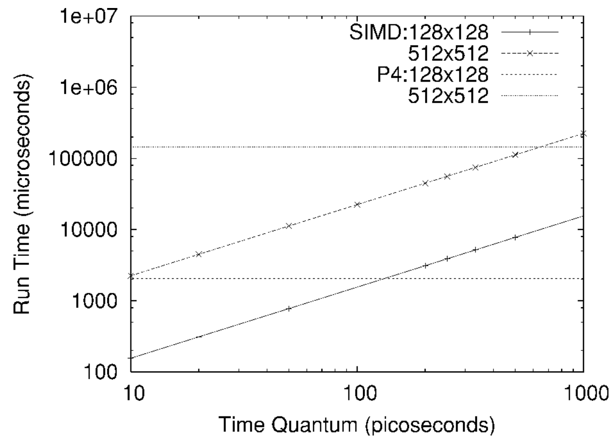


Fig. 16. Effect of operating speed.

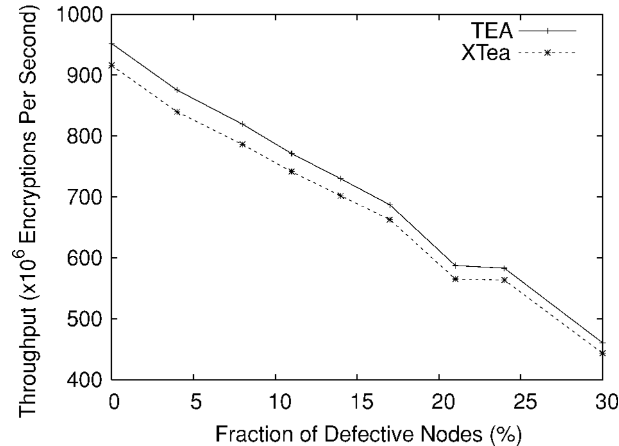


Fig. 17. TEA/XTEA; Graceful degradation of throughput with increasing node defect rate.

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.

7.4 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. 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 17 plots the throughput for TEA and XTEA as node defect rates increase from 0%

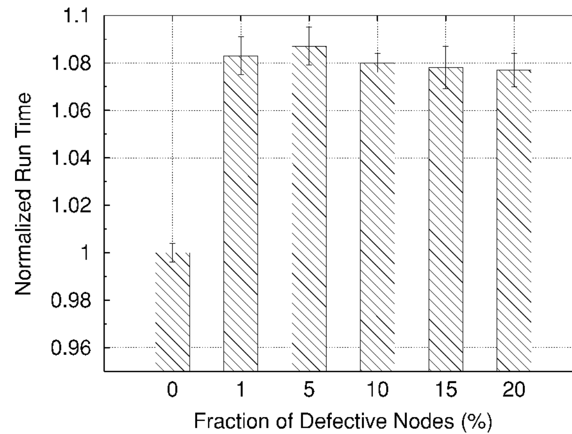


Fig. 18. Matrix multiplication: effect of defects on runtime.

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 32×32 matrix multiply. Figure 18 shows the running time for 32×32 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 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.

7.5 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 severe bandwidth limitation 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 for 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 tool chain to explore compiler optimizations. Other open research areas include modifications to the configuration mechanism to exploit unused links in order 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 in order 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 1ns. 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 [Patwardhan et al. 2006]. 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, which 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) [Dwyer et al. 2004] and the nanoscale active network architecture (NANA) [Patwardhan et al. 2006] use DNA-based self-assembly of nanoelectronic 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 overhead, better node utilization, and uses a single node type.

Researchers have developed FPGA-based reconfigurable architectures [Culbertson et al. 1996; Heath et al. 1998; Goldstein and Budiu 2001] 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) [Lyons and Vanderkulk 1962], N-modular redundancy [von Neumann et al. 1956], NAND multiplexing and hot/cold sparing [DeHon 2003] (particularly in the context of molecular electronic systems). The defect-tolerance scheme used in this article does not rely on redundant computation but isolates defective regions in the system. There has been extensive research on designing and building vector [Espasa et al. 2002; Ciricescu et al. 2003] and SIMD machines [Tucker and Robertson 1988; Ujval et al. 2002, including the Cell processor [Hofstee 2005]. The Cell processor has eight SIMD cores that can be programmed independently unlike the PEs in SOSA. The primary difference 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 in order to exploit the full potential of future nanoscale devices. This article 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.

```
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
```

Fig. 19. Matrix multiplication pseudocode.

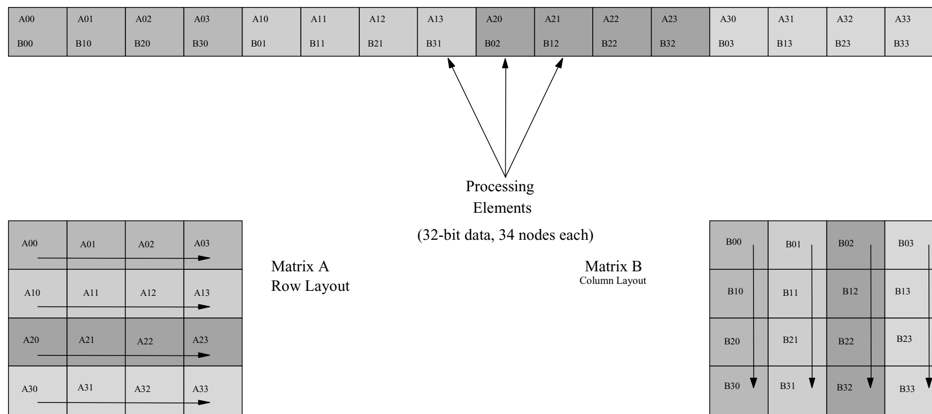


Fig. 20. Matrix layout.

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.

APPENDIX

PROGRAMMING SOSA—MATRIX MULTIPLICATION

We now 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 $N \times N$ matrices A and B, shown in Figure 19.

Since SOSA does not include memory that is 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 20, for two 4×4 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 iteration, and the fourth part

```
; 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))
```

Fig. 21. Matrix multiply assembly code—no optimizations.

moves 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 21 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 these parts. The primary optimizations applied to the third and fourth part include the reuse of microinstructions where possible.

To optimize the accumulate operation, 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 kth iteration, we move the sum 2^k PEs before performing the accumulate. This is depicted in Figure 22 for $N = 16$. It reduces the number of iterations but does not reduce the amount of data that must be communicated. Note that we perform some extra ADD instructions on data elements that do not contribute to the final result. We show the final accumulate code in Figure 23.

We use loop unrolling to optimize the multiplication 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

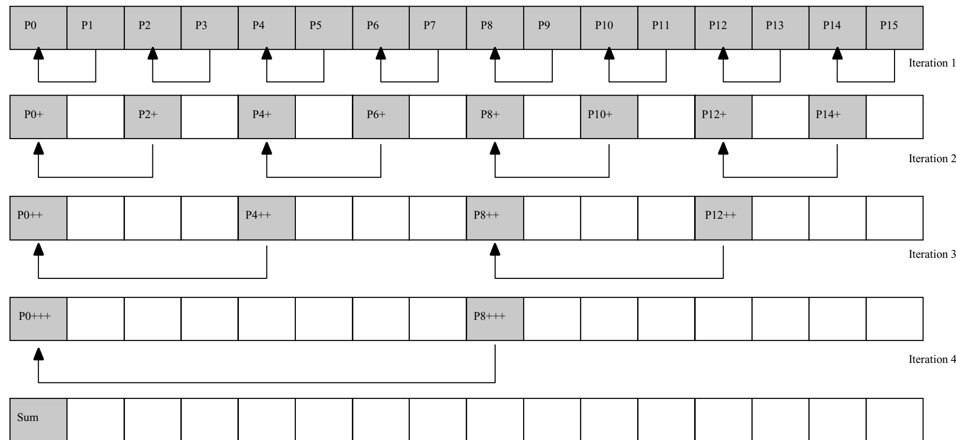


Fig. 22. Logarithmic accumulate.

```
; 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
```

Fig. 23. Logarithmic accumulate—assembly code.

the multiplier. The loop unrolling allows us to reuse microinstructions which helps reduce instruction execution time.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and members of TROIKA for comments and suggestions that improved this work.

REFERENCES

ABELSON, H., ALLEN, D., COORE, D., HANSON, C., HOMSY, G., KNIGHT, T. F., NAGPAL, R., RAUCH, E., SUSSMAN, G. J., AND WEISS, R. 2000. Amorphous computing. *Comm. ACM 43*, 5, 74–82.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35*, 2, 59–67.

BACHTOLD, A., HADLEY, P., NAKANISHI, T., AND DEKKER, C. 2001. Logic circuits with carbon nanotube transistors. *Science 294*, 1317–1320.

BURKE, P. J. 2004. Carbon nanotube devices for GHz to THz applications. *(SPIE) 5593*, 52–61.

CIRICESCU, S., ESSICK, R., LUCAS, B., MAY, P., MOAT, K., NORRIS, J., SCHUETTE, M., AND SAIDI, A. 2003. The reconfigurable streaming vector processor (RSVP). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 141–150.

CULBERTSON, W. B., AMERSON, R., CARTER, R. J., KUEKES, P., AND SNIDER, G. 1996. 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*. 2–10.

DALAL, Y. K. AND METCALFE, R. M. 1978. Reverse path forwarding of broadcast packets. *Commu. ACM 21*, 12, 1040–1048.

DEHON, A. 2003. Array-based architecture for PET-based nanoscale electronics. *IEEE Trans. Nanotech. 2*, 1, 23–32.

DWYER, C., GUTHOLD, M., FALVO, M., WASHBURN, S., SUPERFINE, R., AND ERIE, D. 2002. DNA functionalized single-walled carbon nanotubes. *Nanotech. 13*, 601–604.

DWYER, C., CHEUNG, M., AND SORIN, D. J. 2004. Semi-empirical SPICE models for carbon nanotube FET logic. In *Proceedings of the IEEE Conference on Nanotech.*, 386–388.

DWYER, C., POULTON, J., TAYLOR, R. M., AND VICCI, L. 2004. DNA self-assembled parallel computer architectures. *Nanotech. 15*, 1688–1694.

DWYER, C., PARK, S. H., LABEAN, T. H., AND LEBECK, A. R. 2005. The design and fabrication of a fully addressable 8-tile DNA lattice. *Annual Conference on Foundations of Nanoscience: Self-Assembled Architectures and Devices*, 187–191.

ESPASA, R., ARDANAZ, F., EMER, J., FELIX, S., GAGO, J., GRAMUNT, R., HERNANDEZ, I., JUAN, T., LOWNEY, G., MATTINA, M., AND SEZNEC, A. 2002. Tarantula: A vector extension to the alpha architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 281–292.

GOLDSTEIN, S. C. AND BUDIU, M. 2001. NanoFabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, 178–191.

HEATH, J. R., KUEKES, P. J., SNIDER, G. S., AND WILLIAMS, R. S. 1998. A Defect-tolerant computer architecture: Opportunities for nanotechnology. *Science 280*, 1716–1721.

HOFSTEE, H. P. 2005. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 258–262.

HUANG, Y., DUAN, X., CUI, Y., LAUHON, L. J., KIM, K.-H., AND LIEBER, C. M. 2001. Logic gates and computation from assembled nanowire building blocks. *Science 294*, 1313–1317.

INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, 56–67.

ITRS. 2005. International Technology Roadmap for Semiconductors.

KNUTH, D. E. 1973. *The Art of Computer Programming*. Addison-Wesley.

LINES. A. 2004. Asynchronous interconnect for synchronous SoC design. *IEEE Micro 24*, 1, 32–41.

LYONS, R. E. AND VANDERKULK, W. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM J.*, 200–209.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 161–171.

NEEDHAM, R. AND WHEELER, D. 1997. TEA extensions. Tech. Rep., University of Cambridge, Cambridge, UK.

PARK, S. H., PISTOL, C., AHN, S. J., REIF, J. H., LEBECK, A. R., DWYER, C., AND LABEAN, T. H. 2006. Finite-size, fully-addressable DNA tile lattices formed by hierarchical assembly procedures. *Angewandte Chemie 45*, 735–739.

Patwardhan, J. P., Dwyer, C., Lebeck, A. R., and Sorin, D. J. 2004. Circuit and system architecture for DNA-guided self-assembly of nanoelectronics. *Annual Conference on Foundations of Nanoscience: Self-Assembled Architectures and Devices*, 344–358.

PATWARDHAN, J. P., DWYER, C., LEBECK, A. R., AND SORIN, D. J. 2005. Evaluating the connectivity of self-assembled networks of nanoscale processing elements. In *Proceedings of the IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '05)*, 2.1–2.8.

PATWARDHAN, J. P., DWYER, C., AND LEBECK, A. R. 2006. Self-assembled networks: Control vs. complexity. In *Proceedings of the First International Conference on Nano-Networks (NANONETS)*.

PATWARDHAN, J. P., DWYER, C., AND LEBECK, A. R. 2006. Design and evaluation of fail-stop self-assembled nanoscale processing elements. *IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '06)*.

PATWARDHAN, J. P., DWYER, C., LEBECK, A. R., AND SORIN, D. J. 2006. NANA: A nano-scale active network architecture. *J. Emerg. Technol. Comput. Syst. 2*, 1, 1–31.

PATWARDHAN, J. P., JOHRI, V., DWYER, C., AND LEBECK, A. R. 2006. A defect tolerant self-organizing nanoscale SIMD architecture. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 241–251.

ROBINSON, B. H. AND SEEMAN, N. C. 1987. The design of a biochip: A self-assembling molecular-scale memory device. *Protein Engin. 4*, 1, 295–300.

ROSENBLATT, S., LIN, H., SAZONOVA, V. T. S., AND MCEUEN, P. L. 2005. Mixing at 50GHz using a single-walled carbon nanotube transistor. *Appl. Physics Lett. 87*, 153111.

RUNNELS, L. W. AND SCARLATA, S. F. 1995. Theory and application of fluorescence homotransfer to melittin oligomerization. *Biophysics Journal 69*, 4, 1569–1583.

SCHROEDER, M. D., BIRRELL, A. D., BURROWS, M., MURRAY, H., NEEDHAM, R. M., RODEHEFFER, T. L., SATTERTHWAITE, E. H., AND THACKER, C. P. 1991. Autonet: A high-speed, self-configuring local area network using point to point links. *IEEE J. Selec. Areas Comm. 9*.

SKINNER, K., CARROL, R. L., DWYER, C., AND WASHBURN, S. 2005. Nanowire transistors, gate electrodes, and their directed self-assembly. In *Proceedings of the 72nd Southeastern Section of the American Physical Society (SESAPS)*.

TUCKER, L. AND ROBERTSON, G. 1988. Architecture and applications of the connection machine. *IEEE Comput. 21*, 26–38.

UJVAL, K., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*, 282–288.

VON NEUMANN, J., SHANNON, C. E., AND MCCARTHY, J. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 43–98.

WHEELER, D. AND NEEDHAM, R. 1994. TEA: A tiny encryption algorithm. *Fast Software Encryption: 2nd International Workshop*.

WINFREE, E., LIU, F., WENZLER, L. A., AND SEEMAN, N. C. 1998. Design and self-assembly of two-dimensional DNA crystals. *Nature 394*, 539–544.

YAN, H., PARK, S. H., FINKELSTEIN, G., REIF, J. H., AND LABEAN, T. H. 2003. DNA templated self-assembly of protein arrays and highly conductive nanowires. *Science 301*, 1882–1884.

Received November 2006; revised January 2007; accepted February 2007 by Sally McKee.