Routing in Self-Organizing Nano-Scale Irregular Networks

YANG LIU, CHRIS DWYER, and ALVIN R. LEBECK
Duke University

3

The integration of novel nanotechnologies onto silicon platforms is likely to increase fabrication defects compared with traditional CMOS technologies. Furthermore, the number of nodes connected with these networks makes acquiring a global defect map impractical. As a result, on-chip networks will provide defect tolerance by self-organizing into irregular topologies. In this scenario, simple static routing algorithms based on regular physical topologies, such as meshes, will be inadequate. Additionally, previous routing approaches for irregular networks assume abundant resources and do not apply to this domain of resource-constrained self-organizing nano-scale networks. Consequently, routing algorithms that work in irregular networks with limited resources are needed.

In this article, we explore routing for self-organizing nano-scale irregular networks in the context of a Self-Organizing SIMD Architecture (SOSA). Our approach trades configuration time and a small amount of storage for reduced communication latency. We augment an Euler path-based routing technique for trees to generate static shortest paths between certain pairs of nodes while remaining deadlock free. Simulations of several applications executing on SOSA show our proposed routing algorithm can reduce execution time by 8% to 30%.

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

Additional Key Words and Phrases: Self-organizing, SIMD, data parallel, DNA, nanocomputing

ACM Reference Format:

Liu, Y., Dwyer, C., and Lebeck, A. R. 2010. Routing in self-organizing nano-scale irregular networks. ACM J. Emerg. Technol. Comput. Syst. 6, 1, Article 3 (March 2010), 21 pages.
DOI = 10.1145/1721650.1721653 http://doi.acm.org/10.1145/1721650.1721653

This work is supported in part by the National Science Foundation (CCF-0829911, CCF-0702434), Agilent, IBM.

Authors' addresses: Y. Liu, Department of Computer Science, Duke University, Durham, NC; email: yliu@cs.duke.edu; C. Dwyer, Department of Electrical and Computer Engineering, Duke University, Durham, NC; email: dwyer@ece.duke.edu; A. R. Lebeck, Department of Computer Science, Duke University, Durham, NC; email: alvy@cs.duke.edu.

DOI 10.1145/1721650.1721653 http://doi.acm.org/10.1145/1721650.1721653

1. INTRODUCTION

The trend toward increased functional capabilities on a single chip requires commensurate development of on-chip communication networks. These networks must provide reliable communication without requiring excessive resources. The incorporation of novel nanotechnologies (e.g., nanowires) emerges because the decreasing feature sizes may decrease control over the entire fabrication process (e.g., with self-assembly) and thus increase fabrication defects. As a result, it is increasingly difficult to reliably create interconnects and networks may become highly irregular.

Irregular networks present challenges for programming, system design (e.g., coherence protocols), and developing efficient routing algorithms, particularly for direct networks. It is difficult to map computation and data placement onto irregular topologies. One way to overcome this challenge is to overlay a more conventional regular topology (e.g., tree or ring) on the irregular network. Similarly, by providing the guarantees (e.g., ring ordering) of the overlay network, coherence protocol design can be simplified [Barroso and Dubois 1993; Marty and Hill 2006].

Another challenge is to provide an efficient routing algorithm for a given irregular network. Previous approaches for routing on irregular networks assume the nodes are workstations or complete processors: an environment with abundant resources [Federico and Duato; Chi and Wu 2003]. As we move to the nano-scale domain, the resources dedicated to routing (e.g., routing tables) must be balanced against those required for computation (e.g., functional units, caches, etc.) [Bolotin et al. 2005, 2007]. In a nano-scale network consisting of thousands of nodes, each node has limited resources and many nodes must be grouped together to form a full computational unit. Great challenges are presented since the previous techniques may no longer apply in this environment (i.e., large routing tables are infeasible to simply connect one node to another).

In this article we explore routing for self-organizing nano-scale irregular networks. Our analysis is performed primarily in the context of SOSA [Patwardhan et al. 2006c], a self-organizing nano-scale SIMD architecture, but is broadly applicable to any system that maps logical rings onto irregular networks. SOSA uses a one-time configuration process to organize the nano-scale irregular physical network into a regular physical network (a tree) by disabling certain links, and then maps a regular logical network (a ring) onto the regular physical network. The logical structure of the whole network is a large ring of processing elements and each processing element is a small ring of functional units, called nodes. There are two special nodes, head and tail, in each small ring where data packets enter or leave the processing element.

The head and tail nodes within a single processing element are logically adjacent but may be physically far apart. Operations that rely on the logical adjacency of the head and tail nodes in a ring (e.g., any operation requiring a round-trip of the ring) may incur significant latency due to the physical separation of the nodes. Therefore, we focus on reducing the physical path length between head and tail nodes.

Our approach trades configuration time and a little storage space to reduce communication latency. During the configuration process the shortest physical path between the pair of head and tail nodes in each processing element is discovered and appropriate information is stored within each node to inform routing decisions during execution. For the networks we examine, there is little difference in the head/tail shortest path on the irregular physical network (a graph) versus the regular physical network (a tree). Therefore, our routing optimizations augment existing Euler path routing techniques [Patwardhan et al. 2006c] for trees. In essence, our approach provides express channels between pairs of nodes on an Euler path.

Simulation results show that our routing optimization can reduce overall execution time by 8% to 30% across a set of six benchmark applications. In general, these programs spend at least 20% of their time executing operations that require tail to head communication and obtain at least 8% improvement in execution time. We also provide a proof that our routing optimizations maintain the deadlock-free property of Euler path routing (see Appendix A).

The remainder of this article is as follows. Section 2 provides a brief introduction to SOSA. Section 3 presents statistics of instructions using the logical adjacency of head and tail nodes, and analysis of self-assembled nano-scale networks. Section 4 describes our proposed routing algorithm and Section 5 describes our implementation in detail. Section 6 presents simulation results and analyzes the performance of the proposed routing algorithm. Section 7 summarizes related work and Section 8 concludes.

2. BACKGROUND AND SOSA OVERVIEW

SOSA [Patwardhan et al. 2006c] is a defect-tolerant self-organizing nano-scale SIMD architecture built from a random network of simple computational nodes. SOSA relies on a novel DNA self-assembly process to create small-scale nodes with limited digital compute and storage capabilities.

2.1 DNA Self-Assembled Systems

We assume an assembly process 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 [Patwardhan et al. 2004]. Current self-assembly processes produce limited size DNA grids and thus limit circuit size (e.g., < 10,000 FETs). However, the parallel nature of self-assembly enables constructing 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 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 1 shows a small network of nodes, including

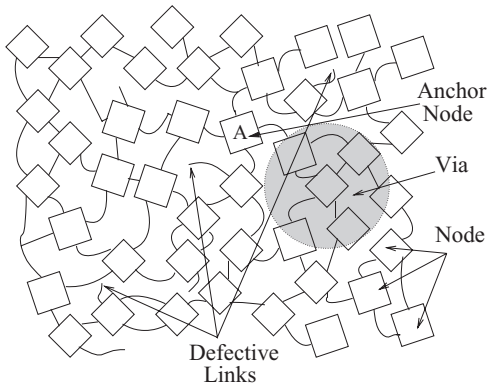


Fig. 1. Self-assembled network of nodes.

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.

2.2 SOSA Overview

The aforesaid self-assembly method results in a large-scale (i.e., billions) arbitrary network composed of limited capability nodes. Networks built using DNA self-assembly techniques have inherent irregularity since it is difficult to control node placement, node orientation, and link growth.

The overall 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 shift instructions to move data between Processing Elements (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, a small register file (32 bits total), a data buffer, 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 virtual channels are used to broadcast instructions (VC0) and route data in opposite directions (VC1 along the depth-first path, VC2 along the reverse depth-first path).

2.2.1 *System Configuration.* The random network of nodes is organized at two levels during a configuration phase. First, since a node is too small to hold an entire 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. However, the first step is to impose a regular topoloty onto the arbitrary graph created through self-assembly. For this, we use a variant of the Reverse Path Forwarding (RPF) algorithm [Dalal and Metcalfe 1978] to map out defective nodes and generate a broadcast tree from the random network. We inject a small packet through an anchor which is broadcast on all the anchor node's active links.

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.

After the broadcast tree is generated, it is traversed in depth-first order (Euler path) and a specified number of sequential nodes are grouped into one Processing Element (PE). The first node in a PE along the Euler path is defined as the head node of the PE and the last node is defined as the tail node. The configuration process maps a logical ring onto the physical broadcast tree and produces a logical abstraction of a set of PEs connected via a ring with a data parallel (SIMD) execution model. Applications can be executed on the network after configuring all PEs.

2.2.2 *Routing Algorithm.* An Euler path-based routing algorithm is used in SOSA for communication both within and between PEs. The Euler path is chosen for two main reasons. First, it is easy to configure. In SOSA, each link is a bidirectional link so the Euler path visits each link exactly twice but in different directions. VC 1 and VC 2 together form an Euler path in SOSA, performing communication in two opposite directions on the logical ring topology. It is simple to configure an Euler circuit by first generating a tree structure and then traversing the tree in depth-first order with the root as both the starting and ending node. Second, when an Euler path is used for routing, each node only needs to store the incoming and outgoing directions for each virtual channel and thus only a few bits are required at each node. In contrast, other methods require significant storage or computational resources and are thus impractical for SOSA.

Although the Euler path-based routing algorithm has its advantages, it is not time efficient. The distance between two nodes along an Euler path can be much longer than the length of the shortest path between them. Consider the system shown in Figure 2 with two 8-node PEs configured from a 16-node network. In this example node 0 is the anchor, PE 0 has head node 0 and tail node 7, while PE 1 has head node 8 and tail node 15. If the Euler path is followed, the path

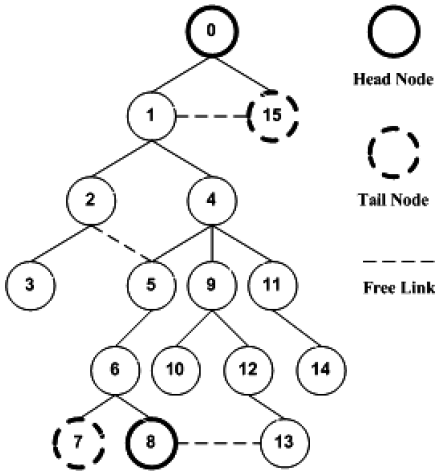


Fig. 2. Euler path and shortest path.

between PE1's tail node (15) and its head node (8) is 15-0-1-4-11-14-11-4-9-12-13-12-9-10-9-4-5-6-8, which has 18 physical hops whereas the shortest path is 15-1-4-5-6-8 which has only 5 physical hops.

This length discrepancy is particularly problematic in SOSA for latency-critical operations, such as predicate setting instructions. For most instructions, information flows from the head node toward the tail node (e.g., the carry bit for addition) of a PE and must visit each intermediate node. However, recall that SOSA is an SIMD processor. To support conditional execution for individual PEs, SOSA provides predicated instructions (i.e., instructions that only execute if a predicate condition is satisfied). Predicated instructions utilize predicate bits stored in the head node of each PE. Predicate bits are set by Predicate Setting Instructions (PSIs) that require propagation of information from the tail node back to the head node. It is PSIs that are most affected by the length of the Euler path routing.

3. NANO-SCALE IRREGULAR NETWORK ANALYSIS

In this section we begin by analyzing the impact of PSIs on execution time and how a new routing algorithm may be able to reduce their latency. We also examine the difference between the shortest path in the graph and the shortest path in the tree formed through the RPF algorithm.

3.1 Execution Time Analysis

The proposed routing algorithm focuses on reducing the physical path length between head and tail nodes to reduce the latency of Predicate Setting Instructions (PSIs) often found on critical control flow paths. PSIs have the highest average latency of any instructions because they generate data packets that must travel from the tail to the head, which is the longest path in a processing element. We analyze the impact of shortening the length of tail-to-head paths on execution time by simulating a nano-scale network of processing elements

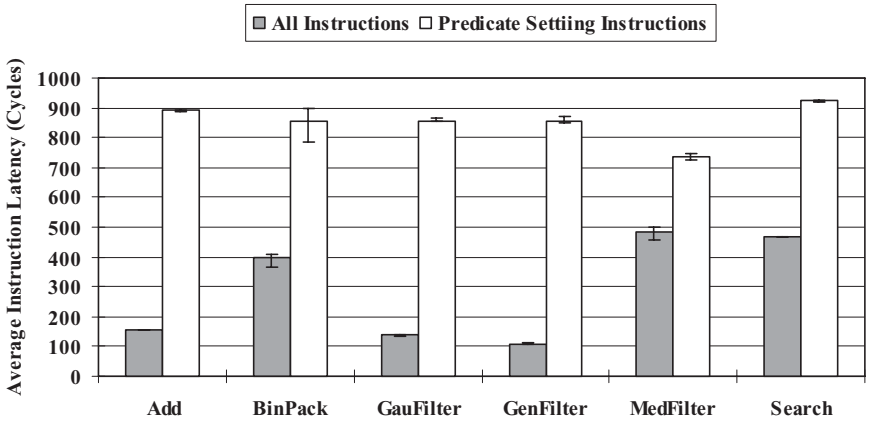


Fig. 3. Average instruction latency.

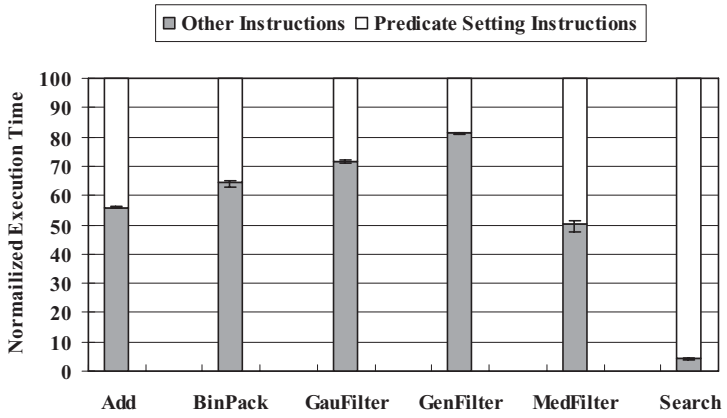


Fig. 4. Normalized execution time.

within SOSA. The study we present is specific to SOSA; however, our analysis is applicable to other systems with request/acknowledge cycles that traverse a network.

We analyze the average latency of PSIs and the fraction of execution time spent on PSIs in 20 randomly generated networks. The networks we use to run the test programs are generated with placement control but without orientation control or interconnect control (see Section 3.2 for details on the network fabrication control), and there are 4500 nodes in each network. The average results of the 20 runs are shown in Figure 3 and Figure 4.

An optimization that reduces the tail and head path length will only reduce the execution time of PSIs since only these instructions require tail-to-head communication. Figure 3 compares the average latency of PSIs with all instructions among six test programs simulated by a timing-accurate SOSA simulator. The SOSA simulator is an event-driven simulator where each activity in a node is an event and its latency is a multiple of "time quantum" defined as the time taken to perform one part of the internode asynchronous

communication handshake [Patwardhan et al. 2006c]. The test programs include *Add*, a software-emulated floating point program for addition and normalization, *BinPack*, a pipelined version of integer bin-packing with a first-fit heuristic using N PEs for N bins, *GauFilter*, *GenFilter*, and *MedFilter* which are image processing programs that apply a generic 3x3 filter, a separable Gaussian filter, and a median filter to an NxN image using N^2 PEs, respectively, and finally *Search*, a program that finds a 32-bit input string in a database using $O(N)$ PEs for N strings.

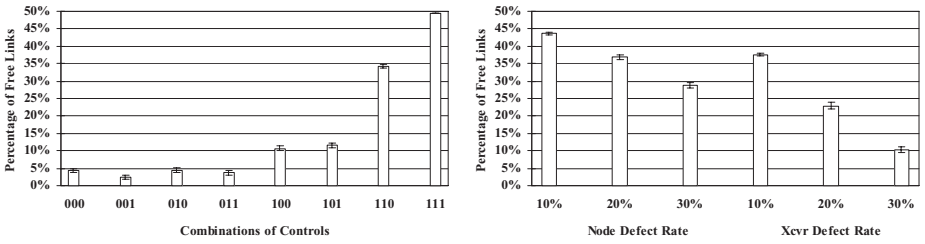
Figure 3 shows that PSIs can have as high as 8.5 times average latency of the overall average instruction latency. Therefore, a small number of PSIs will take a large number of cycles to execute. As shown in Figure 4, PSIs take from 20% to 95% of the total execution time across the test suite. *Add* has instructions to normalize floating point results and *Search* has instructions to compare the input string with strings in the database. Both these two types of instructions are PSIs so a large fraction of execution time is spent on them.

The large fraction of PSIs found in these test programs motivates the need to reduce the path length between tail and head nodes. However, the irregularity of the nano-scale network and limited resource at each node preclude a deterministic shortest path algorithm. To find an efficient optimization, we analyze a variety of network topologies in the next section to gauge the benefit of three simple routing strategies on program execution: the Euler path, the shortest path in the graph, and the shortest path in the tree.

3.2 Free Link Analysis

Within the context of a self-assembled nano-scale network there are three principle controls over the integration of components (i.e., random versus regular or periodic): (1) placement, (2) orientation, and (3) node interconnection) [Patwardhan et al. 2006a]. Future self-assembly technology may provide all three types of control; however, whether and when full control can be achieved remains an open question. To fully analyze the characteristics of nano-scale irregular networks we simulate all eight combinations of control and also networks with full control but different fractions of defective nodes or transceivers. Placement control is modeled as either precisely placing a node on a grid or randomly placing it in a fixed area. Orientation control is modeled as either aligning a node both vertically and horizontally or allowing a random rotation about its center. Finally, interconnection control is modeled as either straight wires or wires formed through a guided random walk, with parameters obtained from actual DNA self-assemblies.

We analyze networks with different numbers of nodes: 3000, 4500, 6000, 7500, and 9000. For each number of nodes and each combination of control, 20 networks are generated and analyzed. For each number of nodes and each fraction of defective nodes or transceivers, 20 networks with full control are also generated and analyzed. All the following results in this section are the average across networks with differing numbers of nodes. We use a 3-bit binary number to represent different control combinations where the leftmost bit indicates placement control, the middle bit indicates orientation control, and the



(a) networks without defects (b) networks with defects (full control 111)

Fig. 5. Free link percentage in networks with different controls and defect rates.

rightmost bit indicates interconnect control. A bit with value 1 means that we have the corresponding control over the integration.

As shown in Figure 2, the dashed link between node 15 and node 1 is used to form the shortest path between nodes 15 and 8 in the graph. However, the tree structure does not include this link in the Euler path and it will be unused, thus we define it as a free link. Since free links may be used to form shorter paths than the Euler path, we evaluate the potential impact of free links on the tail-to-head path length. This is an upper bound on the possible improvement over an Euler path route since discovering the actual shortest path will require additional resources and increase node complexity and possibly instruction latency.

Figure 5(a) shows the percentage of free links in the networks without defects, and Figure 5(b) shows the percentage of free links in the networks with different node defect rates and transceiver defect rates. We can see that in the networks without defects, the free link percentage is around or less than 10%, except when there is at least both placement and orientation control (110 and 111). Even with almost 50% free links in the networks with full control (111), as the fraction of defective nodes or transceivers increases the percentage of free links decreases. Although free links could be used to form paths shorter than the Euler path, their benefit may be limited due to the low free link percentage across the networks with only one or two controls or those with high defect rates.

As described in Section 3.1, PSIs requiring tail-to-head data transfer within a PE take a large fraction of total execution time, thus it is more practical to focus on shortening the path between tail node and head node within the same PE rather than trying to reduce the path between every pair of nodes. In order to check how much improvement we can obtain by following the shortest paths, we compare the average lengths of the Euler path, the shortest path in the graph (which means that free links are used to form the shortest path), and the shortest path in the tree between the tail node and the head node in the same PE.

Figure 6 compares the average lengths of the three different paths in the networks without defects. For all networks without placement control (0XX) the results are nearly identical (see leftmost bars). Similarly, for networks with placement but without orientation control, the results are nearly identical (bars labeled 10X). For all networks except those with full control (111), there is at least a 50% reduction in the average tail-to-head path length if we follow the shortest path in the graph or the tree instead of the Euler path.

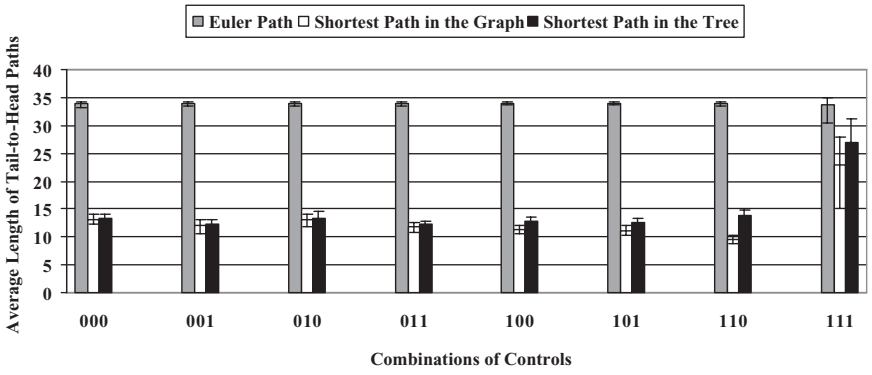


Fig. 6. Comparison of average length of tail-to-head paths in networks without defects.

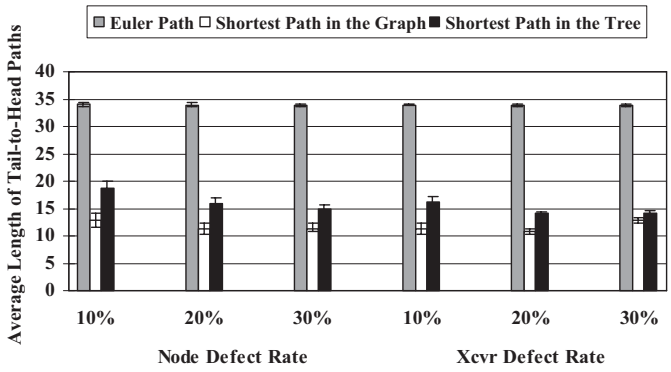


Fig. 7. Comparison of average length of tail-to-head paths in networks with defects.

In networks with full control, the average length of the tail-to-head paths along the Euler path is similar to those networks with other types of control, but the average lengths of the shortest path in the graph and in the tree between the tail and head nodes are both longer. The possible cause of the longer shortest tail-to-head paths in the graph is that there is a smaller fraction of free links connecting two nodes in the same PE, which we call intra-PE free links, and this type of free link has more contribution to the path length reduction. The likely reason why the shortest tail-to-head paths in the tree are longer is that the networks with full control have regular mesh structures and the physical structure of a single PE is more likely to be linear, making the shortest path in the tree overlap more with the Euler path.

Figure 7 compares the average lengths of the three different paths in the full-controlled networks with different node defect rates and different transceiver defect rates. When there are more defects in the networks with full control, the regular mesh structures become more irregular. Consequently, there are more intra-PE free links and the single PE structure becomes more nonlinear. As a result, the average lengths of the shortest path in the graph and in the tree become shorter and so do the differences between them.

The results shown in Figure 6 and Figure 7 establish that in both irregular networks and regular networks with defects, the shortest paths in the graph and in the tree are both much shorter than the Euler path between the tail and head nodes. Moreover, in these two types of networks, the shortest paths in the graph and in the tree have similar average length, which indicates that free links actually do not have a large effect in reducing average tail-to-head path length. It would be more efficient to not use free links to form the shortest path, since in both cases the average tail-to-head length paths are almost the same, but the configuration cost to route within the already configured tree structure is much lower than introducing new complexity to identify and route along the free links.

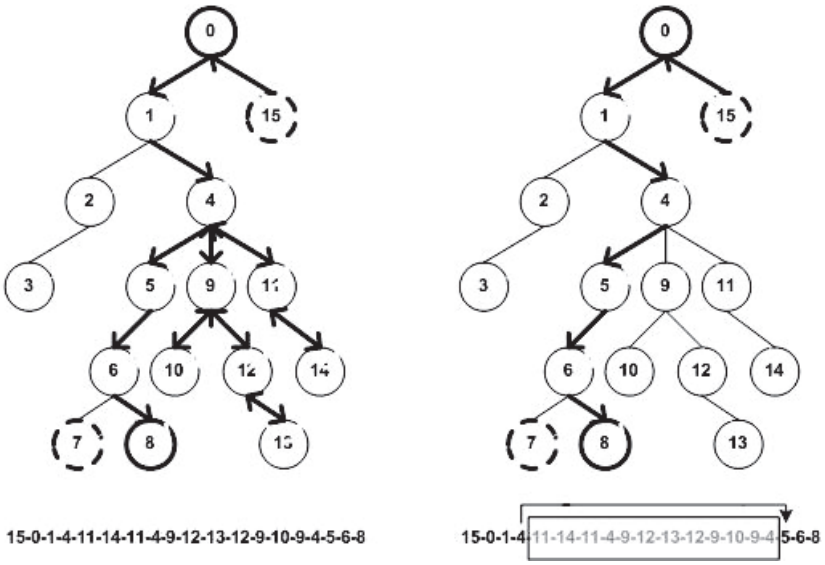
4. NEW ROUTING ALGORITHM

In this section we propose a new algorithm for establishing static routes in nano-scale irregular networks, which follows the shortest path in a generated tree structure between the tail and head nodes within a PE instead of the Euler path. Importantly, each node in a nano-scale irregular network has very limited storage space (a few bits) and computational power (a 1-bit ALU). As a result, we cannot afford a routing table [Sancho 2004] (especially when it is possible to have billions of nodes in our network) or complex computation [Ibanez et al. 2008] for full dynamic routing at each node as in general irregular networks. We thus determine the shortest tail-to-head paths during the configuration process and trade increased configuration time for reduced storage space and computational power during execution. Note that some storage space is still required to route tail-to-head packets; however, the storage space needed is much less than required by a routing table.

4.1 Optimized Tail-to-Head Paths

There are two types of information that need to be collected and stored at each node during the configuration process: (1) if the node is on the shortest tail-to-head path in the generated tree (which we call the optimized tail-to-head path); and (2) if it is, where to forward the tail-to-head packet. Each node must know whether it will perform routing along the optimized tail-to-head path. Tail-to-head data generated by a PSI at the tail node only needs to be seen by the head node and it can bypass as many intermediate nodes as possible. If an intermediate node is bypassed by the optimized tail-to-head path, it must be able to proceed to the next instruction without observing the tail-to-head data. On the other hand, if an intermediate node is not bypassed, it should stall until it receives and sends out the tail-to-head data before it may proceed to the next operation.

If an intermediate node needs to route data generated by a PSI, it also needs to know which neighbor should receive the data. By choosing different destination neighbors from the Euler path at certain nodes, the optimized tail-to-head path can be obtained. The optimization method is based on the following observation: when the Euler path is followed, we may visit an intermediate node more than once in order to transfer data from the tail node to the head node;



(a) Euler path (b) Optimized path

Fig. 8. Two PEs with eight nodes per PE.

when the optimized tail-to-head path is followed, we should visit each intermediate node exactly once. The observation indicates that an intermediate node may have multiple appearances on the Euler path but only appears once on the optimized path between one pair of tail and head nodes. In order to obtain the optimized tail-to-head path, when a node is visited for the first time along the Euler path, the successor node of its appearance nearest to the head node on the Euler path is chosen as the destination neighbor.

Figure 8 shows a simple example with two PEs and eight nodes per PE. PE 0 has node 0 as the head node and node 7 as the tail node; PE 1 has node 8 as the head node and node 15 as the tail node. When PE 1 executes a PSI, a predicate bit is sent from node 15 to node 8. Figure 8(a) shows that the Euler path from node 15 and node 8 is 15-0-1-4-11-14-11-4-9-12-13-12-9-10-9-4-5-6-8 and Figure 8(b) shows that the optimized path is 15-0-1-4-5-6-8. When we already know the reverse depth-first path and there are multiple appearances of one node, such as node 4, on the Euler path, after this node receives the tail-to-head data for the first time, instead of forwarding the data to the successor of this appearance, which is node 11, it can forward the data to the successor of the appearance nearest to the head node, which is node 5. In this way, the Euler path between the first and the last appearances of a certain node is pruned off and we can obtain the optimized path instead of the Euler path.

4.2 Our Algorithm versus Up*/Down* Routing

The original up*/down* routing algorithm and several improved versions [Schroeder et al. 1991; De Pallegrini et al. 2004; Sancho 2004; Ibanez et al. 2008] are widely used in irregular networks with abundant resources. The basic idea

of up*/down* routing is to inhibit certain turns to break cycles in a network, thus obtaining deadlock-free routing. In order to generate the shortest path in a tree, every link in the tree is assigned with up direction if the end is closer to the root or down direction otherwise, and then a legal route is formed by first following several up links and then several down links.

The optimized tail-to-head paths generated by our algorithm are similar to the up*/down* routing in the tree. However, our algorithm focuses on only pairs of tail and head nodes rather than random pairs of nodes and our optimized paths are static as established during configuration. Less randomness and static optimized paths mean less and fixed size storage space. As a result, no full routing tables are needed and the time to look up the routing tables is saved. As a contrast, although there are improved versions of up*/down* routing trying to get rid of routing tables, more computation is actually needed during execution to encode the routing information included in the destination address [Ibanez et al. 2008].

4.3 Deadlock-Free Routing

An important aspect of a routing algorithm is that it must avoid deadlock. SOSA already ensures deadlock-free routing by using Euler path forwarding and providing three virtual channels [Patwardhan et al. 2006c]. Our routing algorithm does not change the routing along VC0 or VC1, but only affects VC2. Since we follow the optimized path by pruning some nodes and links off the Euler path, we are using fewer resources. Combined with the fact that the optimized path is the same as applying up*/down* routing in the generated tree, routing a single tail-to-head packet will not cause deadlock. For the case when multiple optimized tail-to-head paths cross over at one single node, because the resources required by different paths are from different disjoint sets, there is no possibility of deadlock. In Appendix A we prove that our routing algorithm is deadlock free, even if multiple optimized tail-to-head paths cross over at one single node.

5. IMPLEMENTATION

The proposed routing algorithm is divided into two processes, configuration and routing, and implemented in the SOSA simulator. The SOSA simulator is an event-driven simulator. Each activity in a node is an event and its latency is a multiple of "time quantum," which is defined as the time taken to perform one part of the internode asynchronous communication handshake [Patwardhan et al. 2006c]. During the configuration process, the information about whether a node is on the optimized path is collected and stored in the node and the information about where to send the data is stored in the related transceivers of the node. During the routing process, each node decides whether to forward packets along the Euler path or the optimized path based on the combination of the data packet type and the virtual channel.

5.1 Configuration

The configuration process starts after a tree structure is generated from the nano-scale irregular network using the RPF (Reverse Path Forwarding)

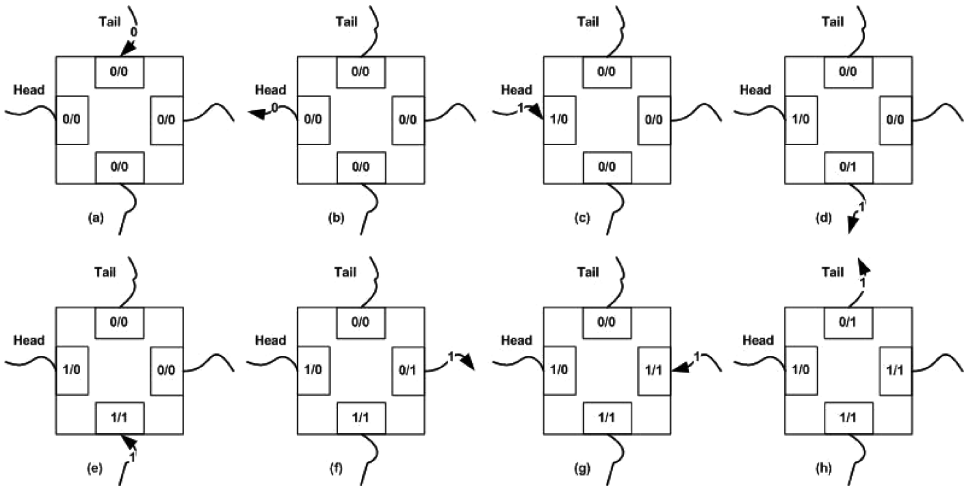


Fig. 9. Configuration process of PE 1 at node 4.

algorithm [Dalal and Metcalfe 1978] already implemented in SOSA. One configuration packet is sent out from the root and forwarded along the Euler path to configure one PE. A configuration bit is added to the configuration packet and initialized to 0. When the corresponding PE is being configured, the configuration bit is set to be 1 at the head node and set back to be 0 at the tail node. When a node receives a configuration packet, if the configuration bit is 1, it means that this node is on the Euler path between the tail and the head nodes of the PE that is currently being configured.

As described in Section 2, each node in SOSA has four transceivers and each transceiver has an input port and an output port with separate buffers. During the configuration process, each port needs one bit of temporary storage space to store the value of the configuration bit when the configuration packet goes through this port. We call the bit stored in the input port of a transceiver the input (I) configuration bit of the transceiver and the bit stored in the output of a transceiver the output (O) configuration bit of the transceiver. The I/O configuration bits may have different values when different PEs are being configured since each configuration packet is only used to configure one PE.

Figure 9 shows how the I/O configuration bits are set at node 4 in Figure 8 when PE 1 is being configured. The node shown has four active transceivers: up, down, left, and right, and connected to node 1, node 5, node 9, and node 11, respectively. The head node of PE 1 is in the left direction and the tail node is in the up direction. Each transceiver has one input port and one output port so we also have four input ports and four output ports. In each transceiver, the left bit shown is the I configuration bit and the right bit is the O configuration bit. The configuration packet for PE 1 arrives at the up input port with configuration bit 0 (step a), leaves the left output port with configuration bit 0 and returns to the left input port with configuration bit 1 (steps b and c), leaves the down output port and returns to the down input port with configuration bit 1 (steps d and e), leaves the right output port and returns to the right input port with

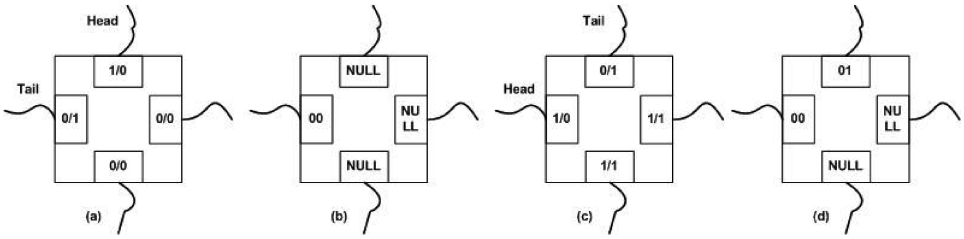


Fig. 10. Output port index setting at node 4.

configuration bit 1 (steps f and g), and finally leaves the up output port with configuration bit 1 (step h).

A node finishes forwarding the current configuration packet after sending out the packet from the transceiver that initially received it. Afterwards, data packet routing decisions are made based on the I/O configuration bits. Each output port in a node is given an index to identify it and two bits are needed to represent this index since we have four output ports per node. One such index is stored in each transceiver as a pointer and we call it as output port index. If a tail-to-head PSI packet arrives at the input port of a transceiver, the output port index will point to the output port that the node should forward the packet.

Figure 10 shows how the output port indices of all the four transceivers are set in node 4 after PE 1 is configured. Figure 10(a) shows the values of all the four I/O configuration bits pairs after PE 0 is configured and Figure 10(c) shows the values after PE 1 is configured. Figure 10(b) and Figure 10(d) show the values of all the four output port indices after the configuration of PE 0 and PE 1, respectively. The values of the output port indices are defined as follows: 00 for the up output port, 01 for the left, 10 for the down, and 11 for the right.

All the output port indices are initialized to be NULL before the configuration of the first PE in the network. After the configuration of each PE i, the values of all four pairs of I/O configuration bits in each node are checked. If a transceiver has I/O configuration bits equal to 0/0 (such as the down and right transceivers in Figure 10(a)) or 1/1 (such as the down and right transceivers in Figure 10(c)), the output port index remains NULL since the tail-to-head packet sent out by the tail node of PE i during PSI execution will not go through this transceiver. If a transceiver has I/O configuration bits equal to 0/1 (such as the left transceiver in Figure 10(a) and the up transceiver in Figure 10(c)), it means that the tail node of PE i is in the corresponding direction (left for PE 0 and up for PE 1). Similarly, if a transceiver has I/O configuration bits equal to 1/0 (such as the up transceiver in Figure 10(a) and the left transceiver in Figure 10(c)), it means that the head node of PE i is in the corresponding direction (up for PE 0 and left for PE 1). As a result, suppose transceiver x has I/O configuration bits 0/1 and transceiver y has I/O configuration bits 1/0, the output port index of transceiver x should be y, since x to y is the direction of the PE tail to PE head, while the output port index of transceiver y remains unchanged. For example, as shown in Figure 10(b), the output port index of the transceiver 01 (left) is set to be 00 (up), and as shown in Figure 10(d), the output port index of transceiver 00 (up) is set to be 01 (left).

Besides the 2-bit storage space at each transceiver to indicate where to forward the tail-to-head packet along the optimized path, we need one additional bit of storage space per node. This bit is called as tail-to-head bit and is used to indicate whether the node is on the optimized tail-to-head path of its own PE (note that nodes can route packets on behalf of other PEs). The tail-to-head bit of each active node is initialized to be 1 before the PE configuration process. After the configuration of each PE i, for each node in PE i, if all active transceivers in a node have I/O configuration bits 1/1, the tail-to-head bit is set to 0, indicating that the predicate data will not go through this node.

5.2 Routing

During the routing process, all data travels in the units of packets. Predicate data packets generated by PSIs are routed along the optimized tail-to-head path and other data packets going from tail to head are routed along the Euler path. The main problem is for each node to determine if the incoming data packet is a predicate data packet or not.

One possible solution is to add one bit in the data packet as a flag. When a data packet is generated and sent out to a head node by a tail node executing a PSI, the flag is set and later each intermediate node receiving this packet knows that the packet should go along the optimized path by checking the flag. However, since only one bit can be transferred through a link at a time and one packet in SOSA only has four bits, adding one bit to the data packet may cause unnecessary overhead.

Another solution, which we implement, is to use the combination of the virtual channel index and control information in the data packet. There are three virtual channels in SOSA: VC0 as the broadcast channel, VC1 as the forward channel, and VC2 as the backward channel. Predicate data packets only go through VC2. However, other data packets, such as those generated by shift instructions, also go through VC2, so more information is needed to differentiate these data packets from predicate ones. There are currently two control bits in each packet indicating whether the packet is a data packet with control bits as 11 or different control packets with control bits as 00, 01, or 10. For VC0 and VC1, these two bits are useful since there can be both data packets and control packets traveling through these two virtual channels. However, for VC2 there are only data packets, therefore we can use one of the other three values representing a control packet to indicate a predicate data packet. By checking both the virtual channel index and the value of the two control bits in the packet, a node can decide whether the incoming packet is a predicate packet or not and route the packet either on the Euler path or on the optimized tail-to-head path.

6. EVALUATION

We compare the performance of the proposed routing algorithm (optimized) against routing along the Euler path (nonoptimized). Using a network size of 4500 nodes, we simulate the six different test programs we used to analyze the execution time and the latency of PSIs in Section 3.1. Figure 11 shows the normalized execution time in networks without defects averaged across 20

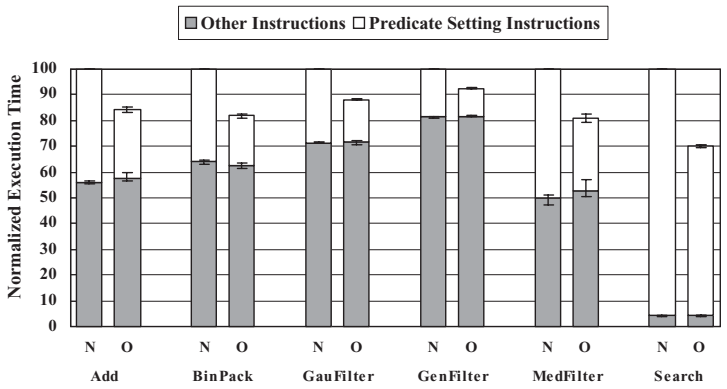


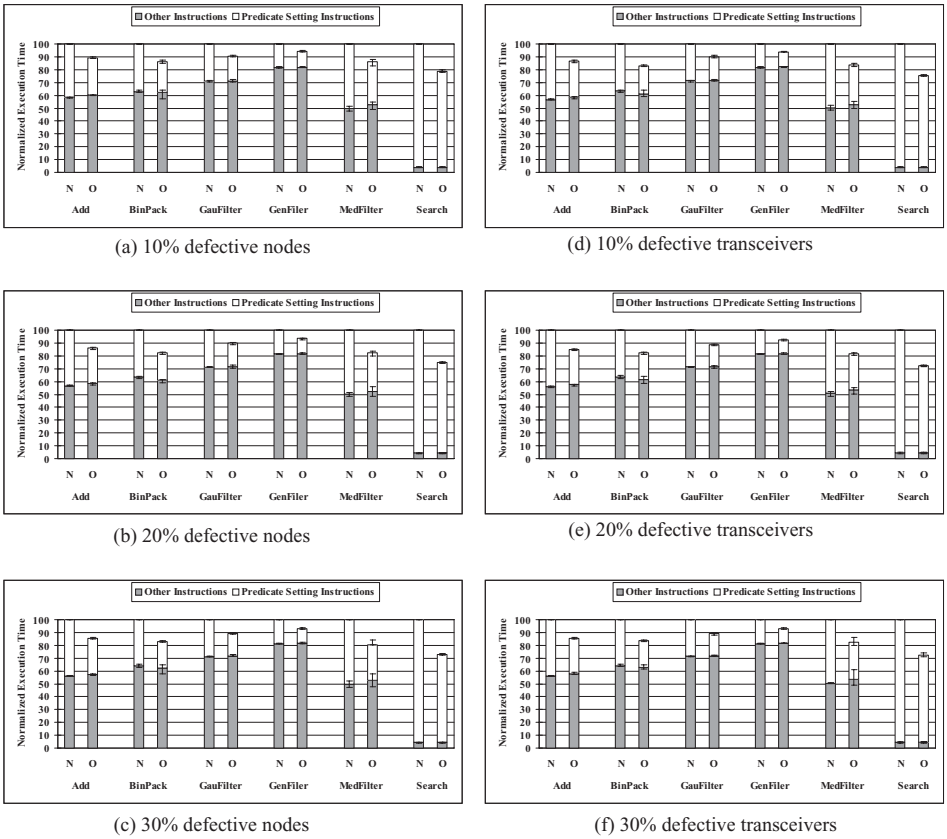
Fig. 11. Benefit of optimized routing in networks without defects.

different randomly generated topologies with placement control but without orientation or interconnect control.

Figure 12 shows the normalized execution time in regular mesh networks (full control over placement, orientation, and interconnect) with varying defect rates for nodes and transceivers, respectively. In each of these graphs, N means nonoptimized and O means optimized.

From these results we can see that there is up to a 30% reduction in total execution time for the optimized routing algorithm. As expected, the execution time spent in other instructions remains almost the same for both optimized and nonoptimized routing. The benefit of optimized routing depends primarily on the fraction of time spent executing PSIs in the base, nonoptimized system. There is no dependence on defect rates of nodes or transceivers in the networks. For all test programs, except *GenFilter*, more than 25% of their execution time is spent executing predicate instructions in the nonoptimized system. These programs exhibit at least 12% reduction in total execution time with the optimized routing algorithm. *GenFilter* spends less than 20% of its execution time on PSIs before the optimization, so the improvement is smaller than other test programs, but we still observe an 8% reduction in total execution time. *Search* has around 90% of its execution time spending on PSIs. However, only around half of the PSI execution time is spent on tail-to-head communication and we can reduce the tail-to-head communication time by around 60%. As a result, the reduction in total execution time of *Search* should be around 27% (90%$\times$50%$\times$45%), which is exactly what is shown in both Figure 11 and Figure 12.

The additional configuration time cost by simple operations to decide the routing direction is not a problem since configuration can be done only once before the system is ready to use. The extra storage space required at each node is totally nine bits and it has little effect on the node size compared with the already existing resources. It is worthwhile to add the extra configuration time and storage space because of the significant performance improvement.



(a) 10% defective nodes

(d) 10% defective transceivers

(b) 20% defective nodes

(e) 20% defective transceivers

(c) 30% defective nodes

(f) 30% defective transceivers

Fig. 12. Benefit of optimized routing in networks with defective nodes or defective transceivers.

7. RELATED WORK

Routing in irregular networks has always been a challenging problem to solve. Several routing algorithms based on different network structures have been proposed to provide high throughput and low latency while at the same time avoiding deadlocks [Federico and Duato; Chi and Wu 2003]. These algorithms work well in irregular networks composed with single processors and workstations. However, as CMOS feature sizes decrease and nanotechnology develops in order to bypass the Red Brick Wall and we enter the field of nano-scale networks, it becomes more difficult to route in irregular networks because each component has less storage space and computation power.

People have proposed several routing algorithms for networks in which each component has fewer resources than processors and workstations but more resources than nano-scale devices. Some algorithms still use meshes [Kumar et al. 2002; Taylor et al. 2002; Wiklund and Liu 2003], while some algorithms work on partially irregular networks using irregular meshes [Bolotin et al. 2005; Schafer et al. 2005]. There are also some algorithms [Palesi et al. 2006; Bolotin et al. 2007] focusing on compressing routing tables in regular mesh-like networks. For nano-scale irregular networks, there is a tree-based routing

algorithm using RPF and virtual channel cut-through [Patwardhan et al. 2006b, 2006c].

8. CONCLUSION

In this article, we propose a new routing algorithm for self-organizing nano-scale irregular networks. Our approach trades configuration time and a small amount of storage space to reduce communication latency. We augment the existing Euler path routing technique for trees to generate static shortest paths between specific pairs of nodes, while remaining deadlock free. Simulations of several applications executing on a nano-scale data parallel architecture show our proposed routing algorithm can reduce execution time by 8% to 30% compared with the original Euler path-based routing algorithm. Moreover, even if there are up to 30% defective nodes or transceivers in the networks, we can still obtain a similar reduction in total execution time. Although our analysis is specific to one system, we believe that the general approach of providing optimized routing between specific pairs of nodes is applicable to a broader set of systems, particularly those that implement logical hierarchical ring topologies on top of irregular physical topologies. In systems with more available resources per node than we have in our nano-scale networks, we could store more information at each node and potentially optimize routing between more pairs of nodes than just the PE tail/head.

APPENDIX A. PROOF

In this section, we prove that the proposed routing algorithm is deadlock free. As described in Section 4.3 the optimized routing algorithm only affects VC2 and uses fewer overall network resources. Combined with the fact that the optimized path is the same as applying up*/down* routing in the generated tree, routing a single tail-to-head packet will not cause deadlock. We need to prove that when multiple optimized tail-to-head paths cross over at one single node, the optimized routing algorithm is still deadlock free. The intuition is that different tail-to-head paths use disjoint resources in the network and thus cannot cause deadlock.

Definition 1. The *tail-to-head Euler path* of a PE is the Euler path between the first appearance of the tail node and the first appearance of the head node.

THEOREM 1. *The tail-to-head Euler path of a PE can be optimized at a node if the node appears at least two times on the Euler path between the head node and the tail node of this PE.*

PROOF. Suppose node i is on the Euler path between the head and the tail of PE *0*.

If node i only appears once on the Euler path between the head and the tail of PE *0*, the destination neighbor of node i on the optimized path is the same as on the Euler path. As a result, no optimization can be done at node i for PE *0*.

If node i appears more than once on the Euler path between the head and the tail of PE *0*, the destination neighbor of node i on the optimized path is the predecessor of the appearance nearest to the head node. As a result, the path

between the first appearance and the last appearance of node i can be pruned off the Euler path and a shorter optimized path can be obtained. □

THEOREM 2. *If the tail-to-head Euler paths of multiple PEs can be optimized at a single node, the input and output resources required by different paths are disjoint.*

PROOF. Suppose both the tail-to-head Euler paths of PE *0* and PE *1* can be optimized at node i. PE *0* has node j as the head node and node k as the tail node. PE 1 has node m as the head node and node n as the tail node.

If we only look at the first appearance of each node, obviously, node j appears before node k and node m appears before node n on the Euler path. Because of Definition 1, either node k appears before node m, or node n appears before node j. Without loss of generality, we assume that node k appears before node m. As a result, the tail-to-head Euler paths of PE *0* and PE *1* are disjoint.

Because the Euler path traverses each link in the network for exactly two times in different directions, each time the Euler path enters or leaves a node, a different resource (an input resource when entering and an output resource when leaving) is used. Since the tail-to-head Euler paths of PE *0* and PE *1* are disjoint, the input and output resources required by different paths are also disjoint. □

REFERENCES

BARROSO, L. A. AND DUBOIS, M. 1993. The performance of cache-coherent ring-based multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 268–277.

BOLOTIN, E., CIDON, I., GINOSAR, R., AND KOLODNY, A. 2005. Efficient routing in irregular topology NoCs. Tech. rep. Department of Electrical Engineering, Technion, Haita, Israel.

BOLOTIN, E., CIDON, I. , GINOSAR, R., AND KOLODNY, A. 2007. Routing table minimization for irregular mesh Nocs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*. 942–947.

CHI, H. AND WU, W. 2003. Routing tree construction for interconnection networks. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-based Processing*.

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

DE PALLEGRINI, F., STAROBINSKI, D., KARPOVSKY, M. G., AND LEVITIN, L. B. 2004. Scalable cycle-breaking algorithms for gigabit ethernet backbones. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom)*.

FEDERICO, S. AND DUATO, J. High-performance routing in networks of workstations with irregular topology. *IEEE Trans. Parallel Distrib. Syst. 11*, 7, 699–719.

IBANEZ, G. A., GARCIA-MARTINEZ, A., CARRAL, J. A., AND GONZALEZ, P. A. 2008. Hierarchical up/down routing architecture for ethernet backbones and campus networks. In *Proceedings of the Conference on IEEE Computer Communications Workshops (InfoCom)*.

KUMAR, S., JANTSCH, A., SOININEN, J. P., FORSELL, M., MILLBERG, M., OBERG, J., TIENSYRJA, K., AND HEMANI, A. 2002. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*

MARTY, M. R., AND HILL, M. D. 2006. Coherence ordering for ring-based chip multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 309–320.

PALESI, M., KUMAR, S., AND HOLSMARK, R. 2006. A method for router table compression for application specific routing in mesh topology NoC architectures. In *Proceedings of the SAMOS VI Workshop: Embedded Computer Systems: Architectures, Modeling, and Simulation*.

PATWARDHAN, J. P., DWYER, C., LEBECK, A. R., AND SORIN, D. J. 2004. Circuit and system architecture for DNA-guided self-assembly of nanoelectronics. In *Proceedings of the Conference on Foundations of Nanoscience: Self-Assembled Architectures and Devices*. 344–358.

PATWARDHAN, J. P., DWYER, C., AND LEBECK, A. R. 2006a. Self-assembled networks: Control vs. complexity. In *Proceedings of the 1st International Conference on Nano-Networks (NANONETS)*.

PATWARDHAN, J. P., DWYER, C., LEBECK, A. R., AND SORIN, D. J. 2006b. NANA: A nano-scale active network architecture. *J. Emerg. Technol. Comput. Syst. 2*, 1, 1–30.

PATWARDHAN, J. P., JOHRI, V., DWYER, C., AND LEBECK, A. R. 2006c. 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.

SANCHO, J. C. 2004. An effective methodology to improve the performance of the up*/down* routing algorithm. *IEEE Trans. Parallel Distrib. Syst. 15*, 8, 740–754.

SCHAFER, M. K. F., HOLLSTEIN, T., ZIMMER, H., AND GLESNER, M. 2005. Deadlock-Free routing and component placement for irregular mesh-based networks-on-chip. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*.

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. Select. Areas Comm.* 9.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro 22*, 2, 25–35.

WIKLUND, D., AND LIU, D. 2003. SoCBUS: Switched network on chip for hard real time embedded systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*.

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. *Sci. 301*, 1882–1884.

Received May 2009; revised August 2009; accepted October 2009 by Frederic Chong