1-1-2011

A Self-Configurable Architecture on an Irregular Reconfigurable Fabric

Avinash Amarnath
Portland State University

A Self-Configurable Architecture on an Irregular Reconfigurable Fabric

by

Avinash Amarnath

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Christof Teuscher, Chair
Douglas V. Hall
Alaa Alameldeen

Portland State University
2011

Abstract

Reconfigurable computing architectures combine the flexibility of software with the performance of custom hardware. Such architectures are of particular interest at the nanoscale level. We argue that a bottom-up self-assembled fabric of nodes will be easier and cheaper to manufacture, however, one has to make compromises with regards to the device regularity, homogeneity, and reliability. The goal of this thesis is to evaluate the performance and cost of a self-configurable computing architecture composed of simple reconfigurable nodes for unstructured and unknown fabrics. We built a software and hardware framework for this purpose. The framework enables creating an irregular network of compute nodes where each node can be configured as a simple 2-input, 4-bit logic gate. The compute nodes are organized hierarchically by sending a packet through a top anchor node that recruits compute nodes with a chemically-inspired algorithm. The nodes are then self-configured by means of a gate-level netlist describing any digital logic circuit. A topology-agnostic optimization algorithm inspired by simulated annealing is then initiated to self-optimize the circuit for latency. Latency comparisons between non-optimized, brute-force optimized and our optimization algorithm are made. We further implement the architecture in VHDL and evaluate hardware cost, area, and energy consumption. The simple on-chip topology-agnostic optimization algorithm we propose results in a significant (up to 50%) performance improvement compared to the non-optimized circuits. Our findings are of particular interest for emerging nano and molecular-scale circuits.

Acknowledgements

I would like to thank my thesis advisor, Christof Teuscher, for the wonderful opportunity given to me to do my research and this thesis. His constant guidance and diligent support has helped me grow as a researcher and successfully finish my thesis.

I am grateful to ECE department faculty and staff for the help and resources given to me to complete my thesis work. I would also like to thank the Computer Action Team at PSU for the computing resources and assistance provided whenever needed.

I would like to thank my Teuscher lab mates for their invaluable help during different times of my research work.

Last but not the least, I would like to thank my parents, my friends and family for being there always and supporting me throughout my course of study. Thank you very much.

Contents

List of Tables

List of Figures

1

Introduction

1.1 Motivation

With continuous scaling to smaller feature sizes, CMOS technology is facing several challenges such as rising costs of fabrication, limits of lithography, and rising test costs [15]. These limitations may need a fundamental shift in the way future integrated circuits are fabricated. The semiconductor roadmap suggests a need to explore alternatives to CMOS devices and fabrication techniques [1]. Nanoelectronics is believed to help in finding a solution to these limitations [15]. It is a field which involves chemistry, physics, biology and engineering. Individual wires, diodes, Field Effect Transistors (FETs) and switches could be manufactured abundantly and cheaply in a test tube [15]. It might be impossible to pattern the nanoelectronic devices in a top-down manner [15]. They would likely be generated by a stochastic self-assembly process in a "bottom-up" manner. Due to the lack of control on the self-assembly techniques, self-assembled interconnects are expected to be largely unstructured or random [32]. It has also been argued [24–26] that nanodevices that are assembled in a largely random manner would scale up better, both from the fabrication and the communication standpoint.

While we can currently build switching devices in various technologies besides CMOS [9, 17, 33], one of the remaining challenges is to assemble and interconnect these switching devices (or logic functions) to larger systems, and ultimately to design a computing architecture that allows to perform reliable computations.

Irregular architectures have been proposed to be a possible solution [15]. One example of irregular architecture, called the Nanocell [16], is shown in Figure 1.1. A network is assumed to be created by randomly placing very small conductive particles called as nanoparticles on a substrate. Molecular switches [10], which are either in "on" or "off" state are then introduced into the substrate and the ends of each switch attach to the nanoparticles. Each nanocell in the random network was configured with a binary string which defined the state of each switch in the

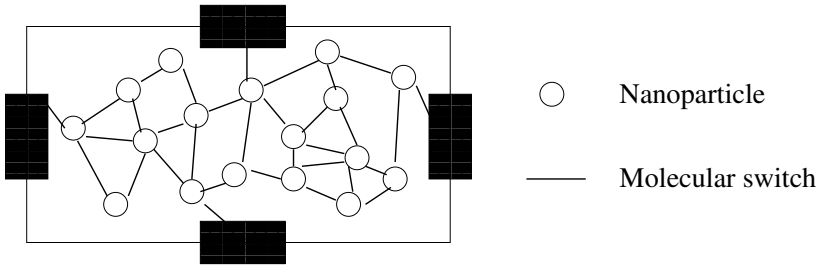


Figure 1.1: A nanocell block. The black boxes are the I/O terminals [16].

cell.

For such irregular architectures to work, certain key ingredients are required such as self-configuration, self-reconfiguration, self-repair, and self-adaptation. Due to the irregular structure, it is also required to sacrifice homogeneity and reliability.

1.2 My contributions

In this thesis, we present a self-configurable computing architecture built on an irregular reconfigurable fabric that addresses some of the challenges we face with emerging self-assembled computing fabrics. In the rest of this report, we refer to two layers of abstraction in our architecture. The higher layer is called the *Circuit Layer* and the lower layer is called the *Device Layer*. Figure 1.2(a) shows the

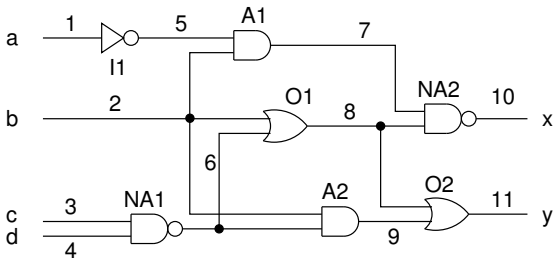
Circuit Layer, which is the logical circuit which is required to be configured onto the device. Figure 1.2(b) shows the *Device Layer*, which is a set of reconfigurable compute nodes interconnected by an irregular fabric of nanowires. We assume a hybrid of CMOS reconfigurable compute nodes combined with a self assembled substrate [21]. We further assume that the entire architecture has an unknown node arrangement with an unstructured and unknown interconnect topology.

To address the challenge of performing computations in such architectures, we focus on a self-organizing strategy that allows building a hierarchical organization of the compute nodes. Each node can communicate through a fixed communication network that is determined by the unstructured fabric. Given a digital circuit *Circuit Layer* to be configured in the form of a gate level netlist, we try to solve the problem of self-configuring the circuit onto the compute node fabric *Device Layer*. The challenge boils down to a mapping problem. In addition, a topology agnostic algorithm inspired by simulated annealing is implemented to self-optimize the circuit mapping for latency. Last but not least, we built a software framework to implement the architecture, evaluate and compare latency, energy and area [4]. My main contributions in this thesis are as following:

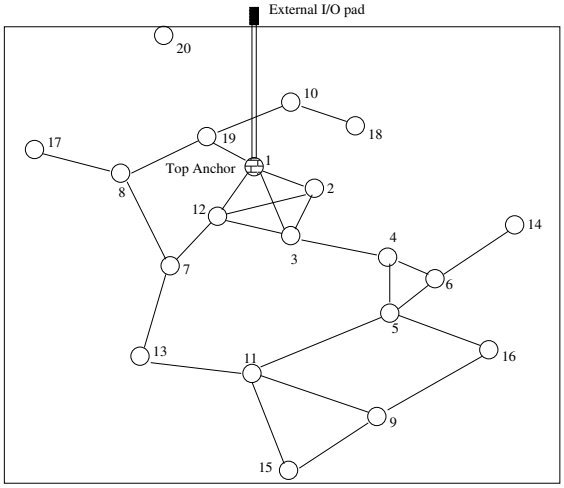
1. Created a hierarchical addressing scheme for routing data packets in a network without any routing table (Sections 3.3 and 4.3).

2. Developed an algorithm which self-configures the network of compute nodes with the given digital circuit. The configuration of the circuit happens without knowledge of the external environment (Sections 3.5 and 4.4).

3. Developed an algorithm which self-optimizes and reconfigures the digital circuit to reduce latency. The self-optimization occurs without the involvement



(a) Circuit layer: A digital circuit which needs to be configured.



(b) Device layer: Irregular reconfigurable fabric of compute nodes.

Figure 1.2: Difference between the device layer and the circuit layer in our architecture. The device layer is a substrate of reconfigurable compute nodes interconnected by a irregular fabric of nanowires. The circuit or logical layer is a higher abstraction, which defines the circuit to be configured onto the device.

of external circuitry (Section 3.7).

4. Built a software framework in MATLAB to do the following (Chapter 4):

   - Initialized a network of random compute nodes. Communication between nodes takes place through different packet modes.
   - Selects a *Top Anchor* and recruits a set of compute nodes hierarchically through a chemically-inspired gradient flow. Assigns addresses to the recruited nodes.
   - Self-configures the compute nodes with a digital circuit.
   - Runs the network for the configured circuit and evaluate latency and energy consumption.
   - Self-optimize the network to reduce latency by reconfiguring the circuit.

5. Built one compute node in VHDL and simulated it for different packet modes successfully (Chapter 4).

6. Synthesized the VHDL node for different node connectivity using Synopsys Design Compiler (Chapter 5).

7. Evaluted the latency and energy improvement for 3 benchmark circuits when our self-optimization algorithm is used. We saw a latency improvement up to 50% and an energy improvement up to 45%.

8. Observed an area overhead of 4% for the self-optimization module. Also observed an energy overhead equivalent to the energy gained whe 14000 circuit data sets were processed for circuit sizes greater than 45 gates for all 3 benchmark circuits.

9. Won an award for my poster entitled "A Self-configurable Computing Architecture on an Irregular Reconfigurable Fabric" at the Sigma Xi Columbia-Willamette organized "Student Research Symposium 2010".

10. Sigma Xi invited me to present a poster on my research work at the "Sigma Xi Annual Student Research Conference 2010" which was held in Raleigh, North Carolina during Nov, 2010.

11. The work of my thesis was also accepted at the "2011 NASA/ESA Conference on Adaptive Hardware and Systems". The paper will be presented in San Diego, California, Jun 6-9, 2011 [4].

2

Background

2.1 ASIC, microprocessors and reconfigurable devices

Application Specific Integrated Circuits (ASIC) and software-programmed microprocessors are two primary methods to compute an algorithm [11]. ASICs are very fast and efficient, but they cannot be altered to perform any other function. A redesign and re-fabrication is required for a different function, which is expensive methodology and slow process. Software-programmed microprocessors are more flexible and can execute different algorithms, but they are a lot slower than ASIC's as they have to read instructions from the memory and decode them before executing [11]. Reconfigurable computing bridges the gap between the fast but non-flexible ASIC and the flexible but slow microprocessors (Figure 2.1).
Reconfigurable devices are made up of an array of computational elements whose configuration bits determine its functionality. These computing elements are connected through a routing network that is programmable. With the help of these resources, any customized digital circuit can be mapped onto the reconfigurable device. Several applications, including data encryption, string pattern-matching, Boolean satisfiability and data compression have shown a considerable speedup with the use of reconfigurable hardware [11]. Compilation tools for reconfigurable devices range from assisting users to map the circuit to the hardware manually or synthesizing a hardware description language to a gate-level netlist. Sophisticated algorithms within the compilation tools optimize the mapping of the circuit.

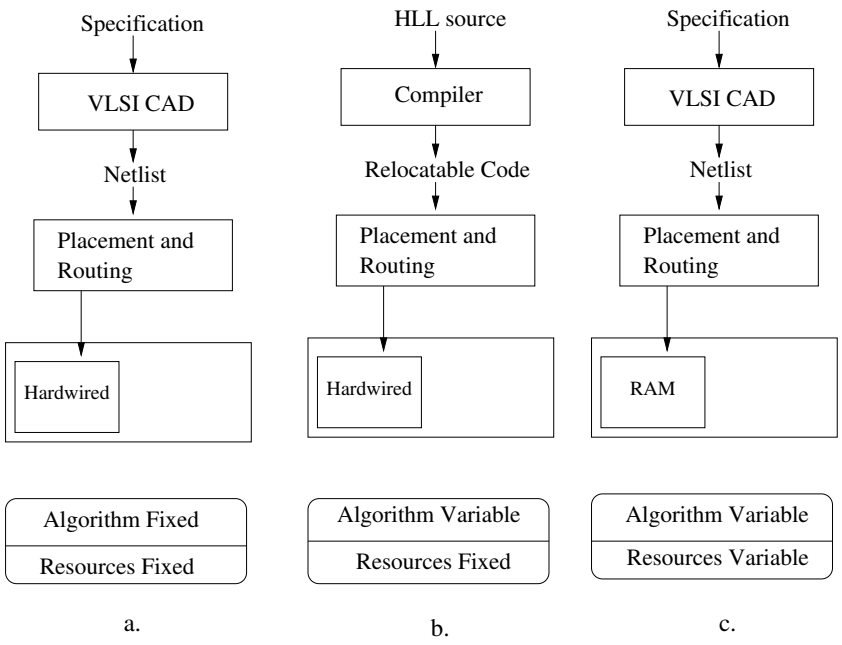


Figure 2.1: a) ASIC; b) Microprocessors ; c) Reconfigurable devices [14].

Field Programmable Gate Arrays (FPGA) are the most common form of reconfigurable computers. Most FPGA logic blocks are configured by setting or resetting SRAM bits (Figure 2.2).

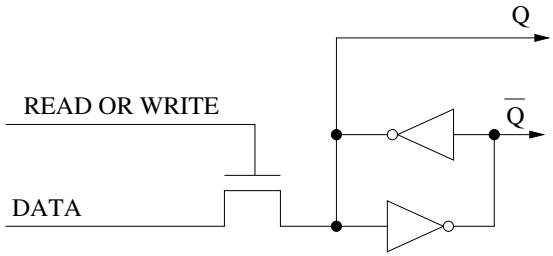


Figure 2.2: SRAM cell.

These bits control both the functionality and the routing between the logic blocks. Each computing block could be as simple as a 3-input Look-up-table (fine grained) or as complex as a 4-bit ALU (coarse grained). The size and complexity of the

basic computing block in a reconfigurable computer is called the granularity. Fine-grained units are useful for bit manipulations while coarse-grained architectures have a lesser communication overhead [14]. A simple 3-input LUT is as shown in

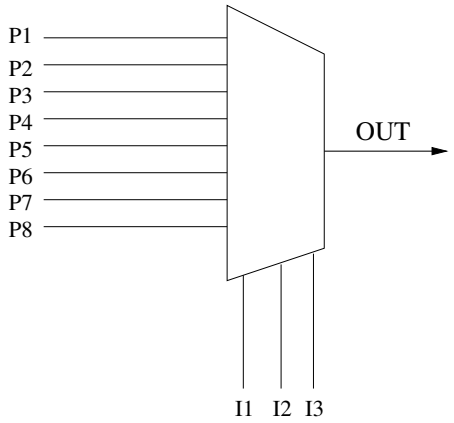


Figure 2.3: 3-input look-up-table (LUT).

Figure 2.3.

Today's reconfigurable computers typically have a regular 2D mesh routing fabric for the logic cells [11]. This is due to the theory that a mesh is the most scalable topology [8]. The routing resources are organized for efficient communication along the rows and columns of the logic cells (Figure 2.4).

2.2 Nanoelectronics and FPGA's

It is believed that nanoelectronics would be mostly used in the form similar to FPGA's [15]. Figure 2.5 shows the CAD flow to make an FPGA act as a computable device. The first step involves converting the Hardware Description Language (HDL) of the user describing a digital circuit into a format that can be implemented on the targeted technology. This is divided into logic synthesis and technology mapping. Logic synthesis involves converting the entire HDL into a

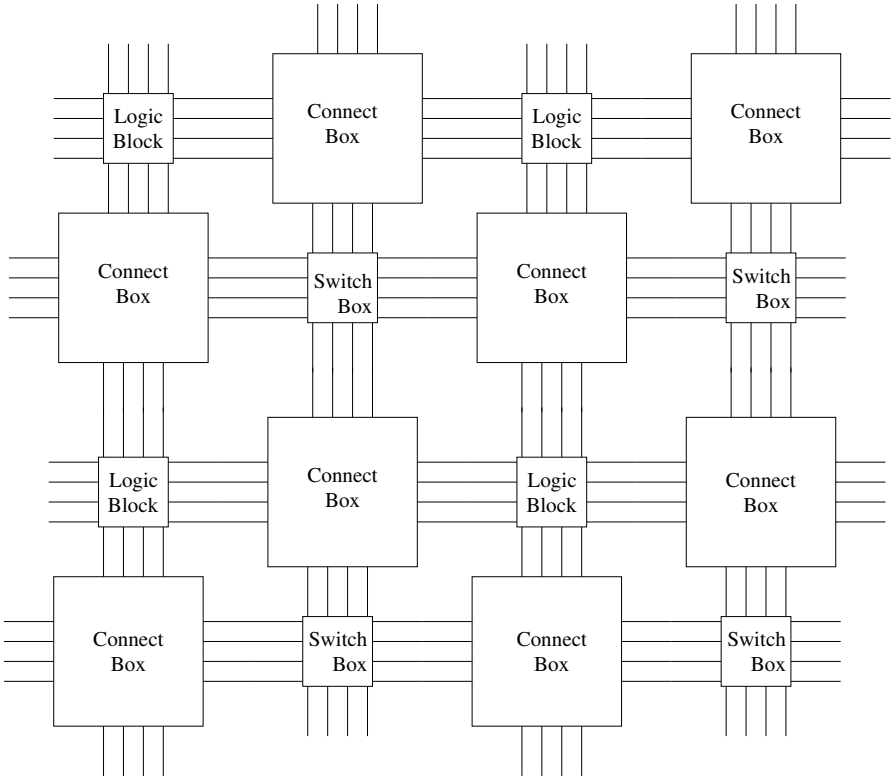


Figure 2.4: A generic FPGA routing architecture.

netlist of common logic gates with interconnections. For example, if/else statements are converted into a multiplexer and a + b is converted to an adder circuit [15]. Technology mapping involves converting the logic gates in a form which can be implemented by the FPGA. This involves programming an SRAM LUT (Figure 2.3) with the required configuration. Nanoelectronics would follow these two steps of FPGA CAD very closely [15]. But the placement and routing on nanoelectronic-based reconfigurable devices would be different from conventional CMOS FPGA's [3] due to the irregular structure and a high defect ratio [15]. This would mean that there cannot be one configuration file for all the devices. Hence, there is a need to implement a methodology to self-configure nanoelectronic-based

reconfigurable devices. This challenge has been addressed in this thesis by our algorithm to self-configure any digital circuit given in the form of a gate level netlist according to the technology (see Sections 3.5 and 4.4).

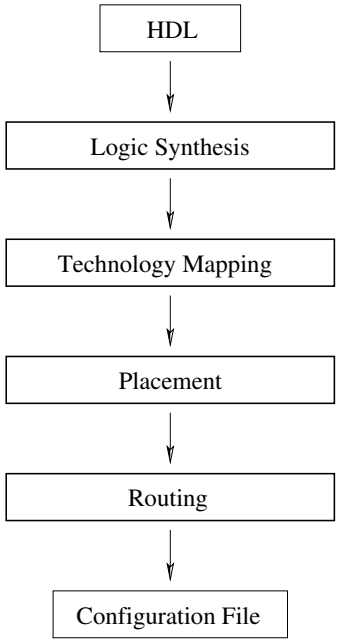


Figure 2.5: CAD flow for FPGA's [15].

2.3 Hybrid of CMOS and nanoelectronics

As mentioned in Section 1.1, the CMOS industry faces several challenges due to the technology scaling. Process variability, transient faults, bulk silicon limits, rising test costs and multi-billion dollar fabs are examples [21]. Nanoelectronics is believed to address some of these problems. Researchers have fabricated and experimented with various nanoelectronic devices to act as switches and interconnects [15]. These include carbon nanotubes [6], semi-conductive nanowires [29] and molecular devices [10]. But it is a challenge to integrate these nanoelectronic

devices and create efficient computing architectures. The small size of nanoelectronic devices would make it very difficult to pattern and place them deterministically [15]. Narayanan *et al.* [19] present a fully programmable digital cellular design for nanodevice-based computational fabrics. They show that their design could achieve a density of 22 times higher than an equivalent 16nm CMOS version for image-processing applications. But this has been just based on simulation and they have failed to address issues like self-configuration, adaptability and defect-tolerance.

Irregular interconnect architectures have been hypothesized to be one type of architecture based on nanodevices [15]. It is generally assumed that self-assembly will be able to create a random network of wires inexpensively [24]. This may be very beneficial for future computing architectures with multi-billion components because it would be very difficult to build a regular interconnect network [24]. Besides the unstructuredness, we have to expect a high fault and defect rate. A number of interconnects may not be used [26]. Hence, fault isolation methodologies is needed.

Patwardhan, *et al.* [21], have presented a defect-tolerant Single Instruction Multiple Data (SIMD) architecture that self-organizes a large number of simple CMOS nodes with high defect rates into SIMD processing elements. They have assumed a DNA-based self-assembly to place a large number of the simple cells irregularly to build a circuit interconnected by nanowires. Each unit cell is a 1-bit ALU which communicates asynchronously with its neighbors. A configuration phase isolates the defective nodes and connects the functional nodes in a logical ring. Communication to the external environment occurs through an anchor node. Configuration starts from the anchor node in a broadcast tree manner. A Instruction Set

Architecture was defined for a 3-register operand with microcoded instructions. Patwardhan, *et al.*, conclude that the performance of this architecture might not be better than conventional processors on most general-purpose workloads.

Assuming the irregular interconnect architecture and drawing inspiration from the self-organizing SIMD architecture (SOSA) [21], we propose a computing architecture made up of a hybrid of a number of simple homogeneous CMOS reconfigurable nodes on a DNA-based self- assembled network. Due to the high number of simple homogeneous CMOS nodes, the cost of technology and fabrication might get amortized much better than current day ASIC's. This approach would avoid a precise control over the entire fabrication process and also counter high defect rates even in the CMOS-based nodes. Challenges for such a design would involve not relying on the underlying network structure, composing more powerful computational blocks from simple nodes, minimizing communication overheads and achieving performance that is comparable to future CMOS-based systems [21].

Though our architecture is based on a reconfigurable fabric of nodes and not simple microprocessor nodes, we draw some inspiration from the SOSA architecture [21]. We propose a fault discovery and isolation step similar to the SOSA architecture. We call this as the *recruitment* phase (see Sections 3.3 and 4.3). We also implement the *Top Anchor* (see Sections 3.3) from the SIMD architecture which acts as the external I/O port for the architecture.

The *Amorphous computing* project also had an influence on our architecture [2]. This project tries to tackle the challenge of effectively making use of a large number of individual computing elements to process data. The authors assume a system of irregularly-placed, asynchronous, locally-interacting elements through chemical means. They propose a pattern formation of these devices inspired by chemical

and biological systems. A language called Growing Point Language (GPL) allows specifying the pattern required such as interconnect topology of an electronic circuit. One step in this process involves a wave propagation, which is inspired by chemical gradient flow. An initial anchor particle is assumed to broadcast messages to its neighbors. These messages can count the hops and mark the nodes it has passed through. After a certain count, the messages are not considered and the nodes which the message has marked will form a region. We draw inspiration for the *recruitment* phase from the amorphous computing project too which we describe in Sections 3.3 and 4.3.

2.4 Networks-on-Chip communication paradigm

Unlike conventional FPGA's, there is not much freedom to pattern multiple routing resources deterministically in an irregular network of nanowires due to the bottom-up manner of fabrication. But conventional FPGA's are also facing the problem of finding the right ratio between routing resources and Look-Up Tables (LUT's). This is due to the increasing demand for routing resources and difficulty in measuring routing resources [27]. Networks-on-Chip (NoC) are hypothesized to be the paradigm for future System-on-Chips [7]. Networks-on-Chip (NoC) do not have the problem of routing and synchronizing the nodes of the network with a global clock as the SoC would be made globally asynchronous but locally synchronous. Challenges related to bus-interconnect technology can be addressed through a layered design of reconfigurable micronetworks [7]. We have proposed using a NoC paradigm for our architecture because of the limitations of the irregular interconnect network and the hypothesis that NoCs would be the future SoCs. It might seem that our architecture might need a lot of communication resources like buffers

and serial-parallel converters in each node compared to the amount of computation taking place in each node. But we provide only a proof of concept while describing our architecture and finding the right communication to computation ratio is beyond the scope of the thesis. A bidirectional NoC architecture has been presented by Ying-Cherng, *et al.* [18]. Hence we assume a bidirectional interconnection in our interconnect network. There are also technologies through which bidirectional serial communication is possible through a single wire [5]. Hence, it is appropriate to assume a single wire bidirectional interconnect network with a NoC paradigm for our architecture.

Packets in a network have a destination address and are routed according to it. There are several algorithms which route packets effectively in regular interconnect networks [7, 20, 31]. These techniques require a routing table to be present in each node. The routing table is like a hash table where there is an entry for every node in the network. Each entry is associated with one of the neighbor nodes. Whenever a packet needs to be routed, its destination address is looked up in the routing table and its next node is determined. This means that there should be a large number of entries in every node. It also means that the size of the routing tables in each node increases linearly with the number of nodes in the network. Also creating the tables either needs a global view of the system or requires a long time to adapt itself [12]. But, as the interconnect network in our architecture is assumed to be irregular, we have explored and presented a novel algorithm and addressing scheme in Section 3.2 to route packets between reconfigurable nodes. We have avoided the use of routing tables and hence a constant hardware cost for the compute node irrespective of the size of the network.

2.5 Simulated annealing

Simulated annealing is an optimization technique which can be applied to many real-world design problems [22]. As the name implies, this technique has an anal-

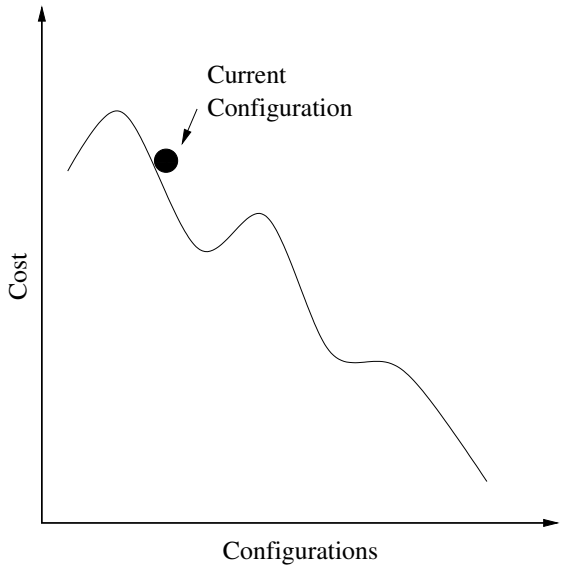


Figure 2.6: Generic effect of simulated annealing. It has an analogy of a ball rolling down a hill, but it can also allow controlled perturbations uphill and thus avoids the solution settling down in a local minima [22].

ogy with the statistical mechanics of annealing in solids. In physics, a low-energy state usually means a highly ordered state. For example in the growth of silicon crystals, the silicon material is heated to a high temperature which will allow many atomic rearrangements. The material is then purposely cooled down slowly until the material freezes into a nice crystal [22]. The simulated annealing approach has been extensively used in VLSI CAD tools to solve placing, floor-planning and routing in ASIC design. Finding the best placement and floor-plan in ASIC design is a NP-complete problem and hence would take exponential time to solve the problem. Simulated annealing is a heuristic approach and gives a reasonable answer in acceptable time [22]. It is very similar to iterative approaches except

that it allows perturbations to move uphill in a controlled fashion (Figure 2.6). In this thesis, we develop and apply a *self-optimization* algorithm inspired by simulated annealing. It is a topology-agnostic algorithm and is used to optimize the configuration of the circuit layer on the device layer to reduce latency (see Section 3.7 and Section 5.1).

3

System Architecture

3.1 Overview

We propose a computing architecture which is made up of a hybrid of reconfigurable CMOS compute nodes that are interconnected by an irregular nanowire network. We take this approach because of the ease and the simplicity in fabricating such devices, e.g., by means of a self-assembly approach.

In the following sections, we will provide an overview of our architecture, the *recruitment* of the nodes, the *self-configuration*, and a *self-optimization* algorithm, which optimizes for latency by reconfiguring the network. We explain these steps with the help of a digital logic circuit (circuit layer). We would be explaining the implementation of the architecture and these algorithms in our simulation framework in Chapter 4. Note that all our algorithms are topology-agnostic, i.e., the network topology does not need to be known by the algorithms. This is essential as we have assumed an irregular interconnect network.

Our architecture is built in the simulator as follows:

1. Place N reconfigurable compute nodes at random locations with a bounded 2D region.

2. For each node, chose a maximum of k_{max} or an average of k_{avg} immediate neighbors that lie within a distance of d_{min} and d_{max}.

3. Establish links between these nodes based on either the k_{max} or k_{avg} values.

This process is inspired by how certain types of nanowires grow in real experiments [28,30]. Figure 3.1 shows a sample reconfigurable fabric composed of $N = 20$ nodes and maximum of $k_{max} = 4$ local connections per node. Note that the resulting graphs are not necessarily planar and that we do not take into account possible short-circuits between wires as they might occur in reality. It can be seen that node 20 is not connected to any other node in the network. This might be possible due to the stochastic placement of the nanowire interconnects in a bottom-up manner. The node functionality is described in Section 3.4. The reconfigurable nodes have different modules. These include the *Inter-node communication module*, *Recruitment module*, *Self-configuration module*, *Data processing module* and the *Self-optimization module*. These are assumed to be fabricated through traditional CMOS technology. We will describe these modules in the following sections and subsequently in Chapter 4.

3.2 Routing packets through the network

We assume a serial single-wire bidirectional interconnect network with a NoC paradigm for our architecture. Common voltage source planes are assumed for the network of nodes. Each node has an *Inter-node communication module* which receives packets from its neighbor nodes, decodes the packets and sends them to other modules according to the *Packet mode* (see Section 4.2). It also receives packets from other modules, routes the packet according to the destination and sends them to the neighbor nodes. The detailed architecture of this module is described in Section 4.1.

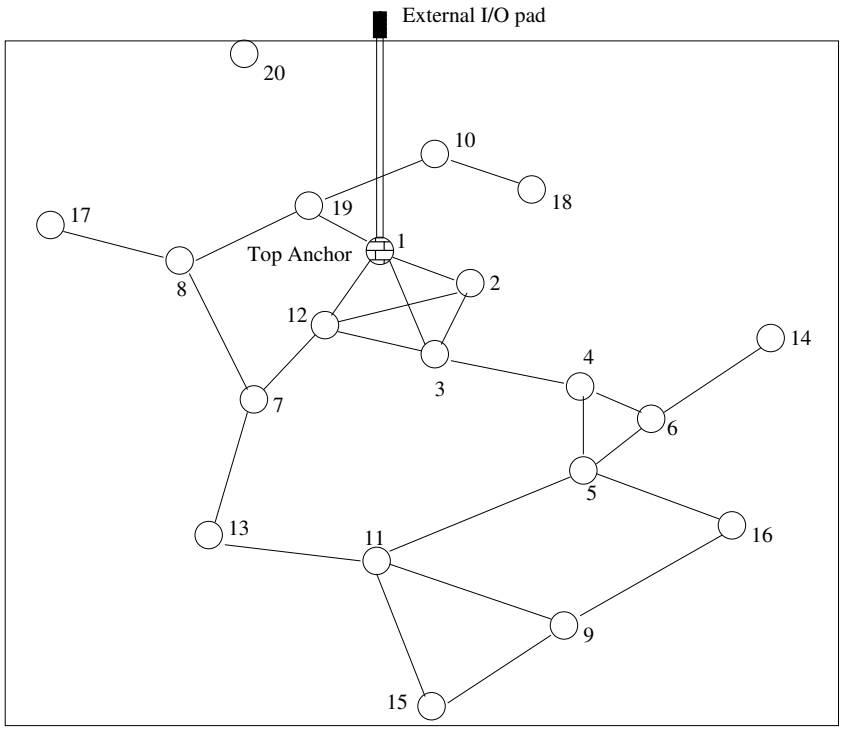


Figure 3.1: A sample reconfigurable fabric composed of $N = 20$ nodes and a maximum of $k_{max} = 4$ local connections per node. Numbers indicate the node identification.

Due to the irregular interconnect network, we cannot use traditional routing techniques. We also aimed for our architecture to be scalable and the hardware costs of each node to be independent of the network size. Hence we developed an addressing scheme and routing methodology which would avoid the use of routing tables.

Addresses are assigned to the compute nodes during the *recruitment* phase (Section 3.3). The recruitment phase organizes the nodes in a logical hierarchy in the form of a tree. Hence, in our routing methodology, we assume packets can only go up and down this logical tree of nodes and the address format we chose supports this routing. Figure 3.2 shows the format of the address of every node. We store the information of the parent and its ancestors all the way up to the *Top Anchor* (Section 3.3) in this address. Each field in the address is assumed to be 3 bits long as we assume a maximum connectivity, k_{max}, of 8 for each node. 3 bits can define when encoded can define all 8 neighbours. When a packet needs to be routed from one node to another, the address values of the current node and the destination node (present in the packet) are compared. There can be three possibilites during this comparison.

- The destination is in the same branch of the virtual tree as the current node and is higher in the hierarchy.
- The destination is in the same branch of the virtual tree as the current node and is lower in the hierarchy.
- The destination is in a different branch of the virtual tree compared to the current node.

If the destination is in the same branch of the logical tree as the current node and is higher in the hierarchy, then the packet is routed to the parent of the current node. Else if the destination is in the same branch of the logical tree but is lower in the hierarchy compared to the current node, then the packet is routed to one of the child branches according to the destination address. Otherwise, the destination is in a completely different branch of the logical tree compared to the current node. The packet in this case is routed to the parent node. As the packet goes higher in the hierarchy, it will eventually find a common branch between the current node and the destination. In this way packets get routed without the use of the routing table.

This method might seem to have a higher delay than other routing algorithms [12]. But it has a few advantages too. Most of the routing techniques require a routing table to be present in each node. The routing table is like a hash table where there is an entry for every node in the network. Each entry is associated with one of the neighbor nodes. Whenever a packet needs to be routed, its destination address is looked up in the routing table and its next node is determined. This means that there should be a large number of entries in every node. It also means that the size of the routing table in each node increases linearly with the number of nodes in the network. Also creating the tables either needs a global view of the system or requires a long time to adapt itself [12]. In our routing methodology, the hardware cost for routing is independent of the network size as there is no need of a routing table to be present. We also do not need a global view of the system to set up the routing.

It might seem the packet length might be very long with this kind of addressing scheme. But we believe the addresses could be encoded similar to IP addresses

in computer networks. In this way, we believe our addressing scheme could be scalable.

Figure 3.2: Format of node address.

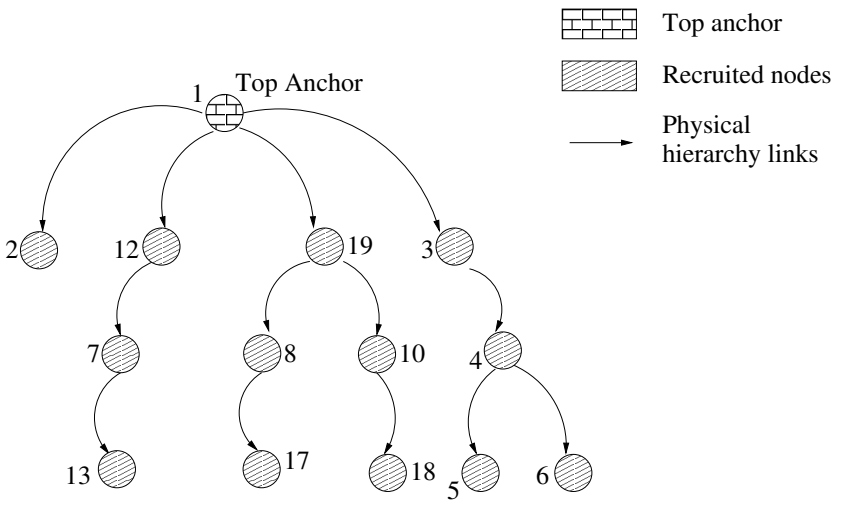
3.3 Anchor and compute node recruitment

In our architecture, a hierarchical control structure is established. Note that this hierarchy is virtual and has no direct representation on the physical reconfigurable fabric of nodes. In principle, any number of hierarchical levels is possible as long as there are enough nodes available. In order to make the system robust, scalable, and decentralized, we have adopted a gradient-based approach that recruits a selected number of levels of nodes. We have taken inspiration from the amorphous computing project for this step [2] (Section 2.3).

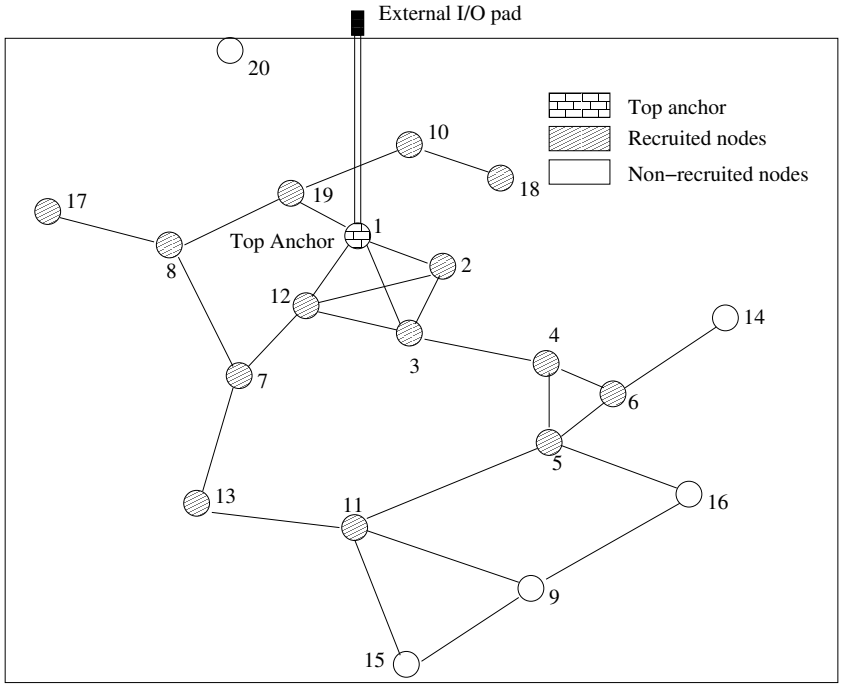
After creating a random initial network (Figure 3.1), a randomly selected node is chosen as a *Top Anchor* node (Figure 3.3(b) - Node ID 1). This node will act as an I/O node to the outside world to receive the digital circuit netlist (circuit layer), the inputs and the outputs in the form of packets. Note that any of the network nodes can function as the *Top Anchor node*.

The recruitment of the nodes is inspired from a chemical gradient flow. In a chemically-inpired gradient flow, an initial anchor particle is assumed to broadcast messages to its neighbors. These messages can count the hops and mark the nodes it has passed through. After a certain count, the messages are not considered and the nodes which the message has marked will form a region [2].

A chemically-inspired gradient flow is initiated from the *Top Anchor* to recruit



(a) Hierarchy of initially recruited nodes. Links represent physical links in the real reconfigurable fabric (see right hand side).



(b) Recruited nodes in the physical reconfigurable fabric.

Figure 3.3: Three hierarchical levels of compute nodes are recruited. Node number 1 is chosen as the top anchor node. At this point, no circuits have been placed in the nodes.

multiple levels of *compute* nodes onto which the gate-level netlist will be mapped on. Gradient packets with a certain initial gradient value are sent through the *Top Anchor* to the neighbors. As they pass through each node, the gradient value of a packet decreases by one and the node is recruited to be a compute node if it has not already been recruited. The gradient flow on the network stops when the gradient value reaches a predefined value. This allows to recruit a certain number of levels of compute nodes within a given network area. In Figure 3.3(b) and Figure 3.3(a), three levels of *compute* nodes are shown to be recruited.

The gradient flow process in our architecture and its algorithm have been explained in detail in Section 4.3. We use 3 different packet modes to achieve this. At the end of the recruitment phase, the recruited nodes are assigned addresses and a virtual hierarchy of nodes are created. These would be the nodes which would be used for computation. In the process of recruitment, faulty nodes and interconnects would be discovered and discarded from the virtual hierarchy of nodes. In this way, static defects and faults are mapped around [15]. In the simulator we have built, only defects in the interconnects are detected and discarded from the logical hierarchy and we assume that the CMOS nodes are perfect. But we believe that it can be extended to detecting defects in the CMOS nodes too with the help if Built-In Self Test (BIST) and different types of feedback packets. But this is beyond the scope of this thesis.

3.4 Node functions and netlist specification

The digital circuit that we place on the reconfigurable fabric is described in the form of a simple gate-level netlist. This netlist can easily be generated by a logic synthesis tool from a high-level language (See Section 2.2).

Table 3.1 shows the list of logic gates each node can be configured into and the associated node IDs that are used in the netlist to identify the node type. We have chosen only 2-input logic gates in our architecture as we are more interested in building the architecture in this thesis rather than exploring the granularity of the reconfigurable unit (See Section 2.1). We have also chosen these 5 logic gates as shown in Table 3.1 as they are the most basic logic functions and any digital circuit can be built with the combination of these gates. The digital logic circuit shown in Figure 3.4 is represented in the form of a netlist in Table 3.2. The identifiers are natural numbers and each line of Table 3.2 represents a logic gate. While the choice of these functions may seem limiting, our basic concept is very easy to extend to more complex nodes and what we present here should be considered merely as a proof of concept.

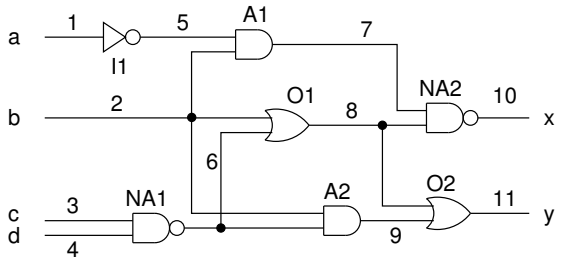


Figure 3.4: Example of a digital logic circuit used throughout the paper to illustrate our approach.

Logic gate	Logic gate ID
AND	1
OR	2
NOT	3
XOR	4
NAND	5

Table 3.1: Logic gates and gate IDs.

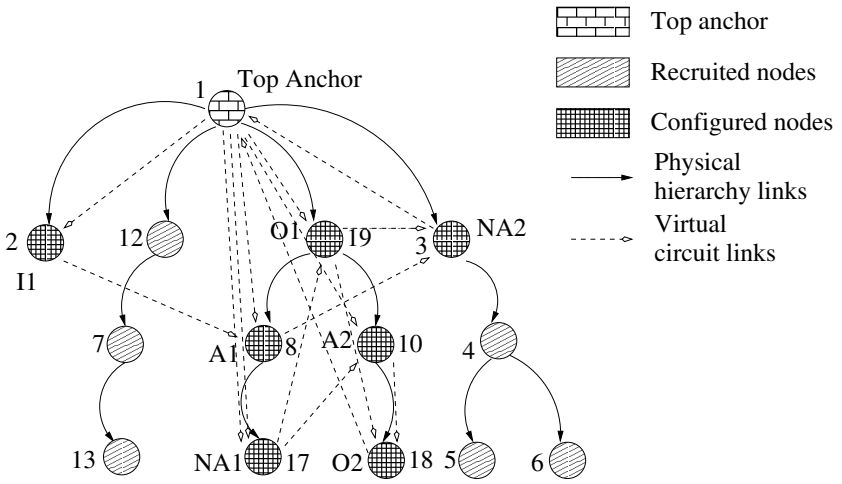
Logic gate ID	Dest. port ID	Source 1 port ID	Source 2 port ID
3	5	1	0
2	8	2	6
5	6	3	4
1	7	2	5
1	9	2	6
5	10	7	8
2	11	8	9

Table 3.2: Netlist example of the logic circuit represented in Figure 3.4.

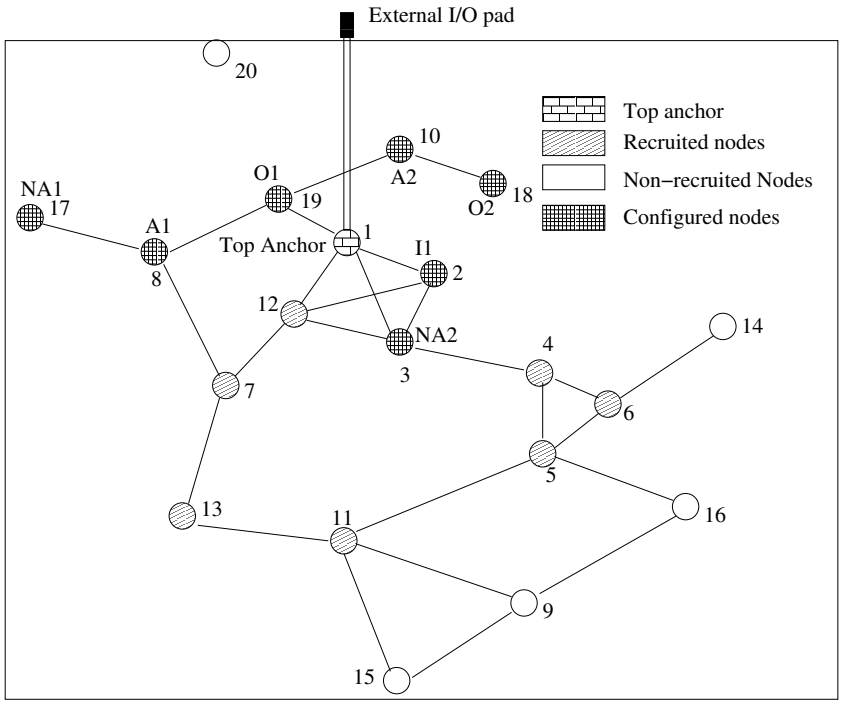
3.5 Configuration

The configuration of the gate-level netlist onto the network of nodes is performed by a *Self-configuration* algorithm which works as follows. First, the configuration is divided into two stages. The first stage involves the *placement* of the netlist onto the recruited compute nodes. Each line of the netlist is sent through the *Top Anchor* in the form of a packet to the compute nodes. If a compute node has no netlist placed in it, it gets placed. Otherwise, the netlist packet is routed to one of its *free* child branches. Each node in the hierarchical network keeps a flag on whether there are any (*free*) compute nodes in each of its child branches. This is done with the help of receiving feedback packets once all the nodes in the child branch have been placed. At the very end of the configuration phase, a feedback packet is finally received by the *Top Anchor*. These steps and the different packet modes used are described in detail in Section 4.4.1.

The second stage of configuration involves *mapping* the source and destination of the placed nodes according to the netlist. A packet sent through the *Top Anchor* and down the hierarchy initializes the second stage of the configuration. Each placed node initializes a packet flow to discover its source and destination. Once found, they are mapped onto the placed node. A feedback packet is received by



(a) Hierarchy of configured nodes. Dashed links represent the circuit connections of the digital logic circuit example shown in Figure 3.4.



(b) The configured nodes in the physical reconfigurable fabric.

Figure 3.5: The digital logic circuit shown in Figure 3.4 is configured and mapped onto the random reconfigurable fabric. There is a significant difference in the mapping between the physical and the virtual links.

the *Top Anchor*, which marks the end of the configuration. Figure 3.5(a) and Figure 3.5(b) show the network with the digital circuit configured onto it. No optimization step was performed at this stage. The detailed mapping phase and the different packet modes used for this step in our architecture has been described in Section 4.4.2.

3.6 Processing data

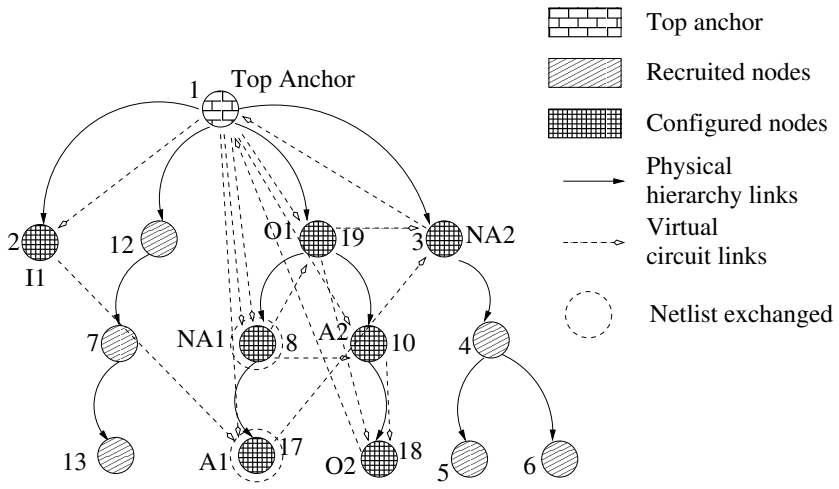
Once the digital circuit has been configured, the network of compute nodes is now ready to process data packets according to the netlist. Each data packet has an associated *netlistID* and *dataID*. The *netlistID* determines which input the data value corresponds to. The *dataID* determines which instance of the input the data corresponds to. For a given 2-input gate, the *dataID* of both the inputs should match before they are processed. As the *Top Anchor* is the only node communicating with the external environment, all computations begin and end at the *Top Anchor*. The design and the steps involved in processing the packet in our architecture are described in Section 4.5.

3.7 Self-optimization of the mapping

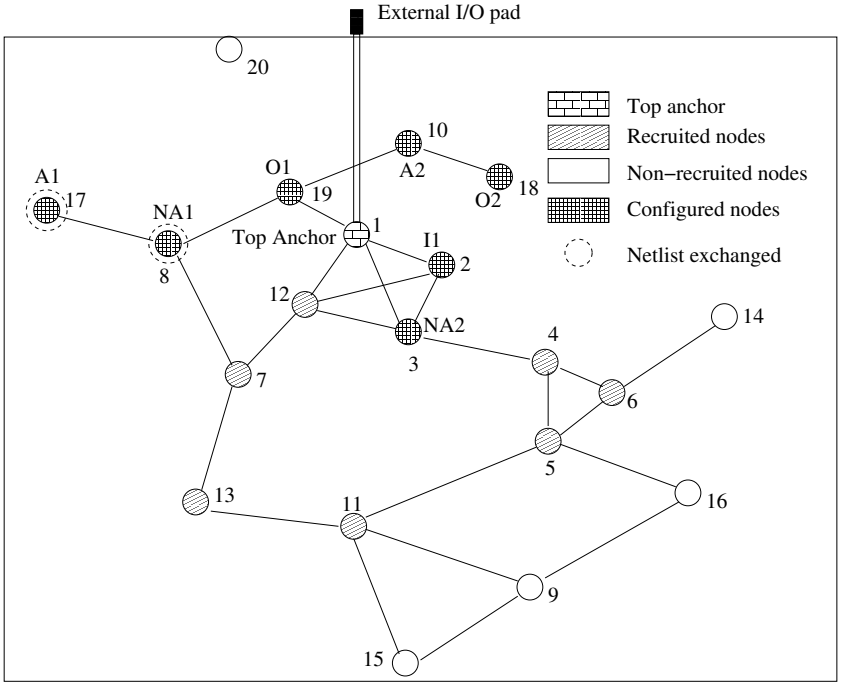
Once an initial placement has been established, we apply a topology-agnostic *Self-optimization* algorithm inspired by simulated annealing. Each node has an optimization module that can access the configuration mapping and netlist of only its immediate neighbor nodes. Unlike conventional FPGA CAD placement and routing tools, no single node in our network and no external controller has access

to the entire configuration mapping of the network. In our case, the simple goal is to get nodes that communicate with each other as physically close as possible, i.e, get the virtual links in the circuit layer to overlap as much as possible with the physical links in the device layer. Ideally, each node that communicates with another node on the logical level should be placed in a way that there is a physical connection between them. This will in most cases not be possible because of the reconfigurable fabrics restrictions and also the way we organize the network of nodes in a hierarchy. Nodes that do not have a direct physical link will communicate by sending data packets through the network. To initiate the self-optimization algorithm, the external controller sends a packet to the *Top Anchor*. This packet carries information about the number of optimization runs similar to simulated annealing. We will be describing the algorithm in detail in Section 4.6.

Figure 3.6(a) shows the network hierarchy and mapping after one optimization step. It can be seen that the virtual gate links common to NA1 (4 links) were more than that of A1 (3 links), and hence the exchange of the netlist and mapping between nodes 8 and 17 took place. Figure 3.7(a) shows the final network hierarchy and Figure 3.7(b) shows the actual network of nodes after 1,000 optimization steps. The latency improved by 33% for this particular digital circuit and network topology (see Section 5.1).

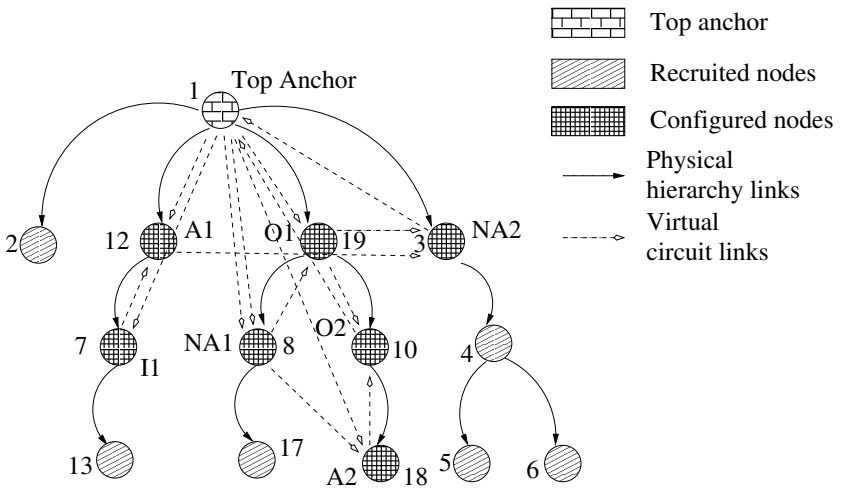


(a) Hierarchy of configured nodes optimized after one optimization step.

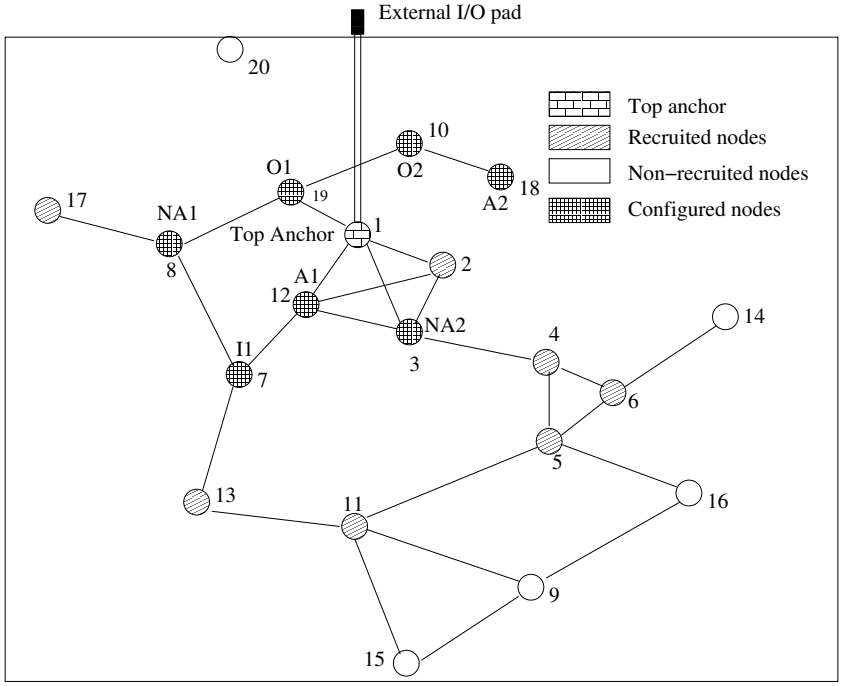


(b) The configured nodes in the physical reconfigurable fabric after one optimization step.

Figure 3.6: Network node configurations after one optimization step.



(a) Hierarchy of configured nodes optimized after $1,000$ optimization steps.



(b) The configured nodes in the physical reconfigurable fabric after $1,000$ optimization steps.

Figure 3.7: Network node configurations after $1,000$ optimization steps. The mapping between the physical and the virtual links almost overlap each other. In this way, the number of hops and eventually the latency to process the data packet is reduced.

4

Simulation Framework

In this chapter, we will be discussing our architecture in detail including the algorithms implemented and the different packet modes used. We have implemented a software and a hardware framework to evaluate the performance of our architecture. The software framework was built using MATLAB. This was used to evaluate the latency and energy consumption (see Chapter 5). A hardware framework of one compute node was designed using VHDL and synthesized using Synopsys Design Compiler for a 65nm CMOS technology. A database of power values was created which was then used by the software framework to evaluate the energy consumption (see Chapter 3.1). The software framework is not cycle-accurate but with the help of the hardware module we have assumed a time value for each iteration while

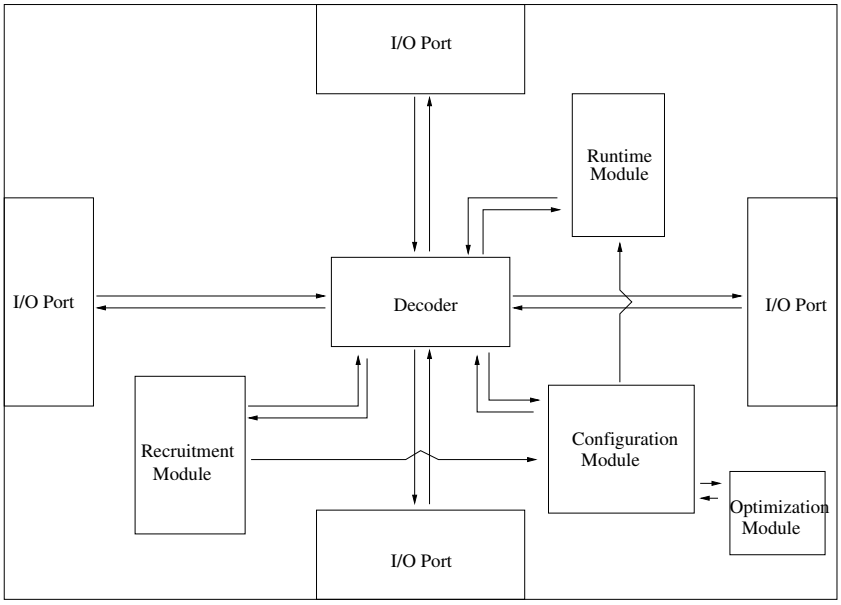


Figure 4.1: Structure of each compute node.

evaluating the performance (see Chapter 5).

A set of compute nodes is initialized and connected randomly but constrained by the number of immediate neighbors (connectivity) and by a minimum and maximum distance between the neighbors. Each node consists of multiple modules which implement the functions required to perform the system operations (Figure 4.1).

4.1 Inter-node communication module

We assume a serial single wire, bidirectional interconnect network with a NoC paradigm for our architecture. For simplicity, a serial communication between the nodes with a synchronous clock is assumed. The communication model can easily be extended to an asynchronous model, however, this is beyond the scope of this thesis.

A communication module called *Switch Node* is implemented in each compute node. The module consists of channel buffers, multiplexors and demultiplexors. It receives packets from its neighbor nodes, decodes the packets and sends them to other modules according to the *Packet mode* (see Section 4.2). It also receives packets from other modules, routes the packet according to the destination and sends it to the neighbor nodes. This module was inspired by the model proposed by Pande, *et al.* [20].

In the implementation of the software framework in MATLAB, we have assumed two iterative states to be present. In the first iteration, every node decodes one packet present in the input buffers, processes the packet and routes resulting packets, if any, to the output channel buffer. In the alternate iteration, the output

channel buffers of every node sends packets, if any, to the input channel buffers of the next node. This makes the simulator not cycle accurate. But with the help of the hardware implementation in VHDL and a nanowire model, we evaluate the latency in this framework. This is further explained in Section 5.1 .

The *switchNode* feeds and gets packets from the decoder module (4.2) present within each compute node. The *Input Module* and the *Output Module* are complementary modules (Figure 4.1).

4.1.1 Packet structure

Each packet is assumed to be 128 bits long. The first bit is called *packetValidBit* and determines whether the packet data is valid (if set) or invalid. The next 3 bits are called *Virtual Channel ID (VCID)* and determine the virtual channel [20] which the packet belongs to. The remaining bits will be discussed in Section 4.2.

4.1.2 Channel buffer

The channel buffer is a circular buffer (Figure 4.2) with a certain channel length. It is the basic building block of the input and output virtual channels. The buffer is built as a 3-state machine with a cyclic state diagram as shown in (Figure 4.3).

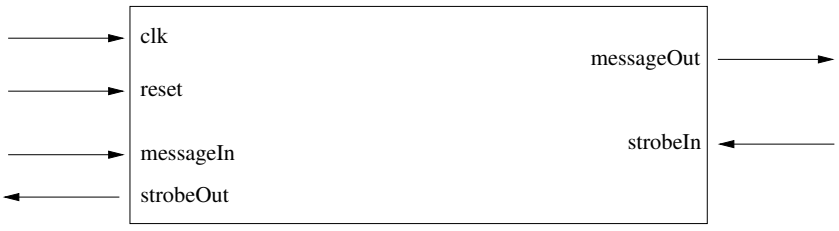


Figure 4.2: Circular buffer - the basic component of the virtual channels.

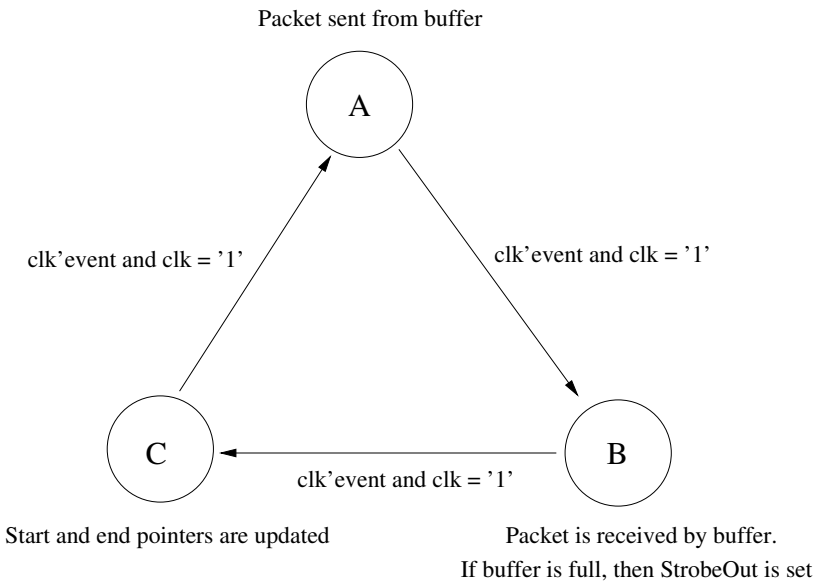


Figure 4.3: State machine for the circular buffer.

In State A, if the channel buffer is not empty, then the packet pointed by the *startPtr* is sent to the next module (emphmessageOut). In State B, a packet is received from another module (*messageIn*, if any, and if the buffer is not full. If it is full, then *strobeOut* is set. In State C, the *startPtr* is updated if the *strobeIn* is reset. *endPtr* is updated if there has been message written into in State B.

4.1.3 Input module

The *Input Module* (Figure 4.4) consists of a *inputPort* which basically is a demultiplexor. It receives a packet from a neighbor, then the input packet is sent to the respective input virtual channel according to the *VCID* of the packet. The input virtual channel is a circular buffer and stores the packet if empty. Otherwise, it is discarded and feedback is given to the previous node that the packet needs to be sent again. Currently, a strobe is set to indicate this but a better communication

protocol could be implemented to address this drawback. The *inputMuxArbiter* and the *inputMux* select one of the virtual channel outputs to be presented to the *decoderModule* through a buffer.

4.1.4 Output module

The *Output Module* (Figure 4.5) consists of a *outputPort* which is essentially a multiplexor. It chooses a packet from one of the output virtual channel based on an arbiter and sends the packet to a neighbor. A demultiplexor feeds packets into the virtual channel according to the *VCID* of the packet from the *decoderModule*.

4.2 Packet decoder module

The *Packet Decoder module* (Figure 4.6) decodes the packet coming from the input ports and loads it into either the *Recruitment module*, the *Configuration module*, the *Runtime module* or the *Optimization module*. It also routes the packets to the necessary output ports based on the destination address.

There are nine packet formats differentiated by different packet modes (Table 4.1). Modes 1-3 are *Recruitment packets* processed by the *Recruitment module* 4.3. Modes 4-8 are *Configuration packets* processed by the *Configuration module*. Mode 9 is the actual data packets which are sent through the network to be processed after the network has been configured with a digital circuit. These packets are processed by the *Runtime module*.

Figure 4.4: Input module.

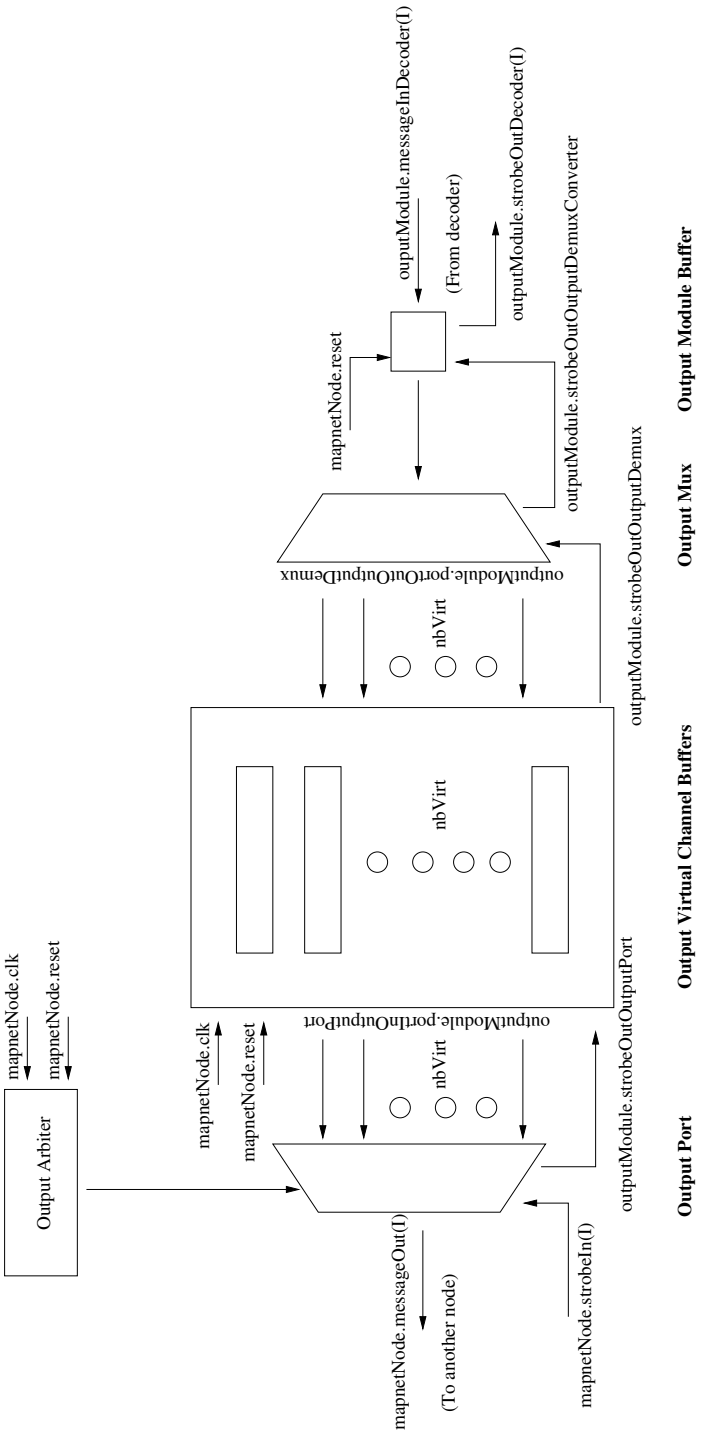


Figure 4.5: Output module.

Packet type	Bits 5-8	Bits 9-28	Bits 29-48	Bits 49-52	Bits 53-56	Bits 57-76	Bits 77-96	Bit 97	Bits 98-99	Bit 100	Bits 101-104
Recruit Init	Mode = 1	Source Address	Dest. Address	Current Gradient	0	0	Max Recruit Gradient	0	0	0	portID
Recruit FB 1	Mode = 2	Source Address	Dest. Address	Child Flag	0	0	0	0	0	0	portID
Recruit FB 2	Mode = 3	Source Address	Dest. Address	0	0	0	0	0	0	0	portID
Config Netlist	Mode = 4	Source Address	Dest. Address	Function ID	Output netlist	Input 1 netlist	Input 2 netlist	Config Bit Flag	Gate ID	Config FB2 Flag	portID
Config Mapping Init	Mode = 5	Source Address	Dest. Address	0	0	0	0	0	0	0	portID
Config FB 2	Mode = 6	Source Address	Dest. Address	Function ID	0	0	0	0	Gate ID	0	portID
Config FB 1	Mode = 7	Source Address	Dest. Address	Netlist ID needed	Source Found Flag	ID Source address	ID Dest. address	0	0	0	portID
Config FB 3	Mode = 8	Source Address	Dest. Address	0	0	0	0	0	0	0	portID
Data Packet	Mode = 9	Source Address	Dest. Address	Data	Netlist ID	Packet ID	0	0	0	0	portID

Table 4.1: Different packet modes.

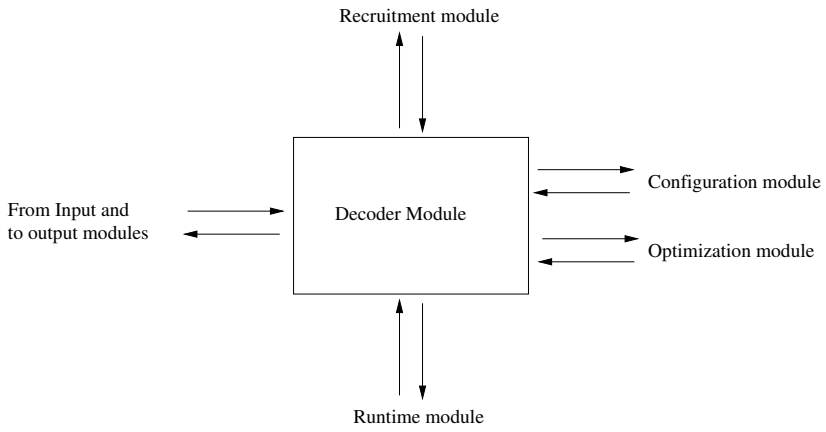


Figure 4.6: Decoder module.

4.3 Recruitment module

The *Recruitment Module* is responsible for recruiting the compute nodes in a hierarchy based on a *chemically-inspired algorithm* (Section 3.3). It communicates with the *Packet decoder module* and deals with packet modes 1 - 3. The following subsections describe the algorithm the framework performs to process these packets.

There are 3 buffers present in the recruitment module, which play a crucial role in completing the recruitment and configuration of the network of nodes successfully.

- Recruitment flag,
- Configuration flag, and
- Config mapping flag.

All three buffers are assumed to have as many bits as the number of ports (connectivity) of each node. All the bits of the *Recruitment Flag* are initialized to 1. Their purpose is to determine the end of different steps and to initiate another

step. This will be discussed in detail in the next few subsections and in Section 4.4.

4.3.1 Recruitment initialization packet (RecInit)

This packet initializes the recruitment process. It is also responsible for assigning the address bits to nodes. The packet contains the *currentGradient* and the *maxGradient* values. It is initially sent to the *Top Anchor* node through the external I/O port. The *currentGradient* value is assumed to be initialized to 0. The *Top Anchor* broadcasts the packet to all its neighboring nodes after updating *currentGradient* value and the *address*. The *currentGradient* value is updated by incrementing the existing value by 1.

If a neighbor node has not been recruited before, i.e., there has been no more than one *RecInit* that has passed through that node, then the *address* will be set as the address for the neighboring node and sets the *source* of the packet as the parent. It also sends a *RecFB1* packet to the sender with the *childFlag* in the packet set indicating that this node can be assigned as a child node. If the *currentGradient* value is less than the *maxGradient* value, then that implies that the *Chemical Gradient* can flow down the gradient i.e. another hierarchy of nodes can be recruited. The *currentGradient* value and the *address* value are updated and a *RecInit* packet is then sent to each of the neighbors, including the parent.

If the neighbor node was already recruited, then a *RecFB1* packet is sent to the sender with the *childFlag* in the packet reset. Algorithm 1 summarizes the above steps.

1: **if** not recruited before **then**
2: Save gradient value, parent address and port ID. Set node as *Compute Node*. Send *RecFB1* packet to parent with *childFlag* set.
3: **if** currentGradient < maxGradient value **then**
4: Send *RecInit* packets to all neighbors with updated gradient values and addresses.
5: **else**
6: Recruitment done through this node. Send *RecFB2* packet to parent.
7: **end if**
8: **else**
9: Send *RecFB1* to sender with *childFlag* reset.
10: **end if**

Algorithm 1: Recruitment of compute nodes initialization packet (RecInit) flow.

4.3.2 Recruitment feedback 1 packet (RecFB1)

This packet is one of the feedback packets which is received after a *RecInit* packet is sent by a node. It updates the relationship of the sender, i.e., a neighbor with itself. If the *childFlag* in the packet is set, then the sender is set as a child node. The bit corresponding to the sender in the *Configuration Flag* and *Config Mapping Flag* are set.

If the *childFlag* in the packet is reset and if the sender is the parent, then the *Recruitment Flag* is updated by resetting the bit corresponding to the parent. If the sender is not even the parent, then that neighbor is updated as not a child or not a parent. This means, according to our model, that the link between these two neighbors is virtually non-existent for packet modes 2-9. Although this might seem like a waste of resources that are not being utilized, it eliminates the need of a routing table (see Section 3.2). Note that such links are later used during the *Optimization Phase* though (see Section 4.6).

After updating the *Recruitment Flag* buffer, all bits of the buffer are checked. If all are reset, that implies that recruitment cannot happen further through that

node and a *RecFB2* packet is sent to the parent to indicate this. Algorithm 2 summarizes this subsection.

1: **if** *childFlag* is set **then**
2: Update neighbor information with the sender as a *child*. Update *Configuration Flag* and *Config Mapping Flag* buffers by setting the bit corresponding to the sender.
3: **else**
4: **if** sender is parent **then**
5: Update *Recruitment Flag* buffer by resetting the bit corresponding to the sender (parent).
6: **else**
7: Update neighbor information with sender as neither child nor parent. Also update *Recruitment Flag* buffer by resetting the bit corresponding to the sender.
8: **end if**
9: **if** all bits of *Recruitment Flag* are reset **then**
10: Recruitment done through this node. Send *RecFB2* packet to parent.
11: **end if**
12: **end if**

Algorithm 2: Recruitment feedback 1 packet (RecFB1) flow.

4.3.3 Recruitment feedback 2 packet (RecFB2)

This packet type is sent by a child to indicate that the recruitment (assigning addresses to the nodes for the required levels of hierarchy) is done through that child node. The bit corresponding to that child node in the *Recruitment Flag* is then reset. All bits of the buffer are then checked. If all are reset, then a *RecFB2* packet is created and sent to the parent node. In that way, the *Top Anchor* finally indicates to the external environment when the recruitment is completed.

1: Update neighbor information with sender as neither child nor parent. Also update *Recruitment Flag* buffer by resetting the bit corresponding to the sender.
2: **if** all bits of *Recruitment Flag* are reset **then**
3: **if** node is a *Top Anchor* **then**
4: Indicate recruitment of network is done.
5: **else**
6: Recruitment done through this node. Send *RecFB2* packet to parent.
7: **end if**
8: **end if**

Algorithm 3: Recruitment feedback 2 packet (RecFB2) flow.

4.4 Configuration module

The *Configuration Module* is responsible for *placing* and *mapping* the netlist of the digital circuit onto the network of compute nodes. Placing can be defined as configuring the compute nodes with each row of the netlist (a 2-input logic gate). Mapping can be defined as the process where the configured nodes discovers where its inputs should be obtained from. Packet modes 4 - 8 help in configuring the digital circuit.

The following subsections describe the purpose of each packet mode and how the configuration module processes them.

4.4.1 Placing of netlist

Configuration netlist packet (ConfigNetlist)

This packet type consists of one row of the netlist of the digital circuit that we want to configure onto the network. The *Top Anchor*, on receiving this packet from the external environment, forwards the packet to one of the children in the order of which child responded first with *RecFB1* packet during the recruitment

phase. If the *configFB2Flag* was set, then the *Top Anchor* stores the *gateID* of the packet in a *Config FB2 Check buffer*. The importance of the *configFB2Flag* will be explained in the following subsections.

A compute node, on receiving this packet type, will check whether it has been configured with any netlist. In that case, the netlist in the packet is configured onto the node with a *gateID*. If *configFB2Flag* in the packet is set, then a *ConfigFB2* packet is sent to the *Top Anchor*.

If the compute node has already been configured and the *configBitFlag* in the packet is reset, then that implies that the packet has definitely come from one of the children. The *Configuration Flag* buffer in the node is updated by resetting the bit corresponding to that child. If all the bits of the *Configuration Flag* are reset, then the *ConfigNetlist* packet is forwarded to the parent node with the *configBitFlag* reset. Otherwise, the *ConfigNetlist* packet is forwarded to one of the child nodes in which corresponding bit in the *Configuration Flag* is set. The *configBitFlag*, present in the packet, is set.

A summary of the algorithm is presented in Algorithm 4.

Configuration feedback 2 packet (ConfigFB2)

This packet type helps in keeping track of whether certain gates have been configured successfully. In Algorithm 4, it was observed that *gateID* of certain *Config Netlist* packets were stored in the *Config FB2 Check Buffer* of the *Top Anchor*. When a *Top Anchor* receives a *ConfigFB2 packet*, the *Top Anchor* node checks off the *gateID* in the *Config FB2 Check Buffer*. Once all the Gate ID's have been checked off, that implies that the netlist placement steps for the particular digital

1: **if** *Compute Node* and no netlist configured **then**
2: Save netlist and gate ID.
3: **if** *configFB2Flag* in packet is set **then**
4: Send *configFB2* to the *Top Anchor* with the stored gate ID.
5: **end if**
6: **else if** node is *Top Anchor* **then**
7: **if** *configBitFlag* and *configFB2Flag* is set **then**
8: Packet has come from external environment. Store the Gate ID corresponding to packet in *configurationFB2CheckBuffer*. Send the *configNetlist* packet to one of the children where the bit corresponding to it in the *Configuration Flag* buffer is set.
9: **end if**
10: **else**
11: **if** *configBitFlag* is reset **then**
12: *configNetlist* packet has definitely come from a child. Update *Configuration Flag* buffer by resetting bit corresponding to child.
13: **end if**
14: **if** all bits of *Configuration Flag* buffer are reset **then**
15: Send the *configNetlist* packet with the *configBitFlag* reset.
16: **else**
17: Send the *configNetlist* packet to one of the children where the bit corresponding to it in the *Configuration Flag* buffer is set. Set the *configBitFlag* in the packet.
18: **end if**
19: **end if**

Algorithm 4: Configuration netlist packet (configNetlist) flow.

circuit were completed . This is a step which seems to need an unlimited buffer size for the *Config FB2 Check Buffer*. But we chose to inform the external environment that configuration of all the gates have been done. So the *Config FB2 Check Buffer* could also be present in the external environment and hence could reduce the hardware cost.

The algorithm is summarized in Algorithm 5.

1: **if** node is a *Top Anchor* **then**
2: Mark as *configFB2* received for the particular gateID in *Config FB2 Check Buffer*.
3: **else**
4: Route packet to *Top Anchor*.
5: **end if**

Algorithm 5: Configuration feedback 2 packet (configFB2) flow.

4.4.2 Mapping the netlist

Configuration mapping init packet (Config Map Init)

Once the placement of the digital circuit onto the network is done, the nodes need to discover the sources of their input packets according to the netlist data. A *Config Map Init* packet is sent to the *Top Anchor* which initializes this process in all the nodes. The *Top Anchor* forwards this packet to all its children. A configured compute node on receiving this packet also forwards it to all its children. It then creates a *Config FB1* packets for each of its inputs. This packet contains the netlist needed and also the compute node address. The *Config FB1* packets are broadcast to the parent and all the children.

If the compute node is not configured and all the bits of the *Config Mapping Flag* are reset, that implies that there is no further mapping needed through this node

down the hierarchy. A *Config FB3* packet is then created and sent to the parent indicating that mapping phase has been completed through this node.

Algorithm 6 summarizes the processing of the Config Map Init packet.

1: Forward *configMapInit* to all children
2: **if** node is a *Compute Node* and is configured with a netlist **then**
3: Setup *configFB1* packet with *sourceFound* flag reset for every input netlist ID. Send these packets to parent and all children.
4: **else**
5: Node has no netlist configured.
6: **if** all bits of *Config Mapping Flag* are reset **then**
7: Configuration mapping done for current node and its children. Send *configFB3* packet to the parent.
8: **end if**
9: **end if**

Algorithm 6: Configuration mapping init packet (configMapInit) flow.

Configuration feedback 1 packet (Config FB1)

This packet type helps in finding the source for a given netlistID. If the *sourceFound* flag in the packet is reset, that implies that the source has still not been found. If the compute node is configured or if the node is a *Top Anchor*, then the required netlist ID is checked with the *Output Netlist ID* of the node. If it matches, it implies that this node would be the source of the data for the given netlist ID. Another *Config FB1* packet is prepared with the *source* field set to be the current node address. The *sourceFound* flag is also set and the packet is routed to the address present in the *destination* field in the packet. If there is no match in the netlist, the *Config FB1* packet is forwarded to all the children and parent except the one it was received from.

If the *sourceFound* flag is set and the *destination* address matches the current node

address, then the address present in the *source* field is saved to the *Input Netlist ID* corresponding the required netlist ID present in the packet. This implies that the *Config FB1* packet has successfully discovered the source for that particular input. If all the inputs are mapped and all the bits of the *Config Mapping Flag* are reset, that implies the configuration mapping is done for the current node and its children. A *Config FB3* is setup and sent to the parent to indicate this.

If the *destination* address field does not match the current node address, then the packet is routed according to hierarchical addressing.

Algorithm 7 summarizes the processing of *Config FB1* packet.

```
1:  if sourceFound flag in packet is reset then
2:    if node is Compute Node and the netlist ID needed matches the output
      netlist ID of the node then
3:      Source found for netlist ID. Save destination of the netlist ID to the
        outputMapping buffer. Send configFB1 to netlist ID destination with
        sourceFound flag set.
4:    else
5:      Forward configFB1 packet to parent and children except to the node
        from where it received it.
6:    end if
7:  else
8:    Source to netlist has been found.
9:    if destination of configFB1 packet reached then
10:     Save source address of packet to corresponding inputMapping buffers.
11:     if all inputs mapped and all bits of Config Mapping Flag are reset then
12:       Configuration mapping done for current node and its children. Send
          configFB3 packet to the parent.
13:     end if
14:   else
15:     Route packet to destination.
16:   end if
17: end if
```

Algorithm 7: Configuration feedback 1 packet (configFB1) flow.

Configuration feedback 3 packet (Config FB3)

This packet type is sent to the parent to indicate when the mapping phase is done for a node and for all its children. The *Config Mapping Flag* buffer keeps track of whether the children have finished the mapping phase. Upon receiving this packet, a recruited node updates its *Config Mapping Flag* buffer by resetting the bit corresponding to the child it received the packet from. The *Config Mapping Flag* buffer is then checked. If all the bits in the buffer are reset, the node is done with mapping, all its inputs are configured, then another *Config FB3* is prepared and sent to the parent.

Algorithm 8 summarizes the processing steps.

```
1: Update Config Mapping Flag buffer by resetting bit corresponding to child.
2: if all bits of Config Mapping Flag are reset then
3:    if node is Top Anchor then
4:       Indicate configuration mapping of network is done.
5:    else
6:       Configuration mapping done for current node and its children. Send
         configFB3 packet to the parent.
7:    end if
8: end if
```

Algorithm 8: Configuration feedback 3 packet (configFB3) flow.

4.5 Data processing module

4.5.1 Runtime data packet

This packet type contains the actual data which needs processing after the network has been configured with a digital circuit. The packet consists of 4-bit data, the netlist ID, the packet ID and the destination address. The packets are stored in

a buffer once it reaches the destination and are processed only if all the inputs for the particular packet ID are ready. The output is then forwarded to all the required nodes according to the destinations marked in the *outputMapping* buffer. Algorithm 9 summarizes the steps in processing a runtime date packet.

1: **if** destination of *runtimeData* packet reached **then**
2: Push data with packet ID into input buffer corresponding to the packets netlist ID and packet ID.
3: **if** All inputs of a packet ID are ready **then**
4: Process inputs according to configured function. Send output to all output addresses in *outputMapping* buffers.
5: **end if**
6: **else**
7: Route packet to destination.
8: **end if**

Algorithm 9: Runtime data packet (runtimeData) flow.

4.6 Self-optimization module

Once an initial placement has been established, we apply a topology-agnostic *Self-optimization* algorithm inspired by simulated annealing (see Section 2.5). Each node has an optimization module that can access the configuration mapping and netlist of only its immediate neighbor nodes. Though we have not implemented a packet-based method to acquire the information of the neighbors, we assume it could be possible easily in the same way we implemented the different packet modes discussed in Sections 4.3, 4.4 and 4.5. Unlike conventional FPGA CAD placement and routing tools, no single node in our network and no external controller has access to the entire configuration mapping of the network. In our case, the simple goal is to get nodes that communicate with each other as physically close as possible, i.e., get the virtual links in the circuit layer to overlap as much as possible

with the physical links in the device layer. Ideally, each node that communicates with another node on the logical level should be placed so that there is a physical connection between them.

To initiate the self-optimization algorithm, the external controller sends a packet to the *Top Anchor*. The packet contains information about the number of optimization runs similar to simulated annealing. It also contains a number called the *Optimization factor*. The *Top Anchor* then sends an initialization packet to a random recruited node 'a' in every iteration. If the recruited node is configured with a netlist data, then an iterative comparison is made with the configuration information of the each of its neighbors 'b' as follows.

We assume two variables *oldHops* and *newHops*. As discussed in Section 4.4, the configuration mapping process involves storing the source and destination addresses of the inputs and the outputs of the netlist, respectively. We assume a mapping information 'm1' for the node 'a' and mapping information 'm2' for a neighbor node 'b'. We evaluate the variable *oldHops* as the sum of sum of hops of all source and destination address of mapping 'm1' from node address of 'a' and of all source and destination address of mapping 'm2' from node address of 'b'. We then evaluate *newHops* in a similar way as to *oldHops* except that the mapping information in the two sums are switched i.e. the mapping information are assumed to have exchanged between the neighbor 'b' and the chosen node 'a'. An array of *oldHops* and *newHops* are eventually created for each neighbor of node 'a'.

We then compare the ratio of *oldHops* with the corresponding *newHops* values. If the ratio is greater than the *Optimization factor* for at least one of the neighbor

comparisons, then the netlist of one of those neighbors is exchanged with the chosen node 'a'. The chosen node 'a' then sends a packet to the *Top Anchor* indicating it to start of a *Configuration mapping* phase (Section 4.4.2). The algorithm for the self-optimization has been summarized in Algorithm 10.

It can be seen from the algorithm that each node communicates with only its im-

```
1: for optimization_runs = 1 to MAX_OPT_RUNS do
2:    Top Anchor sends an optimization initialization packet to a random
      compute node 'a'.
3:    if Compute node 'a' is configured with mapping 'm1' then
4:       for neighbor'b' = 1 to Allneighboursofcomputenode'a' do
5:          Get neighbor's address and netlist and mapping 'm2' if they exist
6:          oldHops = Sum(Hops of source and destination of 'm1' from 'a') +
            Sum(Hops of source and destination of 'm2' from 'b')
7:          newHops = Sum(Hops of source and destination of 'm1' from 'b') +
            Sum(Hops of source and destination of 'm2' from 'a')
8:       end for
9:       if oldHops >= OPT_FACTOR * newHops then
10:         Exchange netlist. Initialize stage 2 of configuration.
11:      end if
12:   end if
13: end for
```

Algorithm 10: Self Optimization algorithm.

mediate neighbors during the optimization procedure. The sum of hops from the source and destination can be evaluated based on simply comparing the addresses of the node and the mapping information of the neighbors in a similar way to routing a packet to its destination (Section 3.2).

5

Evaluation Metrics

We built a simulator to evaluate the performance using MATLAB. It initializes an irregular network of compute nodes with one node selected as a *Top Anchor*. The recruitment, self-configuration, circuit data processing and self-optimization steps are simulated using different packet modes. Different performance parameters are evaluated by this simulator.

We also built a single compute node in VHDL. We successfully simulated it for different packet modes. We also synthesized the node for a 65nm CMOS technology for different node connectivity values. We obtained certain performance parameters from this model too.

We evaluate different metrics using our framework. These are described further in the following subsections.

5.1 Latency

As the data moves between the nodes in the form of packets, we measure latency to determine the time taken for a set of input data to get processed. Each packet in the simulator has an additional field which is incremented by the (1) interconnect delay (d_{wire}) and (2) the node delay (d_{node}) while it passes through the network.

In order to evaluate the interconnect delay (d_{wire}), we have implemented a nanowire model as described in [23]. Nodes send data packets between each other through a 15nm x 15nm Cu nanowire. Each nanowire is separated from other nanowires by a distance sufficient for the nanowire to have negligible wire-coupling capacitance.

Silicon dioxide is the insulator surrounding each nanowire. Hence, the capacitance per unit distance would be 1.0354pF/m [23]. The resistance per unit length of the nanowire is calculated to be 330 ohms/μm. The unit length of the network created by the framework is estimated to 5000μm, which is a conservative value, considering that the area of each node for a 10-port and buffer channel length of 10 obtained by using the Synopsys Design Compiler was 250,000 sq. μm for a 65nm node technology. As data is transmitted serially, the number of bits of each data packet influences the speed of computation. For each logic circuit simulated, the maximum distance d_{Max} of a nanowire and the number of bits per packet are taken as inputs. The interconnect delay is calculated with a simple RC delay model as follows [23]:

$$\begin{aligned} d_{wire} &= \text{Nb. of bits per packet} \times {d_{Max}}^2 \times \text{R} \times \text{C} \\ &= \text{Nb. of bits per packet} \times {d_{Max}}^2 \times 330 \times 1.0354 \times 5000^2\text{ns} \\ &= \text{Nb. of bits per packet} \times {d_{Max}}^2 \times 8.5422\text{ns} \end{aligned}$$

A clock of 500 Mhz was assumed for each node. A database of successful simulation times for different packet modes was created while simulating the VHDL node model. The times were recorded assuming that only one packet was being processed. These values were then used in the MATLAB framework to evaluate the node delay (d_{node}). When the packets are waiting in the buffers, the latency for the packet was incremented by the time it took to process the previous packet. For a 128-bit packet, the interconnect delay and the node delay for all packets were in the same order of magnitude. Hence it was a reasonable assumption to make and implement the MATLAB framework to have two iterative states (see Section

4.1).

$$\text{Latency of a packet} \quad = \quad \text{Total } d_{wire} + \text{Total } d_{node}$$

The average latency is defined as

$$\text{Average latency} = \frac{\text{latencies of each packet received}}{\text{Total number of packets received by } \textit{Top Anchor}}$$

5.2 Energy

The (1) interconnect energy (E_{wire}) and (2) the node energy (E_{node}) are the main components of the total energy evaluated. In the MATLAB framework, each node keeps updating its E_{node} and E_{wire}. Note that E_{wire} is updated only when a packet is being sent from that node.

An energy model was created for the interconnect energy (E_{wire}) as described by Pande *et al.* [20] for 1-bit of information sent. A switching activity of 0.5 was assumed for the evaluations [13]. An operating voltage of 1V and a nanowire model as introduced by Snider *et al.* was implemented [23].

$$\begin{aligned} E_{wire} &= 0.5 \times \text{C per unit length} \times \text{Vdd}^2 \times \\ &\quad \text{Switching Activity} \times \text{Total wire length} \\ &= 0.5 \times 1.0354 \times 10^{-18} \times 1^2 \times 0.5 \times \\ &\quad \text{Total wire length } (\mu m) \times 10^3 \text{mJ} \\ &= 2.5885 \times 10^{-16} \times \\ &\quad \text{Total wire length } (\mu m) \ \ \text{mJ} \end{aligned}$$

We built VHDL models for a compute node by varying the number of ports and buffer channel length. Each node was successfully simulated with different packet modes and also for different if-else conditions in each packet mode. The clock rate was set to 500 MHz. The switching activity [20] and simulation times were recorded for each simulation using Mentor Graphics Modelsim. Each of the nodes was then synthesized for a 65nm CMOS node technology using the Synopsys Design Compiler. The Synopsys Power Compiler was then used in combination with the switching activity values to create a database of average power values for each packet mode and each if-else condition. These values were then exported to the MATLAB framework. We then created a database of energy values from the power value database and simulation time database. The energy of each node (E_{node}) was incremented by the corresponding energy value from the database. In this way, we were able to evaluate E_{node}. We implemented this energy model as there wasn't enough computing resources available for the Synopsys Design Compiler to synthesize a large network of nodes and evaluate the power consumption. We also assumed that idle power or energy was zero when there are no packets in the node to be processed. Figure 5.1 shows the energy consumed by a single node for different packet modes as a function of connectivity of that node. We see that *config Map Init* packet consumes the highest energy compared to all other packets. This is expected as there is a broadcast performed by the configured node to discover the sources of its input defined in the netlist. It can also be seen the energy increases linearly for all packet modes as a function of connectivity.

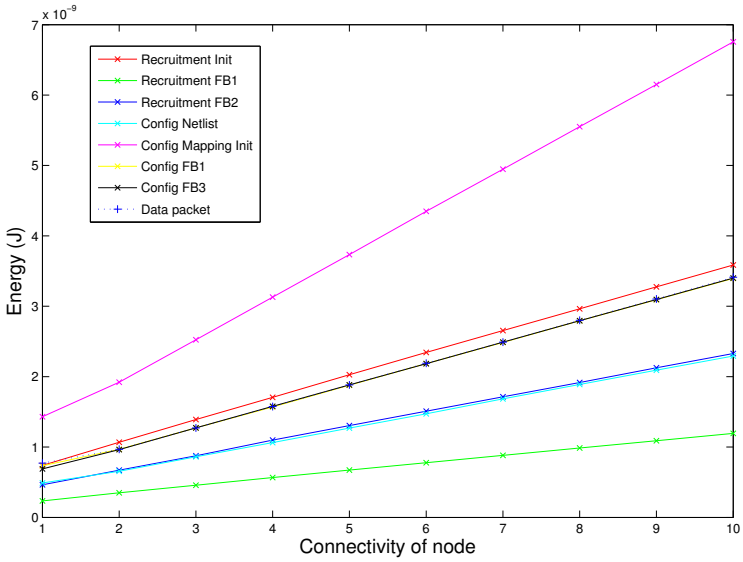


Figure 5.1: Energy consumption by a single node for different packet types as a function of connectivity (neighbors) of that node. We see that *config Map Init* packet consumes the highest energy compared to all other packets. This is expected as there is a broadcast performed by the configured node to discover the sources of its input defined in the netlist. It can also be seen the energy increases linearly for all packet modes as a function of connectivity.

The total energy for the network of nodes is

$$\text{Total energy} \quad = \quad \text{Sum of all } d_{wire} + \text{Sum of all } d_{node}$$

The average energy is defined as follows

$$\text{Average Energy} = \frac{\text{Total Energy}}{\text{Total number of packets received by } \textit{Top Anchor}}$$

5.3 Brute-force optimization of the mapping

To identify the extent to which the mapping could be optimized for latency with a global view of the configuration mapping, we apply the *Brute Force Optimization*

algorithm. In every iteration in this optimization algorithm, two nodes 'I' and 'J' are considered. The *oldHops* and *newHops* value are calculated in a manner similar as described in Section 4.6. The hops are also then compared in a similar manner and the netlist exchanged. The iteration loops are reset again to their initial values. The optimization ends only when there is no exchange of netlist in a complete set of nested loop iterations. The algorithm is summarized in Algorithm 11.

```
1: for I = 1 to MAX_NODES do
2:   for J = I + 1 to MAX_NODES do
3:     if Compute node 'I' is configured with mapping 'm1' or Compute node
       'J' is configured with mapping 'm2' then
4:       oldHops = Sum(Hops of source and destination of 'm1' from 'I') +
         Sum(Hops of source and destination of 'm2' from 'J').
5:       newHops = Sum(Hops of source and destination of 'm1' from 'J') +
         Sum(Hops of source and destination of 'm2' from 'I').
6:       if oldHops >= OPT_FACTOR * newHops then
7:         Exchange netlist. Initialize stage 2 of configuration
8:         Break For loops and reset values of I and J.
9:       end if
10:    end if
11:  end for
12: end for
```

Algorithm 11: Brute force optimization algorithm.

6

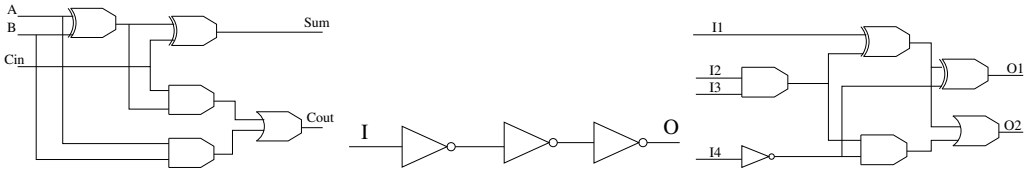
Experiments and Results

A series of experiments were conducted to evaluate the performance and the cost of the proposed architecture. We measured mainly 4 different parameters in our experiments. These included (1) the average latency, (2) the average energy, (3) the recruitment time, and (4) the configuration time. The average latency and average energy were measured for (1) non-optimized configuration, (2) self-optimized configuration, and (3) brute force optimized configuration. We varied different network parameters, including the degree distribution of node connections, the average connectivity and the network size. We mostly experimented with 3 benchmark digital circuits: (1) a multi-bit full adder, (2) a inverter chain, and (3) a random digital circuit with 4 inputs and 1 output. The three circuits are shown in Figure 6.1. We varied the size of the circuit for each of them. We compared our results with a regular mesh network of nodes.

We define certain terminology used in our experiment as follows:

- K - Connectivity of each node.
  - K_{max} - Maximum connectivity of a node in the network.
  - K_{min} - Minimum connectivity of a node in the network.
  - K_{avg} - Average connectivity of a node in the network.
- D - Length of interconnect between nodes.
  - D_{max} - Maximum interconnect length.
  - D_{min} - Minimum interconnect length.

– D_{avg} - Average interconnect length.



(a) 1 bit adder.

(b) 3-gate inverter chain.

(c) 6-gate feedforward Random Boolean network.

Figure 6.1: The three circuit types used for the experiments.

6.1 Experiment 1 - Comparison with 2D mesh

Results were obtained for a random network of N=100 nodes, a maximum connectivity $K_{max} = 4$, and a 2D mesh with uniform connectivity of 4. 100 data packets were sent and we measured the average latency over 10 runs by varying the number of logic gates of an inverter chain circuit to be configured. Each data packet was assumed to be sent serially over the nanowire interconnect. Also, the maximum interconnect length of a single wire is assumed to be 4 unit for irregular networks and 1 unit for 2D mesh networks. The mesh networks would be more compact and hence the lower value. We only used the brute force algorithm for the optimization.

Figure 6.2 shows that the 2D mesh logic network has a latency 80% lower for the same number of configured nodes than the irregular logic network. The reason is due to the uniform connectivity of 4 in 2D mesh compared to Kmax of 4 in irregular network. Also the Dmax is assumed to be 4 times smaller in case of 2D

mesh. We also compared the performance after optimizing the initial configuration of the compute nodes on the random network. Figure 6.3 shows the latency improvement after optimizing the initial configuration using the brute force algorithm. The mean improved latency after the optimization was close to 90% better than the initial latency for circuit sizes greater than 5 gates. The improvement is due to the reduction in the number of hops between logical nodes. This result shows that the initial configuration is not optimal and there is lot of potential to improve the latency. This made us explore further.

6.2 Experiment 2 - Recruitment and configuration time

In this experiment, we evaluate the recruitment and configuration time for different circuit sizes. Results were obtained for a random network of N=100 nodes with maximum connectivity K_{max}=4 and a 2D mesh of 100 nodes with uniform connectivity of 4. The maximum interconnect length of a single wire was assumed to be 4 units for irregular networks and 1 unit for 2D mesh networks.

Figure 6.4 shows the recruitment time as a function of number of gradient levels. A linear increase for both random and 2D mesh networks was seen. Figure 6.5 shows the configuration time as a function of number of gates configured. Configuration time also increases linearly with random and 2D mesh networks. These two results show that the recruitment and the self-configuration steps are scalable with network size and circuit size respectively though we might need to explore larger circuit sizes. Figure 6.6 shows the configuration time as a function of circuit size and of the maximum connectivity (K) for a random network. One can see that maximum connectivity didn't have much effect on the configuration time of

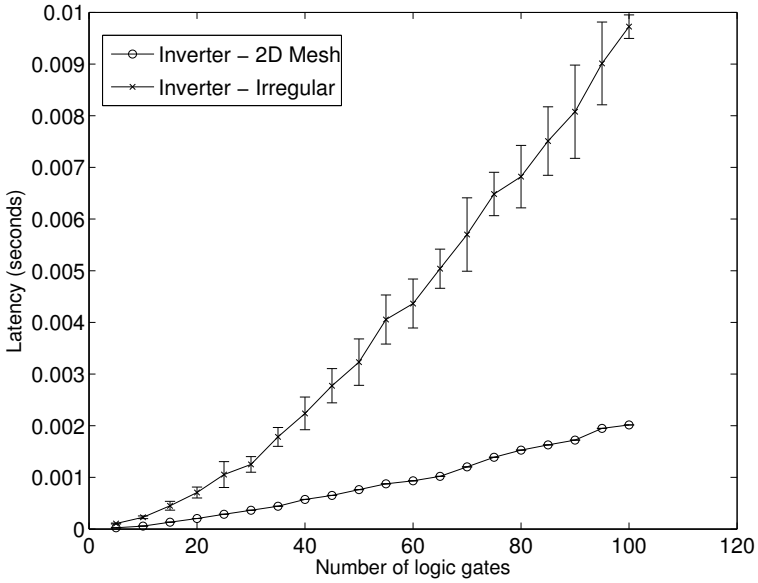


Figure 6.2: Latency comparison between irregular network and 2D Mesh network (non-optimized).

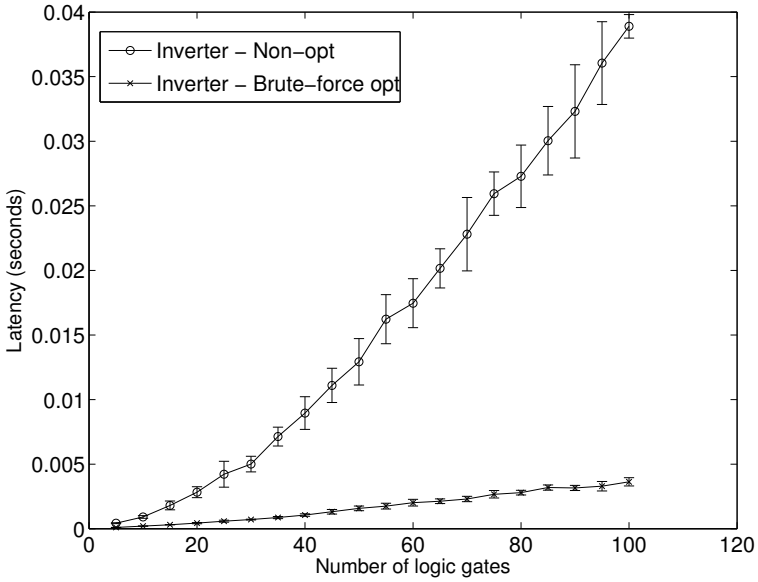


Figure 6.3: Latency comparison between non-optimized and optimized random network.

the circuit. Hence we can say that network topology doesn't have much impact on the configuration time which is an advantage for our unknown and irregular architecture.

6.3 Experiment 3 - Improvement after self-optimization vs circuit size

We implemented the self-optimization algorithm in the MATLAB framework as explained in Chapter 3. We ran experiments to determine the latency improvement due to this algorithm for different circuit sizes for the 3 circuits (1) inverter chain, (2) the multi-bit adder circuit, and (3) the random digital circuit with 4 inputs and 1 output. For all the experiments, we initialized an irregular network of 200 nodes with a maximum connectivity of K_{max}=4. Network size of 200 was chosen as we wanted atleast a 100 nodes to be recruited during the recruitment. Each data packet was assumed to be 100 bits long and sent serially over the nanowire interconnect. After running several experiments, we fixed the optimization factor of 0.9 for the self-optimization. We also fixed the number of optimization runs to 2000 after studying the convergence dynamics.

In Figure 6.7, we can see that the self-optimization algorithm, where each node is independent and has only a local view of the configuration, improved the latency by over 40% for circuit sizes larger than 35 gates for the inverter chain circuit. We could also see that the self-optimization algorithm improves the latency for circuit size larger than 35 gates by over 30% for the random digital circuit and by over 25% for the adder circuit. Figure 6.8 shows that the energy also improves while optimizing for the latency after self-optimization. This is due to the reduction in the number of communication hops. More than 40% less energy is consumed by

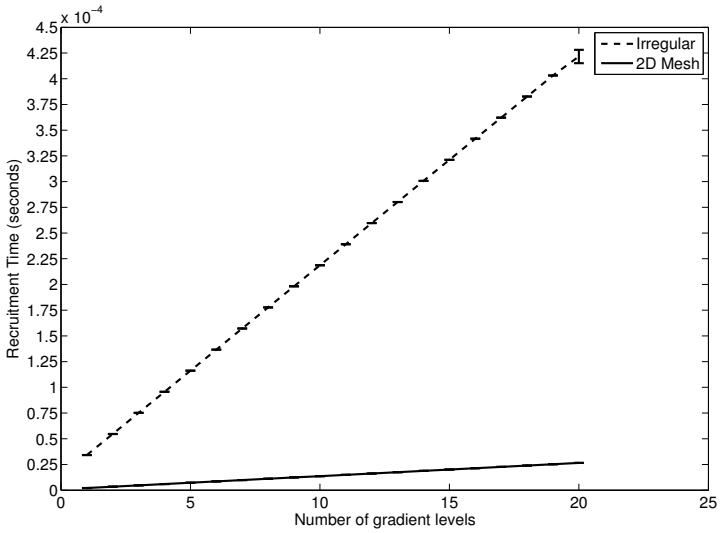


Figure 6.4: Recruitment time comparison between irregular network and 2D mesh network.

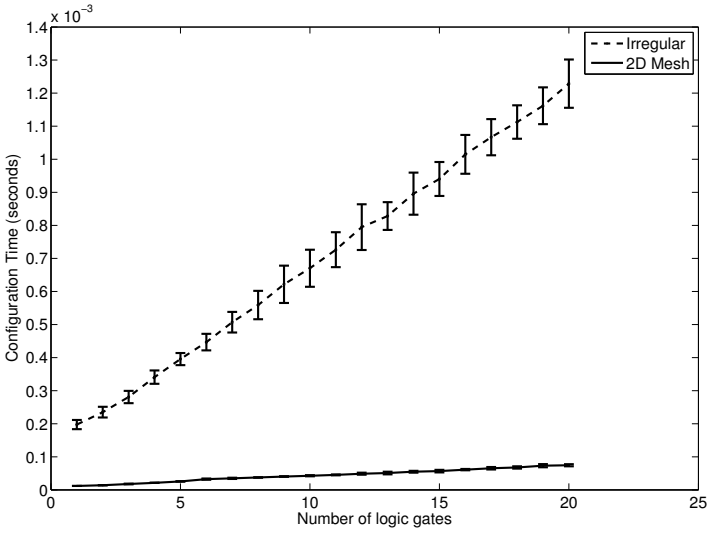


Figure 6.5: Configuration time comparison between irregular network and 2D mesh network.

the inverter chain greater than 35 gates. 30% improvement is seen for the multi bit adder and the random logic circuit. The increase in improvement of latency and energy with circuit size supports the implementation of the self-optimization algorithm.

6.4 Experiment 4 - Improvement after self-optimization vs average connectivity

We ran experiments to see the effect of network topology on the latency improvement. A random network of N=196 nodes was initialized with an average connectivity (K_{avg}) varying between 1.2 to 6.8 and with a maximum connectivity of K_{max}=8. All 3 benchmark circuits were simulated for a gate size of 50. All other parameters were the same as described in Section 6.3.

Figure 6.9 shows that relative latency improvement after self-optimization is invariably independent of the average connectivity. The improvement for the inverter chain is over 45%, for the adder circuit is over 25% and for the random logic circuit it is over 30%. Figure 6.10 shows the energy improvement after self-optimization. It is almost independent of the average connectivity. The energy improves by more than 45% for the inverter chain and more than 25% and 35% respectively for the multi-bit adder and the random logic network. The latency and energy being independent of K_{avg} means that we could implement the self-optimization without being concerned about the average connectivity or topology of the network. This is an advantage as our architecture has an unknown and irregular topology.

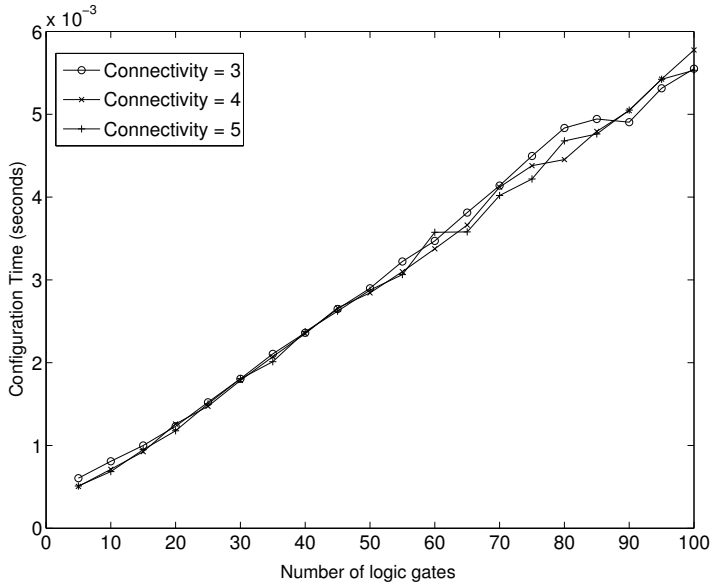


Figure 6.6: Configuration time as a function of K_{max} and the circuit size.

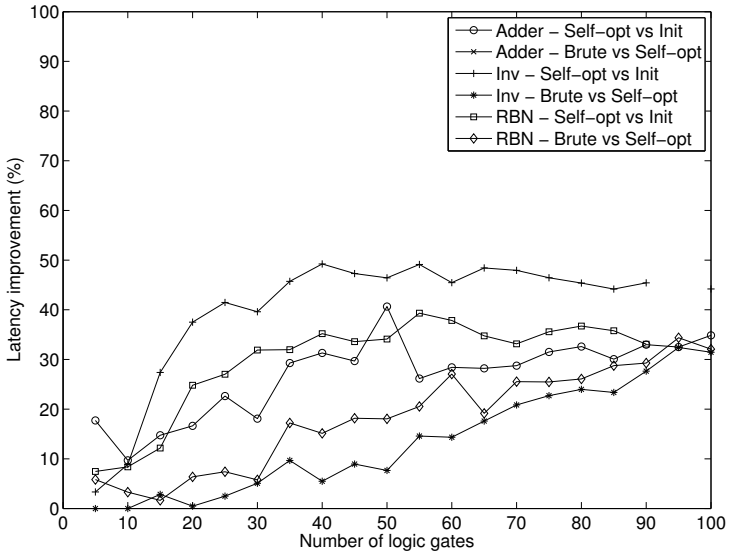


Figure 6.7: Comparison of latency improvement of self-optimization and brute force optimization algorithm over the initial latency for circuit size varying between 5 and 100.

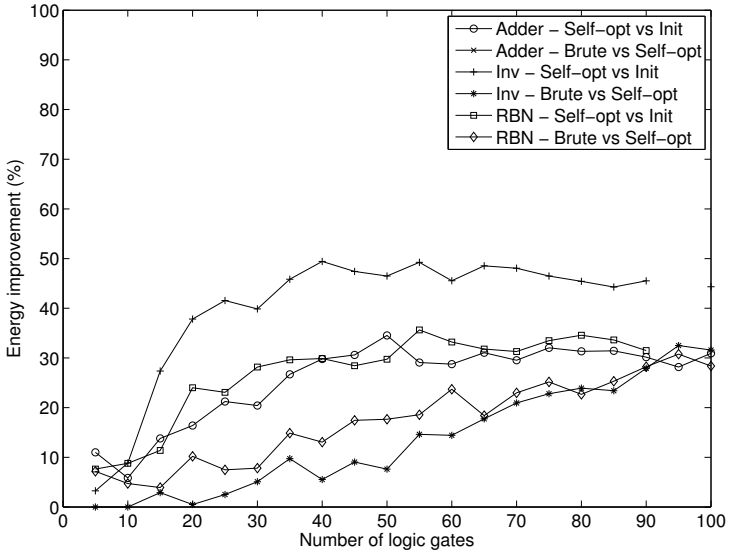


Figure 6.8: Comparison of energy improvement of self-optimization and brute force optimization algorithm over the initial energy for circuit size varying between 5 and 100.

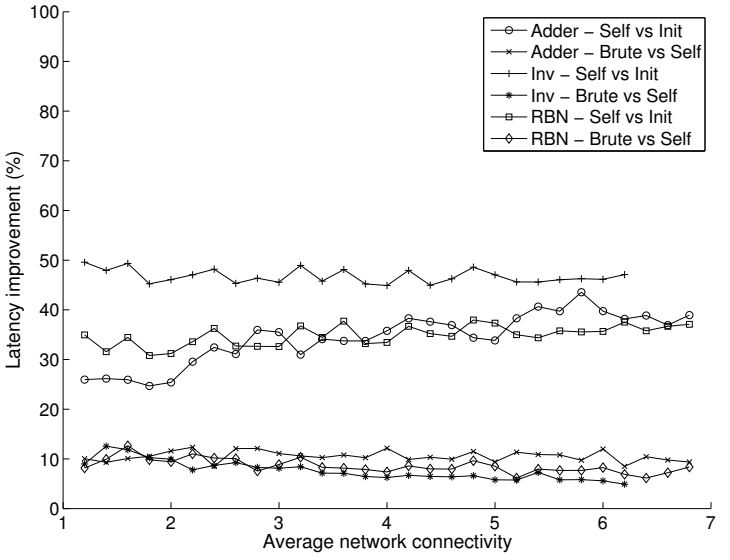


Figure 6.9: Comparison of latency improvement of self-optimization and brute force optimization algorithm over the initial latency for average connectivity varying between 1.2 to 6.8.

6.5 Experiment 5 - Area overhead for the self-optimization algorithm

Figure 6.11 shows the total area of each node for a channel length of 5 as a function of number of ports (connectivity). The area values were obtained from Synopsys Design Compiler and hence do not consider any kind of floor-planning. Figure 6.12 shows the relative area of the different modules in a node with a connectivity, K, equal to 4. The communication module consumes around 80% of the node area mainly due to the channel buffers. Optimizing and reducing the resources required for the communication module is beyond the scope of this thesis.

Figure 6.13 shows the area of the self-optimization module with respect to the total area of node. It can be seen that 4% is the maximum overhead cost and the cost decreases as the number of ports increase. This is quite low compared to the area consumed by the rest of the module. Hence, it is favorable to implement the self-optimization module with respect to the area.

6.6 Experiment 6 - Energy overhead for self-optimization algorithm

Figure 6.14 shows the ratio of the energy overhead for the self-optimization algorithm after 2000 optimization runs over the energy gained for processing one data packet for each of the circuits. It can be seen that for circuit sizes greater than 40 gates, the energy gained due to self-optimization after processing 12,000 data packets would equal the energy spent by the self-optimization algorithm. This means that if we know that the device would be used to configure a circuit and use it more than 12,000 times, then the self-optimization algorithm would be feasible to implement with respect to the energy consumption.

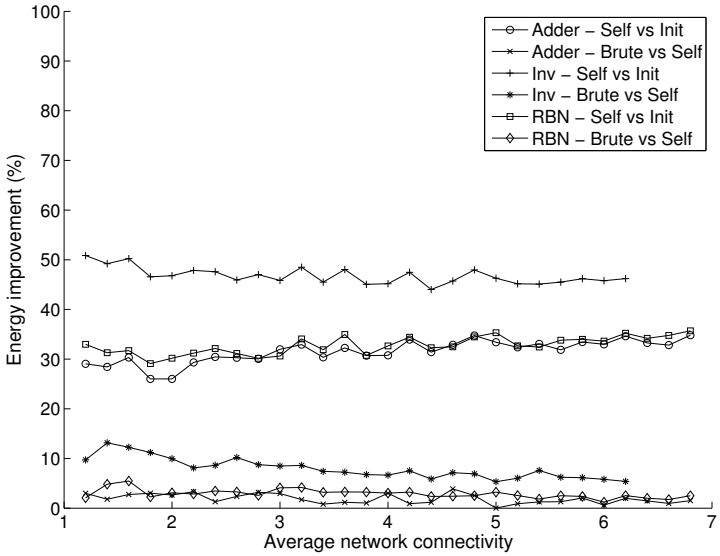


Figure 6.10: Comparison of energy improvement of self-optimization and brute force optimization algorithm over the initial energy for average connectivity varying between 1.2 to 6.8.

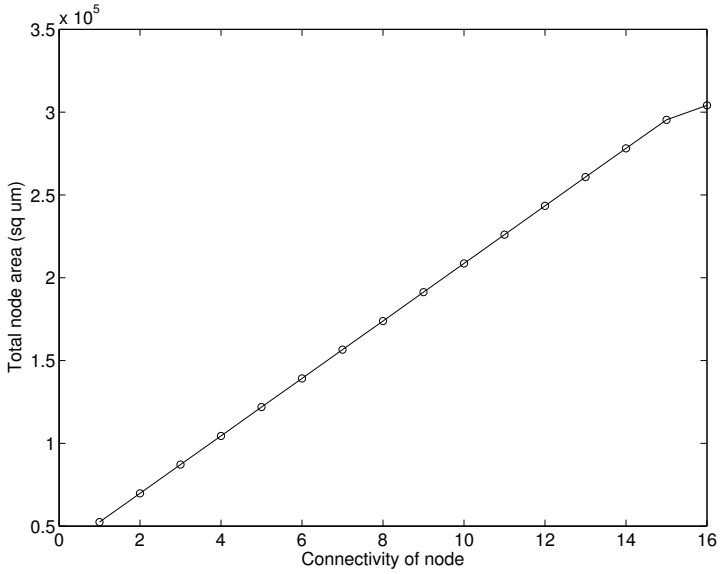


Figure 6.11: Total node area as a function of connectivity for a input and output channel buffer size = 5.

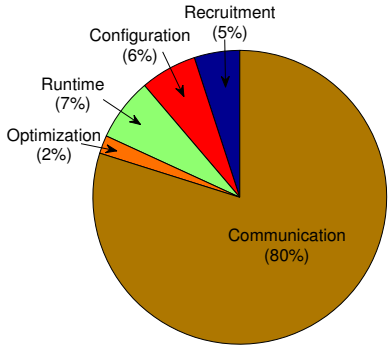


Figure 6.12: Relative area of the modules for a node with K = 4

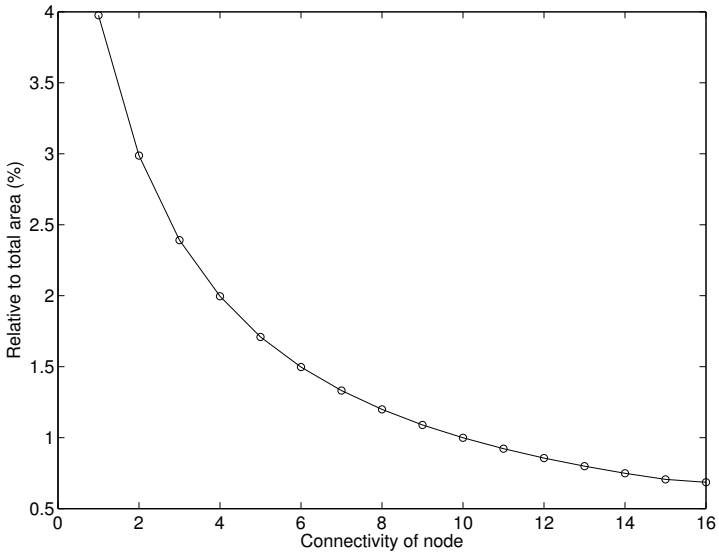


Figure 6.13: Area of optimization module in comparison to total area.

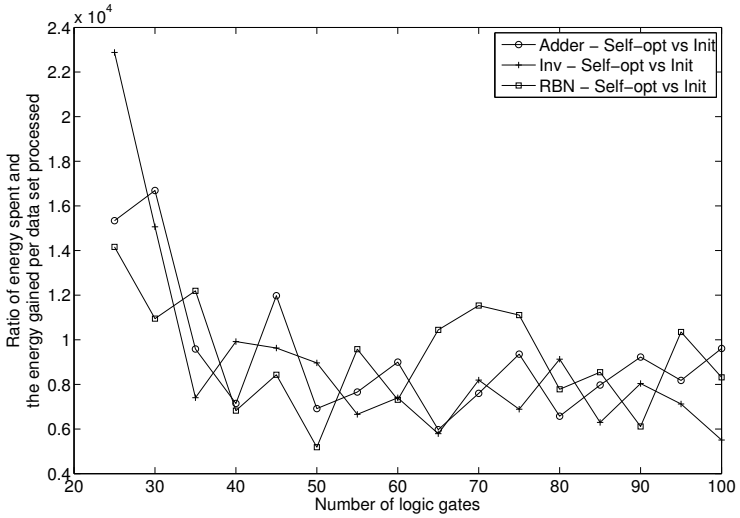


Figure 6.14: Ratio of energy overhead to energy gained for every data packet processed.

6.7 Experiment 7 - Architectural change improves latency

During the course of our experiments, we modified the design of the input module to optimize the latency. Instead of an exclusive buffer at the end of each of the input modules, we chose a common buffer which would feed packets into the decoder module. This avoids a multiplexer present in the decoder module to unnecessarily check buffers in ports where there are no packets to be processed. Though our MATLAB simulator is not exactly cycle accurate, we still see an improvement in the latency as seen in Figure 6.15. The latency improvement for a 10-bit adder circuit increases linearly with the average connectivity. This can be attributed to the fact that the multiplexer in the decoder will need to check more number of ports unnecessarily.

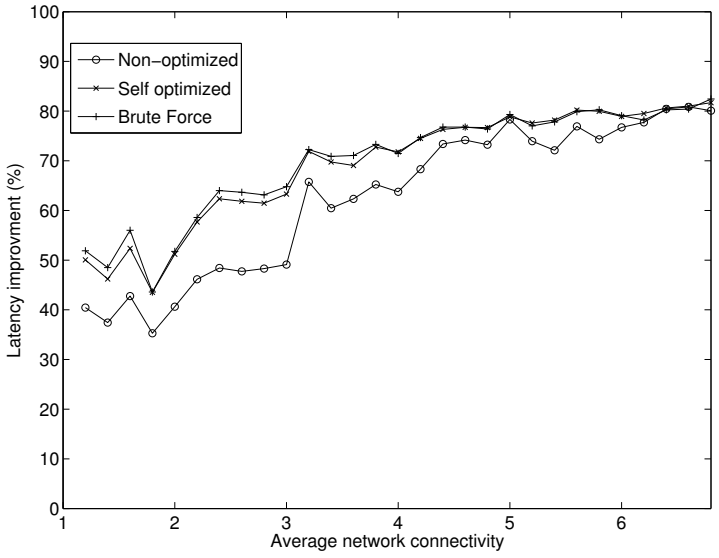


Figure 6.15: Latency improvement for a 10-bit adder circuit when a common input buffer is used over using exclusive buffers as a function of average connectivity.

7

Conclusion

In this thesis, we have presented a computing architecture which could be built by a bottom-up self-assembled fabrication technique. We have assumed a hybrid architecture where there are multiple homogeneous reconfigurable compute nodes fabricated using CMOS technology are connected through random nanowire interconnects grown in a bottom-up manner. We believe that this presents a simple yet important step toward adaptive and scalable computing architectures for non-classical devices, e.g., self-assembled molecular and nanoscale systems. Using a software simulator, we then demonstrated a technique through which a given digital circuit would get configured onto the network of reconfigurable nodes without any external circuitry and without a global view of the system by any node. We then evaluated crucial metrics including latency, energy and area consumption for 3 benchmark circuits with the help of a compute node synthesized using a 65nm CMOS technology and a nanowire model. We compared these results with a 2D mesh network.

To optimize for latency, we presented a topology agnostic self-optimization technique which reconfigures the initial network with every node having the configuration mapping of utmost its immediate neighbour nodes only. We evaluate the latency and energy improvement for all 3 benchmark circuits. We see a latency improvement up to 50% and an energy improvement up to 45% but, it required us to make a trade off with the latency, energy and area overhead to implement this self-optimization module. We observed an area overhead of 4%. The energy

overhead was compared with the amount of energy gained due to the optimization. It was seen that for circuit sizes greater than 30 for all 3 bencmark circuits, the energy gained to process data for the circuit would be lesser than the energy spent during self-optimization at around 12000 data processes.

Some of the limitations of our design are that the performance in terms of latency does not match current day processors nor does it match a 2D mesh. But the tradeoff is the ease of manufacturing and cost savings eventually.

Future work will focus on further optimizing the reconfigurable logic unit. We would like to make more comparisions between our architecture and present day computing architectures. We also would like to focus on increasing the reconfigurable hardware in each node and finding the right computation to communication ratio. The long term goal is to make such architectures even more adaptive and scalable.

References

[1] International technology roadmap for semiconductors (ITRS), update. *Semiconductor Industry Association*, 2008.

[2] H. Abelson, R. Weiss, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, R. Nagpal, E. Rauch, and G. J. Sussman. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[3] M. J. Alexander, J. P. Cohoon, J. L. Ganley, and G. Robins. Placement and routing for Performance-Oriented FPGA layout. *VLSI Design*, 7(1):97–110, 1998.

[4] A. Amarnath and C. Teuscher. A Self-Configurable computing architecture for unstructured and unknown reconfigurable fabrics. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, San Diego, California, 2011. In Press.

[5] D. Awtrey. Transmitting data and power over a One-Wire bus. *Advanstar*, February 1997.

[6] A. Bachtold, P. Hadley, T. Nakanishi, and C. Dekker. Logic circuits with carbon nanotube transistors. *Science*, 294(5545):1317, November 2001.

[7] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[8] G. Bilardi and F. P. Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27(2):172–182, June 1995.

[9] G. I. Bourianoff, J. E. Brewer, R. Cavin, J. A. Hutchby, and V. Zhirnov. Boolean logic and alternative information-processing devices,. *IEEE Computer*, 41(5):38–46, 2008.

[10] J. Chen and M. A. Reed. Large On-Off ratios and negative differential resistance in a molecular electronic device. *Science*, 286(5444):1550, November 1999.

[11] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[12] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. pages 684– 689, 2001.

[13] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE*, pages 253 –259, June 1992.

[14] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Munich, Germany, 2001. IEEE Press.

[15] M. Haselman and S. Hauck. The future of integrated circuits: A survey of nanoelectronics. *Proceedings of the IEEE*, 98(1):11–38, 2010.

[16] C. P. Husband, S. M. Husband, J. S. Daniels, and J. M. Tour. Logic and memory with nanocell circuits. *Electron Devices, IEEE Transactions on*, 50(9):1865 – 1875, 2003.

[17] J. A. Hutchby, R. Cavin, V. Zhirnov, J. E. Brewer, and G. I. Bourianoff. Emerging nanoscale memory and logic devices: A critical assessment. 41(5):28–32, 2008.

[18] Y. C. Lan, H. A. Lin, S. H. Lo, Y. H. Hu, and S. J. Chen. A bidirectional NoC (BiNoC) architecture with dynamic Self-Reconfigurable channel. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(3):427–440, 2011.

[19] P. Narayanan, T. Wang, and C. A. Moritz. Programmable cellular architectures at the nanoscale. *Nano Communication Networks*, 1(2):77–85, June 2010.

[20] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Computers, IEEE Transactions on*, 54(8):1025–1040, 2005.

[21] J. Patwardhan, C. Dwyer, and A. R. Lebeck. A self-organizing defect tolerant SIMD architecture. *ACM Journal on Emerging Technologies in Computing Systems*, 3(2):10–es, 2007.

[22] R. A. Rutenbar. Simulated annealing algorithms: an overview. *Circuits and Devices Magazine, IEEE*, 5(1):19–26, January 1989.

[23] G. S. Snider and R. S. Williams. Nano/CMOS architectures using a field-programmable nanowire interconnect. *Nanotechnology*, 18(3):035204, 2007.

[24] C. Teuscher. Nature-inspired interconnects for self-assembled large-scale network-on-chip designs. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 17(2):026106, 2007.

[25] C. Teuscher, N. Gulbahce, and T. Rohlf. An assessment of random dynamical network automata for nanoelectronics,. *International Journal of Nanotechnology and Molecular Computation*, 1(4):39–57, 2009.

[26] Christof Teuscher, Neha Parashar, Mrugesh Mote, Nolan Hergert, and Jonathan Aherne. Wire cost and communication analysis of self-assembled interconnect models for Networks-on-Chip. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*, NoCArc '09, page 8388, New York, NY, USA, 2009. ACM. ACM ID: 1645232.

[27] S. Trimberger. Effects of FPGA architecture on FPGA routing. In *Design Automation, 1995. DAC '95. 32nd Conference on*, pages 574–578, 1995.

[28] H. L. Wang, W. Li, E. Akhadov, and Q. Jia. Electrodeless deposition of metals and metal nanoparticle using conducting polymers. *Chemistry of Materials*, 19:520, 2007.

[29] Y. Wu, J. Xiang, C. Yang, W. Lu, and C. M. Lieber. Single-crystal metallic nanowires and metal/semiconductor nanowire heterostructures. *Nature*, 430:61–65, 2004.

[30] P. Xu, S. H. Jeon, H. T. Chen, H. L., G. Zou, Q. Jia, M. Anghel, C. Teuscher, D. J. Williams, B. Zhang, X. Han, and H. L. Wang. Facile synthesis and electrical properties of silver wires through chemical reduction by polyaniline. *The Journal of Physical Chemistry C*, 114(50):22147–22154, December 2010.

[31] W. Zhang, L. Hou, J. Wang, S. Geng, and W. Wu. Comparison research between XY and Odd-Even routing algorithm of a 2-Dimension 3X3 mesh

topology Network-on-Chip. In *Intelligent Systems, 2009. GCIS '09. WRI Global Congress on*, volume 3, pages 329–333, 2009.

[32] V. Zhirnov and D. J. C. Herr. New frontiers: Self-assembly in nanoelectronics. *IEEE Computer*, 34:34–43, January 2001.

[33] V. Zhirnov, J. A. Hutchby, G. I. Bourianoff, and J. E. Brewer. Emerging research logic devices. *IEEE Circuits & Devices Magazine*, 21:36–46, 2006.