Richard Curtis Harting
Undergraduate Thesis: Computation on Self-Organized Networks

Abstract

With the cost and limitations of scaling CMOS rising, researchers are looking into new computing substrates. One such substrate uses carbon nanotube transistors attached to a DNA self-assembled grid. These grids, containing a limited amount of logic, can be randomly linked together through one bit channels. This thesis looks at improving computation on this substrate. First, it explores the different programming trade-offs inherent to a computer architecture on the system. This is done through a case study of floating point arithmetic. As a result of the study, a new instruction was added to the architecture to augment the built-in control structures. In the end, floating point addition is implemented in a way that is competitive with modern computers, given enough nodes. Next, this thesis explores the issue of connectivity in random topologies. Because of the simplicity of the transceiver logic in each node, links that are made of more than one pair of nodes fused together must be logically cut such that only two nodes remain on a link. This process reduces the number of nodes connected to each other in the system. The effect of cutting mechanisms, node density, and defect tolerance were all studied, and a procedure was developed to provide a promising amount of connectivity.

1. Introduction

According to the semiconductor industry roadmap [1], the cost and complexity of building silicon devices is increasing every year. The primary reason for these rising manufacturing costs is the shrinking of the size of transistors in each technology generation. Smaller transistors, with gate lengths now at 45nm, mean that more precision must be used in the manufacturing process. Another downside to smaller transistors is a higher defect rate. In a 500nm process, an error of 10nm is negligible, where as in a 45nm process it could cause the malfunction of a transistor. With rising defect rates and higher chip densities, the yield of chips is also falling, furthering increasing manufacturing costs. Eventually, this scaling will lead to a fundamental limit on the size of transistors.

The reason that manufacturers keep pushing into more expensive technologies is simple: performance. Smaller transistors means higher logic density, which, when used effectively, translates into higher performance. With the self assembled systems, not only would the manufacturing costs potentially be cheaper, but also the massive parallelism that occurs in manufacturing could offer gains in performance. Other than the cost of raw material, the money needed to make one small processing node at a time is roughly the equivalent to the amount needed to make billions. If connected and designed effectively, these nodes could provide a huge computational increase relative to silicon.

For these reasons, researchers have begun to explore alternative substrates and methods of computation. Typically, the research that occurs can be categorized into the following three areas:

- **Devices –** Examples include MOSFETs, Carbon nanotube transistors (CNFETs)[3], and nanowire arrays[4]
- **System of Integration –** Examples include self assembly[5] and photolithography

- **Model of Computation –** Examples include Boolean logic and quantum computation[6]

Though many of these models share similarities, this thesis focuses on experiments that were ran for one node in the space. The devices used are CNFETs, which are attached to an addressable, self-assembled DNA lattice. This system will use a standard Boolean logic for computation.

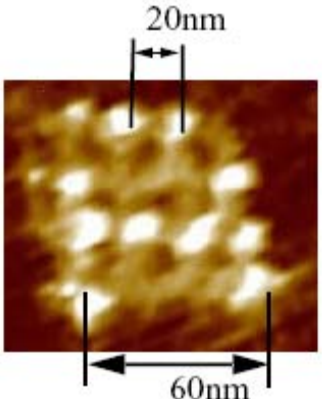

Figure 1 - DNA Assembled Structure [25]

1.1 The Physical Model

The underlying physical system uses a DNA lattice grid in order to selectively place addressable components. In the laboratory, this has been demonstrated by making a grid that been addressed using proteins to recreate letters of the alphabet, shown in figure 1. It is estimated that these grids can be grown to be about 1μm on each side, providing enough addressability for up to 5000 FETs. The advantage of such a system is the parallel assembly with which these grids can be built. Instead of stepping through an etcher one die at a time, *billions* of these grids can be assembled at once. It should be noted here that the yield of this chemical process is not ideal and defects will arise in the self-assembly process.

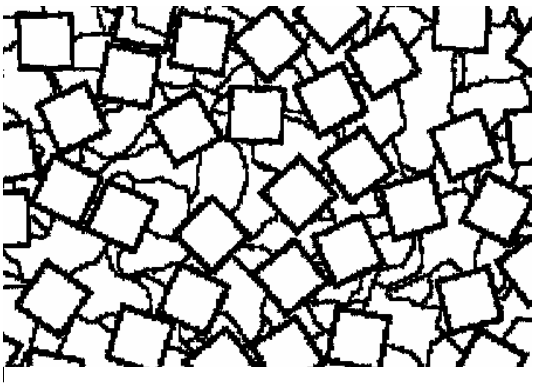

Figure 2 - A sample Topology

Once these nodes have been made, they need to be deposited to a surface and interconnected. The interconnection model assumes that each node grows out wires from each of its transceivers. These wires are grown in a random walk until they run into another wire or node. Figure two shows a topology generated by our in-house simulator.

When using this mechanism of growth, randomness is pervasive at all levels. The nodes may have any number of random defects when they are deposited onto the substrate surface. No control mechanisms are used for the exact placement and orientation of these nodes. The amount of separation of nodes, for example, can only be controlled probabilistically by changing the concentration of grids to be deposited. When the wires are grown, they (theoretically) could connect any two nodes, due to the random turning that is inherent to the process. These wires may also fuse together, joining together more than just two nodes. A downside of these fused links is that if the transceivers only have enough logic to communicate on a point to point basis, nodes must be logically removed (shutting down their transceivers) from the fused link. This haves a very significant impact on net connectivity, as will be seen in section 6.

1.2 Implications of Physical Model

The physical model of the computer that is to be built has a profound impact on the architecture running on top of it. With the DNA self-assembled system, many aspects

need to be considered when designing a computer architecture. The list below is by no means complete, but it provides a listing of many of the factors that influence the computation model.

- **High Defect Rates:** The nodes that are placed have a probability of being defective. Any computing mechanism that will work needs to be able to map out or route around these randomly placed defects. Since there are so many nodes in a system, throwing out an entire systems for a single node defect, such as is sometimes done in silicon, is not feasible.
- **Random Interconnections:** The system cannot assume anything about which nodes are connected to others. Each node may be connected to any number of nodes, depending on the amount of link fusion that occurs.
- **Limited Interconnections:** The connections that are between nodes are only a single wire. This means that the communication of a single bit requires a handshake between transceivers, limiting the amount of bandwidth between nodes. This typically will make communication a bottleneck in these systems.
- **Limited communication to the micro scale:** Wires cannot be placed precisely for nodes to communicate with the outside world. In Duke's implementation, a wire is randomly placed in the system and is responsible for inputting instructions into the system. The nodes must be designed to tolerate this limited communication and random wire (via) placement.
- **Limited Node Resources:** Each node does not have enough resources on it to do a significant amount of computation. In an architecture studied at Duke, SOSA (explained below), each node only has enough logic on it to support one bit computation. The limited amount of transistors also limits the amount of routing that each node can support. A limited routing capability is at odds with a random network, where no network assumptions can be made.
- **Large Amounts of Nodes:** This where the greatest potential lay in nano-scale systems. The sheer number of nodes that could be easily placed in a system could perform massive amounts of computation. Any architecture mapped onto a nano-system must be able to utilize this resource successfully.

1.3 The Computation Model

In order to further study computation models, two architectures were developed at Duke University: the "Nano-Scale Active Network Architecture" (NANA) [7] and the "Self-Organized SIMD Architecture" (SOSA) [8]. Since this thesis deals with SOSA, I will only present a brief explanation of the lessons learned from NANA.

NANA consisted of two different types of nodes randomly placed into a network, containing either compute or memory logic. When the system powered on, it went through a configuration phase of setting up routing gradients and establishing a internal memory system. When executing, instructions packets drift along the different routing gradients looking for resources to execute upon. In the system, only one instruction packet is in flight at a time. Though these packets can contain more than one instruction, the utilization of nodes over time was minimal. This was due to both the execution model of instructions finding resources and the use of heterogeneous nodes.

SOSA looked to remedy this low utilization in two ways. First, all nodes are homogenous. They contain 4 transceivers for communication to neighbors, a 1-bit arithmetic logic unit (ALU), 32 one bit registers, test logic, and control logic. By

providing both a limited amount of computation and memory on a node, it means that operations can be done locally on each node. Once a node receives an instruction broadcast, it has all the resources that it needs to execute it. This eliminates the wasted time and resources of instructions searching for computation nodes. A more thorough description of SOSA is given in the next section.

2. SOSA – Introduction

The system architecture that is used in this thesis is the "Self-Organized SIMD Architecture," or SOSA, developed at Duke University. The system is logically assembled using a reverse path forwarding algorithm [11]. An external controller sends a packet into the system through a wire (the via) and into a node located near it (the anchor). This anchor node will then initiate a breadth first search to assemble a logical tree of all nodes in the system. Once this broadcast has completed, the controller initiates a depth first search (DFS). The DFS causes the nodes to configure into logical processing elements (PEs).

Each PE consists of a head node, a middle series of nodes, and a tail node. Each node in the PE is physically homogenous, but logically has different functionality once configured. The head node acts as the synchronization point for PE's executing instructions. Regardless of when a node receives an instruction, it will not execute it until the head node has sent the proper signal. The middle nodes act as a bit sliced register file and ALU. The bits are arranged from the least significant bit near the head to the most significant bit at the tail. The tail node is responsible for sinking signals and propagating needed values back to the head node.

SOSA is a single instruction, multiple data (SIMD) type architecture. Architecturally, that means that each instruction inserted into the system is executed on all PEs. Instructions are sent into the system through the via. When the node closest to the via receives the instruction, it stores the instruction in its instruction buffer and begins to broadcast the packet throughout the system. Every node that receives the instruction packet does the same: buffer and broadcast. The head node, once finished with the buffer and broadcast, will send a "go" signal to the next node in its PE (through the DFS gradient). Every node that receives both a go signal and has an instruction in its buffer will execute the instruction and propagate the go. Any information that needs to travel from one node to the next, such as carry bits, will propagate as well.

If all processing elements are executing every instruction, the question of data dependant control arises. For example, consider a program where R1 and R2 each contain a value. The goal of the program is to place the larger of the two numbers in R1 and the smaller of the two in R2. In a conventional processor this is done using an if statement in high level code, or branch instructions in assembly. These mechanisms provide a means for jumping to different parts of a program and not issuing specific instructions. In a SIMD system, however, this does not work since instructions can not be sent selectively to PE's. Instead, SOSA uses predicated instructions. Each PE can set a predicate bit, found in the head node, based on a conditional instruction. Next, an instruction that is marked as predicated is broadcasted into the system. For each PE receiving the instruction, if the predicate bit is set, the PE executes the instruction. If the bit is not set, the PE does not execute the instruction. The major drawback to predicated instructions is that no means of skipping instruction clauses exist. There is no concept of

the common case in SOSA programming since all instructions for all possible scenarios must be broadcast.

2.1 Memory

SOSA does not implement a memory system. The only form of internal storage comes in the form of the 32 (or 16, below) registers found inside of each PE. This provides three separate difficulties. First, it limits the amount of data in the working set that can be placed in the system. In none of the programs that were implemented for SOSA was this an issue, but it still is an upper bound.

Second, in most parallel systems, there is a notion of shared memory or message passing. SOSA has neither. No shared memory exists within the system due to a lack of centralized control and memory system. Messages that are passed cannot be addressed to specific PE's, but can only be sent to the adjacent PE via a PESHIFT operation. This means that any data that needs to be shared between nodes must either be broadcast in from above (by a series of shifts and increments) or passed around all the PE's via the PESHIFT operation.

Finally, there is no form of instruction memory in the system. As was discussed earlier, the only form of program control is through predicated instructions. This means that each PE can't have its own instruction cache to execute and skip around instructions quickly. It also means that all instructions that a node executes must be broadcast from the anchor node, a potential bottleneck for the system.

2.2 Communication

Name	**Old Code**	**New Code**	**Explanation**
Repeat Counter	Add R1 R2 R3 Add R1 R2 R3 Add R1 R2 R3	Add R1 R2 R3 3	Three instruction broadcasts turn into one instruction and a counter
Register Modifiers	Add R1 R2 R3 Add R2 R3 R2 Add R3 R4 R1	Add R1+ R2+ R3- 3	Each repeat of the instruction modifies the registers in the subsequent repeats
No Registers	Add R1 R2 R3 Shift R1 Add R1 R2 R3 Shift R1	Add R1 R2 R3 Shift Add Shift	As long as the registers (and their modifiers) do not change, they can be reused by instructions

Table 1 - Ways of Reducing Instruction Bits Broadcast

Almost exclusively, communication is the bottleneck in SOSA programs. This communication latency comes in two forms: instruction broadcast and data passing. Data passing occurs as an instruction is executing. Carry bits must be propagated during add instructions, results must be relayed to the head in comparison operations, and the predicate bit is sent down through the nodes in predicated operations. The implementation of SOSA does allow for the designer to make decisions to reduce this latency. In one configuration of the system, the PE has 34 nodes, 32 of which contain 1 bit of the 32 registers. Instead, each of the middle nodes can have 2 bits of the register file, making the PE only 18 nodes in length, lessening the amount of inter-node communication. The cost of this, however, is that this configuration only allows for 16

registers, instead of 32. For the undertaken studies, unless otherwise stated, this is the configuration used.

The other form of communication bottleneck is the amount of time that it takes a system to broadcast instructions into each of the nodes. Every instruction that the nodes execute must be broadcast through the system. Table 1 shows several examples of mechanisms that SOSA implements that help to reduce the number of bits broadcast.

2.3 Instruction Set

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

Table 2 - SOSA Instruction Set [8]

Table 2 shows the SOSA instruction set. With only a few differences, the ISA is similar to many other instruction sets, though more limited. Gone are all memory based and branch operations. New instructions are those that set and manipulate predicate bits, predicated instructions, and PE shifts.

2.4 Goal for programming

The motivating guideline when writing assembly code for SOSA is to limit the amount of communication, especially the bits broadcast. This can be done by using the mechanisms listed above, such as the repeat counters. The number of instructions broadcast into a system can also be modified by algorithmic changes to the program. Understanding how to over come the communication bottleneck is essential in order to effectively implement a floating point library.

3. Floating Point Study

3.1 Motivation

The motivation for studying floating point is twofold. First, it was the first application that was neither embarrassingly parallel nor tuned for a massively parallel system studied for SOSA. Floating point contains many different types of exceptions and cases that need to be handled. It also contains a non-trivial control flow in normalizing exponents and setting exponents equal to each other in addition and subtraction. For these reasons, floating point may be considered representative of a more general class of programs. Finding ways of improving floating point algorithms may lead new paradigms and ways of improving other algorithms.

Floating point operations are also a key component of scientific code, which can often be very parallel. If floating point were to be demonstrated to take an amount of time competitive with a normal processor, then scientific computing would become a promising application domain for SOSA.

3.2 Experimental Setup

This section of the thesis used a custom simulator for SOSA. It models timing to the granularity of handshakes over the interconnect.

3.3 Introduction To Floating Point

Floating point is the representation of numbers in a computer in what is essentially scientific notation. It is used to store large numbers, small numbers, and fractional amounts. Representations of floating numbers usually consist of a sign bit, some amount of exponent bits, and a mantissa. The number stored is equal to the mantissa multiplied by a base raised to the exponent.

The IEEE 754 standard is the predominant set of rules for floating point representation [12]. It dictates that the base of floating point numbers be 2, and also lays out rules for representation and rounding of numbers. In implementing floating point for SOSA, the goal was to stay in the spirit of IEEE single point standards, in the sense that there is a sign bit, 8 bit exponent (with the same bias), and 24 bit mantissa, totaling 33 bits. The IEEE standard has a total of 32 bits since the leading (most significant) bit of the mantissa is implicitly 1. For SOSA, it is made into an explicit 1. In IEEE floating point, in order to be considered normalized, the leading bit of the mantissa must be 1.

Many of the IEEE's standards, however, were selectively dropped in the process of implementing the floating point library. First, there is no explicit representation of 0 as there is in IEEE (a 0 exponent and 0 mantissa), since there is no implicit 1. All numbers with a mantissa of 0 will behave correctly as 0's when doing math. The library, however, does not support infinity and NaN (not a number). The reason for this is because each of these relatively unlikely cases requires their own special rules for doing math. This means that in a SIMD machine, the code that checks for these cases and executes based on a predicate bit must be broadcast at every operation. It is possible to write this code if it is of high importance, but it was not written for this project.

Finally, perhaps the most significant deviation from the IEEE floating point (other than the high radix, below) is the removal of all rounding. IEEE uses three extra bits called the guard, round, and sticky bits in order to minimize the amount of rounding error that occurs on normalization. Our current implementation does not have these bits and simply truncates on normalization. This means that over many operations the floating point numbers will monotonically decrease with respect to the actual expected values. It also means that the margin of error will be larger than a system that rounds since the amount of data being lost (that which is stored in the guard and round bits) is higher than in the IEEE implementation. The reason for the elimination of these bits is to both reduce runtime and the length of a given PE. Again, if the loss of runtime is less important than the loss of precision, it is possible to implement these three bits.

3.4 Original Implementation

The process of adding (or subtracting) floating point number typically occurs in three sections. The exponents must be set equal, the correct operation must occur, and sum must be renormalized. The renormalization can occur if the mantissa is too large (>24 bits) or if the mantissa is too small (<24 bits). Each of these sections will be treated separately.

3.4.1 Exponent Equality

The code in figure 3 is the basic way of doing exponential equality. Essentially, a check is preformed that sees if the exponents are equal, and if they are not, shifts the

mantissa of the number with the lowest exponent from MSB to LSB and increases the corresponding exponent by 1. Since this is a SIMD architecture, the program must perform this check 24 times (the number of bits in the mantissa). This is because the central controller broadcasting the instructions does have any way of telling if the exponents are actually equal and breaking the loop. After 24 shifts, if the exponents are still not equal, the mantissa of the smaller number will be 0 and it will be safe to do the operation. At this point in the code, it is guaranteed that the number with the smaller absolute value has its mantissa in R2 and its exponent in R5. R14 contains a mask that adds 1 to the exponent.

```
*Repeat 6 24 #External Control insn:
             #Broadcast the next 6 insns 24 times
SETLT R5 R4 R5      #If exp2 < exp1
CPPRED R5 R2 R2     #copy predicate bit
PRSHIFTML R2 R2 R2  #shift mant2
PRADD R5 R14 R5     #Add 1 to exp2
PRESET R2           #Clear the predicate bits
PRESET R5
```

Figure 3 - Radix 1 Exponent Equality

3.4.2 The operation

If the operation is an addition if the sign bits of both addends are the same. If they are different, then a subtraction is preformed. The sign of the result is equal to the sign of the addend that has the largest absolute value.

3.4.3 Normalization

Normalization can occur either because the mantissa has 25 bits or if it has 23 or less. While the sum can be at most 25 bits, there is no lower bound on the number bits in the difference between two numbers. With both operations, the process is a check to see if the mantissa is greater (less than) 24 bits and, if it is, then shifting from MSB to LSB (LSB to MSB) and adding (subtracting) 1 from the exponent. The code for this loop can be found in Figure 4. R3 is the sum mantissa, R4 is its exponent, R14 is a mask with a single '1' at bit 24.

```
#Renormalize in case mantissa is > 24 bits
SETLT R3 R14 R3
PINV R3 R3 R3       #If mant >= 25bits
PRSHIFTML R3 R3 R3  #Shift and Add
CPPRED R3 R4
PRADD R4 R14 R4
PRESET R3+ 2

#Renormalize in case the mantissa < 24 bits
SHIFTML R14 R14 R14 #mask setup
SHIFTML R4 R4 R4
*REPEAT 4 26
SETLT R3= R14=  R3+ 2  #if R3 is <24 bits
PRSUB R4 R14 R4        #Subtract 1 from exp
PRSHIFTLM R3 R3 R3     #Shift mantissa 1
PRESET R3+ 2
```

Figure 4 - Normalization

3.5 High Radix

In figures 3 and 4, each loop must be ran 24 times, shifting one bit at a time. Since the program is communication bound, the number of instructions broadcast from the external controller has a direct impact on the runtime. Instead of shifting one bit at time 24 times, for example, it would be quicker to shift 2 bits at a time, thirteen times. In these shifts, two must be added to the exponent instead of one. The name for this type of implementation is called base or radix 2^2. If the shifting occurred 4 at a time, it would be base 2^4 an so forth. Figure 5 shows the code for exponent equality and normalization, respectively, for radix 4. The registers contain the same information (with the masks slightly shifted) as in the previous example.

```
*Repeat 6 13
SETLT R5 R4 R5
CPPRED R5 R2 R2
PRSHIFTML R2 R2 R2 2
PRADD R5 R14 R5
PRESET R2
PRESET R5
```

Figure 5 - Exponential Equality: Base 4

In order to maintain 24 bits of precision, the size of mantissa in a base 4 implementation must be 25 bits. This is because, in the worst case, a normalized number can have a leading '01' instead of a leading '1-'. Also, since the only number added to the exponent is 2, it is guaranteed that if exponent starts even, it will never become an odd number. This makes the LSB (a 0) redundant and it can be made implicit. The same holds true for numbers with higher radices. Table 3 shows the number of bits needed to implement each variant of floating point while maintaining the same 24 bit worst case precision and range of exponents (using implicit zeros).

Base	Sign Bits	Exponent Bits	Mantissa Bits	Total	Nodes per PE
2	1	8	24	33	19
2^2	1	7	25	33	19
2^4	1	6	27	34	19
2^8	1	5	31	37	21
2^16	1	4	39	44	24

Table 3 - Register Size for Various Bases

Although there is an increase in the number of nodes in a PE, there is still a significant decrease in the run time needed to do addition. Table 4 shows the decrease in the number of combined instructions broadcast in the normalization and equality loops. The final column is the number of flops for each base given a fixed number of nodes, 1 million, with a nanosecond time unit.

Base	Loop Iterations	Loop instructions Broadcast	Flops per 1M nodes
2	24	240	3.64E+08
2^2	13	130	6.35E+08
2^4	7	70	9.52E+08
2^8	4	40	1.21E+09
Integer	N/A	N/A	9.50137E+11

Table 4 - High Radix Performance Information

There are two important notes here about high radix floating point. First, it is not IEEE approved. IEEE 754/854 standards only allow for the use of base 2 and 10. The reason for this is that for a given number of bits the highest precision can be obtained with a base 2. The base 10 is allowed for financial matters. Also, high radix and the speed vs. precision argument has been around since the dawn of floating point. IBM mainframes, for example, used and still support a hexadecimal (2^4) version of floating point. [13] The observation has always been that you sacrifice precision for speed when doing higher radix floating point operations. [14]

3.6 Gather Go

Typically, the exponential equality portion of floating point addition is described as "shifting one mantissa by the difference in the two exponents." In order for this to happen in the SOSA system, however, there needs to be a mechanism to feed data back into the control logic. This need spawned the GATHER_GO instruction. It takes one register (R) and a repeat counter (C) and feeds the C least significant bits of R into the head of each PE. Next, the PE executes the subsequent instruction the number of times that was just brought into the head.

The PE needs a minimal amount of additional hardware to implement this instruction. The least significant bits are sent into the head of the PE though the built in shifting mechanism. The next instruction sent out is predicated and has a maximum repeat counter, all previously included in SOSA. During this predicated execution, however, the head node sends the predicate bit based on the new counter being greater than zero. This control logic, five bit counter, and decrementer are the only new logic. Ideally, in the head node five of the predicate bits could be used for this function, along with the included ALU. This makes the only hardware overhead the control logic.

Although the Gather Go is useful in this context, it is not immediately apparent where else it can be used. It is only useful when one instruction is repeated a specified, known number of times. It does not work for normalization, for example, since there is no quick way to find the number of leading 0's before the first 1. It does work, however, for operations such as multiplication as will be shown later in the paper.

Table 5 shows the code that is needed to implement exponential equality using gather go, taking eight instructions. Figure 6 shows the normalized runtime of different forms of the implementation of floating point addition. The largest gains with gather go occur at lower bases, which have longer loops.

Code	Explanation
`SHIFTML R4 R4 R4 24` `SHIFTML R5 R5 R5 24`	Shifting the exponents into the LSBs of their respective registers

`SUB R4 R5 R3`
`SETGT R3 R12 R12`
`PRCPREG R12 R3`
	Find the difference between the two exponents. If the difference is greater than 24 (R12), then put 24 as the value of the difference.

`GATHER_GO R3 5` `PRSHIFTML R2 24`	The Gather and Go instruction takes the 5 LSB's of R3 and puts them into a special register in the head node. The next instruction (which must be predicated) then executes the number of times found in the loop counter.
`SHIFTLM R4 R4 R4 24`	Putting the exponent in the correct place

Table 5 - Gather and Go Exponent Equality

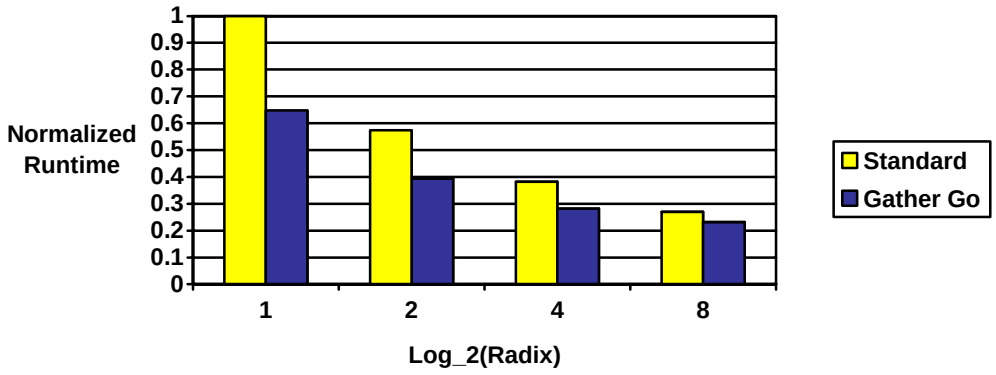

Figure 6 - Runtimes of Standard and Gather Go Addition

3.7 Binary Normalization

The final optimization that was made to floating point is an algorithmic change to the normalization loop. Instead of executing the loop using shifts of equal lengths, the loop is executed using shifts of 16, 8, 4, 2, and 1. For example, if a mantissa needed to be shifted by 13 bits, it would not execute the 16 shift (based on predication), execute the 8, 4, not the 2, and finally execute the last single shift.

Although significantly reduced, there is still a performance gain for going to a higher radix. This is because the last iteration of the loop can be removed as one moves from 2 to 2^2 to 2^4 etc. For example, in radix 2^4, the first one can be in any one of three positions and thus the checks for 2 and 1 do not need to be preformed. The results can be found in the new normalized graph in figure 7. The numbers in figure 7 for binary normalization use the gather and go methodology of exponent equality.

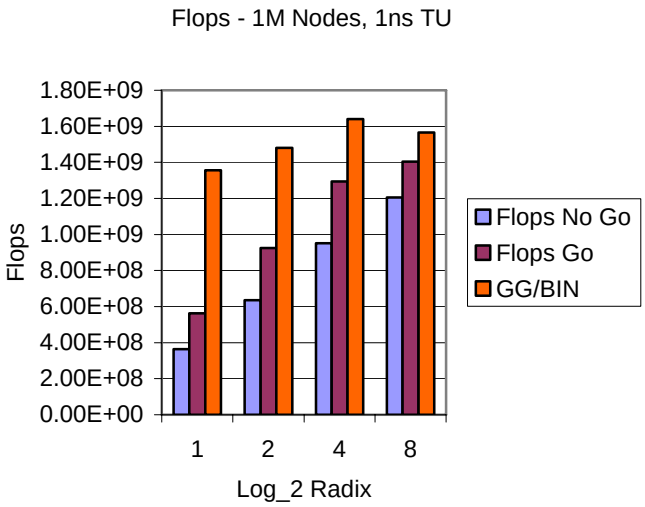

Figure 7 - Flops for Binary Normalization

3.8 Discussion

This analysis looked at many aspects of the SOSA system. At the surface it showed the potential for one million nodes to execute floating point code at about 1.3GFlops, using a base of two. The hexadecimal version of floating point allows for over 1.6GFlops. While these numbers are lower than the 200Gflops that some graphics cards offer [15], SOSA has a much higher scaling potential. Estimates for the growth of nodes in a system expect that between 10^9 and 10^{12} nodes could be grown at once [8]. Scaling linearly (fairly optimistic), approximately $2.8*10^{11}$ nodes are needed to equal the performance, in flops (360T), of IBM's BlueGene/L [16]. These numbers provide optimism that the domain of scientific computing is relevant to SOSA. A limiting factor in this domain, however, may be the amount of communication needed to pass data between PE's.

High radix floating point operations demonstrated both the advantage of the repeat counters and the disadvantage of broadcasting many instructions into the SOSA system. Unfortunately, these types of optimizations may need hand tuning in order to receive the best performance. A compiler, for example, may handle loop unrolling but a change such as binary normalization may be too complex. This is another point to be made about SOSA: writing efficient programs is tedious. A semester's worth of research went into improving the performance of floating point addition by a factor of four. The implementation time for larger, more complex programs may be higher.

SOSA also helps to study control structures that are inherent to SIMD systems. Predication offers some form of control, yet it is not efficient. The largest problem is the lack of capability in SOSA to handle uncommon cases gracefully. The system must broadcast all exception clauses into the system as predicated instructions. The GATHER_GO instruction does offer some control relief in that it can modulate the amount of times that an instruction is executed. This is good for essentially one line 'for' loops with a data dependant stopping point. While not widely applicable, this instruction does offer benefits.

Overall, programming and worthwhile computation seems possible on SOSA. This code, however, must often be hand tuned or highly parallelizable to achieve peak performance.

4. Integer Multiplication Study

Integer multiplication is another program that would be benefited by gather go. SOSA does not have a native multiplication instruction, so it must implement the function using a series of shifts and adds. With the shift and add loop, there is a direct trade-off between the number of available registers and the run time. Using the register modifiers and repeat counters built in to SOSA, loop unrolling can take advantage of all free registers. Alternatively, GAHTER_GO can be used in multiplication between shifts. A multiplicand can be added to the product a specific amount of times (found in the other multiplicand) then shifted. This method uses only three registers. Figure 8 shows the code for both forms of multiplication.

```
*REPEAT 3 8                    *REPEAT 3 8
CPSHIFTLM R2 R4+ R4+ 4         GATHER_GO R2 4
PSHIFTML R1 R4+ R1 4           PRADD R3 R1 R3 16
PRADD R4+ R3 R3 4              SHIFTLM R1 4
```

Figure 8– Integer Multiplication without (left) and with (right) Gather and Go

Type	Registers Used	Run Time (normalized)
Unrolled into 1 Register	4	1
Unrolled into 2 Registers	5	0.574
Unrolled into 4 Registers	7	0.323
Unrolled into 8 Registers	11	0.198
GG, 5 bits at a time (7 iterations)	3	0.467
GG, 4 bits at a time (8 iterations)	3	0.403
GG, 3 bits at a time (11 iterations)	3	0.467
GG, 2 bits at a time (16 iterations)	3	0.600

Table 6 - Integer Multiplication Runtimes

Table 6 shows the amount of registers used and the run time for the different variations of processes. The results show that if there are more than five registers available, then loop unrolling provides the best run time. If there are less than five registers, however, the GATHER_GO mechanism is the best. Interestingly, there is a minimum in the run time of the various versions of the gather go multiply. This is because at two bits at a time, many instructions are broadcast into the system and broadcast communication is the bottleneck. When executing five bits at a time, the system becomes bound by the communication latency of the predicated operations, not the instruction broadcast. This is a demonstration of just two of the trade-offs that are at work when designing programs to run on SOSA.

5. Recommendations for SOSA

After studying SOSA for several months, I was able to gain an appreciation of many of the intricacies in the system. Below are some of the improvements that may be worth looking into in the future.

5.1 Head & Tail Fusion

In comparison instructions, data must traverse the PE from head to tail to make the correct comparison. After this, the result must propagate from the tail back to the head, which stores the predicate bits. A possible mechanism for avoiding this reverse traversal is to fuse the head and the tail of the PE, making it a logical ring. This way of ordering processing elements may also provide benefits to PE shifts in that each PE can be thought of as a ring with the head node connected to the head node of the next PE. During PE shifts this would limit the amount of communication that needs to be sent through a PE.

One way of configuring such as system is, instead of grouping PE's during a depth first search, group them by subtree. Essentially a PE would not be made in a system on the DFS until a node had just the right amount of nodes, between 1-2 times the length of a PE, under it. This means that the only communication that would occur in this subtree is intra-PE and the only communication that would occur above it is inter-PE. This would potentially prevent specific nodes from being as much as a bottleneck in the system. The downside to this type of configuration is that the number of PE's that can be configured is much less. To offset this loss of PE's the configuration would need to offer a large speedup.

5.2 Data Memory

Processing elements in SOSA only have 16 (or 32) registers. For many processes, this is not a large working set. My idea to help counterbalance this would to have specific nodes or PE's dedicated to data memory in SOSA. In order to avoid the same loss of utilization as NANA, what I propose is configuring PE's into memory. For example, if each memory PE could work for 4 (or 8) compute PE's, each compute PE could have 8 (or 4) more registers available to it for storage. In order to configure this systme, there needs to be a separate routing entry in each node that points to its memory PE (mPE).

The cost of such a system would be the additional control logic in each node and the computational power lost to memory. SOSA would require new instructions for storing and loading data into the mPE's. Ideally, each mPE would be able to know where the data its receiving is coming form. As an example, in a the hypothetical STORE R3 M1 instruction, each PE would send its R3 to the memory PE. The mPE would place this data in the register equal to $4*(PE_{Num})+1$. Also, given even more logic, a shared memory system could be placed into these nodes for data aggregation and manipulation, since data is being sent to a central location. Currently, the aggregation of data occurs through PE shifts. No work has been done to look into adding mPE's into SOSA.

5.3 Instruction Cache

One of the major bottlenecks in SOSA, if not the largest bottleneck, is the broadcasting of instructions to all nodes. This broadcast not only takes time to reach all PE's, but it also creates bandwidth contention with instructions that nodes are executing. In order to combat this, an instruction cache could be placed into the system at random locations. Via a special instruction sent from the external controller, instructions could be stored in this cache. Another instruction would tell PE's to execute the instructions

stored in the cache. This would limit the amount of bits broadcast into the system for small, repeated sections of code. In order to study if this would be a useful addition to SOSA, code needs to be analyzed for repeated sections. An initial analysis was not optimistic about instruction caches that could only hold three instructions. In order to allow for localized conventional code control (branches) instruction caches may need to be too large to be physically feasible.

5.4 Multiple Instruction Vias

The motivation for having multiple instruction vias is the same as having the instruction cache. The difference here, however, is instructions are broadcasted in from different points in the system. The configuration phase of the system would still be done from one via, but broadcasting instructions would occur from different vias. In order for PE's to differentiate from the same instruction broadcasted twice and receiving the same instruction from two different places, the nodes need to store a new instruction gradient. This would allow for a lower latency broadcast of instructions. It would also lead to the ability for different nodes to execute different instructions based on their instruction via. Each of these PE's would still be able to pass data to the domains of other instruction vias. In order to exploit this new model, however, complex studies of how to divide programs between execution sections would be needed.

5.5 Wired OR

In order to combat the issue of needing to broadcast ever possible exception clause into the system, a wired OR could provide a mechanism of feedback to the external controller. Given a comparison instruction, using this mechanism, a controller could poll the entire system to see if an exception clause was encountered by any PEs. To do this, every head node in the PE satisfying the condition would attempt to propagate a 1 back to the controller. If necessary, the controller would then broadcast the exception clause into the system.

This type of mechanism would have several issues, however. First, the controller needs to know that a 0 has been received. Merely, listening for a 1 will not do here since there is no way of telling how long a propagation will take. A way of doing this would be to have each PE notify the PE next to it that it has completed, circling the ring of PEs. This would be a very long latency operation. Also, a probability analysis would be needed to figure out what is the actual probability of needing to broadcast the instructions and the expected time savings in doing the OR polling.

6. Connectivity Study

Given the limitations of the physical layer presented in the introduction, and the knowledge that communication is a bottleneck in the SOSA computer system, I next studied network connectivity. Previous work on the topic at Duke was an exploration on how deposition variables and defects played a role in the total connectivity of a system[9]

The definition of connectivity used in this study is the number of nodes that are reachable from an via that is placed in a system. This count is preformed in simulation by a random selection of a node to act as the anchor that initiates a breadth first search over all nodes in the system. Any node that receives a packet marks itself as visited and forwards the packet out of all links (transceivers) that are non-defective. Should a node receive additional packets, it sinks them. The number of nodes touched in this method divided by the total number placed in the system is equal to the connectivity.

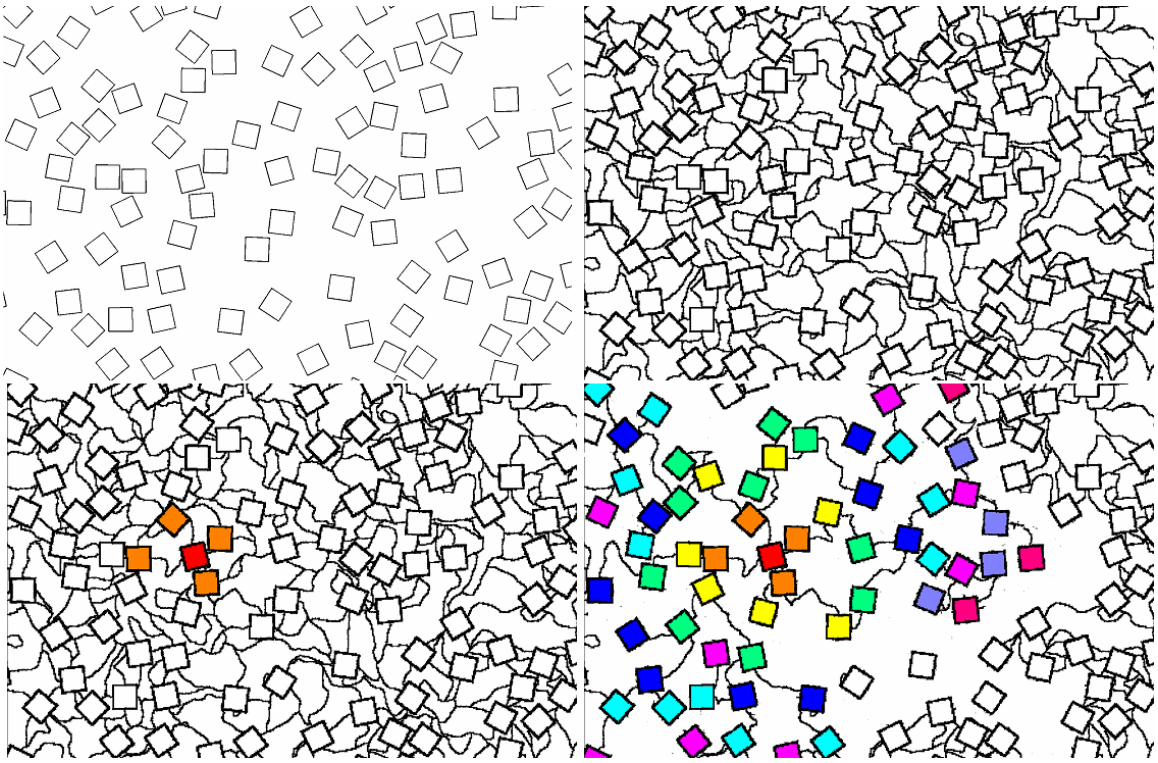

Figure 9 - The Growth and Cutting of a Topology

Patwardhan, et al.'s study of connectivity looked at topologies that were generated using a network growth simulator. This simulator placed nodes of the same size into a specified area then grew wires, using a random walk, out of each of four transceivers. This growth of wires allowed for fused links between more than two nodes to form. This growth model was taken to be close to the physical description given in the introduction. The simulator allowed a decision about what combination of the orientation (rotation), placement of nodes, and growth of wires were random. In the paper, the treatment of the fused link buses was varied. It was found that when all parameters were random, connectivity was high, nearly 100%, if all nodes on a link could communicate without restriction. When the links are modeled such that only two nodes can control a bus and all other transceivers are permanently cut off, the connectivity drops to approximately 3-5%. This is because of both fact that the cutting of these links occurs, and the specific methodology of cutting that was used in the paper (random). Section 6 provides an in depth look into this phenomenon.

6.1 Tools and Modifications

My studies of network connectivity use a modified version of the simulator used in reference 9. In order to study some of the different mechanisms responsible for connectivity, I modified the growth simulator to support heterogeneous node sizes and a flexible amount of transceivers.

6.2 Effect of the Number of Transceivers

The first study that I made into the connectivity of nodes was to look at the effect of increasing the number of transceivers that are present on each node. Using the random cutting of links, it was shown that increasing the number of transceivers in each node has

a large impact on the connectivity of the system. Increasing the number of transceivers, and thus outgoing wires, on each node from 4 to 6 provided an increase in connectivity from 6% to 57%. Going from four transceivers to ten provided an increase in connectivity to 94%.

Though the connectivity was at 94% at a peak, this number is not realistic in the current physical model. Having ten transceivers in each node would require more transistors than can currently be placed on the DNA scaffold. These numbers, however, do provide an idea of the gains that can be made if ways of including more than four transceivers, physically or logically, and growing more than four wires becomes possible.

7. Cutting Changes

As was indicated in the previous section, the connectivity of four transceiver nodes in the physically modeled system is low. The reason for this is that fused links get cut into links with only two nodes attached. This cutting is preformed at the power on of a system. Essentially, each node first does a built in self test (BIST) on itself. If any of the compute or control logic is found to be faulty, the node turns itself and its transceivers off[1]. If a transceiver is found to be defective, only that transceiver is turned off. After the BIST, each transceiver that is still on attempts to discover neighbors with which it shares a link. If more than two nodes share a link, a random back-off mechanism is used to choose the two nodes that will communicate with each other. Each node on a link has an equal probability of selection. Finally, the broadcast gradient is sent in from the via and the configuration of processing elements begins.

7.1 Waiting for the Broadcast

The first change that could be made in the configuration process is for the nodes to wait until they have received a broadcast packet before cutting the nodes. In this scheme, no transceivers cut themselves off the links at power on. When a node receives a broadcast packet, over each of its live links it sends a signal that notifies other nodes on the link to randomly select one to pair with the node receiving the packet. If two nodes on the same link simultaneously receive a broadcast, only one will remain on the link and pair with a node that has not received the packet. Image 9 shows the process of node placement (upper left), wire growth (upper right), the seeding of BFS (lower left), and the system after broadcast completion (lower right).

With this change implemented, the first study looks at the change in connectivity, still using a random selection model, when nodes wait for the broadcast. The results are promising. Instead of having a connectivity of 4%, a system with 1200 4-transciever nodes has a connectivity of approximately 65%! This indicates that when the links cut randomly, they tend to form isolated “islands” of nodes. It is imperative that nodes wait until they receive a broadcast packet before cutting the links.

7.2 Number of Vias

The amount of vias used in the system also has an effect of how much connectivity exists in a topology. With a system with one via, a single micro-scale wire is placed into the node system to send signals from the outside world. A node that is located near the wire is made the anchor and becomes the root of all the system trees. In systems with more than one anchor, the system will power on and configure around each anchor node, one at a time, resetting after every trial. After all the possible configurations have been

[1] The model is a fail-stop model and does not take into account Byzantine-type failures.

polled, the controller selects the via that has the most connectivity corresponding to it. This sort of configuration helps to improve the average connectivity in different systems and reduce the variability. Table 7 shows minimum and average connectivity numbers that were taken from a system with a varying amount of anchor nodes. These numbers were taken over $5000/\#_{vias}$ runs on the same topology. Since ten vias provided the highest amount of connectivity and is physically reasonable, all simulations will be of that system.

Number of Vias	Minimum Connectivity	Average Connectivity
1	0.06%	71.0%
2	0.06%	72.9%
3	66.3%	73.7%
5	71.4%	74.2%
10	72.3%	74.7%

Table 7 - Effect of Additional Vias

7.3 Exploiting Self Information

Though 65% is a large gain for the connectivity, the goal of getting as close as possible to the bus based 100% remained unsatisfied. If nodes are randomly selected to be placed on the wire, there is a chance that the random decision may be a "bad" decision. An example of this if three nodes were on a link, one received the broadcast and the remaining two needed to arbitrate for the link. If one link had no children and the other link had 3, the one with three reachable children would provide the most connectivity benefit if added to the system[2]. The random selection process assigns equal weights to each of the nodes. Ideally, the node that was optimal to achieve maximal connectivity would be selected, typically the one with three children.

The next cutting scheme studied does exactly that. Assuming an omnipotent knowledge of all nodes on a fused link, the node with the most live transceivers is selected. In the case of a tie, one of the nodes in the tie is selected randomly. The definition of a live transceiver in this study is one that that has not been cut off from their links or found to be defective. Aside from tie selection, this is a deterministic process. Physically, it is difficult to achieve this form of deterministic selection on a wire, but the method provides an idea of potential connectivity gains. The amount of gain realized in this process is an increase from about 65% to 71%.

In order to achieve a similar selection model with a physical design that is easier to implement, I studied the connectivity model that uses a modulated random back-off. The idea in the random selection of nodes is that each node tries to establish themselves on a link, and if it detects a conflict, it backs off for a random amount of time and retries, much like Ethernet [10]. This method varies the amount of time each node remains silent as an exponential equal to the amount of transceivers that they have live. This leads to a probability of a node selection probability on a link of:

[2] This is the case for the simple example. However, there are situations where this would not be optimal. An example of this is where the node with 3 children having an alternate route to the anchor.

$$P_{selection}(Node_n) = \frac{B^{T(n)}}{\sum_{i \text{ in } N} B^{T(i)}}$$

In the above equation B is an arbitrary base, taken to be 8 in this experiment. T(i) is the amount of transceivers alive on node i and N is the set of all nodes on a fused link. For this experiment, the connectivity actually increased to about 74%. This is close to the previous result and could be slightly higher because of the random selection method may select more optimal cuts than the greedy algorithm. Since this method provides a large amount of connectivity at a low physical cost, it is what all other studies use.

7.4 Using Local Information

If the greedy algorithm proved to be successful using only information from one node, what would happen if information was gathered from more than one level of the tree? There could be a situation for example, if one node had three children, all with no children, and one node had two children, all with three children. Using the deterministic algorithm applied from above, the first node would be connected, tying four nodes to the anchor. If the second node was chosen, however, twelve nodes would have been incorporated into the tree. The new method of cutting nodes looks into how acquiring more information on different levels affects connectivity. When nodes are arbitrating a cut, a node will add the number of transceivers it has alive to the amount that its children have alive. The node with the highest number is then deterministically selected. This process can be done for any amount of levels down the tree via recursion.

Simulating this system with a recursion depth of 2, 3, and 4 levels showed no significant difference in connectivity between the nodes. This lack of change between the no recursion model and the three different amounts of recursion indicate that either no new information is being added to the greedy algorithm's decision process, or that the information that is added has no impact. While this is an interesting result, implementing this type of system is more difficult than both the greedy algorithm that operates on one level and the modulated back-off, currently making implementing this system a no-win situation.

7.5 Using Global Information

The last type of information utilization is global. In this scheme, the anchor initiates a global broadcast throughout the system. In this broadcast, no cutting occurs, and each node remembers the node this is up the tree from itself. Next, starting with the children, a count is sent up the tree so that every node knows how many nodes are in its (uncut) subtree. The anchor reinitiates the broadcast, this time cutting links deterministically based on the counts found in each node. This scheme is extremely difficult to implement physically and offered no significant connectivity gain, therefore it was not furthered studied.

7.6 Discussion

Overall, the simple switch from nodes randomly cutting themselves to nodes waiting until the broadcast initiation to cut had the largest effect on connectivity. Implementing this feature would require minimal changes to the transceiver logic. For additional connectivity with a low implementation cost, a modulated back-off mechanism is best. Nodes know how many transceivers that they have alive by polling after BIST and transceivers asserting a signal on a wire to see if there are at least one response.

Implementing systems that exploit more than individual information do not currently seem to be worth the additional logic, in terms of connectivity.

Adding anchors to a system helps to serve two purposes. First, it raises the average amount of connectivity in a system. It also raises the minimum amount of connectivity found in a system. For future studies in this paper, all systems are taken to have 10 anchors and implement the modulated back-off scheme.

8. Effect of the Density

Using the modulated random back-off method of selecting nodes on a link, I studied the effect of the density of nodes placed onto a substrate. Table 8 shows the effect that increasing the number of nodes in a system has on the total amount of connectivity. In the table, 2200 nodes was the maximum number of nodes that could be placed in the constrained area. It was found that once about 1200 of the possible 2200 nodes are placed, the connectivity percentage reaches a relatively flat, increasing plateau. There is, however, a drop of from 2000 nodes to 2200 of about 4% (from 74% to 70%). This process was repeated for several different topologies with similar results.

Number of Nodes	Connectivity
400	54.7%
800	68.1%
1200	73.9%
1600	75.5%
2000	76.1%
2200 (Max)	69.3%

Table 8 - Connectivity as a Function of Node Density

8.1 Discussion

The first point to be made about these results is that although the percentage of nodes connected in the system decreases as the number of nodes increases from 2000 to 2200, the absolute number of nodes connected increases. This means that if nodes can be considered to have a free cost, then they should still be placed as densely as possible on the substrate. If the nodes are a limiting factor in the manufacturing process then designers should aim for a point that is near eighty percent maximum density. This will provide some leeway in the exact concentration while still achieving the maximal amount of node utilization. The next section challenges the assertion that with cost-free nodes should be packed densely as possible.

The results from this study do not take into account other factors that may be affected by the density. For example, although not studied, the lengths of the wires increase as node placement grows more distant. If these small changes have a non-negligible on the communication latency, it may have a significant impact on performance. Power density, if large enough, may also play a limiting factor in the selection of node density. If nodes are too dense and give off a significant amount of power, the system may produce more heat than can be cooled.

9. Effect of the Defect Model

Another part of the study of connectivity in the system is that of effects of defects. All the previous studies in selecting a cutting methodology and the effect of density assumed perfect nodes and transceivers. This is a study into what happens with the removal of that assumption.

There are two defect models that are studied: node failures and transceiver failures. During the BIST at the power on of the system, if a node discovers that its control or compute logic is defective, it powers down. This power down removes the node and all of its transceivers from the system. If a transceiver fails, then it powers itself down but the node remains functional. These defects are assigned in simulation by taking each node (or transceiver) independently, generating a random number between 0-1, and comparing it to the reliability.

9.1 Theory

Taking the case of a perfect tree of 2000 nodes with 4 transceivers (3 children), there is a total of 8000 transceivers, including those on the leafs of the tree. If the reliability of nodes is 0.75, then a total of 500 nodes will be shut down. Also being shut down with the nodes are their transceivers, 2000 in total. In this case if a node fails, then itself and its entire sub-tree is cut off from the graph, so the connectivity will be well below ideal value of 0.75 (all non-defective nodes). The probability of a node and its entire sub-tree failing is equal to R_N, or 0.75. The probability of any given node is not connected is equal to $1-R_N^{L+1}$, where L is equal to the level of the node in the tree (the root being 0).

Now, lets examine the case of what happens in the same tree when only the transceivers have defects. For this example, assume that R_t is 0.75 and that all transceivers are independent. In this case, a total of 2000 transceivers will fail and shut down, the same amount as in the node defect model. The number of nodes completely cutoff from the system is equal to $2000*(1-Rt)^4$, or about 8. Given these numbers, it seems apparent that the number of nodes connected in this tree would be higher than the number in tree from the previous paragraph.

Simple intuition in this case, however, is wrong. Looking at the tree, if the link above a node were to fail, the node below it and its entire sub-tree become disconnected. This is the same amount of nodes that get disconnected as in the scenario where the node at the bottom of the link becomes disconnected. This link fails whenever either of the two transceivers fail. Thus, the probability of a link failing and taking out an entire sub-tree is equal to $(1-Rt)^2$. This means that the probability of any given node being disconnected from the network is equal to $1-R_t^{2L}$.

The take away point from this theory is that given independent failures, that transceiver logic need to have a reliability that is approximately equal to $R_N^{0.5}$. Though this assumes an ideal tree with no redundant links, the qualitative idea still holds. In practice, the power that R_N needs to be raised to find the equivalent R_t is less than 2, indicating that transceivers are not as critical in the actual system. Physically, protecting the transceiver logic may not actually be needed. If failures in a system are the result of individual transistor failing, then the amount of transistors in each component becomes a factor. If transceivers have half the amount of logic than the rest of the node and each transistor failed independently, then R_T would inherently equal $R_N^{0.5}$. If this is not the case, balance between the reliabilities can be done using common protection techniques, provided enough resources exist.

9.2 Results

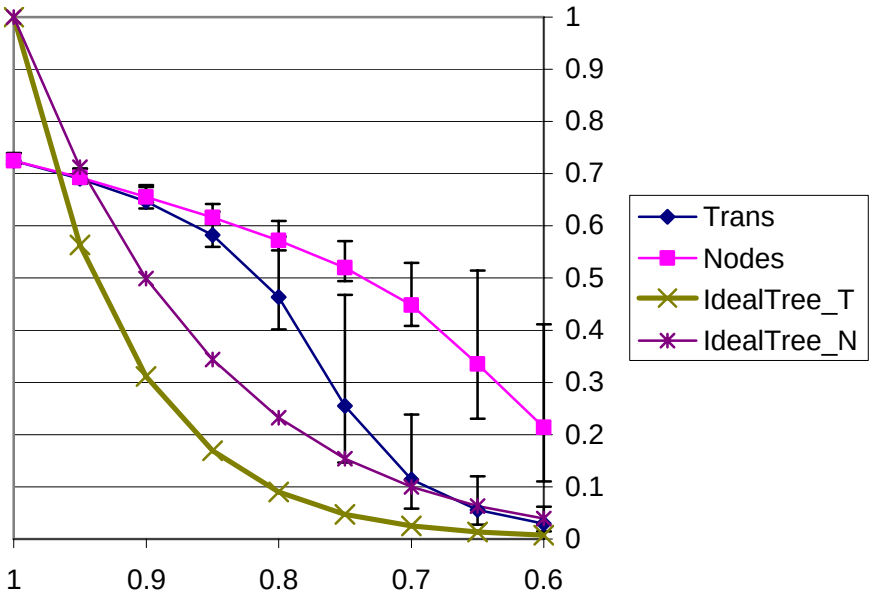

Figure 10 - The Effects of Defects (90/10 error bars)

Figure 10 shows the graph the percent of connected nodes as the reliabilities of either the nodes or the transceivers are varied. The lines labeled IdealTree_[N/T] show the amount of connectivity expected if all the nodes and links formed a perfect tree. Since the connectivity is above these numbers, it shows that that the system contains inherent redundancy between connected links. This result is not surprising, given the random nature of the growth, but it does provide a sanity check for the model.

When obtaining the actual results, only one of the reliabilities were varied, while the other was held constant. As expected, the connectivity for the system is lower at a given reliability for transceivers. The point at which the connectivity of the two systems become equal, however, isn't when $R_t = R_N^{0.5}$, but rather when $R_t \approx R_N^{0.6}$. This is an indication that the system performs better under the presences of transceiver defects than the ideal system. This may be an indication that there is more redundancy build into the wiring of the nodes than in the nodes themselves.

9.3 Density Defect

Another set of experiments that were ran for the system was a study of how well defects are tolerated for different densities of nodes. It was previously shown that in a defect free system, that there is a plateau for connectivity percentage and the amount of nodes placed in a fixed area. The reason for the drop off in this percentage as the amount of nodes increases may be because of the lack of routing space for links to fuse and add redundancy to a system. If this was the case, as the defect rate was increased, the system with maximal density would behave more like the ideal tree.

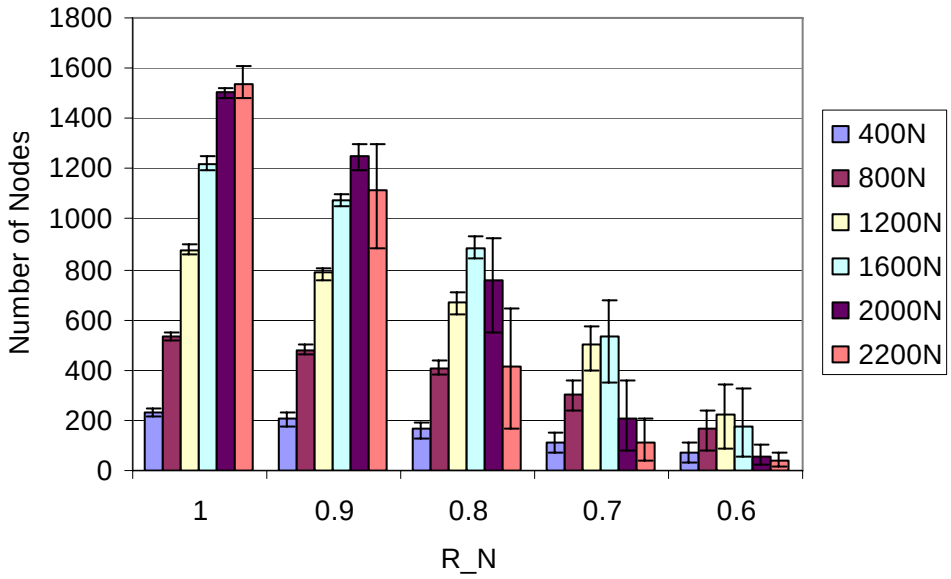

Figure 11 - Nodes connected as a function of R_N (90/10 error bars)

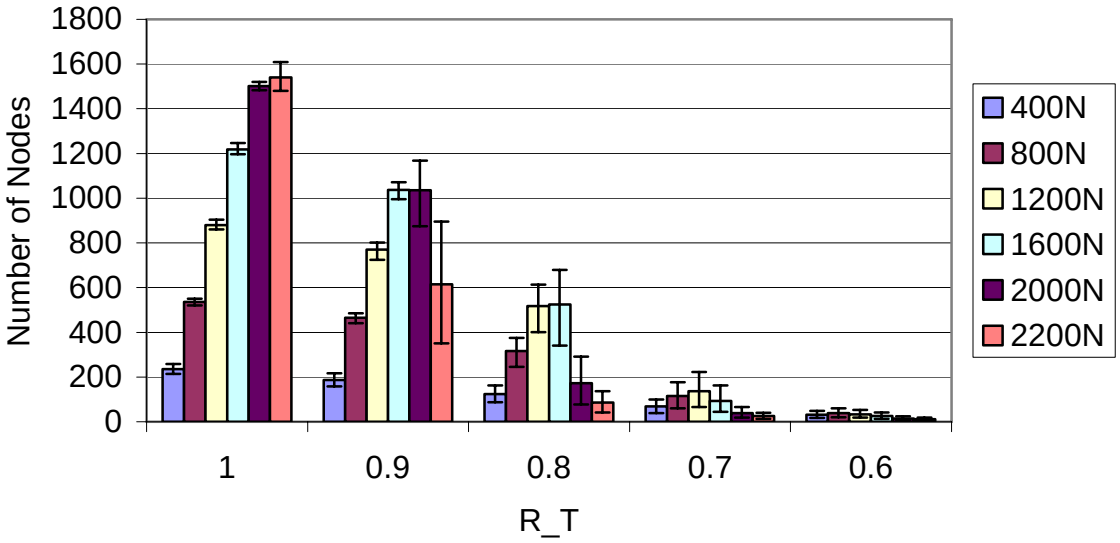

Figure 12 - Nodes Connected as a Function of R_T (90/10 error bars)

Images 11 and 12 show the effect of defects on node and transceiver reliabilities, respectively, in systems with varying node densities. A very important feature of these graphs is that the y axis is not the percent of nodes connected (as in previous graphs) but rather the total number of nodes that are connected to the anchor. This means that a system that has 1600 nodes with a reliability of 0.7 will have more nodes connected to

the anchor than a system with 2200 nodes with the same reliability. The reason for this is most likely to be from the lack of wired redundancy that occurs when there is more room for wires to grow. This indicates a trade-off between having too much wiring space (too many fused links) and too little (not enough fused redundancy).

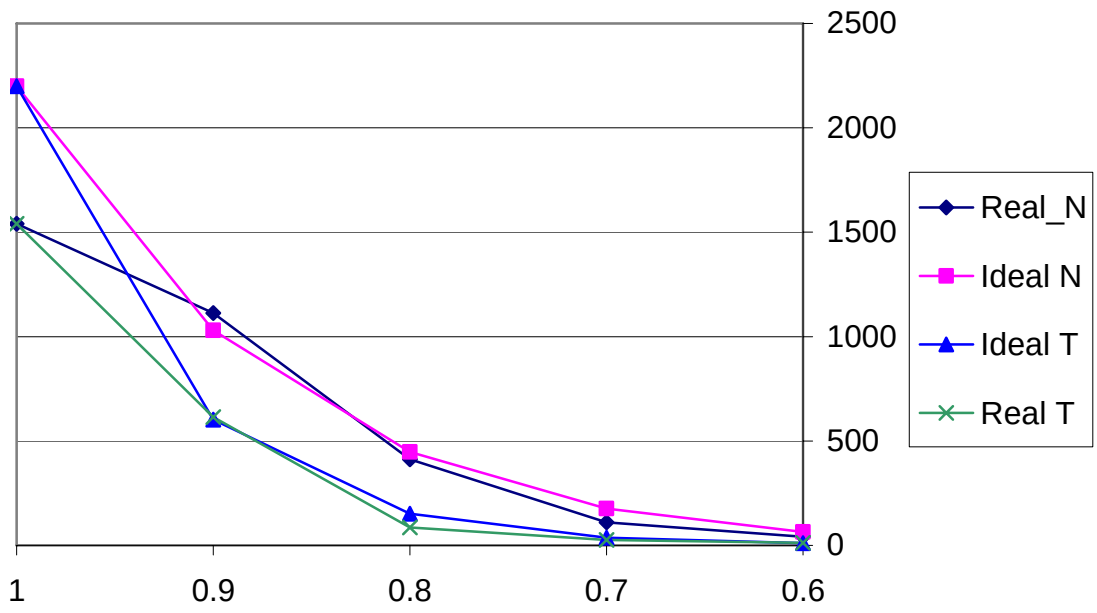

Figure 13 - Theoretical and Actual Connectivity of 2200 node system

Image 13 is a graph of the connectivity of nodes with specified reliabilities of transceivers and nodes plotted along with the theoretical number for a perfect tree calculated using the number of nodes placed in the system (2200). This helps to prove the theory that there is less redundancy inherent in the 2200 node system than the 1200 node system (figure 10). The number of connected nodes is only slightly higher than the theoretical. This indicates that there is still some redundancy built into the system, since there were only 1540 nodes originally connected, but not the amount of redundancy as systems with smaller amounts of nodes.

9.4 Discussion

The study of defect tolerance has yielded many insights into how to design a system. First, transceivers that are found in nodes should function with a probability that is approximately equal to the square root of the probability of a node failing. Second, adding too many nodes to a system is harmful in the presence of defects. This is not a cost analysis form as done in the previous section, but rather raw connection numbers. From these studies, it seems that the optimal point in the density is approximately 0.75 of the maximum. This yields a large some of the largest, most robust connectivity numbers in the presence of defects.

10. Balanced Tree, PEs

The final study of the connectivity of node systems looked into tree shape and formation. The results can be seen in table 9. The trees that form from the given configurations were by no means bushy. Instead of having an ideal three children per node, each tree had about 1.5 nodes per children. This number did not vary as the density of nodes changed.

Nodes	Children per Interior Node	Number of Levels	Theoretical Levels	Number of Children	Visited
400	1.5	36.6	14.3	82.4	257.6
800	1.5	38.1	15.5	174.8	527.8
1200	1.5	42.0	17.0	285.1	870.9
1600	1.5	47.2	17.5	407.1	1219.3
2000	1.5	68.6	19.0	477.0	1494.6
2198	1.5	81.0	18.9	501.3	1554.5

Table 9 - Tree Statistics

The trees were not balanced, either. Instead of having an ideal number of levels ($\log_{1.5}$ #), the trees have many more. For example, instead of the ideal 19 levels that 1500 nodes should occupy, they occupied 68. While this number is indicative of the shape of the trees, it is not necessarily bad.

The optimal PE formation, if the number of hops from head to tail is the metric, is a straight line down the tree. This sort of formation occurs when a depth first traversal encounters long branches. Having a tree that is not properly balanced, therefore, would decrease the amount of hops necessary for intra-PE communication. This will provide a speed increase in operations where only intra-PE communication is used, that is, everything put PE shifts. With PE's configured in chains, there is no bottleneck in these operations. When PE's are in miniature trees, however, the bottleneck becomes the root node, or the head of the PE.

In the case of PE shifts, however, this wisdom changes. If all PE's were subtrees, routing from one head to another would involve going up and down a hop in a balanced tree. If the tree were instead a chain, data would need to traverse through an entire PE to get to the head of the next one. There exist scenarios where a PE at the bottom of an unbalanced tree must send its data through the anchor node to communicate with the next PE. The shorter the distance to the anchor, the less time this communication takes to occur. This difference in communication overheads leads to an interesting trade-off in performances. If processes rely broadcasting instructions and PE shifts (as most do), then PEs should be comprised of small sub-trees in a bushy, balanced system. If a program is compute bound, however, the best connection configuration would be a straight line of nodes.

11. Related Work

The study of floating point and high base systems has been theoretically studied since the middle of the twentieth century. Many papers, for example, have been written on the

theory of high radix numbers [2]. A detailed discussion of the need for standards in floating point operations, a counter argument to the removal of IEEE compliance, can be found in [17]. The Thinking Machines CM-1 architecture [18] is similar to SOSA in that it consists of many nodes using predicated instructions. While some of the aspects of the architecture are physically infeasible, such as implementing a regular hypercube communication network, other aspects may be helpful to implement. Interestingly, the CM-1 machine has a global OR operation, which can check for exceptions. This would also be very beneficial to SOSA, greatly reducing the amount of instructions broadcast. The trade is that the latency of the instruction would likely be long and that if there are a billion different PE's there may be a high probability of detecting an unusual case.

There has also been work looking into how to best implement floating point numbers in systems without any specific hardware floating point support. Some systems take a fixed point approach to arithmetic [19]. This occurs when the compiler knows what the exponents will be in operations and does not need to explicitly store them. This way software will manage the exponents, leaving only integer computation. This does not work for SOSA, however, since there are be too many processing elements for exponent management to occur implicitly. Most similar to our work is that in [20], where the authors implemented a software version of floating point for automotive applications. The authors made several modifications that were similar to ours, such as using a high radix.

12. Conclusions

In this thesis, I have studied several aspects of nanocomputing. First, I improved the runtime for floating point operations in SOSA. I demonstrated many of the ways that can be used to better tune general assembly code for SOSA, by demonstrating algorithmic changes and the addition of a new control instruction. It has provided lessons that can be applied to other processes, such as multiplication, as well as demonstrated the feasibility of a key component of scientific computing.

SOSA runs on a substrate that is abstracted away, but that still plays an important factor in the design of the system. Connectivity numbers for the physical model were raised from about 5% to 74% using a new methodology for cutting fused links. Properties of these physical systems were also studied, such as the ideal density. A small node density can cause too many links to fuse and a low connectivity. A maximal density does not provide enough room for redundant links and causes connectivity problems in the face of defects.

13. Acknowledgements

I would like to thank my advisors, Professors Chris Dwyer and Alvin Lebeck, for the guidance and support in this project. I would also like to thank Dr. Daniel Sorin, who taught me just about everything I know about computer architecture. I would also like to thank my family who has constantly supported me throughout my years at Duke. This support would end if they were not included in the acknowledgements in my capstone project.

14. References

[1] International Technology Roadmap for Semiconductors, 2005

[2] P. Johnstone, F. Petry, "Higher Radix Floating Point Representations," *Proc. Of 9th Symposium on Computer Arithmetic*, pp 128-135, 1989.
[3] A. Bachtold, P. Hadly, T. Nakanishi, C. Dekker. "Logic Circuits with Carbon Nanotube Transistors," *Science*, Vol. 294, pp 1317-1320, 2001.
[4] Y. Huang, X. Duan, Y. Cui, L. Lauhon, K. Kim, C. Lieber. „Logic Gates and Computation from Assembled Nanowire Building Blocks," *Science*, Vol. 294, pp. 1313-1316.
[5] C. Dwyer, S.H. Park, T. LaBean, A. Lebeck. "The Design and Fabrication of a Fully Addressable 8-tile DNA Lattice," *Foundations of Nanosciences: Self-Assembled Architectures and Devices*, pp 187-181, 2005.
[6] M. Steffen, L. Vandersypen, I. Chuang. "Toward Quantum Compuataion: A Five-Qubit Quantum Processor," *IEEE Micro*, pp 24-34, March-April 2001.
[7] J. Patwardhan, C. Dwyer, A. Lebeck, D. Sorin. "NANA: A Nano-Scale Active Network Architecture," *ACM Journal on Emerging Technologies in Computing Systems*, 2(1):1-30, 2006.
[8] J. Patwardhan, V. Johri, C. Dwyer, A. Lebeck. "A Defect Tolerant Self-Organizing Nanoscale SIMD Architecture" *International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.
[9] J. Patwardhn, C. Dwyer, A. Lebeck. "Self-Assembled Networks: Control vs. Comlexity," *International Conference on Nano-Networks*, 2006.
[10] Ethernet Back-off
[11] Y. K. Dalal and R. M. Metcalfe. "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM*, 21(12):1040–1048, 1978.
[12] David Goldberg. "What Every Computer Scientist Should Know About Foatingpoint Arithmetic." *ACM Computing Surveys*, 23(1):5-48, 1991.
[13] G.M. Amdahl, G. A. Blaauw, F.P. Brooks. "Architecture of the IBM System/360," *IBM Journal of Research and Development,* pp 87-102, April 1964.
[14] R.P. Brent, "On the Precision Attainable with Various Floating-Point Number Systems," *IEEE Transactions on Computers*, C-22, pp. 601-07, 1973.
[15] Taken from http://ati.amd.com/products/radeonx800/index.html
[16]M. Ohmacht, R.A. Bergamaschi, S. Bhattacharya, A. Gara, et al. "Blue Gene/L Compute Chip: Memory and Ethernet Subsystem," *IBM Journal of Research and Development*, Vol 49, pp 255, 2005.
[17] W. Kahan, "Why do We Need a Floating-Point Arithmetic Standard?," 1981
[18] L. Tucker, G. Roberstson. "Architecture and Applications of the Connection Machine." *IEEE Computer*, pp 26-28, Aug. 1988.
[19] RISC Machines Ltd. "Fixed Point Arithmetic on the ARM", *Application Note 33*, 1996.
[20]D. Connors, Y. Yamada, W. Hwu. "A Software-Oriented Floating-Point Format for Enhancing Automotive Systems," *Workshop on Compiler and Architecture Support for Embedded Computing Systems*, 1998
[21] S.H. Park, C. Pistol, S.J. Ahn, J. Reif, A. Lebeck,C. Dwyer, T. LaBean. "Finite-Size, Fully Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures," *Angewandte Chemie,* 45:735–739, Jan. 2006.