Fault-Tolerant Parallel Prefix Computation: An Algebraic Approach

by

Zachary A. Kissel

A Thesis
Submitted to the College of Computer and Information Science
In Partial Fulfillment of the Requirements for
The Degree of Masters of Science

College of Computer and Information Science
Northeastern University
Boston, MA
September 2007

© 2007 Zachary A. Kissel
Abstract

The intent of this thesis is to investigate language-based algebraic approaches to model parallel prefix computation with faulty circuits. Such approaches provide mechanisms for denoting and reasoning about circuit constructions, yielding proofs of correctness that are easier to verify. This thesis studies bounds on the number of replicas needed for a circuit to be fault tolerant when a class of Byzantine faults is considered. Furthermore, the trade-off between frequency of voting and the amount of replication needed are considered.
Acknowledgments

Writing this thesis would not have been possible without many people. Firstly, the person most responsible for the pages in your hands. Riccardo, thanks for introducing me to process calculus. Especially thanks for all the mentoring and insight.

I owe many thanks to my parents. Thanks for encouraging me to explore and for always believing in me. Especially thanks for fostering my inquisitive side.

To my dad who has taught me so much about engineering and life. Thanks for lending me my first programming book, who knew that book would have been the start of all of this.

To my mother who lent an ear to listen to me ramble about fault tolerance and most of all for teaching me to follow my dreams. Thanks for helping me keep perspective.

To my brother who keeps me saner than he knows. Thanks for convincing me there is always time to step away from the work. Most of all thanks for bringing out my adventurous side.

Finally, to Wendy, you more than anyone understands the joy, stress and late nights that is thesis writing. Thanks for all the companionship. Most of all, thanks for always being there through it all – our journey together is just beginning.

Zachary A. Kissel
Boston, September 2007
# Contents

Abstract .................................................. ii
Acknowledgements .......................................... ii
Contents ..................................................... v
List of Figures .............................................. vii
List of Tables ............................................... ix

1 Introduction ............................................. 1
  1.1 Chapter Overviews .................................... 2

2 Background .............................................. 3
  2.1 Models .................................................. 3
    2.1.1 PRAM Model ........................................ 4
    2.1.2 Message Passing Model ............................ 5
    2.1.3 Circuit Model ..................................... 7
  2.2 Parallel Prefix Computation .......................... 8
    2.2.1 Parallel Prefix Computation on the Chain Topology 8
    2.2.2 Parallel Prefix Computation on the Mesh Topology 9
    2.2.3 Circuits For Parallel Prefix Computation ......... 10
  2.3 Fault Tolerance ....................................... 11
    2.3.1 Fault Models ...................................... 12
    2.3.2 Write-All Problem ................................ 13
    2.3.3 Byzantine Generals Problem and Consensus ....... 13
    2.3.4 δ-Loose Computation Circuits ................... 16
    2.3.5 Fault Tolerant Sorting Networks ................. 17
    2.3.6 Formal Methods for Fault Tolerance ............. 18
CONTENTS

3 An Algebra For Prefix Circuits 21
  3.1 Haskell ........................................... 21
    3.1.1 Types ........................................ 22
    3.1.2 Lists ......................................... 22
    3.1.3 Functions ..................................... 23
    3.1.4 User Defined Data Types ..................... 24
  3.2 Hinze Circuit Algebra ............................... 25
    3.2.1 Laws .......................................... 28
    3.2.2 A Haskell Realization ......................... 30
  3.3 Circuit Constructions ............................... 32

4 A Fault-Tolerant Prefix Circuit Algebra 35
  4.1 A Normal Form ..................................... 35
  4.2 Fault Tolerance ................................... 38
    4.2.1 Replication ................................... 38
    4.2.2 Total Majority Vote ............................ 38
    4.2.3 Faults, Fault Patterns, and Fault Families ... 40
  4.3 Achieving Fault Tolerance .......................... 41
    4.3.1 Fault Family $F_1$ ............................. 41
    4.3.2 Fault Family $F_{vr}$ .......................... 42
  4.4 Arbitrary Voting and Fault Family $F_v$ .......... 45
  4.5 Circuit Constructions ............................. 48
  4.6 Modified Replication ................................ 49

5 Conclusion and Future Work 53
  5.1 Summary of Results ................................ 53
  5.2 Directions and Auxiliary Results .................. 54
    5.2.1 Mapping Distributed Algorithms to Circuits .... 54
    5.2.2 More General Fault Models ..................... 55
    5.2.3 Improving the Algebra ........................ 55

Bibliography 57
## List of Figures

2.1 The Ladner-Fischer construction for $P_0(n)$ ............... 12  
2.2 The Ladner-Fischer construction for $P_k(n)$, $k \geq 1$. .... 12  
3.1 Fan on $n$ wires .................................. 26  
3.2 Identity on $n$ wires ................................. 26  
3.3 Circuit $[3,2,1] \triangleright fan_3$ .......................... 27  
3.4 Circuit $fan_3 \prec [3,2,1]$ ............................... 27  
3.5 Circuit $scan_4$ ......................................... 30  
4.1 Circuit $\hat{fan}_2 = fan_2 \times fan_2 \times fan_2$ .......... 39  
4.2 Circuit $scan_5^4$ ...................................... 45  
4.3 Circuit $lf 0 4$ ......................................... 48  
4.4 Circuit $scan_4$ ......................................... 48  
4.5 Circuit $bk_4$ .......................................... 49  
4.6 Circuit $scan_6$ ......................................... 50  
4.7 Circuit $lf 0 6$ ......................................... 50
List of Tables

3.1 Basic laws of the Hinze algebra .................................. 28
3.2 Magnitude and stretch laws of the Hinze algebra ............. 29
4.1 Voting laws .............................................................. 39
5.1 Necessary number of replicas per fault family. ............... 53
In Computer Science there is often a need to take a sequence \(a_1, a_2, \ldots, a_n\) of objects and combine them in such a way that you end up the sequence of prefixes \(a_1, a_1 \oplus a_2, a_1 \oplus a_2 \oplus a_3, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_n\), where \(\oplus\) is a binary operation on objects. These objects are traditionally drawn from a semigroup, which provides the composition operation \(\oplus\).

**Definition 1.0.1. (Semigroup)** A semigroup is a set \(S\) together with a binary operation \(\oplus\) such that: \(\oplus\) is associative and all elements of \(S\) are closed under the operation \(\oplus\).

Formally, the prefix computation problem is defined as follows. Let \((S, \oplus)\) be a semigroup. Given an ordered sequence \(x_1, x_2, \ldots, x_n\) of elements of \(S\), compute the \(n\) prefixes of this sequence, where the \(k\)th prefix is \(x_1 \oplus x_2 \oplus \cdots \oplus x_k\).

It is not too complex to define a sequential algorithm to solve the prefix computation problem. Moreover, it can be shown that this can be accomplished in time \(\Theta(n)\), where \(n\) is the length of the sequence.

**PREFIX-COMPUTE** \((x_1, x_2, \ldots, x_n)\)

1. for \(i \leftarrow 2\) to \(n\)
2. do \(x_i \leftarrow x_{i-1} \oplus x_i\)

Algorithm **PREFIX-COMPUTE** takes as input the sequence \(x_1, x_2, \ldots, x_n\). The algorithm then proceeds to compute all partial prefixes. For each element in the sequence beyond the first, the operation is applied to the previous
element’s value and the current element’s value. The result of this operation is taken as the current element.

The need to compute prefixes arises in areas diverse as job scheduling, polynomial interpolation, and simulation of finite state automata [19, 2, 27, 9].

This thesis is a first step in an algebraic investigation of a parallelized form of this problem in the context of unreliable systems. The faults considered are a subset of Lamport’s Byzantine Faults [29].

The main results of this thesis are:

- The addition of fault tolerance to an existing circuit algebra.
- An algebraic mechanism for reasoning about majority voting
- A tight bound on the number of replicas needed for a circuit to be fault tolerant with respect to a certain type of fault.

This thesis concludes with interesting directions that should be explored in the future work. Namely, around a general recurrence relation to generate fault tolerant circuits.

1.1 Chapter Overviews

This thesis is organized into the following chapters

- Chapter 2 presents the background on the parallel realization of the prefix computation problem. This chapter also provides a survey of the research surrounding prefix computation and general fault tolerance.

- Chapter 3 formally introduces the Hinze Algebra for prefix circuits as well as foundations of Haskell. This algebra is later extended to allow us to argue our results.

- Chapter 4 is the central chapter of this thesis. In this chapter fault tolerant extensions to the Hinze algebra are explored. Tight bounds for the number of replicas needed in order for a circuit to be fault tolerant are also discussed.

- Chapter 5 explores future work and reviews the main results of this thesis.
Chapter 2

Background

This chapter introduces the parallel prefix computation problem. This is the parallel realization of the prefix computation problem. Like many other problems, attempts to parallelize the prefix computation problem has led to more complex algorithms. In order to understand the parallel solutions, we must understand parallel computing models. This chapter begins with this discussion. We conclude with a discussion of major works in the field of fault tolerance.

2.1 Models

As with all forms of analysis, we must define the model we will use. The models we consider are the Parallel Random Access Machine (PRAM) model, the Message Passing model and the Circuit model. The PRAM model is concerned with parallel algorithms, the Message Passing model is concerned with distributed algorithms, and the Circuit model is concerned with distributed algorithms that occur in close proximity with no shared resources. Parallel algorithms make the assumption that memory is available locally and often shared between the parallel threads of execution. Distributed algorithms, on the other hand, assume memory is geographically distributed. This means the threads of execution do not implicitly share information. Algorithms in the Circuit model have constraints similar to the Message Passing model. These models, as will be shown, have their places in computing.
2.1.1 PRAM Model

The most pervasive parallel computing model is the Parallel Random Access Machine (PRAM) model. This model is an extension of the RAM model [22, 26, Chapter 2].

Definition 2.1.1. (PRAM Model) The Parallel Random Access Machine (PRAM) is a theoretical machine consisting of $p$ processors with shared memory of size $m \geq p$. Each processor $p_i$ is a RAM computer in and of itself. Moreover, each $p_i$ has private memory. It is assumed, like in the RAM model, that local memory access, read or write, takes unit time [22].

The RAM model’s notion of running time can be extended to be used in the PRAM model. The running time of a PRAM algorithm, denoted $T_p(n)$, is the time it takes the algorithm to complete on an input of size $n$ using $p$ processors. The PRAM model takes parallel operations in unit time that is, two operations that occur in parallel have the same running time as one of the operations occurring in the RAM model.

Unlike in the RAM model, running time does not completely describe the efficiency of a PRAM algorithm. In the PRAM model we are also concerned with the total amount of work that can be completed in a given amount of time.

Definition 2.1.2. (Parallel Work) The work $w(p, n)$ that a PRAM algorithm could potentially perform is given by $w(p, n) = pT_p(n)$, where, $p$ is the number of processors and $n$ is the amount of work to be done.

Together with the notion of running time, parallel work characterizes the efficiency of a PRAM algorithm. We would like the measures of both running time and work to be optimal. Optimality of running time is measured in terms of what is known as speed up.

Definition 2.1.3. (Speedup) The speedup $S$ of an algorithm is given by $S = \frac{T(n)}{T_p(n)}$, where $T(n)$ is the running time of the optimal RAM model algorithm on input of size $n$ and $T_p(n)$ is the running time of the PRAM algorithm executed on input of size $n$ using $p$ processors.

Definition 2.1.1 talks of shared memory available to all processors in the system. We have not, however, talked about how this shared memory is accessed. There are four basic memory access models employed by the PRAM model.
2.1. MODELS

• Concurrent Read Concurrent Write (CRCW). In this memory model every processor can read and write to all shared memory locations at the same time.

• Concurrent Read Exclusive Write (CREW). In this memory access model every processors can read from a given memory location at the same time, but only one processor can write to a memory location at a particular point in time.

• Exclusive Read Concurrent Write (ERCW). In this memory model only one processor can read from a given memory location at a time, but multiple processors can write to the same memory location.

• Exclusive Read Exclusive Write (EREW). In this model only one processor can read or write to a given memory location at a given time.

All of the literature pairs one of these memory access models with their invocation of the PRAM model.

2.1.2 Message Passing Model

The PRAM model is just one model used to study parallel algorithms. Another pervasive model is the Message Passing model or Distributed model. This model is like the PRAM model except there is no notion of shared memory. In the Message Passing model we define a communication topology that describes which processors can communicate directly with other processors. This communication is done through the use of Send and Recv operations. These operations are used to explicitly move information around a distributed system. Analysis of these algorithms depend on the same measures as the analysis of PRAM model algorithms. However, the Message Passing model also takes into account analysis of the communication complexity. An analysis of communication complexity is obtained by calculating an upper bound on the number of routing steps it takes to get a message from the sender to the receiver.

Several communication topologies have been explored in literature. The most common topologies are:

• The chain topology of size n, a topology where processors are labeled $p_i$ obeying the rule that for $1 \leq i \leq n$ processors $p_i$ can communicate
with processors $p_{i-1}$ and processor $p_{i+1}$. Processor $p_1$ may only communicate with processor $p_2$ and processor $p_n$ may only communicate with processor $p_{n-1}$.

- The ring topology of size $n$, a topology where processors are labeled $p_i$, obeying the rule that processor $p_i$ can communicate directly with $p_{(i-1) \mod n}$ and processor $p_{(i+1) \mod n}$.

- The mesh topology of size $n \times m$, a topology where processors are labeled with coordinates of the form $p_{i,j}$. In the mesh topology, the direct communications allowed are exactly those allowed if one was to treat the row, $i$, and column $j$ as distinct chain topologies.

- The cube topology of size $n \times m \times l$, a topology where each dimension is a treated as a mesh when speaking of direct communications between processors. Furthermore, each processor is labeled as $p_{(i,j,k)}$.

- The star topology, a topology where all communication must travel through one central node. This means that every processor, except the central node, has a direct communication link only with the central node.

These topologies directly impact the efficiency as well as the design of distributed algorithms. Three measures are usually studied:

- **Degree**: the number of communication links attached to a processor

- **Communication Diameter**: the maximal number of links that must be traversed in the optimal communication between two processors. This measure puts a strict upper bound on running time as well as communication time.

- **Bisection Width**: the minimum number of links that have to be removed to separate the communication topology into equal size sub-topologies. This measure serves as a bound on the movement of data through the distributed computer.

For clarity we treat communication topologies as graphs where a node in the graph is a processor and an edge, in the graph is a bidirectional communication path between two nodes. For further information on graphs and
2.1. MODELS

associated theory see [7]. Naturally, the three important measure involved in communication topologies have a graph theoretic analog. Degree is just the degree of a node in the graph. Communication Diameter is the diameter of the graph. Bisection Width is the bisection width of a graph.

2.1.3 Circuit Model

The Circuit model is similar to the Message Passing model in that distinct wires that must work together by acting as input to a gate. In its purest form, the Circuit model is given by the following definition.

Definition 2.1.4. (Circuit Model) A circuit is a directed acyclic graph. Each node of in-degree two represents an operation on its two inputs. Nodes of in degree two are known as gate nodes. All other nodes have in-degree zero and are labeled with an integer between 1 and $n$. These form the input nodes of the circuit [19].

The circuit operates by propagating data from the input nodes through the graph, following the wires (edges) of the circuit (graph). Each time a gate node is encountered the associated operation is performed. The result of the binary operation is stored at the given node. Finally, the node value is propagated over all outgoing wires (edges) of the current node. This process continues until the circuit has been completely traversed. The results of the circuits computation lies at the end of the circuit.

In practice we refer to wires having a value, and the point where two wires meet represents a gate operation. The result of that gate operation is placed on the output wire.

The main measures of efficiency of a circuit of size $n$ in the Circuit model is circuit depth $D(n)$ and the number of nodes $C(n)$ in the circuit. The depth of a circuit is given by the longest direct path from the input nodes to the output nodes. We determine the length of this path by counting the nodes. The measures in the Message Passing model also apply, but are not really focused upon in the literature.

The measure of circuit depth is important because it represents the parallel computation time. This means that the running time $T(n)$ of a circuit is proportional to the depth of a circuit ($T(n) \propto D(n)$). The maximum number of product nodes, $C(n)$, is important because it measures the number of computing units (gates) needed by the circuit.
The Circuit model of computations shows up in the study of comparator networks as well as a parallel prefix computation. This model is our main focus for the remainder of this thesis.

2.2 Parallel Prefix Computation

In Chapter 1 we discussed the prefix computation problem. Using the PRAM model from Definition 2.1.1 we can extend the problem to the parallel case. This extension is known as the Parallel Prefix Computation Problem.

Definition 2.2.1. (Parallel Prefix Computation) Let \((S, \oplus)\) be a semigroup. Given an ordered sequence \(x_1, x_2, \ldots, x_n\) of elements of \(S\) compute the \(k\) prefixes of this sequence, where the \(k\)th prefix is \(x_1 \oplus x_2 \oplus \cdots \oplus x_k\). Furthermore, each \(x_i\) is initially assigned to processor \(p_i\).

2.2.1 Parallel Prefix Computation on the Chain Topology

One of the earliest PRAM model solutions to the Parallel Prefix Computation Problem was presented by Kruskal, Rudolph, and Snir in 1985 [27]. The algorithm makes use of the EREW shared memory model.

\[
\text{PARALLEL-PREFIX-COMPUTE}(x_1, x_2, \ldots, x_n)
\]

1. for \(j \leftarrow 0\) to \(\lceil \log_2 n \rceil - 1\)
2. do for \(i \in \{2^j + 1, \ldots, n\}\)
3. doPar \(t_i \leftarrow x_{i-2^j} \oplus x_i\)
4. for \(i \in \{2^j + 1, \ldots, n\}\)
5. doPar \(x_i \leftarrow t_i\)

It can be shown that the \text{PARALLEL-PREFIX-COMPUTE} algorithm has running time \(\Theta(\log_2 n)\) and has a work measure of \(\Theta(n \log_2 n)\). From this measure and what we know about solving the algorithm in the serial case, we can assert that the speedup of this algorithm is \(\Theta\left(\frac{n}{\log_2 n}\right)\). This algorithm, in its message passing form, is a central building block for much of the subsequent work on the parallel prefix computing problem.

\text{PARALLEL-PREFIX-COMPUTE} extends to the Message Passing model assuming we have two functions \text{SEND}(val, dest) and \text{RECV}(val). SEND sends
2.2. Parallel Prefix Computation

a value \( val \) from the current processor to the destination processor \( dest \) relaying through other processors if it must. \texttt{Recv} receives the value \( val \). The resulting algorithm is \texttt{MP-Parallel-Prefix-Compute}.

\[
\text{MP-Parallel-Prefix-Compute}(x_1, x_2, \ldots, x_n)
\]

1. \texttt{for} \( j \leftarrow 0 \) \texttt{to} \( \lceil \log_2 n \rceil - 1 \)
2. \texttt{do for} \( i \in \{2^j, \ldots, n\} \)
3. \texttt{doPar} \texttt{SEND} \((x_i, p_{i+2^j})\)
4. \texttt{if} \( i \neq 2^j \)
5. \texttt{then} \texttt{Recv} \((x_{i-2^j})\)
6. \( x_i \leftarrow x_{i-2^j} \oplus x_i \)

2.2.2 Parallel Prefix Computation on the Mesh Topology

Eğecioğlu and Srinivasan [9] extended the work of Kruskal, Rudolph, and Snir in order to solve the parallel prefix computation problem on an \( \sqrt{n} \times \sqrt{n} \) mesh architecture in the Message Passing model. They introduce two algorithms, \texttt{Algorithm-A} and \texttt{Algorithm-B}.

In \texttt{Algorithm-A} we label the processors such that the bottom left hand corner of the mesh is processor \( p_{(1,1)} \) and the upper right hand corner of the mesh is \( p_{(\sqrt{n},\sqrt{n})} \). Each processor \( p_{i,j} \) is given the value \( x_{(j-1)\sqrt{n}+i} \).

- Preform the \texttt{MP-Parallel-Prefix-Compute} algorithm on each column bottom up.
- Perform the \texttt{MP-Parallel-Prefix-Compute} algorithm on the top row of the mesh
- The values at processors \( (1,1), \ldots, (1,\sqrt{n}-1) \) to each process is broadcast down columns \( 2,3,\ldots,\sqrt{n} \).
- As the values are broadcast the broadcasted value is incorporated with the current value of the processor using the binary operation.

We can now discuss the running time of algorithm \texttt{Algorithm-A}. We can see that phase one takes \( \Theta(\log_2 \sqrt{n}) \), and so does phase two. Phase three can be shown to takes \( \Theta(\sqrt{n}) \) time to complete. Therefore, the running time of \texttt{Algorithm-A} is \( \Theta(\sqrt{n}) \). Comparing this to the sequential algorithm for
solving the prefix computation problem we obtain a speedup of $S = \frac{\sqrt{n}}{\sqrt{n}} = \sqrt{n}$. This algorithm is not, however, optimal. Recall, that communication time is the third central measure in a distributed algorithm. This measure is what Eğecioğlu and Srinivasan seek to optimize. They present their optimal solution via Algorithm-B.

Algorithm-B makes the following assumptions: each processor $p_{(i,j)}$ is assigned either $x_{(j-1)\sqrt{n} + i}$ if $i \leq \frac{\sqrt{n}}{2}$ or $x_{j\sqrt{n} + \frac{\sqrt{n}}{2} + 1}$ otherwise. We now proceed to sketch the algorithm.

- As in phase one of Algorithm-A calculate the partial prefixes using the algorithm of Kruskal, Rudolph, and Snir. However, this time start from the top and bottom columns working towards the center of the mesh.

- Send the values from rows $\frac{\sqrt{n}}{2}$ to $\frac{\sqrt{n}}{2} + 1$ and perform the binary operation saving the result in row $\frac{\sqrt{n}}{2} + 1$, this is done for all columns. In parallel, the value at position $(\frac{\sqrt{n}}{2} + 1, j)$ is sent to position $(\frac{\sqrt{n}}{2}, j + 1)$ and perform the binary operation saving the result at location $(\frac{\sqrt{n}}{2}, j + 1)$. This is done for each column except columns less than $\frac{\sqrt{n}}{2}$.

- Next, the algorithm of Kruskal, Rudolph, and Snir is used to compute the prefixes across rows $\frac{\sqrt{n}}{2} + 1$ and $\frac{\sqrt{n}}{2}$.

- Now, the values in rows $\frac{\sqrt{n}}{2}$ are broadcast all the way to row $\sqrt{n}$. Each time the value passes through a processor, in row greater than $\frac{\sqrt{n}}{2} + 1$ the binary operation is performed and stored at the current processor location. In parallel send the values in position $(\frac{\sqrt{n}}{2} + 1, j)$ to position $(\frac{\sqrt{n}}{2}, j + 1)$, $\forall j < \sqrt{n}$. If the current row is not row $\frac{\sqrt{n}}{2}$ perform the binary operation and store the result at the current processor.

It can be shown that Algorithm-B does not improve upon speedup or running time. It does, however, reduce the number of communication steps. According to Eğecioğlu and Srinivasan, this is the optimal solution to the problem on the mesh computer [9].

2.2.3 Circuits For Parallel Prefix Computation

Thus far we have considered solutions to the parallel prefix computation problem on chains and meshes. These, and similar topologies, are not the
only way in which this problem has been approached. Earlier work on the parallel prefix computation problem considered solutions achieved through the use of circuits [19]. Ladner and Fischer introduced such a solution in the form of a recursive construction. This section makes use of Ladner and Fischer’s product circuit, or circuit for short.

**Definition 2.2.2. (Product Circuit)** A product circuit is a circuit in the sense of Definition 2.1.4, where each gate computes the product of its two inputs.

The construction that Ladner and Fischer present defines a family of circuits denoted $P_k(n)$. Where $k$ is a parameter, greater than or equal to zero, chosen for the construction and $n$ is the size of the prefix computation problem. The variable $k$ can be used to control the depth of the circuit. This allows us to define a family of circuits that solve the parallel prefix computation problem on a sequence of size $n$.

The circuit $P_k(n)$ is constructed recursively as shown in Figures 2.1 and 2.2. Note that in the definition of $P_k(n)$ in Figure 2.2, if $n$ is odd, then the right-most wire would pass through $P_{k-1}(n/2)$ From their construction, Ladner and Fischer argue that the depth is bounded above by $k + \lceil \log_2 n \rceil$ and the number of product nodes in the circuit is bounded above by $2 \left(1 + 1/2^k\right) n$. These bounds hold only as long as $n \geq 1$ and $0 \leq k \leq \lceil \log_2 n \rceil$. From these bounds, we can conclude that the construction yields circuits of depth, and therefore running time, $O(\log_2 n)$. It also can be shown that the construction has an asymptotic upper bound on the number of product gates of $O(n)$. Again, we see a speedup of $O\left(\frac{n}{\log_2 n}\right)$.

The Ladner-Fischer circuit is not the only circuit that computes the parallel prefix; there is also the Brent-Kung circuit construction [6] and the Lin-Hsiao circuit construction [21]. The Brent-Kung circuit construction, like the Ladner-Fischer circuit construction, has logarithmic depth. These circuits constructions are discussed in depth in Chapter 3.

### 2.3 Fault Tolerance

The algorithms presented thus far have assumed the availability and correct operation of the underlying hardware and software. This, however, is not always the case. More often than not, systems fail to operate reliably. This observation has not gone unnoticed. Considerable research efforts have been
spent towards analyzing unreliable (faulty) systems. This body of work is branded with the term Fault Tolerant Computing. We conclude this chapter with a discussion of important problems in fault tolerance.

### 2.3.1 Fault Models

Intuitively, a fault model describes how a system or algorithm can fail. Moreover, these models allow us to reason about a system or algorithms resilience to types of faults. We consider three fault models that have a hierarchical structure of increasing complexity. Due to this hierarchical structure, if an algorithm is fault tolerant in one fault model it is fault tolerant in all fault models below the current model. The models we consider are Fail-Stop, Fail-Stop Restart, and finally the most complex Byzantine.

- **Fail-Stop (crash):** In this fault model processing elements fail and then no longer participate in finding the solution to the problem.

- **Fail-Stop Restart:** In this fault model the processing elements fail and for an indeterminate amount of time not participate. Then at some
later time the processing elements begin participating in the algorithm from the point at which it stopped.

- Byzantine: In this fault model processing elements fail exhibiting unconstrained behavior. The processing elements can act as Fail-Stop elements, Fail-Stop Restart elements, or may unreliably communicate results to other processors.

In general all of the failure models are taken in conjunction with what is known as a fault pattern. A fault pattern \( f \) is a sequence of tuples \((f_1, f_2, \ldots, f_n)\), where \( f_\tau \) describes all the failures that occur at time \( \tau \). Study of specific fault models are often constrained to analyzing the behavior of families of fault patterns in the given model. Furthermore, literature often takes the fault pattern to be constructed by an adversary.

### 2.3.2 Write-All Problem

The idea of fault tolerance was studied for a long time in the PRAM model by many researchers. The fault model normally considered was the Fail-Stop model. This model received frequent attention through solutions to the classic Write-All problem, introduced in 1989 by Kanellakis and Shvartsman [17].

**Definition 2.3.1. (Write-All Problem)** Given a zero valued array of \( n \) elements, write a value of one into each array location using \( p \leq n \) faulty processors. Furthermore, assuming that at anytime at least one processors is functioning properly [17].

The most efficient solution to this problem was given by Anderson and Woll within the confines of the Fail-Stop model [1]. The Fail-Stop Restart model and Byzantine models have also been applied to the Write-All problem in [10] and [24].

### 2.3.3 Byzantine Generals Problem and Consensus

Perhaps the most famous investigation into fault tolerance came through the introduction of the Byzantine Generals problem which is a way to cast the consensus problem. Briefly, the consensus problem is the problem of having a set of distributed parties agree on a specific value or action. The Byzantine
Generals problem considers this problem in the presence of Byzantine faults. Consider the following situation. There are several divisions of the Byzantine army encamped outside of an enemy city. Each division is commanded by its own general. Assuming that the generals can only communicate by messenger how can they decide whether to attack, or retreat from, the enemy. Moreover, it is known that some of the generals are not trustworthy (traitorous) [29]. This is the so called Byzantine Generals Problem and stated formally by Definition 2.3.2. When this problem has a solution the solution is said to be a Byzantine Consensus. In order to have a solution to the Byzantine Generals problem the agents must communicate synchronously.

**Definition 2.3.2. (Byzantine Generals Problem)** A commanding general must send an order to his \(n - 1\) lieutenant generals such that:

1. All loyal lieutenants obey the same order.
2. If the commanding general is loyal, then every loyal lieutenant obeys the commander’s order.

Lamport, Shostak, and Pease proved an upper bound on number of generals that can be traitorous and still reach consensus.

**Theorem 2.3.1. (Impossibility of Consensus)** To reach consensus in the presence of \(t\) traitorous generals there must be \(3t + 1\) generals.

The proof of this result can be seen in [29]. The solution is developed in terms of a recursive algorithm. The algorithm assumes there are \(n\) generals. One general is the commander and the other \(n - 1\) generals are lieutenant generals. The algorithm goes as follows [29]:

- If the number of traitors \(m\), is zero
  1. The commander sends his value to every lieutenant.
  2. Each lieutenant use the value he receives form the commander, or the value RETREAT if the lieutenant does not receive a value.

- If \(m > 0\)
  1. The commander sends his value to every lieutenant.
2.3. **FAULT TOLERANCE**

2. For each \( i \), let \( v_i \) be the value Lieutenant \( i \) receives from the commander, or else be retreat if he receives no value. Lieutenant \( i \) acts as the command and performs the algorithm on \( m = m_1 \) to send the value \( v_i \) to each of the \( n - 2 \) other lieutenants.

3. For each \( i \), and \( j \neq i \), let \( v_j \) be the value Lieutenant \( i \) received from lieutenant \( j \) in step (2), or else RETREAT if lieutenant \( i \) did not receive a value. Lieutenant \( i \) then uses a majority, or voting function, to determine what the agreed upon value.

**Definition 2.3.3. (Majority Function)** Let \( V \) be a finite set of values. The majority function \( \text{Maj}(V) \) returns the value in \( V \) that occurs most frequently.

The majority function is often called the voting function; that name is adopted in this thesis. It can be shown that Theorem 2.3.1 does not hold when there is no central decision maker. Consider the case when we have three agents in a system that wish to reach consensus on the result of a computation. If we allow one of these agents to lie about the result we still obtain consensus using the voting function. According to Theorem 2.3.1 we would need four agents to handle one untrustworthy agent. We have just argued a counterexample that shows this is not the case when we do not have a leading agent in the system. Theorem 2.3.2 handles the case when all agents in the system contribute equally to the decision.

**Theorem 2.3.2. (General Consensus)** To tolerate \( m \) faulty agents in a system and still reach consensus, there must be \( 2m + 1 \) agents in the system.

**Proof.** Consider the case where we have \( 2m + 1 \) agents, where \( m \) is the number of faulty agents. Furthermore assume that all agents are performing the same computation and uses a majority vote to determine the correct result. Since \( m \) of the agents in the system are faulty there are \( m + 1 \) agents in the system that are correct. Because \( m < m + 1 \) we know that the majority vote favors the correct computation carried out by the \( m + 1 \) agents.

We can show that the \( 2m + 1 \) bound is tight. Assume, by way of contradiction, that only \( 2m \) agents are needed in the system. Further assume that \( m \) of these agents are faulty. This means there are \( m \) faulty agents and \( m \) fault free agents. Since the number of fault free agents equal the number of faulty agents a majority vote will not yield a solution. A contradiction has been reached. Therefore, at least \( 2m + 1 \) agents are needed. \( \square \)
2.3.4 δ-Loose Computation Circuits

Consensus is one problem investigated in terms of fault tolerance but it often requires more resources than are available. Motivated by comparator networks [18, 26] Gál and Szegedy formulated what is known as δ-loose computation. Intuitively this means that one allows for the answer to be slightly incorrect in exchange for reduced replication. By using δ-loose computation, Gál and Szegedy were able to introduce a level of fault tolerance with limited hardware resources [14].

**Definition 2.3.4. (Comparator Network)** A comparator network is a circuit with gates known as comparators. A comparator has two inputs \( v_1 \) and \( v_2 \) and two outputs \( v'_1 \) and \( v'_2 \). The comparator performs the following operation on its inputs:

\[
\begin{align*}
    v'_1 &= \min (v_1, v_2) \\
    v'_2 &= \max (v_1, v_2)
\end{align*}
\]

**Definition 2.3.5. (δ-Loose Computation)** For any computation device \( M \) we say that \( M \) δ-loosely computes a binary function \( f \) if

1. Whenever \( f(x) = 1 \), \( M(x) = 1 \)

2. if \( f(z) = 0 \) for every \( z \) with \( d(x, z) \leq \delta n \), then \( M(x) = 0 \).

Where \( d(x, z) \) denotes the Hamming distance, or number of different bits, between \( x \) and \( z \). Moreover, \( M \) outputs a random value if \( f(x) \) does not output a value that falls into one of the two conditions [14].

Gál and Szegedy go on to define the notion of γ-faulty circuits. Using the notions of δ-loose computation and γ-faulty circuits they were able to develop a circuit for computing symmetric functions in a fault tolerant manner.

**Definition 2.3.6. (γ-Faulty Circuit)** We say that a circuit is γ-faulty if at most γ fraction of the gates on each level of the network are faulty [14].

**Theorem 2.3.3. (Fault Tolerant Symmetric Functions)** For any \( \delta > 0 \) there is a \( \gamma > 0 \) such that for any symmetric function \( f \) there is a synchronized circuit with following properties:
2.3. FAULT TOLERANCE

1. If an adversary destroys a $\gamma$ fraction of the gates on every level (including the input level), the circuit still computes $f$ in a $\delta$-loose manner.

2. The size of the circuit is $O(n)$.

3. The depth of the circuit is $O(\log_2 n)$.

Their formal constructions are beyond the scope of this thesis. Intuitively, however, the solution is realized through the use of $\epsilon$-halvers and the fact that any symmetric function can be represented by a 0-1 string of length $n - 1$ where the $i^{\text{th}}$ element of the string is one if and only if the function yields one when exactly $i$ inputs are one.

**Definition 2.3.7.** ($\epsilon$-Halver) An $\epsilon$-halver is a bounded depth comparator network with the property that for any set of the $l$ smallest (largest) inputs, where $l \leq n/2$, at most $\epsilon l$ elements are among the last (first) $n/2$ outputs [14].

It is most convenient to think about $\epsilon$-halvers as comparator networks that that separates the $n$ inputs into the $n/2$ largest and $n/2$ smallest misplacing no more then $\epsilon$ of the values.

### 2.3.5 Fault Tolerant Sorting Networks

There has also been a fair amount of effort toward developing fault tolerant sorting networks.

**Definition 2.3.8.** (Sorting Network) A sorting network is a comparator network where the final output is a monotonically increasing sequence [18].

Important in this area is due to Leighton, Ma and Plaxton, and to Sun and Cerny. The work of Leighton, Ma and Plaxton provides a bounds on the depth of the sorting network when the network is allowed to produce an incorrect result with negligible probability [20]. (Roughly speaking a probability is negligible if the probability is effectively zero) These bounds hold under the conditions that every comparator fails with some non-negligible probability

Sun and Cerny provide a sorting network that produces, with high probability, the correct result in the presence of faults. Their result holds under the assumption that comparators fail deterministically [30]. The work of Sun and Cerny makes use of the idea of a critical comparator.


**Definition 2.3.9. (Critical Comparator)** A critical comparator is a comparator that operates on two adjacent wires.

A comparator is critical when the network cannot produce the correct sorted sequence should that comparator fail. Sun and Cerny develop a fault-tolerant solution that uses the notion of critical comparators and critical stages. A critical stage is a region where all wires participate in one comparison, if every comparison in this region fails the sorting network fails to sort correctly [30]. Their solution comes at the cost of additional comparators in the network. The work on fault-tolerant sorting networks is interesting, for the purposes of this thesis because it gives a glimpse into the limited work done in terms of fault-tolerant circuits.

### 2.3.6 Formal Methods for Fault Tolerance

Researchers have investigated ways to prove an algorithm to be fault tolerant. Lamport, Shostak, and Pease achieved their results through the use of traditional mathematical arguments. Limited work has also been done through the use of process calculus. Process calculus, in its most basic form, provides tools for establishing properties. The use of process calculus allows algorithm and systems designers to reason about complex systems formally. Process calculus also allows for the proofs by automatic means.

Yeung and Schneider developed the idea of distributed recovery blocks as an approach to fault tolerance in general software design [31, 32]. Their methods were proved using CSP a process calculus developed by Hoare in 1978 [16].

Process calculus has also been used by Nestman, Fuzzati, and Merro to explore the Byzantine Consensus problem. They were able to develop a process calculus in which they could provide formal proofs for many Byzantine consensus algorithms [13]. The consensus problem was further studied in terms of process calculi by Francalanza and Hennessy in [12].

Most recently there has been work towards extending the simply typed \(\lambda\) calculus [3] in order to make it fault, tolerant by Walker, Mackey, Ligatti, Reis, and August in [25]. They achieved their results through a statically typed \(\lambda\) calculus. This form of the typed \(\lambda\) calculus treats fault tolerance as a type safety property. This makes a guarantee of fault tolerance a statically checkable property. The intuition surrounding the work of Walker et. al. is to replicate every \(\lambda\) calculus expression three times and vote on the result.
after the operation has completed. They assign a colored type to each piece of data allowing only data of the same colors to be combined. This gives each replicated λ calculus expression a color.

Formal methods is the realm of this thesis. It is the purpose of this thesis to extend an algebra so it can be used to express fault-tolerant algorithms for the parallel prefix problem within the Circuit model.
Chapter 3

An Algebra For Prefix Circuits

This chapter introduces the Hinze algebra, which is used to describe and reason about circuits. Intuitively, the Hinze algebra provides operators for constructing circuits inductively from smaller circuits, and comes equipped with an equational theory that equates circuits that define the same function from inputs to outputs. The Hinze algebra is defined in Haskell, therefore, a brief introduction to Haskell is provided. This chapter also investigates the forms of prefix circuits as well as their expressions in the Hinze algebra. A few additional laws and constructs are also be discussed.

3.1 Haskell

Haskell is a functional programming language that implements a typed $\lambda$ calculus. It is a lazily evaluated language. This means expressions are not evaluated until they are needed. Although we will not make use of laziness here. This feature allows Haskell to conveniently handle structures such as infinite lists. The main distinction of Haskell is that it is side-effect free. This implies referential transparency: meaning an expression can be replaced with its value without changing the program. We will just be scratching the surface of this language; for a good in depth discussion of functional languages and Haskell please see [5, 23].
CHAPTER 3. AN ALGEBRA FOR PREFIX CIRCUITS

3.1.1 Types

Types in any programming language refer to the expected form of data as well as the data returned as a result of an operation. Haskell requires a type for everything. There are a few base types as well as the idea of type variables provided by the language. Some of the more common base types are integers (Integer), Booleans (Boolean), and reals (Double). There are also type variables which allows any type to be substituted. In this thesis, as tradition dictates we use Greek letters to indicate type variables. We can say that an object in Haskell has a specific type by using the following syntax:

\[(\text{Object}) :: (\text{Type})\]

For example, to assert that the number two has type Integer, we would write \(2 :: \text{Integer}\). Typing is discussed more as the need arises.

3.1.2 Lists

Lists and list processing is central to most functional programming languages, and Haskell is no different. Lists in Haskell are enclosed in brackets. For example, \([1, 2, 3]\) is a list of three elements. Moreover, we denote the empty list as \([\,]\).

Lists are concatenated using a double plus (\(+\)). For example \([1, 2, 3] + [4, 5, 6]\) evaluates to \([1, 2, 3, 4, 5, 6]\). One may also construct a list by composing elements together using a colon (\(\:\)) for example \(1 : 2 : 3 : []\) is equivalent to the list \([1, 2, 3]\). This form of composition is also used to compose an element and a list for example \(1 : [2, 3, 4]\) is equivalent to the list \([1, 2, 3, 4]\). Another way to construct a list is through the use of list comprehension this is similar in form and meaning to the idea of set comprehension in mathematics. An example of list comprehension is: \([x * x | x ← [1, .., 10]]\). This notation means that we construct a list that contains the square of every element of the list \([1, .., 10]\). We could optionally have created a comma separated list of constraints. The notation \([1, .., 10]\) also deserves some mention. It allows the programmer to define a list containing a range of elements. This range notation may be familiar to those readers that have experience with Pascal.

Thus far we have given no mention to types used with lists. A list can have any (fixed) type. It must be the case, however, that every element of a list be of the same type. A list is said to be of type \(T\) if these constraints are
satisfied, written \([T]\). For completeness, it should be noted that the empty list is polymorphic and therefore has every type.

### 3.1. Functions

A function is an operation that maps elements from one set to another set. Just as mathematical functions have types, expressed in mapping notation, Haskell has functions with types expressed in a similar mapping notation.

We know, for example, that most functional programming languages have a way in which to obtain the first element of a list. In Haskell this function is called \textit{head}. We would write the type expression for \textit{head} as

\[
\textit{head} :: [\alpha] \rightarrow \alpha
\]

This means the function \textit{head} takes a list of elements of type \(\alpha\) and returns an element of type \(\alpha\). This type expression denotes a polymorphic function. A polymorphic function is a function that can use any type. Thus, the same function \textit{head} can be applied to lists of \textit{Integers} as well as lists of \textit{Booleans}.

We can write a similar type expression for \textit{tail}. Function \textit{tail} is the function that takes a list and return a list with the first element missing.

\[
\textit{tail} :: [\alpha] \rightarrow [\alpha]
\]

In general a function can be of any arity with its types separated by arrows (\(\rightarrow\)). The return type of the function is always given by the last type in the type expression. A function of arity greater than one is known as a higher order function. The evaluation of these functions is achieved through a process known as currying. Currying allows for a function of multiple arguments to be evaluated by applying the function one argument at a time and returning a function after each argument is processed. For example, consider the function \(f \ x \ y = x \ast y\). When \(f 3\), say, is evaluated, it returns a new function, \(g_3\), which can be thought of as having definition \(g_3 \ y = 3 \ast y\). Then, \((f 3) 4\) evaluates to 12, as expected. Expressions such as \((f 3) 4\) are typically written simply as \(f 3 4\).

Like other functional languages, Haskell allows anonymous functions, called lambda expressions. A lambda expression is written \(\lambda v \rightarrow \langle expr\rangle\), where \(\langle expr\rangle\) is any valid Haskell expression that may refer to \(v\).

We have discussed how to denote the type of a function, higher-order functions and how to construct a lambda expression. We now discuss how
actually define a function in Haskell. Say, for example, we wish to define a function, \textit{avg}, that takes two integers and returns their arithmetic mean. The type signature for this function would be

\[
\text{avg} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Double}
\]

To write a function we use a notation akin to mathematical notation. We write the function name on the left hand side followed by its parameters separated by spaces then an equal sign followed by the expression. So, for our \textit{avg} function we would have

\[
\text{avg} \ x \ y = (x + y) / 2
\]

Taken together with its type we have

\[
\text{avg} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Double}
\]

\[
\text{avg} \ x \ y = (x + y) / 2
\]

To apply a function like \textit{avg}, we simply write the function name followed by the arguments to the function. In the case of \textit{avg}, we could, for example, write \textit{avg} 5 2 to compute the arithmetic mean of five and two.

We introduce just one more native Haskell functions. That function is \textit{map}, which takes a function and a list and applies that function to every element of a list, then returns the resulting list. This is captured by its type equation.

\[
\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
\]

This function is extremely useful and crops up in many applications.

\subsection{User Defined Data Types}

Sometimes the data types predefined in Haskell are not enough. Haskell allows a programmer to add new types to the language, in particular through the use of the \texttt{data} keyword. Data declarations have the following form:

\[
\texttt{data} \ \langle \text{name} \rangle \ \langle \text{opt\_type\_param} \rangle = \langle \text{cname} \rangle \ \{ \langle \text{elements} \rangle \},
\]

where \textit{name} is the name of the new data type, \textit{opt\_type\_param} is an optional type parameter, \textit{cname} is the name for the constructor and \textit{elements} is a comma separated typed list of members of the new data type. Members of
the new data type maybe of any type including a function type. It should be noted that `opt_type_param` is used to make the new data type polymorphic.

An example of a data type declaration for a cartesian coordinate is given as:

```haskell
data Coord = Point {x :: Integer, y :: Integer}
```

When using a new data type we often elide the field names. We construct a value of type `Coord` by writing `Point(10, 20)`, for example. In this notation field values are given in the order that they appear in the definition. To demonstrate how one could operate on this new data type, we construct a function that translates a point along the $x$ axis.

```haskell
xTranslate :: Integer → Coord → Coord
xTranslate scale Point(x, y) = Point(x + scale, y)
```

Notice that in order to deconstruct the data type we need to place variables in the constructor as an argument to the function. (This is a form of pattern matching, which is used to break data apart based on its structure.) These variables can be referenced in the body of the function. We could further modify our data declaration to be polymorphic in the type of the $x$ and $y$ coordinates. This can be achieved using the following data declaration.

```haskell
data Coord α = Point {x :: α, y :: α}
```

Now `xTranslate` can be changed only to work on `Integer` coordinates by doing the following.

```haskell
xTranslate :: Integer → (Coord Integer) → (Coord Integer)
xTranslate scale Point(x, y) = Point(x + scale, y)
```

Notice that all that needed to change was the type annotation. The function declaration did not need to change.

### 3.2 Hinze Circuit Algebra

The Hinze circuit algebra, or algebra of scans, is a small algebra developed by Ralf Hinze to aid in the description of prefix circuits [15]. It provides two powerful reasoning tools: the ability to reason inductively on the structure of expressions denoting circuits to prove properties of circuits; and laws that
allow for reasoning about equivalence of circuit expressions. Traditionally, prefix circuits are described graphically. This, for example, was done by Ladner and Fischer to describe their circuits in [19]. Hinze’s algebra, on the other hand, allows for a concrete definition of circuit constructions that can be easily manipulated.

The algebra is made up of a set of two base elements, a fan on \( n \) wires denoted \( \text{fan}_n \) and an identity on \( n \) wires denoted \( \text{id}_n \). A fan can be thought of as a broadcast from the leftmost wire of a circuit to all other wires. The identity on \( n \) wires can be thought of just \( n \) wires that carry a value. Figures 3.1 and 3.2 depict \( \text{fan}_n \) and \( \text{id}_n \) respectively. Formally, a fan on \( n \) wires is a broadcast of the value on wire 1 to wires 2 through \( n \). When the broadcasted value reaches its destination it is composed with the value on the destination wire, using the semigroup operation of the underlying prefix computation problem. It should be noted that \( \text{id}_0 \) represents a circuit with no wires and that \( \text{id}_0 \) and \( \text{fan}_0 \) denote the same circuit, as do \( \text{id}_1 \) and \( \text{fan}_1 \). Circuits also have a notion of size. The size of a circuit is measured by the number of wires it has. We denote the size of circuit \( c \) by \( |c| \). For example, we have \( |\text{fan}_3| = 3 \).

Fans and identities are not very useful by themselves. We need a way to compose the base elements of the algebra. There are two ways to compose elements. The first is parallel composition and is denoted by a multiplication symbol (\( \times \)). The second is sequential composition and is denoted by a semicolon (\( ; \)). Composing two elements of the algebra in parallel allows for the operations of the elements to occur at the same time. Composing
two elements of the algebra in sequence forces the operations of the two elements to occur one after the other. Two elements may only be composed in sequence if and only if they are of the same size. The algebra imposes an order of operations where parallel composition binds tighter than sequential composition. An example of a circuit that uses both parallel and sequential compositions is

$$\text{fan}_3 \times \text{fan}_2; \text{fan}_5$$

There are two additional operations that are of importance. These operations are known as stretches. Not every circuit we wish to create can be constructed out of broadcasts. A stretch allows us to have a multicast operation. Stretches come in two forms denoted $<$ and $>$. A stretch takes a circuit $f$ and a stretch vector $x$ such that $|x| = |f|$. The stretch $>$ with a stretch vector of $[3, 2, 1]$ applied to $\text{fan}_3$ looks like Figure 3.3 and is denoted $[3, 2, 1] > \text{fan}_3$. Notice that the $>$ stretch places $i - 1$ wires in front of wire $i$. This can be thought of as stretching the existing circuit and connections inserting an appropriate amount of wires as dictated by the stretch vector.

The $<$ stretch has a similar interpretation. However, the number of wires expressed in the stretch vector are placed after the original circuits wires. Using the previous example’s stretch vector of $[3, 2, 1]$ we can write $\text{fan}_3 < [3, 2, 1]$. This is represented graphically in Figure 3.4. It should also be noted that stretches bind tighter than sequential and parallel composition.
CHAPTER 3. AN ALGEBRA FOR PREFIX CIRCUITS

3.2.1 Laws

Thus far we have given an algebra without laws that we can use to manipulate expressions. We now summarize the laws presented by Hinze as well as derive an additional result that become important in Chapter 4. In the following tables $f$, $g$, and $h$ represent arbitrary circuits in the Hinze algebra. Table 3.1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id_{</td>
<td>f</td>
</tr>
<tr>
<td>$f; id_{</td>
<td>f</td>
</tr>
<tr>
<td>$id_{</td>
<td>f</td>
</tr>
<tr>
<td>$f \times id_{</td>
<td>f</td>
</tr>
<tr>
<td>$f; (g; h) = (f; g) ; h$</td>
<td>Associativity of Sequential Composition</td>
</tr>
<tr>
<td>$f \times (g \times h) = (f \times g) \times h$</td>
<td>Associativity of Parallel Composition</td>
</tr>
<tr>
<td>$(f \times g); (f' \times g') = (f; f') \times (g; g')$</td>
<td>Distributivity</td>
</tr>
</tbody>
</table>

Table 3.1: Basic laws of the Hinze algebra

only captures the most common algebraic laws. Hinze also presents laws for magnitude calculations and laws for stretches, given in Table 3.2. There are more stretch laws that Hinze introduces; however, we do not need them in this thesis. A full list of the laws can be found in [15]. We now introduce one additional property that becomes important in Chapter 4.

**Theorem 3.2.1.** $fan_n \times fan_m = fan_n \times id_m; id_n \times fan_m$ holds for all $n$ and $m$.

**Proof.** This derivation goes as follows:

$$fan_n \times fan_m = (fan_n; id_n) \times (id_m; fan_m) = (fan_n \times id_m); (id_n \times fan_m)$$

The property discussed in Theorem 3.2.1 allows us to say that the parallel composition of two fans is in fact embarrassingly parallel [4]. In other words, the two fans have disjoint inputs and outputs.

Using the laws defined in this section, one can argue correctness and equivalence in circuits. Hinze does this for parallel prefix computation, by
3.2. HINZE CIRCUIT ALGEBRA

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id_n</td>
<td>= n$</td>
</tr>
<tr>
<td>$</td>
<td>fan_n</td>
<td>= n$</td>
</tr>
<tr>
<td>$</td>
<td>f; g</td>
<td>=</td>
</tr>
<tr>
<td>$</td>
<td>f \times g</td>
<td>=</td>
</tr>
<tr>
<td>$</td>
<td>\bar f</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>\bar f</td>
<td>$</td>
</tr>
<tr>
<td>$(f; g) \prec \bar x = (f \prec \bar x); (g \prec \bar x)$</td>
<td>Distributive Law over Sequential Composition</td>
<td></td>
</tr>
<tr>
<td>$(f \times g) \prec \bar x = (f \prec \bar x_1) \times (g \prec \bar x_2)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar x \succ (f \times g) = (\bar x_1 \succ f) \times (\bar x_2 \succ g)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>f \times \bar x</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>f \times \bar x</td>
<td>$</td>
</tr>
<tr>
<td>$id \prec x = id$</td>
<td>Magnitude Law of Stretch</td>
<td></td>
</tr>
<tr>
<td>$id \prec x = id$</td>
<td>Magnitude Law of Stretch</td>
<td></td>
</tr>
<tr>
<td>$(f; g) \prec \bar x = (f \prec \bar x); (g \prec \bar x)$</td>
<td>Distributive Law over Sequential Composition</td>
<td></td>
</tr>
<tr>
<td>$(f \times g) \prec \bar x = (f \prec \bar x_1) \times (g \prec \bar x_2)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar x \succ (f \times g) = (\bar x_1 \succ f) \times (\bar x_2 \succ g)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>f \times \bar x</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>f \times \bar x</td>
<td>$</td>
</tr>
<tr>
<td>$id \prec x = id$</td>
<td>Magnitude Law of Stretch</td>
<td></td>
</tr>
<tr>
<td>$id \prec x = id$</td>
<td>Magnitude Law of Stretch</td>
<td></td>
</tr>
<tr>
<td>$(f; g) \prec \bar x = (f \prec \bar x); (g \prec \bar x)$</td>
<td>Distributive Law over Sequential Composition</td>
<td></td>
</tr>
<tr>
<td>$(f \times g) \prec \bar x = (f \prec \bar x_1) \times (g \prec \bar x_2)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar x \succ (f \times g) = (\bar x_1 \succ f) \times (\bar x_2 \succ g)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>f \times \bar x</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>f \times \bar x</td>
<td>$</td>
</tr>
<tr>
<td>$id \prec x = id$</td>
<td>Magnitude Law of Stretch</td>
<td></td>
</tr>
<tr>
<td>$id \prec x = id$</td>
<td>Magnitude Law of Stretch</td>
<td></td>
</tr>
<tr>
<td>$(f; g) \prec \bar x = (f \prec \bar x); (g \prec \bar x)$</td>
<td>Distributive Law over Sequential Composition</td>
<td></td>
</tr>
<tr>
<td>$(f \times g) \prec \bar x = (f \prec \bar x_1) \times (g \prec \bar x_2)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar x \succ (f \times g) = (\bar x_1 \succ f) \times (\bar x_2 \succ g)$</td>
<td>Distributive Law over Parallel Composition where $\bar x = \bar x_1 \oplus \bar x_2$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
<tr>
<td>$\bar y \succ (\bar x \succ f) = \bar z \succ f$</td>
<td>Idempotency of $\succ$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Magnitude and stretch laws of the Hinze algebra

defining a naive circuit $scan_n$ that compute the prefix computation of $n$ wires:

$$
scan_0 = id_0
$$

$$
scan_n = id_1 \times scan_{n-1}; fan_n
$$

It is known that the circuit expression $scan_n$ always produces the correct solution [15]. For example, Figure 3.2.1 is $scan_4$, the solution to the parallel prefix on four wires. Using this fact and the laws from this section one only needs to show that a given circuit expression $c$ is equivalent to $scan_n$ to prove that $c$ correctly computes a parallel prefix.
3.2.2 A Haskell Realization

Hinze gives a compact Haskell realization of his algebra. For simplicity, we draw from a smaller Haskell core syntax than Hinze did. We first introduce how we represent the input to a circuit. Circuit input takes the form of a list of some type α, where α is the type of the semigroup elements. We define a data type, `Circuit`, to represent a circuit in the algebra.

```haskell
data Circuit α = Circ { width :: Integer, apply :: [α] → [α] }
```

This means a circuit is made up of a width that specifies the number of wires in the circuit, as well as a function from input to output that specifies the result of the circuits computation. Given a circuit c of type `Circuit α`, we write `width c` to extract the width component of c and `apply` to extract the function. The simplest function in the implementation is the function that returns the size of the circuit.

```
|·| :: (Circuit α) → Integer
|f| = width f
```

Note here that a loose syntax has been used to define the size operator. The definition of the identity on n wires is straightforward:

```
id :: Integer → (Circuit α) → (Circuit α)
id n = Circ n (λu → u)
```
3.2. HINZE CIRCUIT ALGEBRA

In a manner similar to \( id_n \), \( fan_n \) can be defined.

\[
fan :: \text{Integer} \to (\text{Circuit } \alpha) \to (\text{Circuit } \alpha)
\]

\[
fan n = \text{Circ } n (\lambda u \to \text{case } u \text{ of } \\
[ ] \to [ ] \\
a : as \to a : [a \oplus b \mid b \leftarrow as])
\]

In the definition of \( fan_n \) we make use of two new Haskell features, pattern matching and \textsf{case} expressions. The \textsf{case} construct allows us to test many patterns to see if we have a match. When a match is found, the code to the right of the arrow is executed. In the code for \( fan_n \), if we see an empty list then we return an empty list. On the other hand, if we see a list composed of the head of the list and the remainder of the list, then we create a new list made up of the head of the list followed by the head of the list composed with every element in the list.

We define four remaining constructs, in Haskell, parallel composition, sequential composition and the two forms of stretches. We define these using a loose syntax, much in the spirit of Hinze’s paper. We first define sequential composition:

\[
; :: (\text{Circuit } \alpha) \to (\text{Circuit } \alpha) \to (\text{Circuit } \alpha)
\]

\[
f;g = \text{Circ } [f] (\lambda u \to \text{apply } g (\text{apply } f u))
\]

The definition of sequential composition is natural. One first computes the result of the first subcircuit, \( f \), and then uses that result to compute the result of the second subcircuit \( g \).

We define parallel composition as previously stated. This causes the two subcircuits to execute at the same time. This is captured by the following Haskell code.

\[
\times :: (\text{Circuit } \alpha) \to (\text{Circuit } \alpha) \to (\text{Circuit } \alpha)
\]

\[
f \times g = \text{Circ } |f| + |g| (\lambda u \to \textbf{let } (x, y) = \textit{splitAt } |f| u \\
\textbf{in } \text{apply } f x \circ \text{apply } g y)
\]

There is a new function used in the definition of parallel composition. The function \textit{splitAt} is a Haskell function that takes an integer \( n \) and a list \( l \), and returns a pair of lists \((l_1, l_2)\) with \( l_1 \) containing the first \( n \) elements of \( l \) and \( l_2 \) containing the remaining elements.
One can also define the stretch operators using a similar process. We first define a function *group* which takes in a vector and a list and produces a list of lists.

\[
\text{group} :: [\text{Integer}] \rightarrow [\alpha] \rightarrow [[\alpha]]
\]

\[
\text{group} [ ] as = [ ]
\]

\[
\text{group} (i : x) as = \text{bs} : \text{group} x \text{ cs}
\]

where \((\text{bs}, \text{cs}) = \text{splitAt} \ i \ as\)

Note that the *where* clause allows a programmer to define a local variable, here the pair of lists \((\text{bs}, \text{cs})\). Using the definition of *group* we can define the \(<\) and \(>\) stretches as shown below.

\[
< :: (\text{Circuit} \ \alpha) \rightarrow [\text{Integer}] \rightarrow (\text{Circuit} \ \alpha)
\]

\[
f \ < x = \text{Circuit} \ \Sigma x \ (\lambda \ u \rightarrow \text{let} \ \text{ys} = \text{group} x \ u
  
  as = \text{apply} f [\text{head} \ y | y \leftarrow \text{ys}]
  
  \text{in} \ (\text{zipWith} \ (\lambda \ x \ y \rightarrow x + + y)
  
  [[a] | a \leftarrow \text{as} [\text{init} \ y | y \leftarrow \text{ys}]) \ )
\]

where \(\Sigma x\) denotes the sum of all the elements of list \(x\), and the function \(\text{zipWith}\) takes a function and two lists and merges the lists applying the function as the list is merged. We can define \(>\) similarly.

\[
> :: [\text{Integer}] \rightarrow (\text{Circuit} \ \alpha) \rightarrow (\text{Circuit} \ \alpha)
\]

\[
x \ > f = \text{Circuit} \ \Sigma x \ (\lambda \ u \rightarrow \text{let} \ \text{ys} = \text{group} x \ u
  
  as = \text{apply} f [\text{last} \ y | y \leftarrow \text{ys}]
  
  \text{in} \ (\text{zipWith} \ (\lambda \ x \ y \rightarrow x + + y)
  
  [\text{init} \ y | y \leftarrow \text{ys} [\text{as} | a \leftarrow \text{as}]) \ )
\]

### 3.3 Circuit Constructions

In Chapter 2, we introduced the Ladner-Fischer circuit. Hinze introduces a way to define this and other circuit constructions within the algebra he defines. These constructions serve as a main focus of our results in Chapter 4. In order to express these circuits Hinze introduces three additional operators to construct circuits: \(>\) (not to be confused with the stretch operator \(\triangleright\)), \(\triangleright\)
3.3. CIRCUIT CONSTRUCTIONS

and \( \text{par} \). These operators can be derived from the operators in Section 3.2. The \( \text{par} \) operator takes a list of circuits as an argument and returns a circuit that is the parallel composition of every circuit in the list. For example \( \text{par}[\text{fan}_2, \text{fan}_3, \text{fan}_2] \) is expanded to \( \text{fan}_2 \times \text{fan}_3 \times \text{fan}_2 \). The \( \triangleright \) operator takes as input a list of circuits \( fs \) and a circuit \( f \); then performs the following operation: \( \text{par } fs; [fs] \triangleright f \), where \( [fs] = \text{map } |:| fs \). Intuitively \( \triangleright \) takes the last output of \( f_i \in fs \) for all \( fs \) and connects it to the \( i \)-th input of \( f \). For example \([\text{fan}_2, \text{fan}_3] \triangleright \text{fan}_2\) is expanded to \( \text{fan}_2 \times \text{fan}_3; [2,3] \triangleright \text{fan}_2\). The operator \( \triangleright \) is given by the following Haskell recursive definition:

\[
\triangleright :: [(\text{Circuit } \alpha)] \rightarrow (\text{Circuit } \alpha) \rightarrow (\text{Circuit } \alpha)
\]

\[
[ | ] \triangleright g = g
\]

\[
(f : fs) \triangleright g = (f : fs) \triangleright g ; id_{j\downarrow -1} \times \text{par } gs
\]

\[\text{where } gs = [\text{fan}_{j\downarrow} | f \leftarrow fs ] ++ [id_1] \]

The use of piecewise definitions is a standard Haskell practice. Note again the use of pattern matching. Intuitively \( \triangleright \) takes a list of circuits followed by a circuit and produces a circuit like \( (f : fs) \triangleright g \) and then composes in sequence a parallel composition of fans, of size equivalent to the individual circuits in the list, shifted one left. For example consider \([\text{fan}_2, \text{fan}_3] \triangleright \text{fan}_2\) this yields the circuit \( \text{fan}_2 \times \text{fan}_3; [2,3] \triangleright \text{fan}_2; id_1 \times \text{fan}_3 \times id_1 \).

We can now define the Ladner-Fischer circuit. We first create a auxiliary function \( \text{double} \). It takes as input a function from a circuit size to a circuit and return a function with the same signature.

\[
\text{double} :: (\text{Integer } \rightarrow \text{Circuit } \alpha) \rightarrow (\text{Integer } \rightarrow \text{Circuit } \alpha)
\]

\[
\text{double } s \ n = (\text{replicate } \lceil n/2 \rceil \text{fan}_2 \triangleright id_1 \triangleright \text{odd } n) \triangleright s \lceil n/2 \rceil
\]

It should be noted that \( \text{replicate} \) takes a number \( n \) and a circuit \( c \) and produce a list containing \( n \) copies of \( c \). Furthermore, the syntax \( [id_1 | \text{odd } n] \) means create a list containing \( id_1 \) if \( n \) is odd otherwise just create an empty list.

Making use of the \( \text{double} \) function we can define the Ladner-Fischer circuit \( \text{lf} \) by the following recursive definition.

\[
\text{lf} \begin{array}{l}
k \ 0 = id_0 \\
k \ 1 = id_1 \\
k \ n = \text{lf } 1 \lceil n/2 \rceil \times \text{lf } 0 \lceil n/2 \rceil ; id_{1 \lceil n/2 \rceil -1} \times \text{fan}_{1 \lceil n/2 \rceil +1} \\
(k + 1) \ n = \text{double } (\text{lf } k) \ n
\end{array}
\]
In accordance with the description in Chapter 2, the definition of the Ladner-Fischer circuit allows for the specification of a depth parameter $k$ as well as the size of the prefix computation circuit $n$. This depth parameter allows us to create a depth-optimal circuit that computes solutions to the parallel prefix computation problem.

As mentioned in Chapter 2, there are additional circuit constructions such as the Brent-Kung circuit construction. This construction can also be expressed in the Hinze circuit algebra. The Brent-Kung circuit is expressed as:

$$b_kn = \begin{cases} 
id_n, & \text{if } n \leq 1 \\
\text{double } b_kn, & \text{otherwise.}
\end{cases}$$

This is similar to the Ladner-Fischer circuit construction except it does not allow us to bound the depth of the circuit. The Lin-Hsiao construction can also be constructed in the Hinze algebra. The construction will not be presented here, but the interested reader is directed to [15].
Chapter 4

A Fault-Tolerant Prefix Circuit Algebra

The previous chapter demonstrated that the Hinze circuit algebra is expressive. However, it was not designed to express circuits that can handle faults. This chapter introduces extensions to the Hinze circuit algebra that allows us to effectively reason about a specific class of faults. We must extend the basic Hinze algebra as defined in [15] to include a majority voting mechanism as defined in Chapter 2. Furthermore, we introduce a notion of replication to the Hinze Algebra.

4.1 A Normal Form

Before we delve into the extensions to the Hinze algebra, we define and argue for the existence of a normal form, which we call sequential normal form (SNF). This normal form simplifies our results, and will be used throughout this chapter. A circuit in SNF is a sequential composition of parallel compositions of basic terms, where a basic term is either a fan, an identity, or a stretch applied to a basic term. For example $\text{fan}_3 \times \text{fan}_2; \text{fan}_5$ is in SNF. One advantage of SNF is that parallel rounds can be determined by looking at the sequential compositions of parallel terms. A parallel round is defined to be all parallel compositions that occur between sequential compositions when a circuit is written in SNF. For example, the following circuit has two
rounds
\[ \text{fan}_2 \times \text{fan}_2 \times \text{fan}_2 \times \text{fan}_2 \times \text{fan}_2; \]
\[ \text{fan}_2 \times \text{fan}_2 \times \text{fan}_2 \times \text{fan}_2 \times \text{fan}_2. \]

In order for SNF to be a normal form it must be the case that every expression in the Hinze algebra has an SNF realization. This is the essence of the following result.

**Theorem 4.1.1.** Given any circuit \( c \), there exists a realization of \( c \) in Sequential Normal Form, denoted \( \text{SNF}(c) \).

*Proof.* (by induction on the structure of expression denoting circuits)

**Base Cases:** The two base cases, when \( c \) is a fan or an identity is trivial, since fans and identities are already in SNF

**Inductive step for \( ; \):** If circuit \( c_1 \) and \( c_2 \) can be written in SNF then \( c_1; c_2 \) can be written in SNF. Let \( c_1 = f_1; f_2; \ldots; f_k \) and let \( c_2 = g_1; g_2; \ldots; g_l \), where \( f_i \) and \( g_i \) are parallel compositions of basic terms.

\[
\begin{align*}
c_1; c_2 & = (f_1; f_2; \ldots; f_k) ; (g_1; g_2; \ldots; g_l) \\
& = f_1; f_2; \ldots; f_k; g_1; g_2; \ldots; g_l
\end{align*}
\]

And therefore this inductive case is true.

**Inductive step for \( \times \):** If circuit \( c_1 \) and \( c_2 \) can be written in SNF then \( c_1 \times c_2 \) can be written in SNF for all \( c \in C \).

\[
\begin{align*}
c_1 & = f_1; \ldots; f_k \\
c_2 & = g_1; \ldots; g_l
\end{align*}
\]

Without loss of generality assume \( k = l \) where the smaller \( c_i \) is padded with \( \text{id}_j \)'s.

\[
c_1 \times c_2 = (f_1; f_2; \ldots; f_k) \times (g_1; g_2; \ldots; g_l)
\]

From Hinze, we know that \( (f; f') \times (g; g') = (f \times g); (f' \times g') \) for all \( f, f', g \) and \( g' \) where \( |f| = |f'| \) and \( |g| = |g'| \). This gives us a basic distributive law which we apply in this case to arrive at:

\[
c_1 \times c_2 = (f_1 \times g_1); (f_2 \times g_2); \ldots; (f_k \times g_k)
\]

This resulting circuit is in SNF so the the induction is true.
Inductive step for \( \rightarrow \): If circuit \( c \) can be written in SNF then circuit \( \vec{x} \rightarrow c \) can be written in SNF. Since \( c \) is in SNF this means that,

\[
c = b_1^1 \times b_2^1 \times \cdots \times b_{k_1}^1; \cdots; b_1^r \times b_2^r \times \cdots \times b_{k_r}^r
\]

where \( r \) is the number of parallel rounds and each \( b_i^j \) is a basic term. Using the above decomposition we can assert

\[
\vec{x} \rightarrow c = \vec{x} \rightarrow (b_1^1 \times b_2^1 \times \cdots \times b_{k_1}^1; \cdots; b_1^r \times b_2^r \times \cdots \times b_{k_r}^r)
\]

\[
= \vec{x} \rightarrow (b_1^1 \times b_2^1 \times \cdots \times b_{k_1}^1); \cdots; \vec{x} \rightarrow (b_1^r \times b_2^r \times \cdots \times b_{k_r}^r)
\]

This is true via the distributive law of stretches over sequential composition in the Hinze algebra. We can further apply the distributive law of stretches over parallel composition to arrive at

\[
\vec{x}^1 \rightarrow b_1^1 \times \vec{x}^2 \rightarrow b_2^2 \times \cdots \times \vec{x}^k_{k_1} \rightarrow b_1^1 \times \vec{x}^2 \rightarrow b_2^2 \times \cdots \times x^r_{k_r} \rightarrow b^r_{k_r}
\]

where \( \vec{x} = x^1_1 + \cdots + x^1_{k_1} = \cdots = x^r_1 + \cdots + x^r_{k_r} \). By definition, each \( \vec{x}^j_i \rightarrow b^j_i \) is a basic term, and the expression is in SNF.

Inductive step for \( \leftarrow \): If circuit \( c \) can be written in SNF then circuit \( c \leftarrow \vec{x} \) can be written in SNF. Since \( c \) is in SNF this means that,

\[
c = b_1^1 \times b_2^1 \times \cdots \times b_{k_1}^1; \cdots; b_1^r \times b_2^r \times \cdots \times b_{k_r}^r
\]

where \( r \) is the number of parallel rounds and each \( b_i^j \) is a basic term. Using the above decomposition we can assert

\[
c \leftarrow \vec{x} = (b_1^1 \times b_2^1 \times \cdots \times b_{k_1}^1; \cdots; b_1^r \times b_2^r \times \cdots \times b_{k_r}^r) \leftarrow \vec{x}
\]

\[
= (b_1^1 \times b_2^1 \times \cdots \times b_{k_1}^1) \leftarrow \vec{x}; \cdots; (b_1^r \times b_2^r \times \cdots \times b_{k_r}^r) \leftarrow \vec{x}
\]

This is true via the distributive law of stretches over sequential composition in the Hinze algebra. We can further apply the distributive law of stretches over parallel composition to arrive at,

\[
b_1^1 \leftarrow x^1_1 \times b_2^1 \leftarrow x^1_2 \times \cdots \times b_{k_1}^1 \leftarrow x^1_{k_1}; \cdots; b_1^r \leftarrow x^r_1 \times b_2^r \leftarrow x^r_2 \times \cdots \times b_{k_r}^r \leftarrow x^r_{k_r}
\]

where \( \vec{x} = x^1_1 + \cdots + x^1_{k_1} = \cdots = x^r_1 + \cdots + x^r_{k_r} \). By definition, each \( b_i^j \leftarrow \vec{x}^j_i \) is a basic term, and the expression is in SNF.

Since all cases are true the theorem is true.
4.2 Fault Tolerance

In this section we present the model of fault tolerance which we use throughout the remainder of this thesis. We implement fault tolerance through the use of replication and majority functions. We also present these extensions as formal constructs for an augmented form of the Hinze algebra.

4.2.1 Replication

All fault-tolerant solutions require some form of replication or redundancy. We define replication in the following way. Given any circuit, $c$ in the Hinze Algebra, let the $n$-fold replication of that circuit, denoted $\hat{c}$, be defined as the parallel composition of $n$ copies of circuit $c$. Thus,

$$\hat{c} = c \times \cdots \times c \upharpoonright_n.$$ 

Let $R(\hat{c})$ denote the number of replicas of $c$ that appear in $\hat{c}$.

Replication must use equivalence among components of the replicated circuit. We enforce this over the wires of the replicated circuit by defining what wires should hold an equivalent value in a replicated circuit throughout the entire computation. To these ends we formally define the notion of a wire class. Let $\hat{c}$ be a replicated form of circuit $c$; the $i$-th wire class is the set $w(i) = \{\omega \mid k + i \mid c\}, 0 \leq k < R(\hat{c})$.

As an example consider $\hat{fan}_2 = fan_2 \times \hat{fan}_2$, pictured in Figure 4.1. Wires in the first wire class are represented using a dashed line, while wires in the second wire class are represented using a solid line.

When dealing with replicas we further assume that when we replicated a circuit $c$, the input to $c$ is also correctly replicated across each replica.

4.2.2 Total Majority Vote

In order to guarantee that a failed wire value in a replicated circuit $\hat{c}$ does not cause $\hat{c}$ to produce the wrong result, a new construction, namely voting, must be introduced. Voting was used by Lamport, Shostak and Pease in their paper on the Byzantine generals problem [29] for the same purpose. It was further demonstrated that majority voting among a set of agents always favors the correct agents as long as only a minority of the agents have failed. In the case that a majority of the agents have failed or the number of failed agents
is the same as the number of correct agents a majority vote produces an indeterminate result. We extend the Hinze algebra with such a construction.

Total voting on wire class $i$, denoted $vote_i$, is a fault free operation that takes the value on each wire in wire class $i$, performs a majority vote, and places the resulting value on each wire in wire class $i$. Like any other construct in the Hinze algebra, $vote_i$ obeys laws. These laws are expressed in Table 4.1. Other properties may emerge when the types of faults being considered is taken in to account. For example if faults only occur at the input to a circuit, majority voting is commutative over both sequential and parallel composition with basic elements. This is not true in general, however. It should be noted

\[
\begin{align*}
vote_i; vote_j &= vote_j; vote_i & \text{Commutativity} \\
vote_i; (vote_j; vote_k) &= (vote_i; vote_j); vote_k & \text{Associativity} \\
vote_i; vote_i &\equiv vote_i & \text{Idempotency} \\
vote_i \times (vote_j \times vote_k) &= (vote_i \times vote_j) \times vote_k & \text{Associativity}
\end{align*}
\]

Table 4.1: Voting laws

that if there are no faults in a replicated circuit $\hat{c}$ with voting, then $\hat{c}$ with voting is equivalent to $\hat{c}$ without voting.
4.2.3 Faults, Fault Patterns, and Fault Families

We now formally define what we mean by the terms fault and fault tolerance. In the context of this thesis a fault refers to the random change of a value on a wire in a circuit. In general a fault negatively influences the execution of a computation, causing the result to be incorrect. Additionally, a fault in a distributed or parallel system may cause unnecessary delays to computation making the computation appear to be starved.

Faults affect circuit computations. Given that a circuit can be thought of as computing a function from input to output, a fault free circuit can be thought of as deterministically computing that function. However, when faults are permitted, nondeterminism is introduced. We no longer can guarantee that the circuit computes the correct result. One fault, at any stage of the computation of the function can drastically change the final result. The goal of fault tolerance, therefore, can be thought of as making sure the circuit always produces the same function, even in the presence of faults.

In order to reason more precisely about specific faults we use fault patterns. A fault pattern \( f \) is an ordered sequence of tuples \((v_1, v_2, \ldots, v_n)\) such that the value position \( i \) in tuple \( j \) contains the wire in class \( i \) that has failed at round \( j \). We use \( e \) to denote the absence of a fault.

We define a fault family \( F \), as a set of fault patterns. We say that a circuit is fault tolerant with respect to fault family \( F \) if the circuit produces the correct result for all \( f \in F \).

There are two families of faults that will be investigated in Section 4.3: \( F_1 \) and \( F_r \). A more general fault family, \( F_v \), will be investigated in Section 4.4.

- \( F_1 \) is the family of faults such that only one wire per wire class can fail during execution.
- \( F_r \) is the family where only one wire can fail per wire class per parallel round.
- \( F_v \) is the family of faults where only one wire can fail per wire class per a specified number of parallel rounds.

These fault families have various applications. The fault family \( F_1 \) is useful for chip construction if one considers the faults to only occur at the input of the circuit. The fault family \( F_r \) has applications in both software and hardware as it allows a tolerance to a large amount of faults for an optimal
amount of replication. Finally, the fault family $F_v$ provides a way to manage trade-offs of hardware costs and fault patterns. This allows designers to take advantage of design-specific fault patterns in order to optimize fabrication costs. These design-specific fault patterns could be based on statistical models of failures or other such methods.

4.3 Achieving Fault Tolerance

In this section we derive tight bounds on the number of replicas needed for a replicated circuit to be fault tolerant with respect to some fault family. We also discuss trade-offs between voting and replication.

4.3.1 Fault Family $F_1$

The fault family $F_1$ allows for one fault to occur per wire class in the replicated circuit. Suppose that we vote at the end of the circuit. In other words, we extend a circuit $c$ to be fault tolerant with respect to $F_1$ using the expression

$$\hat{c};\text{vote}_1,\ldots,\text{vote}_n,$$

for some number of replicas of $c$. How many replicas do we need for a replicated circuit that computes the parallel prefix to be fault tolerant with respect to $F_1$?

**Theorem 4.3.1.** $2n + 1$ replicas are sufficient for a circuit, $\hat{c}$, that computes the parallel prefix to be fault tolerant with respect to $F_1$ where $n = |c|$ and voting occurs only at the end of the circuit.

**Proof.** Assume $R(\hat{c}) = 2n + 1$. We know that a wire can fail only once. Conservatively, suppose that we no longer trust a replica if at least one fault occurs in it. So, we have $n$ faults and therefore $n$ untrusted replicas. This leaves $n + 1$ trusted replicas. Since, $n + 1 > n$ we know that a majority vote results in the correct answer. Therefore, $2n + 1$ replicas allow $\hat{c}$ to be fault tolerant under the conditions specified by the theorem.  

With a little more work we can prove that this bound is tight.

**Theorem 4.3.2.** $2n + 1$ replicas are necessary for a circuit $\hat{c}$ that computes the parallel prefix to be fault tolerant with respect to $F_1$ where $n = |c|$ and voting occurs only at the end of the circuit.
Proof. Assume, by way of contradiction, that \( \hat{c} \) can be constructed such that it is fault tolerant with respect to \( F_1 \), with voting at the end, using only \( 2n \) replicas.

Consider the fault subfamily \( F' \subseteq F_1 \) such that only one fault occurs per replica. \( F' \) represents the worst case faults that can occur in family \( F_1 \). This means the \( n \)-th wire class has \( n \) incorrect values as input to its majority function. The remaining \( n \) replicas is not enough to sustain the incorrect values in the \( n \)-th wire class. Therefore, the vote is indeterminate and we have reached a contradiction. The number of replicas must, therefore, be greater than \( 2n \).

The most straightforward circuit that performs a fault tolerant parallel prefix computation with respect to \( F_1 \), is \( \text{scan}_n \). However, we know from the rules of the Hinze algebra that taking any circuit \( c \) equivalent to \( \text{scan}_n \) will enable us to create a fault tolerant circuit that performs the parallel prefix computation.

### 4.3.2 Fault Family \( F_r \)

In this section we investigate fault tolerance of circuits with respect to fault family \( F_r \), which allow us to a greater number of faults. Furthermore, this allows us to reason about faults that occur between successive components in the circuit. We constrain our discussion to voting that occurs after every parallel rounds.

Can we do better than what is specified by Theorem 4.3.1? We make the following observation. Given any two fans \( \text{fan}_n \) and \( \text{fan}_m \), because of the nature of parallel composition, operations in \( \text{fan}_n \) do not affect actions in \( \text{fan}_m \) and vice versa. In other words, \( \text{fan}_n \) and \( \text{fan}_m \) are embarrassingly parallel [4]. This was shown to be the case in Theorem 3.2.1.

Using the above fact we can deduce the number of replicas we need in order for an arbitrary circuit \( c \) to be fault tolerant with respect to \( F_r \). We first argue in terms of a single fan that Theorem 4.3.1 holds. In this case \( F_r \) is equivalent to \( F_1 \). This is true since there is only one parallel round and therefore voting occurs at the end of the circuit. We know that exactly \( n \) faults can occur. We can conservatively assume that if a wire in a replica fails that replica is no longer trusted. This gives us \( n + 1 \) replicas guaranteed to be correct. Therefore a majority vote favors the correct result. We can use less replicas. We only need to make sure that there are enough replicas...
such that every majority vote yields a correct value. The number of replicas required is demonstrated in the results that follow. These results will only concern themselves with fans as identities can be constructed by the parallel compositions of \( f_{an_1} \).

**Theorem 4.3.3.** The number of replicas \( k(n) \) needed in order for \( f_{an_n} \) to be fault tolerant with respect to \( F_r \) is given by:

\[
    k(n) = \begin{cases} 
    3, & \text{if } n = 1 \\
    5, & \text{otherwise.} 
    \end{cases}
\]

**Proof.** Assume the worst case failure pattern where there is one fault per wire class.

Case \( n = 1 \). Since, \( n = 1 \) there is only one failure affecting only one replica at most. Therefore, it is the case that the other two replicas are correct causing a majority vote to produce the correct result.

Case \( n > 1 \). \( k(n) \) provides 5 replicas in this case. We know that in the worst case each wire class, except the first, sustains two incorrect values. Since, we have 5 replicas we know that we always have at least 3 correct values, per wire class, when a vote occurs. Therefore a correct result is obtained.

It can be further argued that \( k(n) \) is a tight lower bound.

**Theorem 4.3.4.** The minimum number of replicas needed in order for \( f_{an_n} \) to be fault tolerant with respect to \( F_r \) is \( k(n) \).

**Proof.** (by contradiction) Case \( n = 1 \). Assume, by way of contradiction, that we only have two replicas; it is trivial to see that a vote between two different values is indeterminate. Therefore, this case needs three replicas.

Case \( n > 1 \). Assume, by way of contradiction, that we only have four replicas. In the worst case two wires per wire class hold an incorrect value. This means that there exists a wire class where the number of correct values is the same as the number of incorrect values and therefore the majority vote produces an indeterminate result. Therefore, this case needs five replicas.

We now turn our attention to the case of stretched fans, where we define a stretched fan as any sequence of stretches applied to a fan.

**Theorem 4.3.5.** The number of replicas needed in order for a stretched fan to be fault tolerant with respect to \( F_r \) is given by \( k(n) \).
Proof. We know that a single stretch of a fan introduces additional wires. These wires are not in anyway connected to any other wires. This means in the worst case the wire classes introduced by the stretch will only have one incorrect value when performing the majority vote. This means that for the entire stretched fan, in the worst case there will be at least one wire class that has two incorrect values when performing the majority vote. We therefore have the conditions outlined in Theorem 4.3.4 giving us a tight bound \( k(n) \) on the number of replicas.

This argument generalizes to multiple stretches of a fan, by noting that a series of stretches adds wire classes that are not connected in anyway to other wire classes. Therefore these wire classes will only experience one incorrect value when performing a majority vote. Thus the tight bound, \( k(n) \), on the number of replicas still holds.

Theorems 4.3.3, 4.3.4 and 4.3.5, together with our observation about parallel composition, yield a strong result about the number of replicas needed in order to be fault tolerant with respect to \( F_r \) in an arbitrary circuit \( c \).

**Theorem 4.3.6.** The number of replicas needed in order for an arbitrary circuit \( c \) to be fault tolerant with respect to \( F_r \) is given by:

\[
|\hat{c}| = \max_{i, f_i \in c} \{k(|f_i|)\}
\]

where \( \{f_1, \ldots, f_j\} \) are all the basic terms that appear in \( c \).

Proof. We proceed by induction on the number of parallel rounds in \( c \). Assume our circuit is given in SNF.

**Base Case:** Assume we have just one parallel round. We know that we just have parallel compositions of basic terms. Furthermore, we know that that fans in parallel do not affect one another. Therefore, we only need to have enough replicas in order to sustain faults in the largest fan or stretched fan. Assume for a moment that \( f_m \) is the largest fan or stretched fan. Then by corollary 4.3.5 we know the number of replicas we need is given by \( k(|f_m|) \). Therefore the circuit is fault tolerant with respect to \( F_r \) and the base case holds.

**Inductive Case:** Assume we have \( n + 1 \) parallel rounds. We know that the previous \( n \) round guarantees that round \( n + 1 \) has good input. This is true by the inductive hypothesis. Since stretched fans in parallel do not affect each other we can apply Theorem 4.3.5 to the largest fan in the circuit in order to obtain the number of replicas.
4.4. ARBITRARY VOTING AND FAULT FAMILY $F_v$

Theorem 4.3.5 gives us a direct way to defining a fault-tolerant parallel prefix circuit with respect to fault family $F_r$. As a special case, we can define a type of scan $scan^k_m$ for fault family $F_r$. This scan on $m$ wires produces $k$ replicas with voting after each parallel round. Function $scan^k_m$ is defined as

$$scan^k_m = scan^k_{m,m}$$

where $scan^k_{n,m}$ is the following auxiliary function defined inductively on $n$.

$$\begin{align*}
scan^k_{0,m} &= par(replicate \ k \ id_0) \\
scan^k_{n,m} &= scan^k_{n-1,m} ; par(replicate \ k \ (id_{m-n} \times fan_n)) ; vote_1 ; \ldots ; vote_m
\end{align*}$$

To create a circuit on three wires fault tolerant with respect to $F_r$ one would use $scan^5_3$ shown in Figure 4.2, where a voting block is the circuit $vote_1 ; \ldots ; vote_m$.

![Voting Block](image)

Figure 4.2: Circuit $scan^5_3$

4.4 Arbitrary Voting and Fault Family $F_v$

In Section 4.3.2, we were able to derive a formula that gives the number of replicas needed for a circuit to be fault tolerant with respect to $F_r$. We now discuss fault family $F_v$. This is a weaker form of fault patterns where faults need not occur at every round. This lets us be more lenient as to how
much voting we need. This is important for reasons of resource constraint. Allowing the freedom to place voting enables a trade off between the fault that can be sustained and the amount of voting required.

\( \mathbb{F}_v \) is the fault family where we allow one fault per wire class per fault block. Where a fault block is the region of the circuit that occurs between two total votes or the input to the circuit and a total vote. This fault family is equivalent to \( \mathbb{F}_1 \) when there is only one fault block. \( \mathbb{F}_v \) is also equivalent to \( \mathbb{F}_r \) when there is one fault block per parallel round. Thus, \( \mathbb{F}_v \) generalizes both \( \mathbb{F}_1 \) and \( \mathbb{F}_r \).

The \( \mathbb{F}_v \) family allows us the greatest flexibility between frequency of voting and amount of replication. Moreover, it allows us to establish our strongest result which gives us a way to quantify the trade-off between voting and replication. Before we can proceed with introducing our result we must define a notion of a path in our circuit. A path in a circuit is any way to get from wire \( \alpha \) to wire \( \beta \) following a fan or stretched fan.

**Theorem 4.4.1.** Given an arbitrary circuit \( c \), let \( r^i_j \) be the number of paths to wire \( i \) in fault block \( j \). The smallest number of replicas \( k \) needed in order for the circuit to be fault tolerant with respect to fault pattern \( \mathbb{F}_v \) is given by

\[
k = \max_{1 \leq j \leq B} \left\{ \max_{1 \leq i \leq |c|} \{2r^i_j + 3\} \right\},
\]

where \( B \) is the total number of fault blocks in the replicated circuit.

**Proof.** \( k \) can be written as:

\[
k = 2 \left( \max_{1 \leq j \leq B} \left\{ \max_{1 \leq i \leq |c|} \{r^i_j\} \right\} \right) + 3.
\]

Let \( m \) be

\[
m = \max_{1 \leq j \leq B} \left\{ \max_{1 \leq i \leq |c|} \{r^i_j\} \right\}
\]

So, \( k = 2m + 3 \), this mean there exists a wire \( \omega \) in circuit \( c \) that has \( m \) paths to it in some fault block. This means that in the worst case the wire class that contains \( \omega \) has \( m + 1 \) incorrect values, \( m \) for incorrect received values and one for the failing of \( \omega \). So, we have \( 2m + 3 - (m + 1) = m + 2 \) correct values in the wire class \( \omega \) belongs to. We know that \( m + 2 > m + 1 \) and therefore results in a correct majority vote.
4.4. ARBITRARY VOTING AND FAULT FAMILY $F_v$

This bound is also tight. Assume, by way of contradiction, that we have $2m + 2$ replicas. We have already established that there are $m + 1$ incorrect values in the wire class that contains $\omega$. So, we have $2(m+1)-(m+1)=m+1$ correct values. Therefore, the number of correct values equals the number of incorrect values and the majority vote is inconclusive. Thus, we need at least $2m + 3$ replicas and the theorem is true.

We can now use Theorem 4.4.1 to develop circuits that strike an appropriate balance between the amount of voting we perform and the number of replicas we need in order to remain fault tolerant with respect to $F_v$. The most basic relationship that can be derived from Theorem 4.4.1 is that there exists an inverse relationship between the number of fault blocks and the number of replicas. An increase in the amount of voting performed creates a direct decrease in the number of replicas required.

Theorem 4.4.1 can be used to derive the previous results about fault families $F_1$ and $F_r$. In the case of Section 4.3.1 we had the tight bound of $2n + 1$ due to Theorem 4.3.2, where $n = |c|$. We can recast this result using Theorem 4.4.1. We observe the fact that wire $n$ in a circuit $c$ that computes the parallel prefix has exactly $n - 1$ paths. We have to handle this as one fault block due to the model. This gives us $k = 2(n - 1) + 3 = 2n + 1$, which is exactly what was specified in Theorem 4.3.2.

For Theorem 4.3.6, it can also be demonstrated that Theorem 4.4.1 can be used to prove the same result. We know that there is a fault block for every parallel round. Furthermore, by the nature of a parallel round there is only parallel composition of fans, stretched fans, identities or stretched identities in each fault block. This means that the largest possible number of paths to any wire is one. So, $2 \cdot 1 + 3 = 5 = k(n)$ when ever $n > 1$.

Sections 4.3.1 and 4.3.2 present the two extremes in terms of the number of fault blocks. This section unified these previous sections allowing us to assert a fundamental theorem of fault tolerance with in this algebra. Much work can be done to make use of this theorem. Some of this work is outlined in Chapter 5. Through this chapter we have demonstrated a trade off between the number of replicas needed to be fault tolerant and how often we vote. We have ascertained that we can do no better than the need for five replicas. However, this comes at a cost of voting per parallel round. We also determined that we can limit voting to occurring only once at a cost of replicas linear in the number of wires in the circuit. Finally, we unified both the fault families and a function to bound the number of required replicas.
This strong result gives the circuit designer the ability to effectively reason about a scan generator, such as the one defined by Hinze [15].

4.5 Circuit Constructions

In this section we discuss fault tolerant constructions for the Ladner-Fischer and Brent-Kung circuit constructions presented in Chapter 3. Both constructions are discussed with respect to fault families \( \mathbb{F}_1 \) and \( \mathbb{F}_r \).

We know from Section 3.3 that the Ladner-Fischer circuit construction can be made depth optimal. Being a depth optimal construction means that the Ladner-Fischer circuit represents the lowest number of fault blocks and thus the least amount of voting needed out of all circuits fault tolerant with respect to \( \mathbb{F}_r \). Consider, for example the circuit \( \text{lf} 0 4 \). This circuit is described by the equation \( \text{fan}_2 \times \text{fan}_2 ; \text{id}_1 \times \text{fan}_2 \) and depicted in Figure 4.3. Figure 4.4 is the Hinze scan as discussed in Subsection 4.3.1.

![Figure 4.3: Circuit \( \text{lf} 0 4 \)](image)

![Figure 4.4: Circuit \( \text{scan}_4 \)](image)

Comparing the Ladner-Fischer construction to Hinze’s \( \text{scan}_n \) construction, one can observe the fact that the Ladner-Fischer needs the same number of replicas as \( \text{scan}_n \) to be fault tolerant with respect to \( \mathbb{F}_1 \). This is true in the general case as well as the example depicted in Figure 4.3 and Figure 4.4. However, the Ladner-Fischer circuit performs better in certain instantiations of \( \mathbb{F}_v \). For example in the case of voting after every two parallel rounds the Ladner-Fischer circuit requires less replicas than \( \text{scan}_6 \). These notes point
4.6. MODIFIED REPLICATION

out that the Ladner-Fischer circuit allows for trade-offs between voting and replication.

Turning our attention to the Brent-Kung circuit we can observe similar results. We observe that the depth of the Brent-Kung circuit is also minimal. However, with this construction, in general, we can note a decrease in parallelization. An example of the Brent-Kung construction $bk_4$ in Figure 4.5. Comparing $bk_4$, $lf \ 0 \ 4$, and $scan_4$ we notice that the number of replicas needed for all circuit constructions to be fault tolerant with respect to $F_1$ is the same. This is also the true in the case of the $F_v$ fault family. Notice, that in the case that has been presented the fault family $F_v$ is the same for all placements of voting. This is not the case if the depth parameter of the Ladner-Fischer circuit is modified.

4.6 Modified Replication

In this section we discuss the results of Sections 4.3.1, 4.3.2, and 4.4 in terms of a different notion of replication. We consider a replication operation defined in the following way. The $n$-fold replication $\tilde{c}$ of circuit $c$ is the parallel composition of circuits $c_1, c_2, \ldots, c_n$ where $c_i = c$ in the Hinze algebra for all $i$. In other words, we do not require that exactly the same circuit be replicated, only that equivalent circuits are used. We refer to this type of replication
as heterogeneous replication. This type of replication may be used to add diversity to the circuit design [8, 11, 28]. How does this affect our results?

In the case of the fault family $F_1$ we have the same argument as in the proof of Theorem 4.3.2. This is true because every circuit of size $n$ has the property that a failure on wire $i$ affects all wires from $i$ to $n$. Our heterogeneous replicated circuit is no different.

In the case of the fault family $F_r$ we have the same argument as in the proof of Theorem 4.3.6. This is because every circuit is a parallel composition of fans, stretched fans, identities or stretched identities at every round. We therefore can apply Theorem 4.3.6 to every circuit in the heterogeneous replicated circuit. This allows us to arrive at five replicas regardless of the circuit considered.

The interesting case comes when we consider Theorem 4.4.1. This theorem can not be directly applied to a heterogeneous circuit construction. One would be tempted to try to apply the theorem by applying Theorem 4.4.1 to the every fault block of every replica, then taking the maximum value for the number of replicas to use. This provides enough replicas, but it is not a tight bound. For example consider the case of parallel prefix computation on six wires using both the Ladner-Fischer construction and the $scan_6$ construction. Further assume that voting after every third parallel round. For reference purposes $scan_6$ and the Ladner-Fischer circuit on six wires can be seen in Figures 4.6 and 4.7 respectively. By our discussion above we would

![Figure 4.6: Circuit scan6](image1.png)  
![Figure 4.7: Circuit If 0 6](image2.png)

seem to need thirteen replicas. However, only eleven replicas are needed.
4.6. MODIFIED REPLICATION

Consider the following: Assume, the replicated circuit uses a Ladner-Fischer circuit on six wires for one replica and $\text{scan}_6$ for the remaining ten replicas. Further assume that we vote every third parallel round. Using the aforementioned process we need thirteen replicas. We assert that this is more replicas than what is needed.

Consider the worst case fault pattern in the first fault block which, in this case, is the fault block that dictates the number of replicas needed. Fail wire one in the the Ladner-Fischer circuit. This causes an invalid value in the wire classes one through six. Now, in order to create a worst case we enforce the rule that only one fault occurs per replica. The remaining replicas are all $\text{scan}_6$'s. Furthermore we only have to look at the first parallel round. When a failure occurs in wire two it causes an incorrect value in wire class two through six. In general, in the $\text{scan}_n$ circuit we can say that a failure in wire $i$ introduces a bad value in wire classes $i$ to $n$. From this fault pattern we can know that we have one incorrect value in wire class one, two incorrect values in wire class two, two incorrect values in wire class three, three incorrect values in wire class four, four incorrect values in wire class five and five incorrect values in wire class six. In order to have a successful majority vote we need only one more correct replica than the maximum number of incorrect values. Therefore, eleven replicas are sufficient and the natural intuition is incorrect.

It is not clear what function bounds the number of replicas needed in the case of heterogeneous replication in the context of the fault family $\mathcal{F}_v$. It may in fact be the case that no such formula exists. Results regarding heterogeneous replication and the fault family $\mathcal{F}_v$ should be further developed. It seems to be the case that there is a middle ground between number of replicas and the placement and frequency of voting when considering the number of replicas needed in a heterogeneous replicated circuit.

In this section we have argued the trade-offs of constructions for fault tolerant circuits based on three basic circuit constructions. The trade-offs we discussed centered around the amount of voting, the amount of replication, and the amount of parallel operations.
Chapter 5

Conclusion and Future Work

This chapter reviews the main results of this thesis. We conclude with suggestion of new avenues of research based on the work presented.

5.1 Summary of Results

This thesis has provided an initial investigation into an algebra that can be used to reason about fault tolerance in the Circuit model. Additionally we presented trade-offs between parallelization, amount of voting, and amount of replicas needed in order for a circuit to be fault tolerant. Fault tolerance was discussed in the context of three fault families: $F_1$, $F_r$, and $F_v$. In the context of these families a general tight bound on the number of replicas required for fault tolerance was obtained.

\[
F_1(\hat{c}) = 2|c| + 1
\]

\[
F_r(\hat{c}) = \max_{i, f_i \in C} \{k(|f_i|)\}, \text{where } f_i \text{ either a fan or a stretched fan and}
\]

\[
k(n) = \begin{cases} 
3, & \text{if } n = 1 \\
5, & \text{otherwise}
\end{cases}
\]

\[
F_v(\hat{c}) = \max_{1 \leq j \leq B} \left\{ \max_{1 \leq i \leq |c|} \left\{ 2r_i^j + 3 \right\} \right\}, \text{where } B \text{ is the total number of fault blocks in the replicated circuit and } r_i^j \text{ is the number of paths to wire } i \text{ in fault block } j
\]

Table 5.1: Necessary number of replicas per fault family.
We also reasoned about the Ladner-Fischer and Brent-Kung circuit constructions. In the context of these constructions this thesis demonstrates that these constructions present optimal fault tolerance with respect to the fault family $F_1$. These circuit tighten the gap slightly between the two extremes in voting patterns, namely $F_1$ and $F_r$. We went on to reason about heterogeneous replication where we demonstrated the previously discussed bounds on the number of replicas hold when considering fault families $F_1$ and $F_r$. Further analysis was centered around fault family $F_v$ where it was determined that previously presented bounds did not hold. Finally, we conclude with directions that can be explored using the algebra and results proposed by this thesis.

### 5.2 Directions and Auxiliary Results

The work presented in this thesis can be used to explore additional directions. We present the more pertinent future directions in this section as well as some auxiliary tools and results.

#### 5.2.1 Mapping Distributed Algorithms to Circuits

One main avenue of exploration should be centered around mapping the Distributed Computing model on to the Circuit model. This rather straightforward mapping is known as a relay circuit.

A relay circuit associates a set of communication constraints with a circuit. This set of constraints is the edge set, $E$, when the distributed computer is viewed as the graph $G = (V, E)$. The first step in the construction is to pair every input wire with a node of the distributed computer. In order for two wires $i$ and $j$ in the circuit to be able to communicate directly there must be a path from wire $i$ to wire $j$ in the constraint set.

In order to reason about arbitrary relay circuits means we need to extend the Hinze algebra to handle relaying as well as the specification of a relay path. Extending the algebra to admit this notion of a circuit allows for an algebraic method for denoting about distributed algorithm that solve the prefix computation problem.
5.2.2 More General Fault Models

The fault model considered in this thesis is limited to faults that change the value on a wire. The fault models should be extended to consider the complete Byzantine fault model. Looking closer at just two sub-models we can say the following. The Fail-Stop fault model can be considered handled by the work in this thesis, if one treats a Fail-Stop fault as a failure to contribute a value to voting. The Fail-Stop Restart model, however, is not captured by our work.

Extending the reasoning power of the algebra to handle general Byzantine faults would most likely cause an increase in the number of replicas needed to sustain a fault. Though it can be safely assumed that this is quantified by a polynomial in the number of paths.

5.2.3 Improving the Algebra

For the purpose of this thesis, we have assumed that the replicas have their correct input values at the start of the computation. However, we can formalize this by introducing a construct called $\text{bcast}_n$. The $\text{bcast}_n$ construct has the same appearance as a fan, except that the $n-1$ destination wires take on the value that was sent. The stretch operator can be applied to a $\text{bcast}_n$, much in the same way as a fan. The $\text{bcast}_n$ construct can be applied before the input is propagated through the replicated circuit, to map the $n$ values to the each wire in a given wire class.

Fans and stretched fans allow us to express a lot of interesting circuits. However, they fail to allow us the flexibility of creating a fan whose origin is to the right of its destination wires. This can be corrected by creating two new constructs to replace $\text{fan}_n$. These constructs are $\text{rfan}_n$ which is equivalent to $\text{fan}_n$ and $\text{lfan}_n$ which is a fan with it’s origin to the right of it’s destination wires. The stretch operators continue to work as before. These two constructs allow for a greater control over the fan construct.

One can also introduce another construct $\text{cmp}$ that takes an compares two values and puts the larger value on the right wire and the smaller value on the left wire. We would allow for the application of stretch operators to the $\text{cmp}$ construct much in the same way as one may do in the case of fans.

This construct would allow for specification of comparator networks in terms of algebra. Moreover this would allow us to express fault tolerant solutions. An interesting avenue to explore with this extension is to determine
if the bounds on the number of replicas needed also holds in the case of comparator networks.

One major addition to the algebra needed is the ability to derive equivalence relations between fault-tolerant circuits. Currently these laws do not exist and thus proofs of correctness through algebraic means do not exist. Although, this thesis has provided a means for reasoning inductively about algebraic expressions for circuit constructions. Constructing a theory of equivalence will make the algebra an even more powerful tool for reasoning about fault-tolerant circuits.
Bibliography


