Dynamic Diffracting Trees

by

Giovanni M. Della-Libera

S.B. Mathematics,
S.B. Computer Science,
Massachusetts Institute of Technology (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

July 1996

© Massachusetts Institute of Technology 1996. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
July 19, 1996

Certified by ....
Nir Shavit
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Certified by ....
Nancy Lynch
Cecil H. Green Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

OCT 29 1997

Dynamic Diffracting Trees

by

Giovanni M. Della-Libera

Submitted to the Department of Electrical Engineering and Computer Science
on July 19, 1996, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A shared counter is a concurrent object which provides the *fetch-and-increment* operation on a distributed system. Recently, *diffracting trees* have been introduced as shared counters which work well under high load. They efficiently divide high loads into lower loads that can quickly access lock-based counters that share the overall counting. Their throughputs have surpassed all other shared counters when heavily loaded. However, diffracting trees of differing depths are optimal for only a short load range. The ideal algorithm would scale from the simple queue-lock based counter to a collection of counters with a mechanism (such as a diffracting tree) to distribute the load.

In this thesis, we present the *dynamic diffracting tree*, an object similar to a diffracting tree, but which can expand and collapse to better handle current access patterns and the memory layout of the object's data structure, providing true scalability and locality. This tree then assumes each diffracting tree over its optimal range, from the trivial diffracting tree, a lock-based counters, to larger trees that have a collection of lock-based counters. This reactive design pushes consensus to the leaves of the tree, making agreement easier to achieve. It does so by taking advantage of cache-coherence to keep agreement at the higher contention areas of the tree.

Empirical evidence, collected on the Alewife cache-coherent multiprocessor and a distributed shared-memory simulator, shows that the dynamic diffracting tree provides throughput within a constant factor of optimal diffracting trees at all load levels. It also shows to be an effective competitor with load balancing algorithms in produce/consumer applications.

We believe that dynamic diffracting trees will provide fast and truly scalable implementations of many primitives on multiprocessor systems, including shared counters, $k-$exclusion barriers, pools, stacks, and priority queues.

Thesis Supervisor: Nir Shavit

Title: Assistant Professor of Computer Science and Engineering

Thesis Supervisor: Nancy Lynch

Title: Cecil H. Green Professor of Computer Science and Engineering

Acknowledgments

I have so few people to thank, and so much space to do it in. Oops! Strike that. Reverse!

I would like to thank Nir Shavit for suggesting this wonderful project and guiding me throughout its inception and development. We went through many different designs and changes, and I never thought we would find one that was efficient and easy to explain. I also would like to thank Nancy Lynch for her continued pursuit of correctness, which led me to discover the immense benefits that come with actually *thinking* carefully about my algorithm and designing invariants that prove its correctness. I never was so appreciative of induction as I am now. I also thank Nir and Nancy for responding with care and support to the various emails I mailed off in times of stress.

I thank all the friends that supported me in this time-consuming process, but several stand out brightly. Roberto De Prisco and Mandana Vaziri, both officemates over the course of the year, for working with me on problem sets and putting up with my incessant chatter. Carl Hibshman, a biology major and English specialist corrected my entire thesis, a rather long and difficult task. He not only fixed my many pluralization errors, but spotted some typos in my lemma proofs that would not have been discovered if he hadn't been actually trying to understand all of the proofs. Steven Steiner came over in my most troubling hour of proving distinctness and helped me organize my invariants, giving me the impetus necessary to finish the proof. Pratip Banerji and Abiyu Diro listened to my rantings and ravings about the difficulties in my proofs, always calming me down. Matthew Eckstein put me up for this last month as I had already moved all my worldly possessions to Seattle. Finally, Jeff Marshall actually put up with me for the entire second term as we set attendance records at the Galleria Food Court, putting up with all of my mindless musings about my work and inviting me over to relax after a long day at the proof shop.

Finally, I have to thank my family for all of their support over the last 5 years. After being here for 5 years, I had almost forgotten that statistically speaking, I shouldn't be here. But I am, and my family all flocked up to Boston to celebrate in my graduation. The love they have outpoured for me has kept me sane and hopefully my plan to transplant all of them to Seattle will eventually unfold. A special thanks to my mother and grandmothers, the loving women who have watched out for me ever since I came on this earth.

I wanted to put a quote about counting in my thesis. Some battle scene perhaps where a young lieutenant said "If only we knew how many forces they had." Or, picture the familiar marbles in a jug contest. A scene in which the winner remarked "It was just a matter of ratios." However, none really come to mind, so instead I will resort to my high school yearbook quote.

> "An Optimist is one who does math with a pen" - Source Unknown

This is cheesy, I admit. But, I wonder what the comparable quote is for counting things. "An Optimist is one who counts with a diffracting tree?" For some reason though, I think an Optimist would just guess.

Contents

1 Introduction **13**

1.1 Background 13

1.2 Goals 15

1.3 Dynamic Diffracting Trees 16

2 DDT Design **19**

2.1 Diffracting Trees 19

2.1.1 Balancers 19

2.1.2 Counters 22

2.2 Dynamic Diffracting Trees 22

2.2.1 Irregular Diffracting Trees 23

2.2.2 State and Versioning Information 24

2.2.3 Folding and UnFolding 27

2.3 Bookkeeping 34

2.4 Cache Sizing 35

2.5 Folding and Unfolding Policy 35

3 Experimental Results **37**

3.1 Experimental Environments 37

3.2 Index Distribution Benchmark 38

3.2.1 Diffracting Trees and Queue Locks 39

3.2.2 Alewife Results 40

3.2.3 DDT Results on Proteus 44

3.3 Large Contention Change Benchmark 49

3.3.1 Sudden Surge 49

3.3.2 Sudden Drop 50

3.4 Producer/Consumer Benchmarks 51

3.4.1 10-Queens 52

3.4.2 Sparse Producer/Consumer Actions 53

4 DDT Formal Model **55**

4.1 User I/O Automaton 57

4.2 Balancer I/O Automaton 57

4.3 Tree Structure 58

4.3.1 Restrictions on *Status* 61

4.4 Main I/O Automaton 61

5 Automaton Verification **65**

5.1 Distinctness 65

5.1.1 Multi-Sets 65

5.1.2 *Outputs* definition 66

5.1.3 Value Sets 66

5.1.4 *Status* Restrictions on Counters 69

5.1.5 *Count* Properties 70

5.1.6 *Count* Never Decreases 72

5.1.7 Counter_Limit *Count* properties 74
5.1.8 Final *Count* Properties 75
5.1.9 Conclusion 77
5.2 Output Value Limit 78
5.2.1 Mathematical Facts 78
5.2.2 Limits and Value Sets Revisited 79
5.2.3 *ID*, *Status*, and Balancer Properties 81
5.2.4 Active Processor Tracking 82
5.2.5 *ID* tracking 83
5.2.6 *PathID* tracking 84
5.2.7 Processor Travel Plans 87
5.2.8 Conservation of Energy 88
5.2.9 Conclusion 99
5.3 Safety Property 99

6 Implementation Verification **101**

6.1 Request 101
6.2 Increment_Count 102
6.3 Return 102
6.4 Balancer_Request 102
6.5 Balancer_Return 103
6.6 Fold 103
6.7 UnFold 103
6.8 Liveness 103

7 Summary and Future Work **105**

7.1 Continued Work 106

7.1.1 More Performance Results 106

7.1.2 Completion of Implementation Proof 106

7.1.3 Wait-Free Version 107

7.1.4 Timing Scheme 107

7.2 Future Work 107

7.2.1 Message Passing 107

7.2.2 Elimination Trees 108

7.2.3 Scaling Policies 108

7.2.4 Distributed Data Structures 108

List of Figures

2-1 A Balancer at work 20

2-2 Code for Balancing 21

2-3 A counting diffracting tree 22

2-4 An irregular diffracting tree's counting scheme 23

2-5 Definition of node structure 26

2-6 Key elements of a DDT node 26

2-7 Code for main traversal of DDT 27

2-8 Code for counting in DDT 28

2-9 Code for folding 29

2-10 Two cases of Folding, Parent node at Level 1 30

2-11 Two cases of Unfolding, Parent node at Level 1 32

2-12 Code for unfolding 33

3-1 Diffracting Trees of depth 3 with Queue or Spin Locks at Counters on Proteus . 40

3-2 Average Latency of Queue or Spin Lock Counters on Diffracting Tree of depth 3 on Proteus 41

3-3 Diffracting Trees, Queue-Lock Based Counter, and DDT on Alewife from 1-32 Processors 42

3-4 Throughputs of Optimal Composite vs. DDT on Alewife 42

3-5 Latencies of Optimal Composite vs. DDT on Alewife 43

3-6 Diffracting Trees, Queue-Lock Based Counter, and DDT on Proteus from 1-32 Processors 44

3-7 Throughputs of Optimal Composite on Proteus and normalized Alewife 45

3-8 Throughputs of Diffracting Trees and Queue Lock on Proteus 46

3-9 Latencies of Diffracting Trees and Queue Lock on Proteus 46

3-10 Throughputs of Optimal Composite vs. DDT under high contention on Proteus . 47

3-11 Latencies of Optimal Composite vs. DDT under high contention on Proteus . . . 47

3-12 Throughputs of Optimal Composite vs. DDT under lower contention on Proteus 48

3-13 Latencies of Optimal Composite vs. DDT under lower contention on Proteus . . 48

3-14 Average latency of DDT over time in response to sudden surge 50

3-15 Comparison of surges with dynamic and static prism sizing 51

3-16 Average latency of DDT over time in response to sudden drop 52

3-17 10-Queens Performance and Code 53

3-18 Producer/Consumer Performance and Code 54

4-1 Interaction between Well-Formed Users, the Shared Memory System, and Balancers 56

4-2 Balancer I/O Automaton 58

4-3 Main I/O Automaton 62

4-4 Folding and UnFolding *Change* Actions 63

List of Tables

5.1 Conservation of Energy definitions 91

5.2 Proof of Invariant on $request_p$ 92

Chapter 1

Introduction

Coordination problems on multiprocessor systems have received much attention recently. Shared counters in particular are an important area of study because the *fetch-and-increment* operation is a primitive that is being widely used in concurrent algorithm design. Since good hardware support is not readily available, there have been a variety of solutions presented for this problem in software.

1.1 Background

Simple solutions often involve protecting a critical section with either test-and-set locks with exponential backoff (spin-locks) by Agarwal, Anderson, and Graunke [2, 3, 10] or the queue-locks of Anderson or Mellor-Crummey and Scott [3, 15]. These algorithms are popular because they provide great latencies in low load situations, when requests are sparse and mostly sequential in nature. However, they can not hope to obtain good throughput under high loads due to the bottleneck inherent in mutual exclusion.

More sophisticated algorithms proposed have included the combining trees of Yew, Tzeng, and Lawrie [21] and Goodman, Vernon, and Woest [9], the counting networks of Aspnes, Herlihy, and Shavit [4], and the diffracting trees of Shavit and Zemach [20, 18]. These methods are distributed and lower the contention on individual memory locations, allowing for better performance at high loads.

A combining tree consists of a lock-based shared counter and a binary tree which "combines"

the requests that travel up the tree from the leaves. A single processor reaches the counter with several requests, which it can perform by incrementing the counter with the appropriate offset. This tree takes advantage of contention, so performs much better than a simple lock-based counter. The binary tree provides n processors with at best a time of $O(\log n)$ and a throughput of $n/2 \log n$ indices per time step. The pitfall with combining trees is that a single processor's delay or failure in traversing the tree delays those that combined with it indefinitely.

A Bitonic counting network [4] is a data structure isomorphic to Batcher's Bitonic sorting network [5], with a "local counter" at the end of each output wire. At a junction in the network, the first processor that arrives exits on the top wire, the next on the bottom wire, and it continually oscillates. This is the first kind of tree to break away from a single lock-based counter at its root, distributing the counting to several lock-based counters that count with fixed offsets. This allows requests to be independent, making it fault tolerant. These networks have width $w < n$ and depth $O(\log^2 w)$. At their best, the throughput is w and latency a high $O(\log^2 w)$. The biggest disadvantage with counting networks is their rigid network structure. It is unclear how to change the structure of the tree.

The key advantage of counting networks are the k lock-based counters. If the load is too high for a lock-based counter to be effective, divide-and-conquer would encourage the load to be divided into pieces that can rapidly access a lock-based counter. By doing so, the k lock-based counters can efficiently move their lower loads through the shared counter, so the focus shifts to the mechanism for dividing the load. Diffracting trees [20] provide the most effective tool for distributing the load. The trivial Diffracting tree is just a simple lock-based counter. Once the load is high enough that division would benefit performance, diffracting trees of various depths can be used to effectively divide the load. They are constructed from simple one-input two-output computing elements called *balancers* that are connected to one another by wires to form a balanced binary tree. These balancers evenly divide their requests amongst their children. This tree of balancers can then quickly distribute requests to its output wires, which can be connected up to k lock-based counters for a high-performance distributed shared counter. Diffracting trees of various depths provide optimal performance throughout the load range, and the trivial diffracting tree, a queue-lock, provides the best performance under low load. However, a diffracting tree of a certain depth has unwanted costs for lower loads due to its higher latencies, and eventually level out as they become overcapacitated.

The prior art seems to be firmly divided into two camps: the lock-based algorithms which work well in the low load cases and the distributed algorithms that do better under high load. The lock-based algorithms are championed by queue-locks and the distributed algorithms are currently led by Diffracting Trees. One set of experiments revealed that in a low load situation, the throughput over a fixed period of time for a queue-lock counter was 652 operations while the diffracting delivered 46 operations. With the same period of time but with a high load, the queue-lock counter went down to 595 operations, while the diffracting tree rose to 5010 operations. A factor of 10 difference separates each of these sets of numbers.

1.2 Goals

Diffracting trees, from the simple queue-lock based counters to large trees with a collection of counters, provide optimal performance at all load levels. Our goal is to make the diffracting tree structure dynamic, so it can react to the current load and assume the optimal size, guaranteeing the best possible performance.

B.H. Lim recently came up with a reactive scheme [11, 12] that switched between a test-test-and-set lock by Rudolph [17], a queue lock, and a combining tree. This performed well from the low to mid-load ranges, as the combining tree took over for the queue lock. His algorithm only applies to algorithms that have one centralized lock-based counter, which precludes diffracting trees, but gave valuable insights to reactive policy making.

We now focus on the two main insights necessary to create such an algorithm.

- Localize decision making
- Use cache-coherence to make global agreement inexpensive

Localized decision making spares processors from continually deciding on the overall structure of the shared counter, which is what Lim's algorithm requires. A major drawback with global decision making is that processors can get delayed while they wait for a change to occur. By making the changes in the shared counter local to only part of the counter, then the number of processors directly delayed drops significantly, and when other processors arrive in the changed part of the structure, the decision has been made and they can quickly adapt.

Cache-coherence makes localized decision making a reality. Keeping processors in agreement globally is usually an expensive requirement. If an algorithm adds global state which does not often change in high load situations, then this information can be cached, making constant reference to it an inexpensive proposition. If the load in an area of the tree is low, then changes can be made without high cost, which enables localized decision making.

1.3 Dynamic Diffracting Trees

Our algorithm, the Dynamic Diffracting Tree (or DDT), uses these two principles to make diffracting trees dynamic and reactive. A significant change in the load of the system will cause the DDT to expand or collapse into the optimal tree. These changes occur at the end of the tree, a local decision, but one that should be mirrored by the other ends of the tree in a genuine change of load. However, if the tree's memory layout is designed so that different ends of the trees exist in separate parts of memory, the tree may become irregularly formed to give optimal performance. State is added to the nodes of the tree to indicate what kind of node they are, and the caching of this state enables processors to pass through the tree without much delay.

We describe an implementation of the DDT concurrently with a thorough discussion of how the algorithm works. We discuss the scaling policies and the requirements that they must have, and describe the one that was implemented.

We implement the DDT on the MIT Alewife machine of Agarwal, Chaiken, Johnson, Krantz, Kubiatowicz, Kurihara, Lim, Maa, and Nussbaumet [1]. However, the largest Alewife machine only has 32 nodes, limiting the load range we could test with. We show that the Proteus Parallel Hardware Simulator of Brewer, Dellarocas, Colbrook and Weihl [6, 7], which we run up to 256 processes, simulates Alewife well, giving results that are comparable when normalized. We obtain results from various experiments, comparing the DDT to optimal diffracting trees of various depths, simple queue-locks, and implementing a job queue to compete with the load balancing scheme by Rudolph, Slivkin-Allalouf, and Upfal[16]. In the same experiment we ran earlier, the DDT under low load provides 243 operations and under high load provides 3932 operations. The results show that the DDT performs within a constant factor of optimal diffracting trees at all load levels, and future work shows good promise in lowering this factor. A particularly interesting result on the MIT Alewife machine shows a range where the DDT

outperforms all regular Diffracting Trees due to its ability to assume an irregularly-shaped tree structure, taking advantage of locality. We also show that the DDT performs effectively in Producer/Consumer applications.

We define safety and liveness properties that any shared counter should satisfy. We present a specification of the DDT using the I/O Automata of Lynch and Tuttle [13] and formally prove that it satisfies these properties. We sketch an argument which shows that the implementation presented meets the specification.

In summary, we believe that the DDT and its underlying concepts will prove to be an effective paradigm for the design of future data structures and algorithms for multi-scale computing.

This paper is organized as follows: Chapter 2 explains the design of the DDT, presenting the asynchronous shared-memory implementation, and discusses different scaling policies, Chapter 3 discusses the performance results on Alewife and Proteus, Chapter 4 gives the formal description of the DDT, Chapter 5 gives the proof of the specification, Chapter 6 presents the argument that the implementation meets the specification, and Chapter 7 concludes this paper and lists areas of further research.

Chapter 2

DDT Design

We begin by reviewing the basics of diffracting trees and describe in detail the changes necessary to make them dynamic. This includes an implementation of the DDT on an asynchronous, cache-coherent, distributed shared-memory system. Finally, the scaling policy issue is discussed.

2.1 Diffracting Trees

A Diffracting tree [20] consists of *balancers* that are connected to one another by wires in the form of a balanced binary tree, and local counters attached to the final output wires of the tree. A balancer's job is to continually split the number of requests on its input wires onto its two output wires. A local counter counts with an increment based on its depth into the tree, and although its implementation is not restricted, it is assumed that it is a lock-based counter.

2.1.1 Balancers

First, we give the requirements that a balancer must satisfy. We denote by x the number of input requests, or tokens, ever received on the balancer's input wire, and by y_i, $i \in \{0,1\}$ the number of tokens ever output on its ith output wire. Given any finite number of input tokens x, it is guaranteed that within a finite amount of time, the balancer will reach a *quiescent* state, that is, one in which the sets of input and output tokens are the same. In any quiescent state, $y_0 = \lceil x/2 \rceil$ and $y_1 = \lfloor x/2 \rfloor$. Figure 2-1 shows how a balancer could split up 9 distinct process requests (A-I), where the requests without spaces between them happen at the same time.

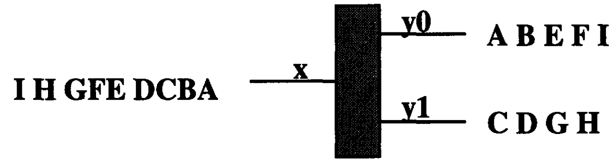


Figure 2-1: A Balancer at work

A simple implementation of a balancer would be a memory location with a lock that toggles between the values 0 and 1. A token entering the balancer obtains the lock, gets the value, stores the inverse value, and exits on the wire indexed by the value obtained, unlocking the bit. This clearly satisfies the properties above, but the lock reduces this problem to the same as that of a lock-based counter, and does nothing to reduce the bottleneck.

Shavit and Zemach present a much better implementation of a balancer [20], which exploits large numbers of requests to pair off processors onto the two output wires and avoid contention. When a processor enters a balancer, it enters that balancer's ID in its position in a global location array, selects a random location in that balancer's *prism*, and swaps the processor's own ID into that location. It attempts to pair off with the processor ID that it receives from the swap. If it fails, it then spins for a certain amount of time waiting for a processor to choose it. If all else fails, it attempts to access the test-and-test-and-set [17] lock described above. If it fails, then it starts all over, trying to reenter the prism [18]. This algorithm is really just an optimization since two processors flipping the bit would leave it unchanged. Figure 2-2 shows the code for the balancing code. This algorithm has been shown to satisfy the properties above [20].

The most important parameters for a diffracting tree are the sizes of the prisms and the spin constants mentioned above. Shavit and Zemach's steady-state analysis [18] found that a tree that would serve P processes should have a depth d and a number of prism locations L such that $\frac{P}{dL} = O(1)$, $L \leq d$, and $L = cd2^d$, where c is a machine-dependent constant. Given the approximate range of load that a diffracting tree would get, a developer can then choose d and subsequently decides upon L. A reactive backoff scheme was also implemented in [18] to provide the best spin constant.

```
type balancer is
    spin:   int
    size:   int
    prism:  array[1..size] of int
    toggle: int    /* 0 or 1 */
    lock:   Lock
    Left:   ptr to balancer  /* wire y0 */
    Right:  ptr to balancer /* wire y1 */
endtype

location: global array[1..NUMPROCS] of ptr to balancer

function diff-bal(b: ptr to balancer) returns ptr to balancer
begin
    location[MYID] = b
    forever
        rand_place = random(b->size)
        his_id = Swap(b->prism[rand_place], MYID)
        if CompareSwap(location[MYID], b, EMPTY) then
          if CompareSwap(location[his_id], b, EMPTY) then
            return b->Left
          else location[MYID] = b
        else return b->Right

        repeat b->spin times
            if location[MYID] != b then
                if b->spin < MAXSPIN then
                  b->spin = b->spin * 2
                return b->Right
        end repeat

        if TestTestSet(b->lock) then
          if CompareSwap(location[MYID], b, EMPTY) then
              k = b->toggle
              b->toggle = 1 - k
              release_lock(b->lock)
              if b->spin > MAXSPIN then
                b->spin = b->spin / 2
              return k
          else
              release_lock(b->lock)
              return b->Right
    endfor
end
```

Figure 2-2: Code for Balancing

2.1.2 Counters

The counters at the end of the tree have an increment equal to their depth into the tree. Their initial values are based on their position in the tree. The "Leftmost" counter, or counter at which a single entrant to the tree would arrive, has initial value 0. The counter with initial value 1 would be the "Leftmost" counter of the subtree rooted by the main root's Right child. It is clear how the ordering then proceeds. Figure 2-3 shows a diffracting tree of size 8 with its output wires ordered for counting.

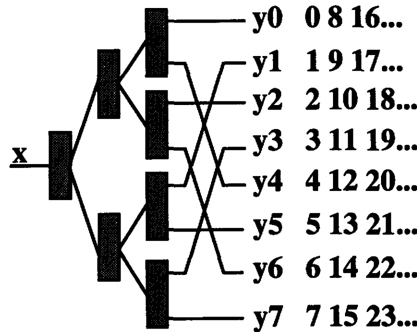


Figure 2-3: A counting diffracting tree

The algorithm for traversing the tree is clear. A token starts at a root node, and balances until it reaches a leaf counter, at which point it interacts with the counter, obtains a value, and exits.

2.2 Dynamic Diffracting Trees

We now start to move to a less rigid structure. Our goal is to allow Dynamic Diffracting Trees to control the following three parameters:

- Depth of Tree
- Prism Sizes
- Spin

By doing so, we gain full control of the parameters on page 2.1.1 that can be set to craft the optimal diffracting tree for a given load. We will employ localized decision making to allow

the tree to change height. As the number of processors P and subsequently the load changes, the tree would optimally expand or collapse to result in the best d and L possible, with the spin constant still determined by the reactive backoff scheme designed in [18].

We now individually discuss each loosening or addition to the original diffracting tree.

2.2.1 Irregular Diffracting Trees

In this kind of tree, we relax the restriction that the tree must be balanced. We only require that balancers have two children and counters are only at leaves. It should be clear that this restriction is not necessary for the algorithm to work correctly, and was instead placed under the assumption that a balanced tree gives the best performance. Abstractly, each balancer and its subtree represents an implementation of a counter, so it could be replaced by a lock-based counter and still function correctly. Figure 2-4 shows how one would set the counter's increment and initial values to make it work correctly.

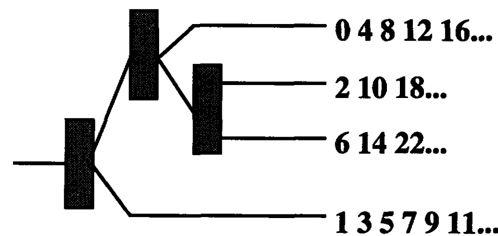


Figure 2-4: An irregular diffracting tree's counting scheme

The reason we must consider this is because it would be very expensive to change an entire diffracting tree to a different size. However, processors can work on different leaves in the tree, expanding or shrinking locally, causing the tree to be irregular at times. It is expected that the rate-making policy at one leaf would ask for the same kind of change as another leaf, due to the average contention levels that the balancers distribute over the entire tree, but if locality dictated that certain ends of the tree were slower than other ends of the tree because of the memory layout of the data structure, then it would be optimal to make the tree irregular to maximize performance on that given memory layout.

2.2.2 State and Versioning Information

Now, once we let the tree size change, we could potentially get trapped into some allocation/deallocation memory issues. We could run out of memory, allocation could take a very long time and bottleneck the processors, or a pointer to a deallocated node could remain around long enough to cause a problem if it was reallocated. Also, since we allow the trees to shrink and grow, it is no longer possible for every processor to initially memorize the structure of the tree, as it could in the diffracting or irregular diffracting tree. So, upon visiting a node of the tree, a processor would need to determine if that node was a counter or balancer.

We solve all of the above problems in the following way. We first add a state variable. This state variable initially only takes three variables, **Counter**, **Balancer**, or **Off**. The first two are clear in the context of merging the types of data structures together. A processor that visits a **Balancer** node balances, and one that visits a **Counter** node counts. However, if a processor visits an **Off** node, then it would know that the tree somehow changed beneath it, so it would trace up the node's ancestral path til it found a node it could successfully visit. We now solve the memory problems by creating a large, balanced tree then creating an initial configuration of balancers and counters from the root and leaving the rest of the tree **Off**. The designer can decide whether a real leaf of this tree can have an allocation call if it wishes to expand or if there is a static limit to the tree. Finally, we add a **Counter_Limit** state for the case where the tree shrinks, which will be explained soon.

This state variable could potentially be an expensive item to check. If the state variable in the root of the tree continually changed, all processors accessing it would be consistently delayed, bringing performance down. However, a sensible scaling policy would prevent the root node to change under high loads. In doing so, processors can utilize cache-coherence to keep accesses to the top level state variables relatively inexpensive. This brings into mind the second main property of the introduction, keeping global agreement inexpensive.

This solution provides distinctness and does not attempt to keep the different parts of the tree in balance. To provide a good balance, a versioning scheme must also be added. More precisely, the goal is to prevent a number from being handed out beyond the current number of requests, to keep the counter in line with more traditional lock-based counters. To better understand how this could be violated with the current design, consider this scenario: The tree

consists of a simple balancer at the root and two child counters. Clearly, one counter hands out the even numbers and one the odd numbers. Assume that the first number to be handed out is 0, and there are 10 requests made. 5 requests go along each of the two wires. After the 10 requests have been sent out, the tree decides to shrink, becoming a counter at the root and making the two children **Off**. Assume that the 5 requests along the even wire arrive, see the **Off** state, and return to the parent, obtaining the first 5 values, 0,1,2,3, and 4. Now, the tree decides to unfold again, and it initializes the two counters to next hand out 5 and 6. Now, the other 5 requests arrive at the odd counter. They receive 5, 7, 9, 11, and 13. Notice the imbalance amongst the 10 numbers handed out.

The key change needed to solve this is that once a balancer becomes a counter, all requests that have passed through that balancer and have not been satisfied have become delinquent. These delinquent processors need to come back to the node and access it again. The way to do this is to install a versioning scheme. When a balancer becomes a counter, the two children need to increase their version numbers, so that if a processor arrives at a node with a distinct version number[1] from that which its parent foretold, it would return to the parent and revisit, to get updated. Each processor caches the versioning numbers throughout its traversal of the tree, and if at any point it finds a differing version number, it traverses back up the tree until it finds agreeing version numbers, which in the worst case is the *Root* node whose version never changes. This versioning scheme can be folded into the state variable in an implementation, to reduce size and complexity, since versioning really is additional state. But, for simplicity, it is kept separate here. The definition of the new node structure is given in Figure 2-5.

[1]Technically, these version numbers are unbounded integers, but they are bounded by the values of the counters, so any implementation which handled the overflow of the counters could handle this as well.

```
typedef State oneof Balancer, Counter, Counter_Limit, or Off

type node is
  node_lock:    Lock
                       /* state and versioning section*/
  state:        State
  ID:           int    /* version of node */
  PID:          int    /* version of children */
                       /* balancer section */
  spin:         int
  size:         int
  prism:        array[1..size] of int
  toggle:       int
  toggle_lock:  Lock
                       /*counter section */
  level:        int
  count:        int
  init:         int
  change:       int
                       /* counter_limit section */
  limit:        int
                       /* binary tree section */
  Left:         ptr to node  /* wire y0 */
  Right:        ptr to node  /* wire y1 */
  Parent:       ptr to node
  Sibling:      ptr to node
endtype
```

Figure 2-5: Definition of node structure

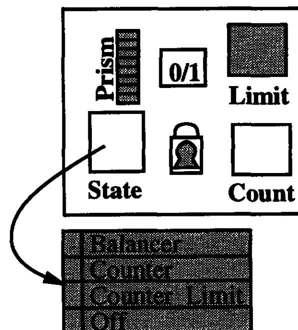


Figure 2-6: Key elements of a DDT node

Figure 2-7 contains the code for the main traversal of a processor through a dynamic diffracting tree. The Bookkeeping work is explained later in this chapter. Figure 2-8 contains the new code for accessing a Counter or Counter_Limit (which will be explained in the next section).

```
root: global ptr to node   /* main root of tree */
Bookkeeping: global array [1..NUMPROCS] of pair

function fetch_incr() returns int
    answer:   int
    IDRecord: array [enumeration of nodes] of int
    n:        ptr to node
begin
    IDRecord[root] = 0
    n = root
    answer = INVALID

    forever
        if (n->ID != IDRecord[n]) then
               n = n->Parent
               continue

        switch n->state
          case Balancer:
            Bookkeeping[MYID] = <n>
            if ((n->state != Balancer) || (n->ID != IDRecord[n]))
               n = n->Parent
               continue
            IDRecord[n->Left] = n->PID
            IDRecord[n->Right] = n->PID
            if n->state == Balancer then
              n = diff-bal(n)
          case Off:
            n = n->Parent;
          case Counter or Counter_Limit:
            answer = increment_counter(n);
            if valid(answer) then
              return answer
            else n = n->Parent
        endswitch
    endfor
end
```

Figure 2-7: Code for main traversal of DDT

2.2.3 Folding and UnFolding

We now describe how the changes in the tree work. The operation occurs locally at the bottom of the tree, *folding* two sibling counters into their parent balancer, becoming a new counter, or

```
function increment_counter(n:ptr to node) returns int
    answer:int
begin
    acquire_lock(n->node_lock)
    if (n->state == Counter or Counter_Limit) and
                      (n->ID == IDRecord[n]) then
        answer = n->count
        n->count = n->count + power(2,n->level)
        if n->count == n->limit then
            n->state = Off
            n->ID = n->ID + 1
        release_lock(n->node_lock)
        return answer
    else
        release_lock(n->node_lock)
        return INVALID
end
```

Figure 2-8: Code for counting in DDT

unfolding a counter into a balancer with two counter children. The algorithms we present here involve possessing 3 locks at one time, but this should be adaptable to having 1 lock at a time in the eventual goal of making this algorithm wait-free. We avoid this complication, however, because it adds extra state to the system and obfuscates the actions which occur. We describe each change below:

Folding

Figure 2-9 contains the code for folding. A processor, upon deciding that a balancer and its children counters need to be folded will attempt to obtain all 3 locks. If it is successful, then it tests whether the 3 nodes are a balancer with two child counters.

```
function attempt_fold(n:ptr to node) returns boolean
    nLeft, nRight, nMax, nMin: ptr to node
    valLimit: int
begin
    nLeft = n->Left
    nRight = n->Right

    acquire_lock(nLeft->toggle_lock)
    acquire_lock(nRight->toggle_lock)
    acquire_lock(n->toggle_lock)

    if (n->state == Balancer) and
       (nLeft->state == Counter) and (nRight->state == Counter) and
       ((nLeft->count != nLeft->change) or (nRight->count != nRight->change)) then

        n->state = Counter
        n->PID = n->PID + 1
        valLimit = MAX(nLeft->count,nRight->count) - power(2,n->Level)
        n->count = valLimit
        n->change = n->count

        Assign nMin, nMax to be nLeft, nRight,
          such that nMin->count < nMax->count

        nMax->state = Off
        nMax->ID = nMax->ID + 1

        if nMin->count < valLimit then
            nMin->state = Counter_Limit
            nMin->limit = valLimit
        else
            nMin->state = Off
            nMin->ID = nMin->ID + 1

        release_lock(nRight->node_lock)
        release_lock(nLeft->node_lock)
        release_lock(n->node_lock)
        return TRUE
    else
        release_lock(nRight->plock)
        release_lock(nLeft->plock)
        release_lock(n->plock)
        return FALSE
end
```

Figure 2-9: Code for folding

Once the locks are obtained and the states are checked, the two child counters' values are compared. Now, the values these two counters hand out are intertwined. They are obviously values that their parent would have handed out as a counter, and they share between them all the values their parent would have handed out, alternating between them. Imagine enumerating a list of numbers that their parent would hand out if it was a counter. Then, one of the child counters hands out the values which appear in the odd positions of the list, and the other hands out the even-indexed values. The ideal situation in folding is that the two counters' values are adjacent on the list. Now, the value contained in the counter's register is the next value to be handed out. If they are adjacent, the parent counter can be set to next hand out the lower of the two numbers, the states are changed (the children are turned Off), and the parent is ready to start counting. This is demonstrated by the first picture in Figure 2-10.

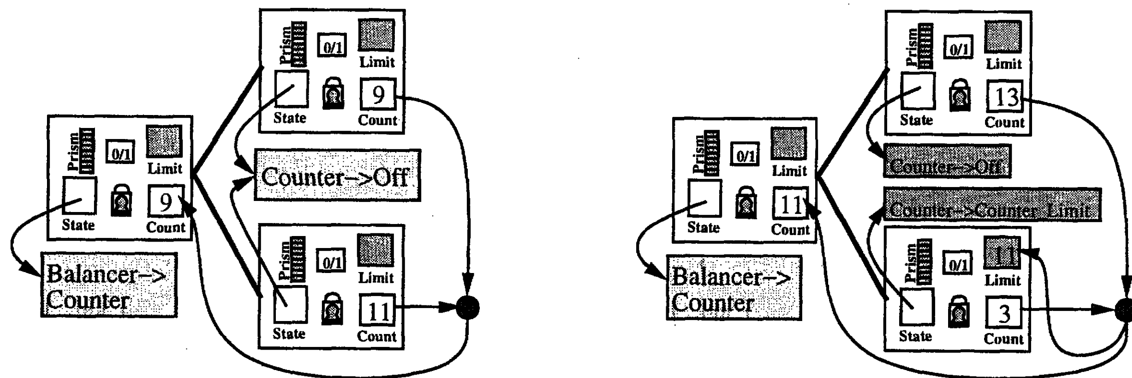


Figure 2-10: Two cases of Folding, Parent node at Level 1

Now, there are cases where the two counters' values are not adjacent on this list. This is the case where some reasoning is required. We take the maximum of the two values, find its position on the list, and move down one notch. This next-lower value on the list is the *limit* value, and it is the value assigned to the parent counter. Now, this *limit* value would normally be handed out by the smaller counter, since adjacent elements on the parent's list are handed out by the different counters. We make the smaller counter a Counter_Limit, which acts just like a Counter, except it has a limit assigned to it of the limit value. If the smaller counter's value reaches the limit value, it turns Off and hands out no more values. The larger counter is immediately turned Off. It is clear that this scheme avoids any over-counting, and a demonstration is presented in the second picture in Figure 2-10.

The only problem remaining is to show that a Counter_Limit has enough requests to use

up all of its available numbers. The parent becomes a Counter, so it does not balance any more processors towards it, and unfolding cannot occur if one of a Counter's children is a Counter_Limit. This is formally proven in the verification section, but informally, the reasoning is as follows: Either these sibling counters have been running from the beginning, or they were unfolded at some point. During unfolding, the values placed in sibling counters are initially adjacent on their parent's list. This is also true upon initialization of the tree. The balancing that occurs has the property that the requests are essentially evenly split amongst the two children. The key insight then is that if one counter's value is more than one higher on the list than the other counter, then it has satisfied more requests. But, due to the balancing process, the other counter should receive the same number of requests, which would let it catch up and fill up all of its values. (It could receive a maximum of one less, but that would imply that it was second on the balancing, which would imply that it started initially larger than the other counter, recovering the one to maintain the balance and allow the adjacency to eventually occur.)

This key insight is what allows this algorithm to work quickly. A scheme could be implemented that allowed for storing the values that weren't handed out in a queue, but this would add another level of complexity and decrease performance.

Unfolding

Unfolding is a bit easier to understand, but has its own challenges. Figure 2-12 contains the code. The same 3 locks are set, the states are checked (we can only unfold a Counter with two Off children.), and then we do the obvious settings. The current counter value can be set to one of the child counters. The next value is then set to the other child counter. Now, the problem here is that we want the balancing to occur so that the extra request always goes to the smaller child counter value. We do this by setting the balancer's toggle bit in the direction of the child with the smaller value. The two different cases are shown in Figure 2-11.

An alternative to the above algorithm would be to keep consistent the role of the Left or first child as the primary child in balancing. Then, the smaller value would always go here. Since the toggle bit would then always be reset to 0, it is just a technical difference. The advantage to the chosen scheme is that it allows each node to have a consistent set of values from which it hands out, simplifying the verification process. In this alternate scheme, each unfolding could

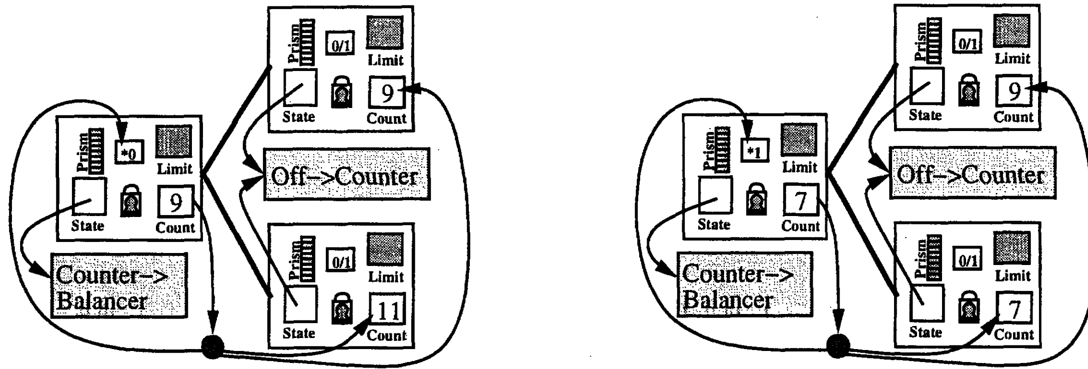


Figure 2-11: Two cases of Unfolding, Parent node at Level 1

assign to a child node a distinct class of values to hand out that it didn't before.

The biggest problem with unfolding is primarily an implementation issue. Consider when these actions would occur. Folding would occur because of below-average contention in that area of the tree. There isn't much delay when the locks are set, since there just aren't that many processors around. On the other hand, unfolding can be a costly process, since it occurs because of high contention. The code presented here releases the parent lock as soon as it's state is set, so that the processors that are waiting to access the counter can sooner find out that it is now a balancer and balance. Some optimizations performed here include having the lock releaser go through and tell all of the processors waiting in the queue that the state has changed, so they can diffract, which gives the balancer a good start with high contention. A future optimization could lie in implementing a tree lock instead of a queue lock so this release could occur even faster.

A specification of this algorithm is proved correct in Chapter 5, and an argument is sketched in Chapter 6 about the correctness of this implementation.

```
function attempt_unfold(n:ptr to node) returns boolean
    nLeft, nRight: ptr to node
    val, ID, i: int
begin
    nLeft = n->Left
    nRight = n->Right
    ID = n->ID
    for i from 1 to NUMPROCS
        if (Bookkeeping[i] = n) return FALSE;

    acquire_lock(nLeft->toggle_lock)
    acquire_lock(nRight->toggle_lock)
    acquire_lock(n->toggle_lock)

    if (n->state == Counter) and (n->count != n->change) and
       (nLeft->state == Off) and (nRight->state == Off) and
       (n->ID == ID) then

        n->state = Balancer
        n->PID = n->PID + 1
        val = n->count
        if ((val - n->init) / power(2,n->level)) mod 2 == 1
            n->toggle = 1
        else
            n->toggle = 0
        release_lock(n->toggle_lock)

        nLeft->state = Counter
        nRight->state = Counter
        nLeft->ID = nLeft->ID + 1
        nRight->ID = nRight->ID + 1
        if ((val - n->init) / power(2,n->level)) mod 2 == 1
            nRight->count = val
            nLeft->count = val + power(2,n->level)
        else
            nLeft->count = val
            nRight->count = val + power(2,n->level)
        nLeft->change = nLeft->count
        nRight->change = nRight->count

        release_lock(nRight->toggle_lock)
        release_lock(nLeft->toggle_lock)
        return TRUE
    else
        release_lock(nRight->toggle_lock)
        release_lock(nLeft->toggle_lock)
        release_lock(n->toggle_lock)
        return FALSE
end
```

Figure 2-12: Code for unfolding

2.3 Bookkeeping

The missing item from the unfolding section was Bookkeeping. We now explain its necessity. A Balancer, upon folding into a Counter, now must force all of the delinquent processors that balanced through it to now return. Versioning causes this to happen. However, imagine that the node now wishes to unfold again. If it becomes a Balancer, then new processors will balance through it and have correct forecasts for the new child Counters. However, imagine a processor that visited the node in its first incarnation as a Balancer. It received the forecast for its children, then also Counters, and began to balance. While it was attempting to diffract or access the toggle bit, all of these changes were to occur, and the node now went into its second incarnation as a Balancer. If this old processor were to diffract against a new processor, then it would upset the balance of the system, since it would arrive at one child Counter, find an incorrect version number, and return to the parent, while its partner from diffraction would arrive at the other child Counter and correctly access it. Now, imagine this happened potentially many times, and each time the old processor went towards the same Counter. When it came time to fold again, there would be no processors left that could bring the troubled Counter back into balance with its sibling.

The solution to this is simple, and the code is given in the main traversal (Figure 2-7) and unfolding code (Figure 2-12). We create a global bookkeeping array, one in which every processor has an entry. A processor, upon visiting a Balancer, registers in its entry of the array the balancer it is visiting. It then rechecks to make sure that the node is still a Balancer with the forecasted ID and enters the balancing section of the code. If the information on the second check was inconsistent with the processor's remembrance, it will go up the tree until it gets back on track. Now, the final piece is a restriction on unfolding. In order for a processor to unfold a Counter, it must traverse the bookkeeping array and make sure no processor is registered as visiting this node as a Balancer. If this traversal is successful, it then unfolds the node if the ID was the same as before the traversal. This is correct for the following reason. A processor sees that the node is a Balancer and puts itself into its array location. An unfolding processor saw that the node was a Counter and that the first processor's bookkeeping entry did not have this node registered. The ordering of these events on a machine that guarantees atomicity per memory location, (as our Alewife machine does guarantee [8], implies that either the first processor will see the update upon its recheck, or the unfolding processor will learn that it was

out of date and will not unfold.

2.4 Cache Sizing

The size of the cache is an interesting study on its own. The steady-state analysis [18] predicts that there should be $cd2^d$ prism locations in the tree, where c is a constant and d is the depth. Now, in this tree, we have changing depths. The solution is for each processor to keep a notion of the tree's current depth. A skeletal cache of the state is enough for a processor to average the depths of the various paths into the tree and come up with an average depth. Given the best experimental constant c and a large enough prism to handle the largest allowed tree, processors can then simply pick a value randomly within their expected prism size. It is expected that in practice eventually processors will approach the same average depth computation, providing the most efficient balancing regardless of the size of the tree.

2.5 Folding and Unfolding Policy

There are three main qualifications that a good scaling policy should meet.

- A policy should react quickly to large changes in the load.
- A policy should keep the overhead that it causes low and factor it into its decision making process
- A policy should keep the number of false positives low and limit many consecutive oscillations

Keeping this in mind, most of our policy exploration was focused on studying the contention at a counter lock. We felt that this was a good estimate for the overall load of the tree. If the lock was always empty when a processor arrived at the counter, then that counter should be folded into its parent. If the lock was always overloaded, then the counter should be unfolded. We found that observing the time it took to access the counter was a good measure. Queue locks have the nice property that the times measured are stable under consistent contention levels, unlike the oscillating times a spin lock would provide.

We then designed our policy around setting thresholds for these times. Passing below a folding threshold or taking longer than the unfolding threshold was a good indication that the local area should change. However, the data structure should not change based on the opinion of one processor. Our final policy is a variant of B.H. Lim's policy in his reactive data structure [11]. It uses a string of consecutive times to allow a change to occur. The minimum number of consecutive times was a constant that was decided upon by experimentation.

This met all three qualifications. A large change in the load will move the time consistently below or above these thresholds and allow for a change. The overhead is low since only one test is needed to see if the time is within the thresholds, stopping any current streaks and allowing the processor to continue. Finally, by requiring consecutive times, a nice hysteresis affect occurs, because it could not immediately change back in the other direction.

Future possibilities include on-line competitive schemes [14] or policies that measure the balancer performance. Since the balancers are tuned by the dynamic prism sizing, a study of the toggling behavior and diffracting rates could reveal a pattern which indicated when it should fold and when its children should unfold.

Chapter 3

Experimental Results

We evaluated a Dynamic Diffracting Tree by implementing it on both a multiprocessor machine and a simulator and running several experiments. The MIT *Alewife* machine developed by Agarwal et. al. [1] was the target machine for this implementation. However, the largest machine only has 32 nodes. We then ran the same experiments on the *Proteus*[1] simulator, developed by Brewer et. al. [7], where we were able to extend our results to 256 processes, and did a correlation to show that the results were comparable. The experiments we performed include index-distribution, sudden spikes and drops in load levels, and producer/consumer runs, all of which demonstrate the advantages of the DDT in a variety of applications.

3.1 Experimental Environments

The MIT Alewife machine consists of a multiprocessor with cache-coherent distributed shared memory. Each node consists of a Sparcle processor, an FPU, 64KB of cache memory, a 4MB portion of globally-addressable memory, the Caltech MRC network router, and the Alewife Communications and Memory Management Unit (CMMU). The CMMU implements a cache-coherent globally-shared address space with the LimitLESS cache-coherence protocol [8]. The LimitLESS cache-coherence protocol maintains a small number of directory pointers in hardware, and handles the rest in software. The Alewife machine guarantees sequential consistency on its cache-coherent memory locations, which means that any processor's memory transactions

[1] Version 3.00, dated February 18, 1993

are ordered consistent with their code base, and memory transactions on a memory location are processed as if they came from a FIFO queue. The Alewife LimitLESS cache coherency policy specifically upon receipt of a write request will require all processors to flush their cached copies of that variable and respond with an acknowledgment before granting the write request.

The primitive that Alewife provides as a read-modify-write operation is *full/empty bits.* Every memory location has a full/empty bit associated with it. This allows for mutual exclusion, since operations are provided which allow a processor to atomically set the bit full if empty and vice-versa. Given this, we deal more abstractly in our remaining discussion about mutual exclusion, taking for granted the availability of queue or spin locks.

Proteus multiplexes parallel threads on a single CPU to simulate the Alewife environment. Each thread has its own complete virtual environment, and Proteus records how much time each thread spends in its various components. In order to improve performance, Proteus does not completely simulate the hardware. Instead, local operations are run uninterrupted on the simulating machine, and this is timed in addition to the globally visible operations to derive the correct local time. This limits its ability to accurately simulate the cache-coherence policy. Proteus does not allow a thread to see global events outside of its local time environment.

3.2 Index Distribution Benchmark

Index-distribution is the simple algorithm of making a request and waiting some time before the request is repeated. In this case, the amount of time between requests is randomly chosen between 0 and `work`, a constant that determines the amount of contention present. `work` = 0 represents the familiar counting benchmark, providing the highest possible contention for the number of processors given. A higher value, usually `work` = 1000 is chosen to give a lower-contention environment. This is a good benchmark to study because it is often used in load-balancing, when the tasks that the processors perform take a varying amount of time, but usually within some predictable level of `work`. We ran this benchmark for a fixed amount of time on the Alewife machine (10^7 cycles), varying the number of processors[2] and the value of `work`. We also ran this benchmark on the Proteus simulator (10^5 cycles), and correlated the results. Since there are usually startup costs, the algorithms are run for some fixed time before

[2]Throughout this paper, each processor only runs one process

the timing begins. This brings into question the fact that the DDT will grow and shrink if the load does not meet well with its initial conditions. Since a separate experiment is conducted to test the changes of the DDT, a substantial startup period will be allowed before timing begins to allow the DDT to best match the input load.

The data collected were the average latency and throughput. The average latency is the average amount of time between the call to `get_next_index()` and its return. The throughput is the total number of `get_next_index()` operations that returned in the time allowed. These are clearly related numbers that can be approximately calculated from each other.

In this benchmark, the algorithms that were run were the Dynamic Diffracting Tree, Diffracting Trees of widths 2, 4, and 8 (and on Proteus, 16 and 32), and a queue-lock based counter. This queue-lock consists of a linked list of processors pointing towards their successors, waiting for their predecessors to wake them up once they are done with the lock. There is a tail pointer which directs new processors to the end of the queue. This code was implemented using atomic register-to-memory-swap and compare-and-swap operations.

3.2.1 Diffracting Trees and Queue Locks

We experimented with two variants of the Diffracting Tree algorithm and with several different prism sizes, and changed the diffracting tree algorithm to use queue-locks instead of spin-locks, which made their performance more robust. For each depth of the diffracting tree, we found the optimal prism size.

We tested both the original Diffracting Tree algorithm [20] and the alternate Diffracting Tree algorithm [18]. The alternate algorithm performed better at all depth and prism sizes. The main difference between these two algorithms occurs after the test-and-test-and-set operation on the toggle bit fails. In the original algorithm, a processor then waited longer to see if it got diffracted. In the alternate algorithm, a processor attempted to enter the prism again. This is better because a processor is more likely to diffract when it re-enters the prism then when it waits around.

The other major update of the original implementations is the use of queue-locks on the counters as opposed to spin-locks. Originally, a test-and-test-and-set loop repeated until it could acquire the lock and increment the counter. This caused the diffracting tree's throughput to

degrade after its peak due to contention. With the addition of a queue-lock, throughput remains steady as higher contention on the lock simply increases the waiting time for the queue. This makes the algorithm more robust, considerably extending the lifetime of a diffracting tree. Figure 3-1, shows a comparison of throughputs of optimal depth 3 diffracting trees with queue- and spin-locks, and Figure 3-2 shows the average waiting time for the two counters.

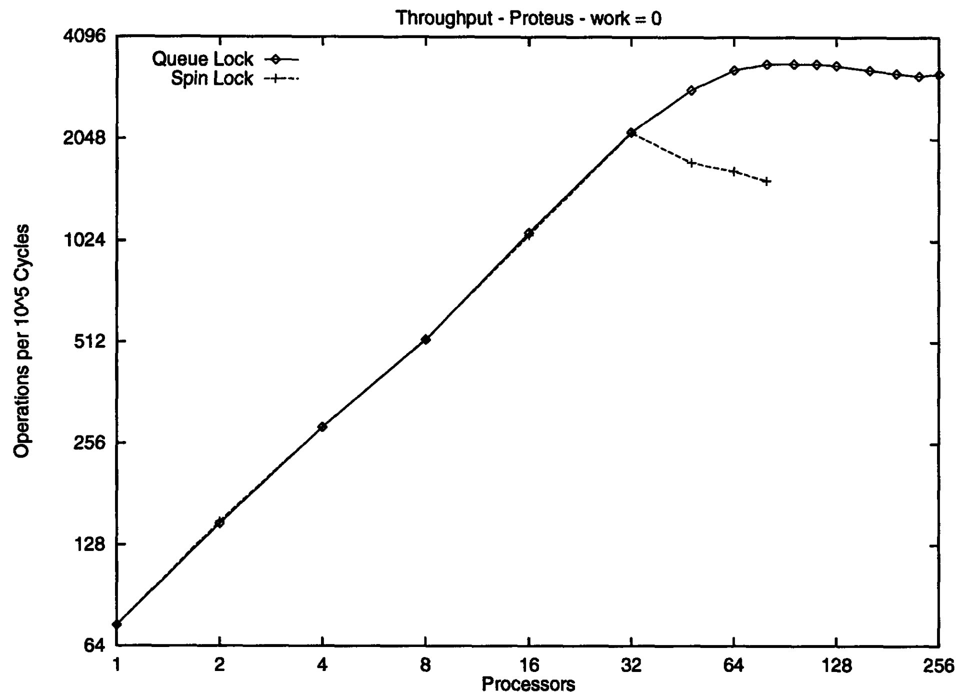


Figure 3-1: Diffracting Trees of depth 3 with Queue or Spin Locks at Counters on Proteus

The Steady-State analysis [18] determined that there should be $cd2^d$ prism locations in the tree, with $c2^d$ locations on each level of the tree, where c is a constant. We experimented by comparing trees at each level and found that $c = 1/2$ was the best factor overall.

3.2.2 Alewife Results

We have the first published performance results for Diffracting Trees on the Alewife machine. We implemented the Dynamic Diffracting Tree directly from the description given in Chapter 2. We configured the dynamic prism sizing to use the constant c found above. We set the number of consecutive timings before a change to be 80, a good experimental number that limited the number of oscillations. Our experiments also determined that the best fold and unfold threshold times were 150 and 800 cycles. Figure 3-3 shows throughputs for a queue-lock based counter,

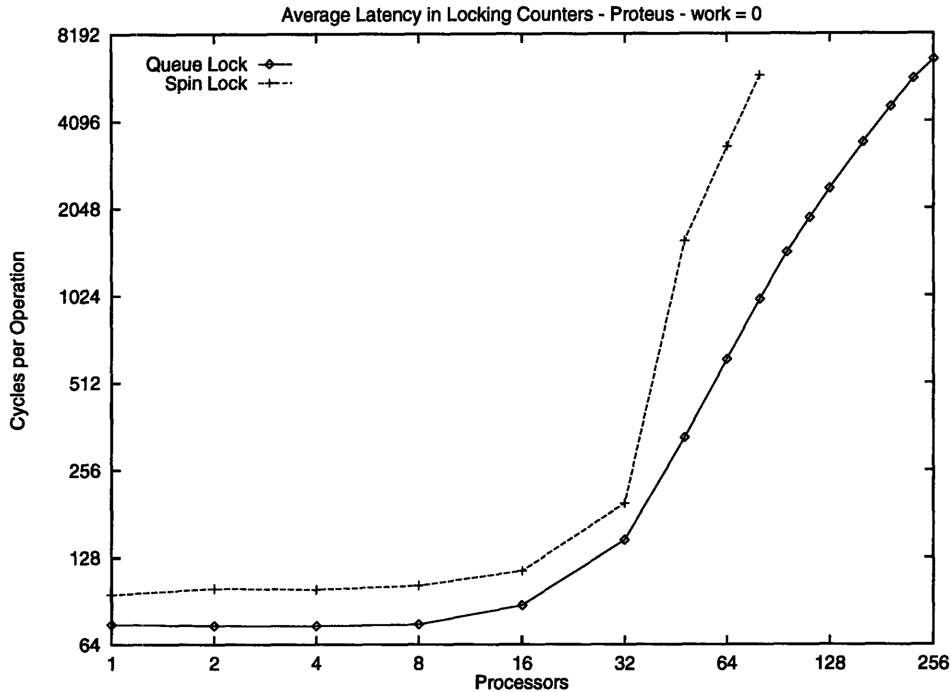


Figure 3-2: Average Latency of Queue or Spin Lock Counters on Diffracting Tree of depth 3 on Proteus

diffracting trees of depth 1, 2, and 3, and the DDT. The most interesting result is that the DDT surpasses all of the diffracting trees shown for a brief range. This is due to its ability to expand only where needed, supplying irregularly sized trees which perform better in this range.

Since the Dynamic Diffracting Tree should represent optimal diffracting trees at each of their optimal points, we have constructed a composite graph of the diffracting tree and queue-lock counter throughputs, with the highest throughput from any diffracting tree or queue-lock counter at a given load level chosen for the graph. We show the optimal composite vs. the DDT for the Alewife in Figures 3-4 and 3-5 under high contention. The throughput and latency appear to stay within a factor throughout its performance. The average factor between the throughput of the DDT and the optimal composite is 1.27.

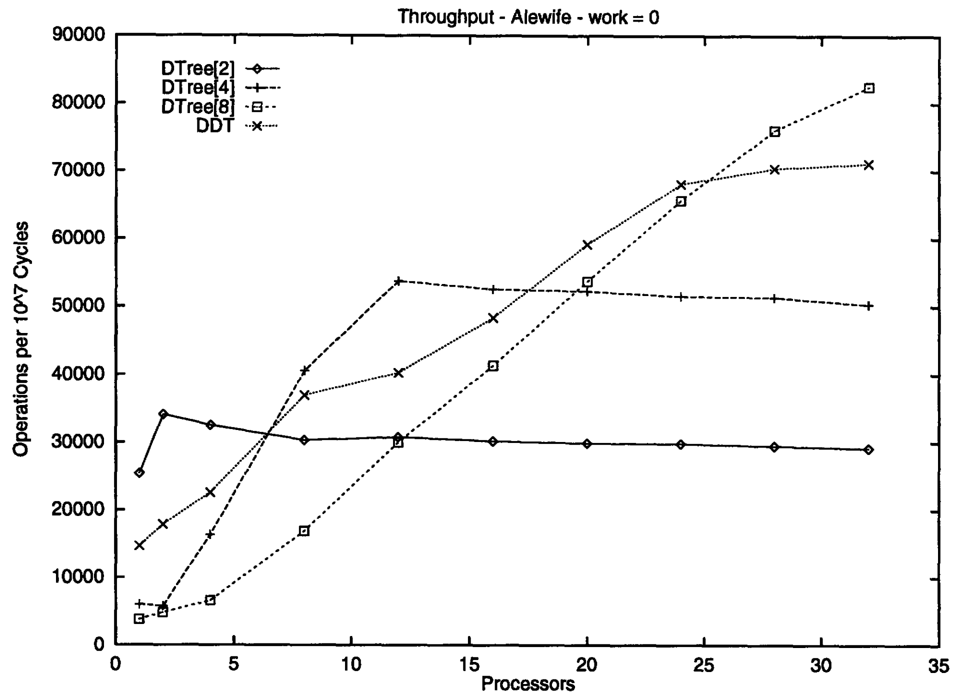


Figure 3-3: Diffracting Trees, Queue-Lock Based Counter, and DDT on Alewife from 1-32 Processors

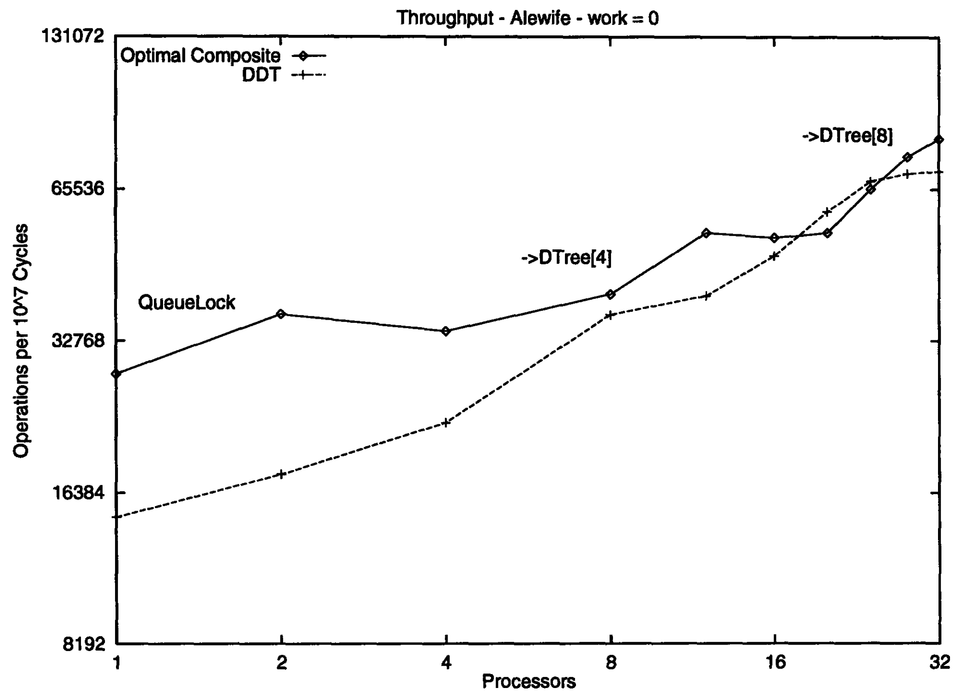


Figure 3-4: Throughputs of Optimal Composite vs. DDT on Alewife

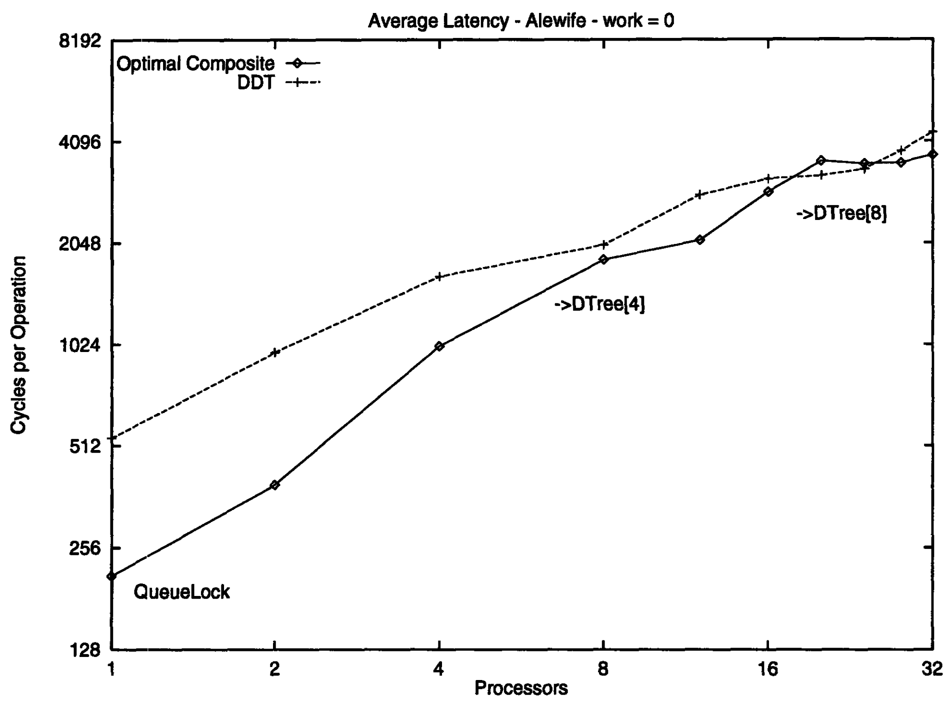


Figure 3-5: Latencies of Optimal Composite vs. DDT on Alewife

3.2.3 DDT Results on Proteus

Unfortunately, the Alewife machine only has 32 nodes. Until larger versions are available, we must rely on simulations to provide higher load level results. We turn to the Proteus simulator, which simulates Alewife's hardware, although it does not fully implement Alewife's LimitLESS cache-coherence policy.

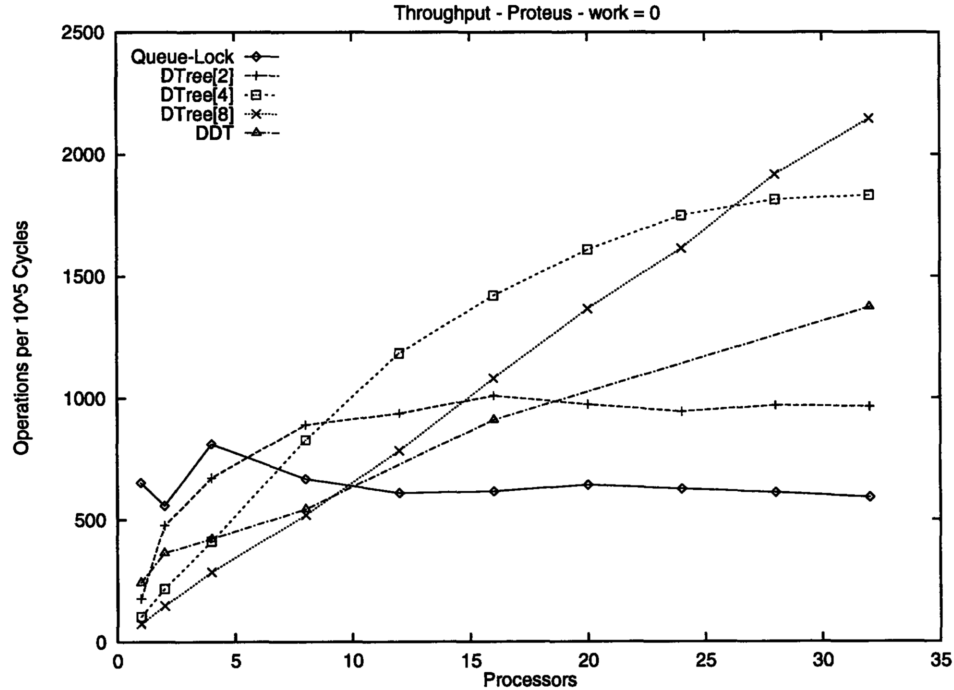


Figure 3-6: Diffracting Trees, Queue-Lock Based Counter, and DDT on Proteus from 1-32 Processors

It is important at this point to compare the results gathered on the Alewife with the Proteus, to make sure that the results can be extended over. Figure 3-6 is the counterpart to Figure 3-3. Notice that the shapes of the Diffracting Trees look similar, although they seem to flatten out more quickly on the Alewife than on the Proteus. But, we really need to see two curves side by side. We construct a Proteus optimal composite for throughput for 1 to 32 processors and normalize the Alewife curve to it. This graph is shown in Figure 3-7. The results show that the Alewife trees have a higher optimal load level, but the graphs still look comparable, a good result for Proteus.

We now extend the Proteus results up to 256 processes, and add Diffracting Trees of depths 4 and 5. Figures 3-8 and 3-9 show the throughputs and latencies of Diffracting Trees of depth

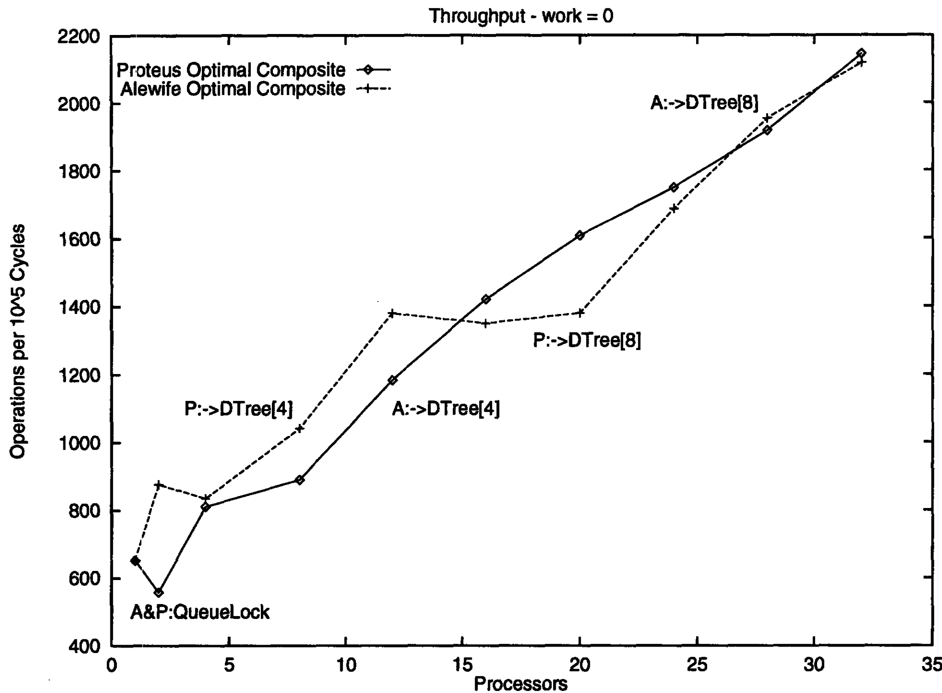


Figure 3-7: Throughputs of Optimal Composite on Proteus and normalized Alewife

0 (queue-lock based counter) through 5.

The Proteus environment is different enough to require a change in some of the constants. The difference in timing mechanisms forced us to move the fold threshold up to 200 cycles. However, the queue-locks had more stable waiting times, enabling us to bring the consecutive timings threshold down to 25.

We show the comparison between the optimal composite and the DDT in Figures 3-10 and 3-11 for high contention ($work$ = 0) cases, and 3-12 and 3-13 for lower contention cases ($work$ = 1000). The results showed that Proteus charged more for the overhead required in computing the changes, but this seems to be a constant factor that is machine-dependent. This could be attributed to the cache-coherence differences between the two architectures. For the high contention case, the average factor between the two throughputs was 1.56, and in the low contention case, the average factor was 1.41.

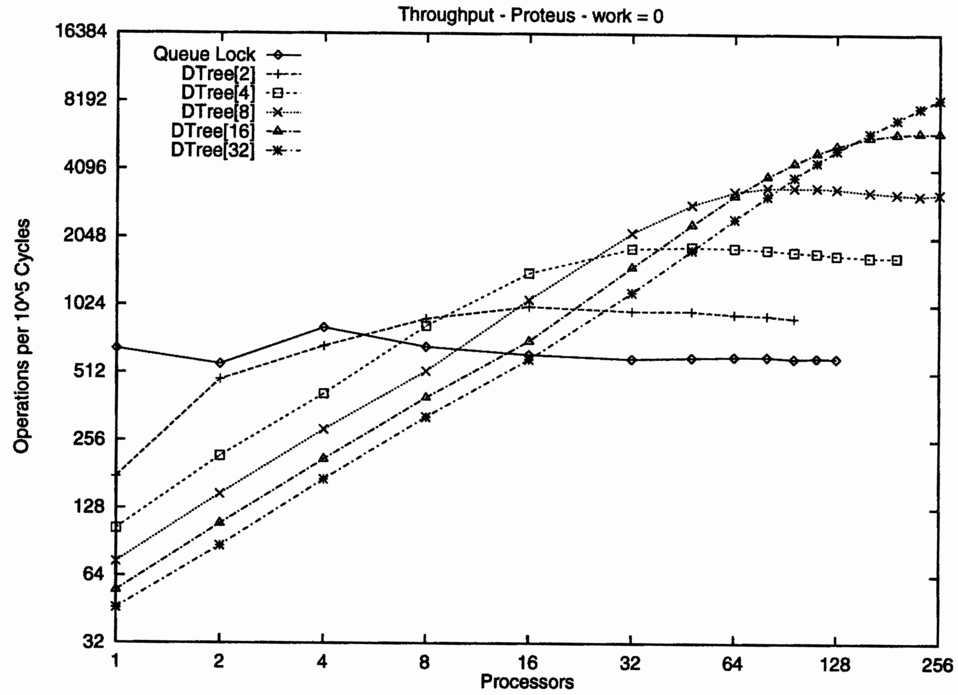


Figure 3-8: Throughputs of Diffracting Trees and Queue Lock on Proteus

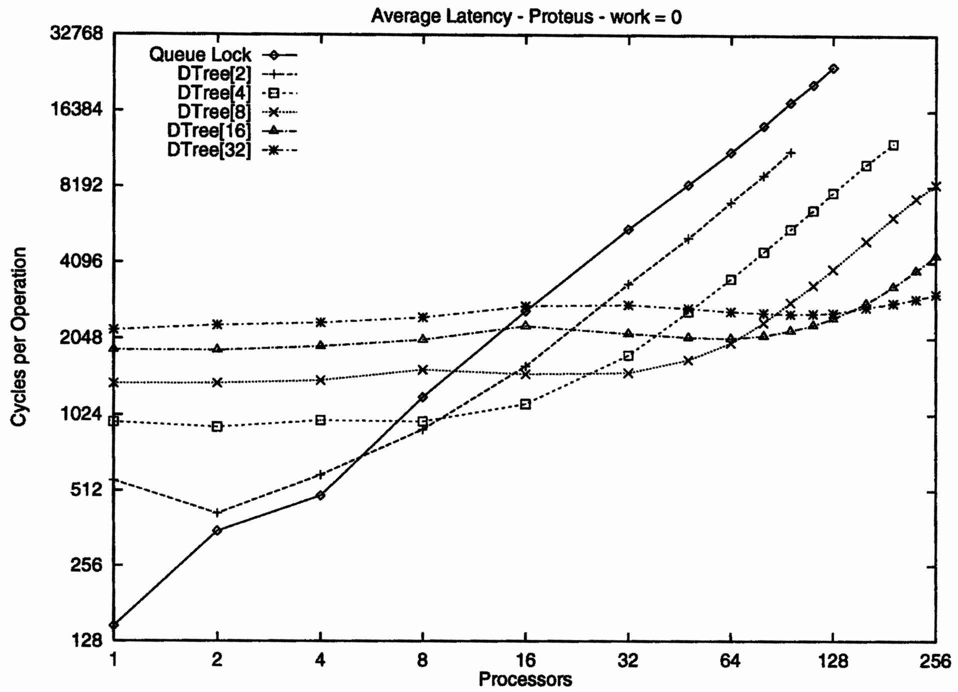


Figure 3-9: Latencies of Diffracting Trees and Queue Lock on Proteus

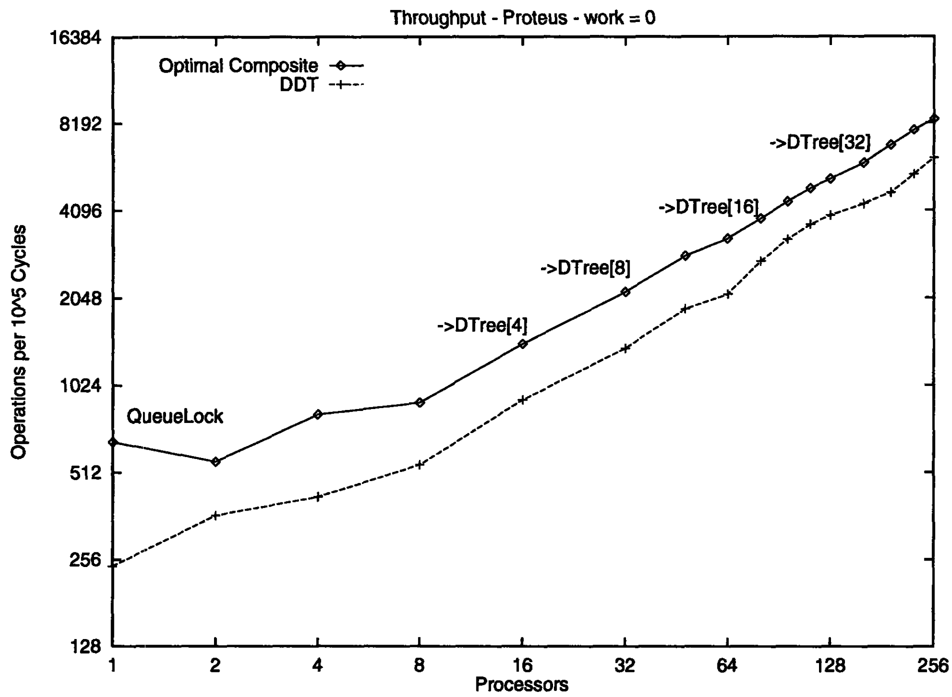


Figure 3-10: Throughputs of Optimal Composite vs. DDT under high contention on Proteus

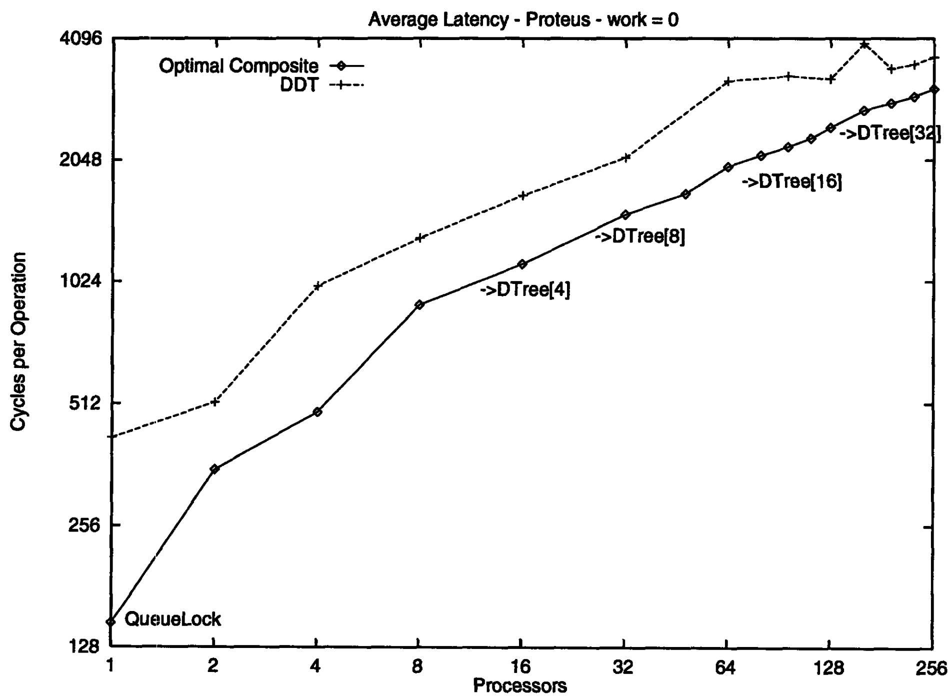


Figure 3-11: Latencies of Optimal Composite vs. DDT under high contention on Proteus

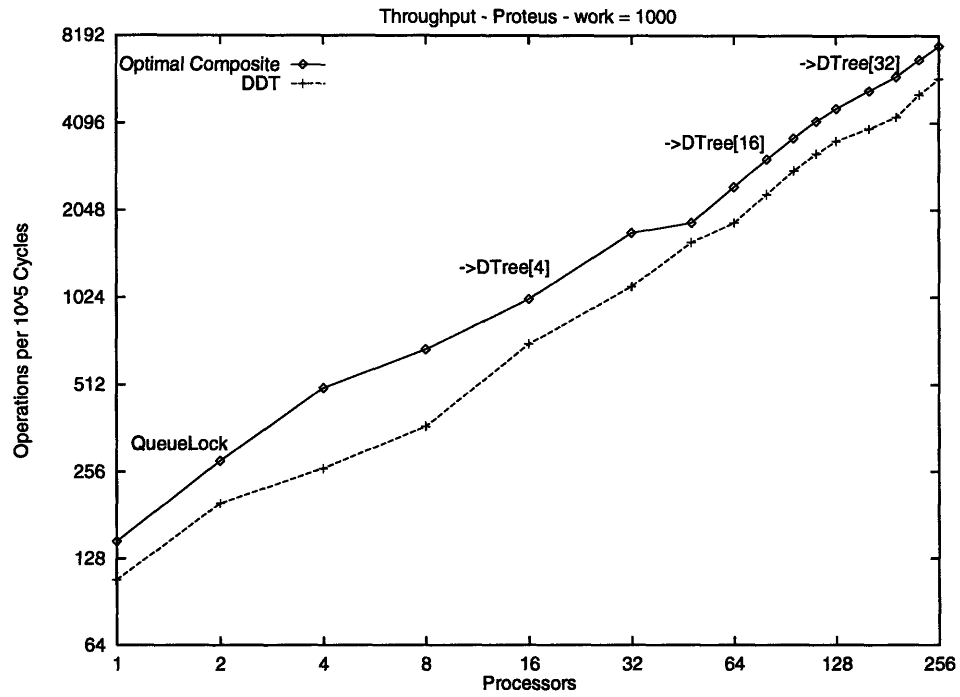


Figure 3-12: Throughputs of Optimal Composite vs. DDT under lower contention on Proteus

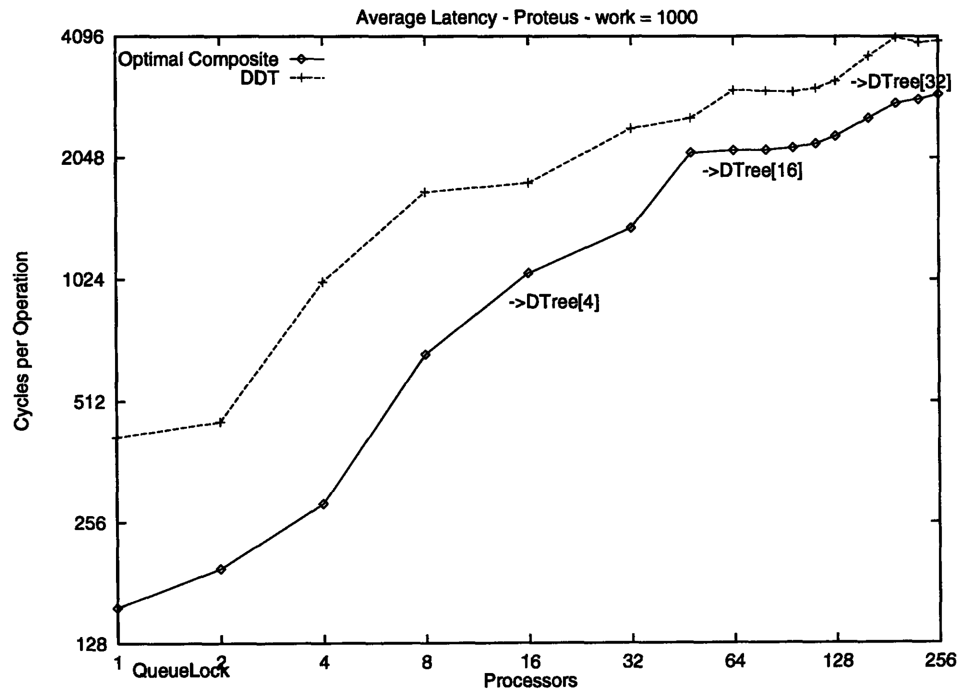


Figure 3-13: Latencies of Optimal Composite vs. DDT under lower contention on Proteus

3.3 Large Contention Change Benchmark

We measured the response of a DDT to a sudden change in contention levels, measuring the average latency of the DDT in fixed width intervals before and after the change occurred, graphing the change in the average latency over time. Here, the system constant for the number of consecutive timings was set at 10 to better handle sudden changes.

3.3.1 Sudden Surge

We ran the index-distribution benchmark with 32 participating processes for a fixed amount of time and `work` = 0, to allow the tree to best fit the load. The tree sized to a depth 3 tree. We then started timing for four time intervals of 25,000 cycles, and allowed an increase in the number of processors to 256, timing for 400,000 additional cycles. The tree grew to depth 5. Figure 3-14 shows the plot of these measurements. As you can see, it takes about 100,000 cycles for the curve to level off, which given an eventual average latency of 4,000 cycles, indicates that it took about 25 equivalent passes through the tree to expand 2 levels, which is what would be expected with the consecutive timings constant set at 10. The throughput before the change occurred was around 340 operations per 25,000 cycles. At the top of the spike, the throughput goes up to around 440 operations, and as the latency drops off, the throughput rises quickly to 1500 operations and remains steady.

The plot also contains Diffracting Trees of depth 3 and 5 with their average latency at 256 processors, which are what the DDT emulates before and after the change. Here is a good example of the tradeoff that a developer must consider in choosing to use the DDT. Imagine that the developer initially used the diffracting tree of depth 3. The triangle on the left formed by the DDT and the depth 3 Diffracting Tree represents the spike in latency that the algorithm must necessarily absorb in order to change, and is a loss to the developer. However, the quadrilateral-like shape formed between the DDT and the Diffracting Tree of depth 3 to the right of the triangle is the region that a developer gains in using the DDT. Of course, the developer could choose to use the depth 5 tree all along, but the DDT outperforms this tree in the lower load case, which would most likely be the common case here.

Now, one of the reasons the spike is so high is due to dynamic prism sizing. The sudden surge in processors all still think that the tree is of depth 3, so have a smaller prism range

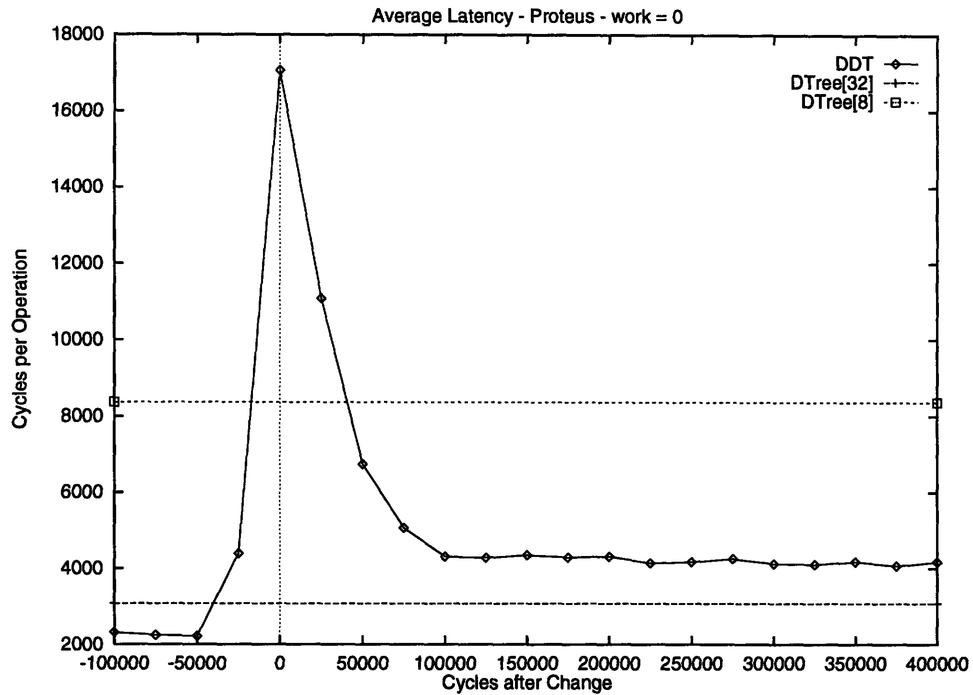


Figure 3-14: Average latency of DDT over time in response to sudden surge

to diffract through. We ran the experiment with statically sized prisms, large enough to best handle the new load of 256 processors, and a comparison is shown in 3-15. As you can see, the bigger prism has a smaller spike in latency, but pays for it with less than optimal performance before the surge. A developer whose system is volatile may wish to choose a higher constant for the dynamic prism sizing in order to handle surges in load but still maintain near optimal levels of performance.

3.3.2 Sudden Drop

We now tried the opposite experiment, with the same settings, dropping from 256 to 32 processes. It takes about 75000 cycles to level off at 2500 cycles. At an average latency of 2500 cycles, this would take around 20 passes, which again seems to be directly caused by the consecutive timings constant of 10 and the 2 levels of folding that are required.

The same trade-off described for the surge is evident here for the developer who currently uses a Diffracting Tree of depth 5.

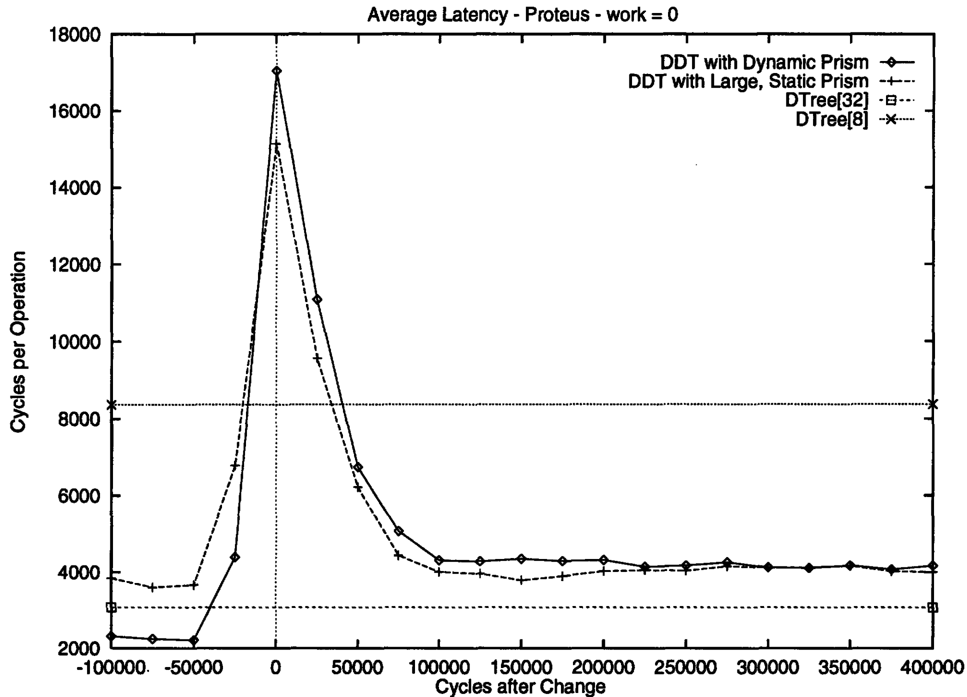


Figure 3-15: Comparison of surges with dynamic and static prism sizing

3.4 Producer/Consumer Benchmarks

Job pools are a collection of jobs that need to be performed by the various processors in the system. Any processor can enqueue (produce) a new job into the pool or dequeue (consume) a job in order to perform it. The shared counter implementation of a job pool consists of two shared counters and an array. To enqueue a job, a processor requests a value from the producer counter and places the job at that location in the array. To dequeue a job, a processor requests a value from the consumer counter and goes to find a job in that location.

An alternative job pool scheme consists of one of many load balancing techniques. Here, processors keep local job pools from which they choose jobs to execute, and participate in load balancing to trade their job allocations. The best load balancing scheme known is by Rudolph, Silvkin-Allalouf, and Upfal (RSU) [16]. In RSU, a processor about to dequeue a job attempts to load balance with probability inversely proportional to the size of its job pool. If it decides to load balance, it picks a processor at random and attempts to equalize their job pool sizes.

In high load situations where processors frequently enqueue and dequeue jobs, the class of load balancing algorithms currently outperforms the shared counters. The lock-based counters

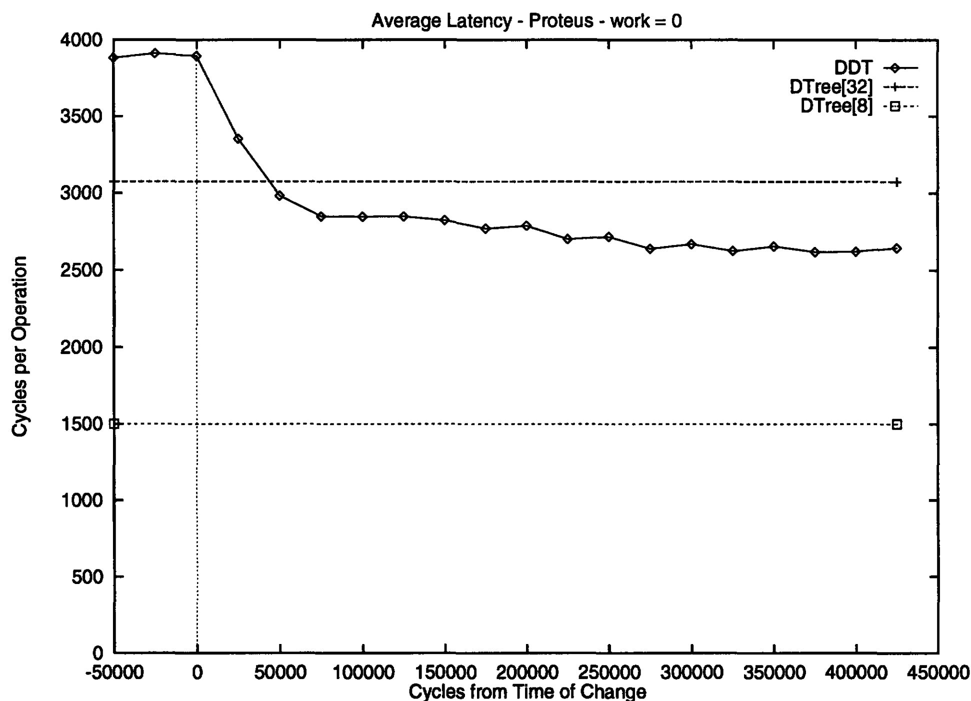
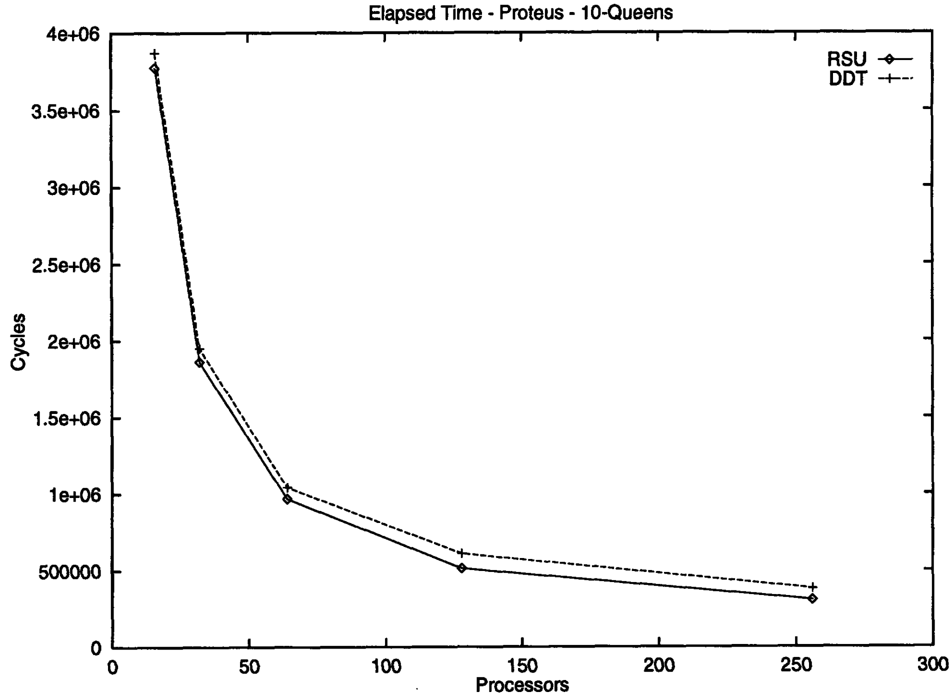


Figure 3-16: Average latency of DDT over time in response to sudden drop

do well against RSU in the low load levels, and the distributed counters seem to come close to RSU's level of performance, but overall, no shared counters has been able to effectively compete with RSU. We now show that the DDT has become an effective competitor.

3.4.1 10-Queens

The n-Queens problem is a good problem to test the DDT on. Here, every consume operation will produce 10 new jobs at a higher depth until a limit is hit. The recursive nature of the algorithm leads it to apply different load levels on the producer and consumer functions. Under low loads, the counters can become lock-based algorithms and compete effectively against RSU. As the number of processors participating increases, the trees can grow larger to give the distributed performance necessary to compete with RSU. Figure 3-17 shows how close the diffracting tree comes to RSU in total time elapsed throughout the differing load levels.



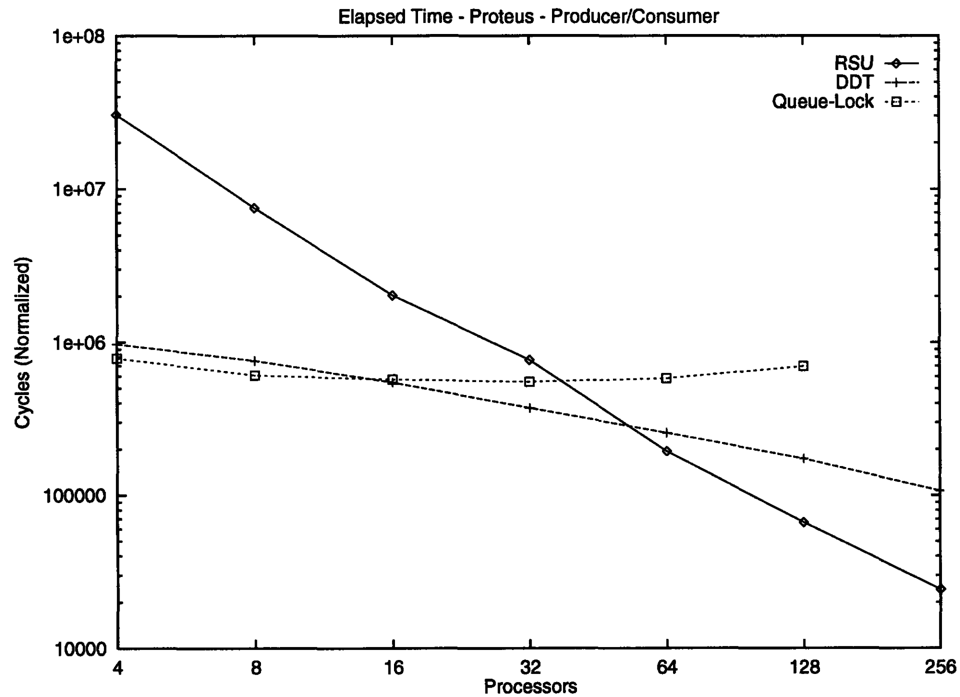

```
Initialization
    produce one instance with depth=0
repeat
    instance = consume();
    wait 8000 cycles;

    if instance's depth < 3 then
      produce 10 instances with depth greater by 1
until all instances have been consumed
```

Figure 3-17: 10-Queens Performance and Code

3.4.2 Sparse Producer/Consumer Actions

The pitfall of RSU and the other load balancing algorithms is the poor performance that occurs under sparse access patterns. To exhibit this, we make half the active processors consumers and the other half producers. Producers initially produce a job and wait until that job is consumed before they produce a new job. This continues until a total of 2560 jobs have been completed. This creates a sparse access pattern in the system since any load balancing transaction could at most shift one job, which is the necessary consumption for the production to continue. We run this system for RSU, a DDT job pool, and a queue-lock job pool. We measure the time elapsed between the beginning of the benchmark until 2560 elements are consumed, and show

the results in Figure 3-18. As one can see, the DDT provides near queue-lock performance in low-loads, and approaches the performance of RSU in higher loads.




```
producer:
    repeat
        produce(val);
        wait until the element is consumed;
    until a total of 2560 elements are consumed

consumer:
    repeat
        consume()
    until a total of 2560 elements are consumed
```

Figure 3-18: Producer/Consumer Performance and Code

Chapter 4

DDT Formal Model

We use Lynch and Tuttle's I/O Automata [13] as the framework for our model. This framework allows us to clearly specify our model as the composition of automata that clearly describe our system. Executions of this system start in some initial state and act through actions that are enabled, moving the system from state to state. We now turn to our specific problem.

The counting problem involves the allocation of values to P *users*, $U_1, \ldots, U_P$. It is formally modeled by a system that contains P processes that acccept requests for values and return with output responses. Each process is the agent of a specific user U_p.

In order for the allocation to be consistent with a counter, there are several properties that must be satisfied. We assume that the system returns values from $\mathbf{N} = \{0, 1, 2, \ldots\}$.

Property 4.1 (Safety Property) In any state of an execution, the values output are distinct and in the set $\{0, \ldots, k-1\}$, where k is the number of requests.

Property 4.2 (Liveness Property) In any fair execution, if a finite number of requests appear, then the number of outputs will equal the number of requests.

Our implementation of such a system consists of P processes that accept input requests (or tokens) from users and shepherd the tokens through a shared memory hierarchical data structure using balancers to fairly distribute the tokens throughout the structure. The hierarchical data structure, known as the Dynamic Diffracting Tree, is an infinite binary tree whose states determine the paths taken. It is dynamic because these states are subject to change in a non-deterministic way.

We will model our system as the composition of two I/O automata. The main I/O automaton is a shared memory system. It consists of processors that act as agents of the user and non-deterministic actions that can change states. This automaton has access to various shared variables, all part of one large binary tree structure. The second I/O automaton consists of a composition of an infinite number of balancers. To solve the counting problem, we also model the users as automata, can formally compose well-formed users with the system. See Figure 4-1 for the total composition.

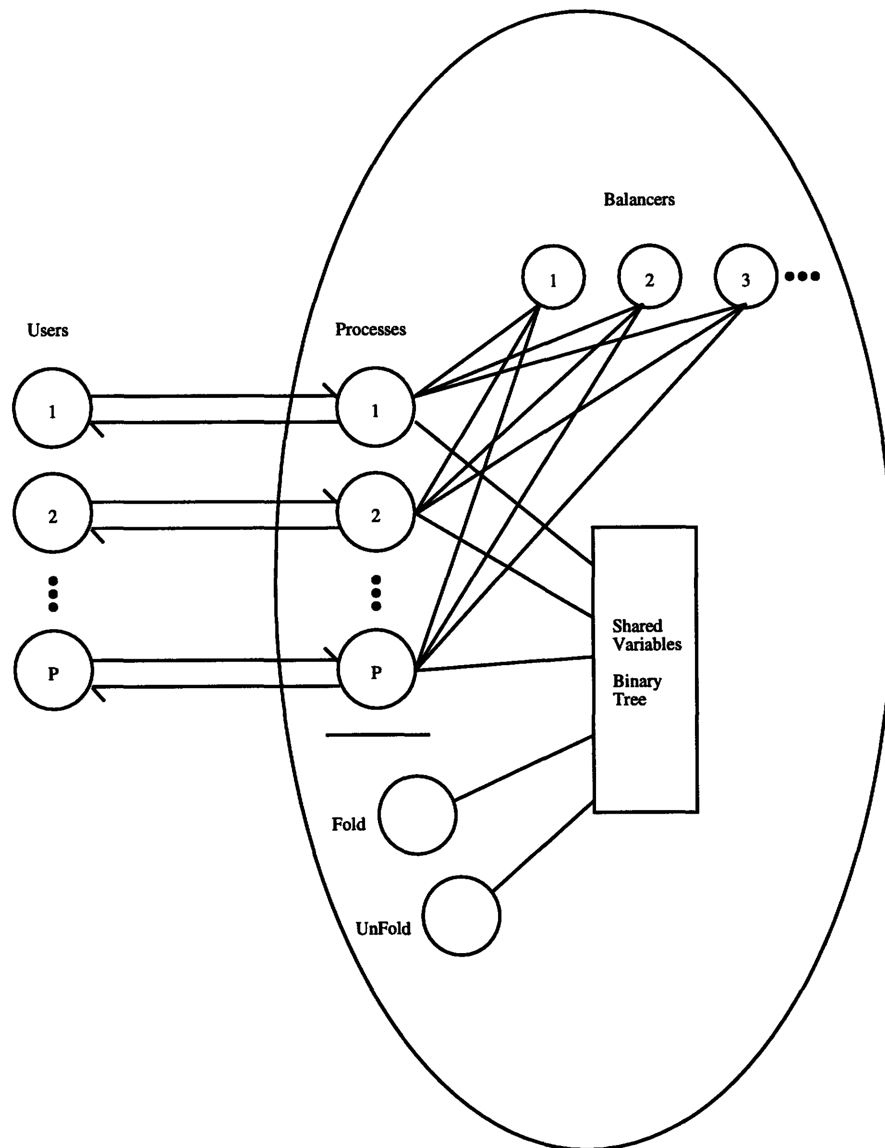


Figure 4-1: Interaction between Well-Formed Users, the Shared Memory System, and Balancers

We first describe the user automata that interact with the system.

4.1 User I/O Automaton

Each user communicates with its process agent to obtain values from the DDT. The only restriction we assume is a handshake protocol in which U_p invokes $request_p$ to obtain a value, and waits for a $return(V)_p$ action before it can request a new value. This cyclic sequence is defined to be *well-formed* for user U_p, and can easily be modeled by a state machine.

We now define the balancer automata that are accessed by the main process automata.

4.2 Balancer I/O Automaton

Balancers are used to distribute the tokens throughout the tree. Specifically, they are used to pass a token from a parent node to a child node, attempting to split the number of tokens evenly between the children.

Balancers have an input wire and two output wires. The number of tokens ever received on the input wire is denoted by x and the number of tokens output on its two output wires are denoted y_0 and y_1. Balancers are formally defined to satisfy the following two properties.

Property 4.3 (Balancer safety) In any reachable state, $x \geq y_0 + y_1$, $y_0 \leq \lceil x/2 \rceil$, and $y_1 \leq \lfloor x/2 \rfloor$.

Property 4.4 (Balancer Liveness) In any fair execution, if there are a finite number of input tokens x to the balancer, then $x = y_0 + y_1$.

As the tree changes, new balancers will be needed for each node. We will create an infinite number of balancers for each node, and process automaton can agree on which balancer to use. For now, we define distinct balancers $b \in B$. We guarantee for now that B is a countable set, and later give a precise definition for B.

Figure 4-2 shows an I/O automaton for a balancer that clearly preserves these properties when composed with our main process automata whose actions invoking the balancer are well-formed.

Before we define the main process automaton, we turn our attention to defining the hierarchical data structure in our system.

Signature:

Input:
$bal_req_{p,b}$

Output:
$bal_ret(V)_{p,b}, V \in \{0,1\}$

States:

$\mathbb{P}_b \subseteq \{1, \ldots, P\}$, initially $\emptyset$
$x_b \in \mathbb{N}$, initially 0
${y_0}_b \in \mathbb{N}$, initially 0
${y_1}_b \in \mathbb{N}$, initially 0

Transitions:

$bal_req_{p,b}$
Effect:
$x_b := x_b + 1$
$\mathbb{P}_b := \mathbb{P}_b \cup \{p\}$

$bal_ret(0)_{p,b}$
Precondition:
$p \in \mathbb{P}_b$
${y_0}_b < \lceil x_b/2 \rceil$
Effect:
${y_0}_b := {y_0}_b + 1$
$\mathbb{P}_b := \mathbb{P}_b - \{p\}$

$bal_ret(1)_{p,b}$
Precondition:
$p \in \mathbb{P}_b$
${y_1}_b < \lfloor x_b/2 \rfloor$
Effect:
${y_1}_b := {y_1}_b + 1$
$\mathbb{P}_b := \mathbb{P}_b - \{p\}$

Figure 4-2: Balancer I/O Automaton

4.3 Tree Structure

The tree of nodes is an infinite binary tree rooted at *Root*. The initial start state of the system can have a variety of values stored in shared memory, because the state of each node can be one of several values based on other nodes in the tree.

N is the collection of nodes in our system.

There are four immutable functions for every node $n \in N$, which behave exactly as the commonly understood definitions of these terms in any binary tree. Here **Left** and **Right** are the left and right children.

Functions:

Parent(n) $\in N_\perp$[1]
Left(n) $\in N$
Right(n) $\in N$
Sibling(n) $\in N$

We have to allow Parent(n) to take on $\perp$ because *Root* has no parent, and we can conveniently assign Parent(*Root*) $:= \perp$. What follows is a list of rules which these relationships satisfy, all of

[1] $\perp$ is another name for ***null*** or ***unknown***, and $A_\perp \triangleq A \cup \{\perp\}$

which satisfy the commonly understood definitions.

$$\text{Parent}(n) = \bot \implies n = Root \tag{4.5}$$

$$\text{Parent}(\text{Left}(n)) = \text{Parent}(\text{Right}(n)) \tag{4.6}$$

$$\text{Sibling}(\text{Left}(n)) = \text{Right}(n) \tag{4.7}$$

$$\text{Sibling}(\text{Right}(n)) = \text{Left}(n) \tag{4.8}$$

$$\text{Left}(n) = m \implies \text{Parent}(m) = n \tag{4.9}$$

$$\text{Right}(n) = m \implies \text{Parent}(m) = n \tag{4.10}$$

We also define two subsidiary relationships:

$\text{Anc}(n) \subseteq N$
$\text{Des}(n) \subseteq N$

These stand for the sets of Ancestors and Descendants. Here are the rules for determining these sets, which makes them satisfy their commonly understood definitions.

$$\text{Parent}(\text{Left}(n)) \in \text{Anc}(\text{Left}(n)) \tag{4.11}$$

$$\text{Parent}(\text{Right}(n)) \in \text{Anc}(\text{Right}(n)) \tag{4.12}$$

$$\text{Left}(n) \in \text{Des}(n) \tag{4.13}$$

$$\text{Right}(n) \in \text{Des}(n) \tag{4.14}$$

$$\text{Anc}(\text{Parent}(\text{Left}(n))) \subseteq \text{Anc}(\text{Left}(n)) \tag{4.15}$$

$$\text{Anc}(\text{Parent}(\text{Right}(n))) \subseteq \text{Anc}(\text{Right}(n)) \tag{4.16}$$

$$\text{Des}(\text{Left}(n)) \subseteq \text{Des}(n) \tag{4.17}$$

$$\text{Des}(\text{Right}(n)) \subseteq \text{Des}(n) \tag{4.18}$$

$$n \notin \text{Anc}(n) \tag{4.19}$$

$$n \notin \text{Des}(n) \tag{4.20}$$

$$\text{Anc}(Root) = \emptyset \tag{4.21}$$

Now, there are two additional immutable values for each node $n \in N$

Values:

$\mathsf{Level}(n) \in \mathbb{N}$
$\mathsf{Init}(n) \in \mathbb{N}$

These describe for each node $n \in N$ its depth in the tree and its initial value as a counter. Here are the formulas which generate these values.

$$\mathsf{Level}(Root) = 0 \tag{4.22}$$

$$\mathsf{Level}(\mathsf{Left}(n)) = \mathsf{Level}(n) + 1 \tag{4.23}$$

$$\mathsf{Level}(\mathsf{Right}(n)) = \mathsf{Level}(n) + 1 \tag{4.24}$$

$$\mathsf{Init}(Root) = 0 \tag{4.25}$$

$$\mathsf{Init}(\mathsf{Left}(n)) = \mathsf{Init}(n) \tag{4.26}$$

$$\mathsf{Init}(\mathsf{Right}(n)) = \mathsf{Init}(n) + 2^{\mathsf{Level}(n)} \tag{4.27}$$

Finally, each node consists of several shared variables which can be accessed throughout the entire automaton.

Shared variables:

$Status \in S$, initially $Status \in S_0$
$Count \in \mathbb{N}$, initially $\mathsf{Init}(n)$
$Change \in \mathbb{N}$, initially $\mathsf{Init}(n)$
$Limit \in \mathbb{N}$, initially $\mathsf{Init}(n)$
$ID \in \mathbb{N}$, initially 0
$PID \in \mathbb{N}$, initially 0
$toggle \in \mathbb{N} \times \{0,1\}$, initially $\{(n,0) : n \in \mathbb{N}\}$

The *Status* variable controls how the processors pass through the tree. Here are the definitions:

S is the collection of possible states for a node and S_0 is the collection of possible initial states for a node.

$S := \{\mathsf{Balancer}, \mathsf{Counter}, \mathsf{Counter_Limit}, \mathsf{Off}\}$

$S_0 := \{\mathsf{Balancer}, \mathsf{Counter}, \mathsf{Off}\}$

4.3.1 Restrictions on *Status*

Here are the restrictions on the initial value of *Status* as the node fits into the tree structure.

$$Root.Status \neq \mathsf{Off} \tag{4.28}$$

$$\mathsf{Parent}(n).Status = \mathsf{Counter} \implies n.Status = \mathsf{Off} \tag{4.29}$$

$$\mathsf{Parent}(n).Status = \mathsf{Off} \implies n.Status = \mathsf{Off} \tag{4.30}$$

$$n.Status \neq \mathsf{Off} \iff (\mathsf{Parent}(n) = \bot) \vee (\mathsf{Parent}(n).Status = \mathsf{Balancer}) \tag{4.31}$$

We now define the indexing scheme for the composition with the balancer automata. $B = N \times \mathbf{N}$. We know that N and $\mathbf{N}$ are countable, so a simple dove-tailing argument shows that B is countable. We will refer in the automaton to $b \in B$ as $b =< n, i >$, where $n \in N, i \in \mathbf{N}$.

Finally, before we finish defining the automata, it will be useful to define some macros to simplify the readability and save the mathematics for the proofs. Their definitions will be clearly explained throughout their use in the proof.

Macros:

$$\mathsf{Increment}(n) = 2^{\mathsf{Level}(n)}$$

$$\mathsf{Toggle_Init}(n, v) = \mathsf{Mod}(\frac{v - \mathsf{Init}(n)}{2^{\mathsf{Level}(n)}}, 2)$$

$$\mathsf{Fold}(n, vL, vR) = \mathsf{Max}(vL, vR) - 2^{\mathsf{Level}(n)}$$

$$\mathsf{UnFold}(n, v) = \begin{cases} v & \mathsf{Mod}(v - \mathsf{Init}(n), 2^{\mathsf{Level}(n)}) \\ v + 2^{\mathsf{Level}(n)-1} & \mathsf{Otherwise} \end{cases}$$

4.4 Main I/O Automaton

Figures 4-3 and 4-4 describe the main I/O automaton. When the change actions, shown in Figure 4-4, do not act, the steps taken by the automaton are clear. A process, upon invoking the input request, goes through balancers, (formally invoking balancer automata) balancing out throughout the tree until it reaches a counter, upon which it increments the counter, obtains a value, and returns an output value to its user.

Now, if a change action acts, changing the states of the tree, then the process' actions become more complicated. When a process originally traverses the tree, it records version

Signature:

Input:

$request_p$

Output:

$return(V)_p, V \in \mathbb{N}$

States:

$node_p \in N_\perp$, initially $\perp$
$bal_wait_p \in \{\text{True}, \text{False}\}$, initially False
$answer_val_p \in \mathbb{N}_\perp$, initially $\perp$
$ID_p \in N \times \mathbb{N}_\perp$, initially $\{(n, \perp) : n \in N\}$

Shared Variables:

Root, the root of tree
Node data structure:
$Status \in S$, initially $Status \in S_0$
$Count \in \mathbb{N}$, initially $\text{Init}(n)$
$Change \in \mathbb{N}$, initially $\text{Init}(n)$
$Limit \in \mathbb{N}$, initially $\text{Init}(n)$
$ID \in \mathbb{N}$, initially 0
$PID \in \mathbb{N}$, initially 0
$toggle \in \mathbb{N} \times \{0, 1\}$, initially $\{(n, 0) : n \in \mathbb{N}\}$

Transitions:

$request_p$
  Effect:
    $node_p := Root$
    $answer_val_p := \perp$
    $bal_wait_p := \text{False}$
    $ID_p[Root] := Root.ID$

$bal_req_{p,<n,i>}$
  Precondition:
    $answer_val_p = \perp$
    $bal_wait_p = \text{False}$
    $n_p.Status = \text{Balancer}$
    $i = n.PID$
    $n.ID = ID_p[n]$
    $n \in \text{Anc}(node_p) \cup \{node_p\}$
    $\forall n' \in \text{Des}(n) \cap \text{Anc}(node_p) \cup node_p$
      $n'.ID \neq ID_p[n']$
    Effect:
      $node_p := n$
      $ID_p[\text{Left}(node_p)] := node_p.PID$
      $ID_p[\text{Right}(node_p)] := node_p.PID$
      $bal_wait_p := \text{True}$

$bal_ret(V)_{p,<n,i>}$
  Effect:
    if $V = toggle_p[i]$
      $node_p := \text{Left}(node_p)$
    else $node_p := \text{Right}(node_p)$
    $bal_wait_p := \text{False}$

$return(V)_p$
  Precondition:
    $V = answer_val_p \neq \perp$
  Effect:
    $answer_val_p := \perp$

$inc_count(n)_p$
  Precondition:
    $answer_val_p = \perp$
    $bal_wait_p = \text{False}$
    $n.Status = \text{Counter_Limit}$ or Counter
    $n.ID = ID_p[n]$
    $n \in \text{Anc}(node_p) \cup \{node_p\}$
    $\forall n' \in \text{Des}(n) \cap \text{Anc}(node_p) \cup node_p$
      $n'.ID \neq ID_p[n']$
  Effect:
    $answer_val_p := n.Count$
    $n.Count := n.Count + \text{Increment}(n)$
    if $n.Count = n.Limit$
      $n.Status := \text{Off}$
      $n.ID := n.ID + 1$

fold(n) and *unfold*(n) are in Figure 4-4.

Tasks:

$\{return(V)_p : V \in \mathbb{N}\}$

Figure 4-3: Main I/O Automaton

```
fold(n)
  Effect:
    if n.Status = Balancer and
        Left(n).Status = Counter and
        Right(n).Status = Counter and
        (Left(n).Count ≠ Left(n).Change or
        Right(n).Count ≠ Right(n).Change)
      n.Status := Counter
      n.Count := Fold(n, Left(n).Count,
        Right(n).Count)
      n.Change := n.Count
      n.PID := n.PID + 1
      ∀m ∈ {Left(n), Right(n)}
        if m.Count < n.Count
          m.Status := Counter_Limit
          m.Limit := n.Count
        else
          m.ID := m.ID + 1
          m.Status := Off
```

```
unfold(n)
  Effect:
    if n.Status = Counter and
        Left(n).Status = Off and
        Right(n).Status = Off and
        n.Count ≠ n.Change
      n.Status := Balancer
      n.PID := n.PID + 1
      n.toggle[n.PID] := Toggle_Init(n, n.Count)
      ∀m ∈ {Left(n), Right(n)}
        m.ID := m.ID + 1
        m.Status := Counter
        m.Count := UnFold(m, n.Count)
        m.Change := m.Count
```

Figure 4-4: Folding and UnFolding *Change* Actions

information about the nodes it visits and receives forecasted information about nodes it will visit. If, upon arrival at a node, the forecasted information is now out of date, the processor will go back up the tree until it finds a point at which its recorded information is correct, and start again down the tree. If it eventually reaches a counter with correctly forecasted information and can successfully access it, then it can exit with its obtained value.

This automaton is not program counter structured. It focuses entirely on the flow of a processor's thread of control throughout the tree. $node_p$ contains the node that the processor has arrived at. If the forecasted information is correct, this is the node that a processor would access. If it was incorrect, then this node serves as the base from which it searches up the tree to find a node which it can access. $answer_val_p$ contains the value that it has obtained. It is initially $\perp$, a precondition for most of the actions, and once a value is obtained, the only action which is enabled is $return(v)_p$. The preconditions for $bal_req_{p,<n,i>}$ and $inc_count(n)_p$ are similar except for the *Status* variable, which forces processes at a **Balancer** node away from $inc_count(n)_p$ and processes at a **Counter** or **Counter_Limit** node away from $bal_req_{p,<n,i>}$. Finally, once a process accesses a balancer automaton, bal_wait_p goes true to keep that process automaton from having any other actions enabled until the balancer automaton returns. Similarly, if $inc_count(n)_p$ is enabled and occurs, it sets a value to $answer_val_p$, which only enables the output action.

The IDs are the version information we mentioned earlier. This is necessary for the changes that can occur. Figure 4-4 shows the two allowed changes. Either two siblings who are counters *fold* into their parent, making their parent a counter, or a counter with two off children *unfolds* into a balancer and two children counters. IDs are necessary for the following reason. A balancer sends processes to children nodes to evenly maintain a split. If a balancer folds, the processes enabled on its children need to return to the parent to obtain a value. However, if the new counter then unfolds, then the processes that remained inactive will see the same state configuration they saw before and will attempt to access the counter they were originally sent to access. This access could not be predicted by the new balancer, because it does not know whether the process saw the change. This requires a versioning scheme to be added.

Each node's *ID* value really refers to the number of changes that a node has gone through in conjunction with its parent and sibling. Each parent keeps a copy of the children's *ID* value in their *PID* value. This is the forecasting indicated before. When a processor balances through a balancer, it records the *PID* value as the predicted values of either of the children it might arrive at. Since *Root* has no parent, its ID never changes. In order to access a node for balancing or counting, the forecasted value has to agree with the node's current ID. If it does not, then the processor moves up the tree until it finds values that do agree, which always as a last resort is *Root*. This is the reason for the complicated precondition in the $inc_count(n)_p$ and $bal_req_{p,<n,i>}$ actions. As soon as a change is made, these preconditions allow a processor to immediately "move" up the tree, by enabling the processor on the correct node. If a processor is stuck inside a balancer, then as soon as it leaves the balancer, it will move up the "tree" until it is enabled on the correct node. This is a more general model, and an implementation could safely do this by walking the processors up the tree until it finds values that agree, since these values can not decrease, hence never come back into agreement.

With that, we now prove the safety property hold for the composition of the main and balancer I/O automata and well-formed users.

Chapter 5

Automaton Verification

We now prove that the automaton composed of well-formed users and the main and balancer I/O automata satisfies the safety property. The safety property says that in any state of an execution, the values output are distinct and in the set $\{0, \ldots, k-1\}$, where k is the number of inputs.

We shall prove separately the distinctness property and the limit on the output values.

5.1 Distinctness

The goal here is to prove that the output values are distinct. However, there is a delay between the action in which the value is chosen and the output action. It would also be useful to formally keep track of the values that have been chosen for output over the entire system. To solve these problems, we shall introduce a history variable *Outputs* that contains all the values that have been chosen for output. Since a value could be chosen more than once (although this is what we will prove will not happen), we shall make *Outputs* a multi-set.

5.1.1 Multi-Sets

The multi-set M contains multiple copies of elements. Notationally, $M(i) = j$ if there are j occurrences of i in M. If $M(i) > 0$, then $i \in M$, otherwise $i \notin M$.

5.1.2 *Outputs* definition

The specification for *Outputs* is as follows.

Outputs multi-set of $\mathbb{N}$, initially $\emptyset$.

Any action that assigns a non-$\perp$ value v to *answer_val* also does the following:

$$Outputs := Outputs \cup \{v\}$$

We now justify why recording values assigned to $answer_val_p$ is equivalent to recording values that are the argument of the output action.

Lemma 5.1 If Output action $return(V)_p$ occurs in an execution, then in the last state of that execution, $V \in Outputs$ and $V \neq \perp$.

Proof: This follows from the type of the action and the definition of *Outputs*. ■

We will prove distinctness by proving invariants that restrict the values that $n.Count$ can obtain. Specifically, we will show that in every state of the execution, a value that $n.Count$ can obtain is not in *Outputs*.

First, we create the sets from which $n.Count$ can obtain values and prove that the values are restricted to these sets.

5.1.3 Value Sets

The binary tree has a recursive method of dividing the values it works with. Each node in the tree has a set from which it can hand out values. The *Root* node counts from the entire set **N**. We give a definition of $\mathsf{Values}(n)$ below:

$$\mathsf{ValueSet}(n,k) := \{k + i * 2^{\mathsf{Level}(n)} : i \geq 0\}$$
$$\mathsf{Values}(n) := \mathsf{ValueSet}(n, \mathsf{Init}(n))$$

We first show that siblings split their parent's values completely.

Lemma 5.2 $\mathsf{Values}(n) = \mathsf{Values}(\mathsf{Left}(n)) \cup \mathsf{Values}(\mathsf{Right}(n))$, $\mathsf{Values}(\mathsf{Left}(n)) \cap \mathsf{Values}(\mathsf{Right}(n)) = \emptyset$.

Proof:

$$\begin{aligned}\mathsf{Values}(n) &= \{\mathsf{Init}(n) + i * 2^{\mathsf{Level}(n)} : i \geq 0\}\\ \mathsf{Values}(\mathsf{Left}(n)) &= \{\mathsf{Init}(\mathsf{Left}(n)) + i * 2^{\mathsf{Level}(\mathsf{Left}(n))} : i \geq 0\}\\ \mathsf{Values}(\mathsf{Right}(n)) &= \{\mathsf{Init}(\mathsf{Right}(n)) + i * 2^{\mathsf{Level}(\mathsf{Right}(n))} : i \geq 0\}\end{aligned}$$

We can rewrite the last two using equations 4.23, 4.24, 4.26, 4.27 as follows:

$$\begin{aligned}\mathsf{Values}(\mathsf{Left}(n)) &= \{\mathsf{Init}(n) + 2i * 2^{\mathsf{Level}(n)} : i \geq 0\}\\ \mathsf{Values}(\mathsf{Right}(n)) &= \{\mathsf{Init}(n) + (2i+1) * 2^{\mathsf{Level}(n)} : i \geq 0\}\end{aligned}$$

Finally, we rewrite it as follows, substituting $j = 2i$ and $j = 2i + 1$.

$$\begin{aligned}\mathsf{Values}(\mathsf{Left}(n)) &= \{\mathsf{Init}(n) + j * 2^{\mathsf{Level}(n)} : j\mathsf{even}, j \geq 0\}\\ \mathsf{Values}(\mathsf{Right}(n)) &= \{\mathsf{Init}(n) + j * 2^{\mathsf{Level}(n)} : j\mathsf{odd}, j \geq 0\}\end{aligned}$$

This then satisfies both parts of the lemma. ■

Next, we show that siblings alternate the values they obtain from their parents, such that the distance between consecutive values for a child is double that of two consecutive values of its parent.

Lemma 5.3 For all $n \in N$, if $V \in \mathsf{Values}(n)$, then $V + 2^{\mathsf{Level}(n)} \in \mathsf{Values}(n)$, if $\mathsf{Parent}(m) = n$, either V or $V + 2^{\mathsf{Level}(n)} \in \mathsf{Values}(m)$, but not both, and if $V \in \mathsf{Values}(n)$, $V \notin \mathsf{Values}(\mathsf{Sibling}(n))$.

Proof: If $V \in \mathsf{Values}(n)$, then $\exists i$ *s.t.* $V = \mathsf{Init}(n) + i * 2^{\mathsf{Level}(n)}$. But, $V + 2^{\mathsf{Level}(n)} = \mathsf{Init}(n) + (i + 1) * 2^{\mathsf{Level}(n)} \implies V + 2^{\mathsf{Level}(n)} \in \mathsf{Values}(n)$. Now, by Lemma 5.2, if $V \in \mathsf{Values}(n)$, then either $V \in \mathsf{Values}(\mathsf{Left}(n))$ or $V \in \mathsf{Values}(\mathsf{Right}(n))$ but not both. However, the minimum difference between distinct elements in Values(m) is $2^{\mathsf{Level}(m)} = 2 * 2^{\mathsf{Level}(n)}$. But $(V + 2^{\mathsf{Level}(n)}) - V =$

$2^{\mathsf{Level}(n)} < 2 * 2^{\mathsf{Level}(n)}$. So, either V or or $V + 2^{\mathsf{Level}(n)} \in \mathsf{Values}(m)$, but not both. The final part of the lemma is true directly from Lemma 5.2. ■

We now show that nodes with no relationships have distinct value sets.

Lemma 5.4 $m \notin \mathsf{Anc}(n) \cup \mathsf{Des}(n) \cup \{n\} \implies \mathsf{Values}(m) \cap \mathsf{Values}(n) = \emptyset$.

Proof: If m is not an Ancestor or Descendant of n, and $m \neq n$, then there exists a least common ancestor a such that m and n both have a as an ancestor but do not share either of the children of a as an ancestor. Without loss of generality, assume that $\mathsf{Left}(a)$ is an ancestor of m and $\mathsf{Right}(a)$ is an ancestor of n. $\mathsf{Values}(m) \subseteq \mathsf{Values}(\mathsf{Left}(a))$ and $\mathsf{Values}(n) \subseteq \mathsf{Values}(\mathsf{Right}(a))$ by Lemma 5.2. But, by the same lemma, $\mathsf{Values}(\mathsf{Left}(a)) \cap \mathsf{Values}(\mathsf{Right}(a)) = \emptyset$, which implies that $\mathsf{Values}(m) \cap \mathsf{Values}(n) = \emptyset$. ■

We now prove our first main invariant. We show that $n.Count$ is always a value in its value set. Our all invariants refer to statements that are true in every reachable state of an execution of our system.

Invariant 5.5 $\forall n$, $n.Count \in \mathsf{Values}(n)$.

Proof: This is initially true, since $\forall n \in N$, $n.Count = \mathsf{Init}(n)$, and $\mathsf{Init}(n) \in \mathsf{Values}(n)$.

Now, we only need to consider the actions which change *Count* and inductively assume that this invariant was true for all $n \in N$ before the action occurred. There are three actions to consider: $inc_count(n)_p$, $fold(n)$, and $unfold(n)$.

$inc_count(n)_p$ can add $\mathsf{Increment}(n)$ to $n.Count$. Originally, $n.Count \in \mathsf{Values}(n)$ and $\mathsf{Increment}(n) = 2^{\mathsf{Level}(n)}$. By Lemma 5.3, this preserves the invariant.

Next, consider $fold(n)$. Here, $n.Count := \mathsf{Fold}(n, \mathsf{Left}(n).Count, \mathsf{Right}(n).Count)$. By induction, $\mathsf{Left}(n).Count \in \mathsf{Values}(\mathsf{Left}(n))$ and $\mathsf{Right}(n).Count \in \mathsf{Values}(\mathsf{Right}(n))$. By Lemma 5.2, this implies that $\mathsf{Left}(n).Count$ and $\mathsf{Right}(n).Count \in \mathsf{Values}(n)$. Now, $\mathsf{Fold}(n, vL, vR) = \mathsf{Max}(vL, vR) - 2^{\mathsf{Level}(n)}$. We know that $\mathsf{Max}(vL, vR) \in \mathsf{Values}(n)$. $\mathsf{Max}(vL, vR) \neq \mathsf{Init}(n)$ because $\mathsf{Max}(vL, vR) > \mathsf{Min}(vL, vR)$ (vL and vR are necessarily distinct by Lemma 5.2), and $\mathsf{Min}(vL, vR) \in \mathsf{Values}(n)$. So, $n.Count \in \mathsf{Values}(n)$ and is not equal to $\mathsf{Init}(n)$, so $n.Count - 2^{\mathsf{Level}(n)} \in \mathsf{Values}(n)$, and the invariant is preserved.

Finally, consider $unfold(n)$. The changes are $\mathsf{Left}(n).Count := \mathsf{UnFold}(\mathsf{Left}(n), n.Count)$

and $\mathsf{Right}(n).Count := \mathsf{UnFold}(\mathsf{Right}(n), n.Count)$. It is known that $n.Count \in \mathsf{Values}(n)$. Now, consider what $\mathsf{UnFold}(m, V)$ returns, where $n = \mathsf{Parent}(m)$. $\mathsf{Mod}(V - \mathsf{Init}(m), 2^{\mathsf{Level}(m)})$ is only equal to zero precisely when $\exists i$ *s.t.* $V = \mathsf{Init}(m) + i * 2^{\mathsf{Level}(m)}$, which means that $V \in \mathsf{Values}(m)$. However, by Lemma 5.3, if $V \notin \mathsf{Values}(m)$, then $V + 2^{\mathsf{Level}(m)-1} \in \mathsf{Values}(m)$, and the invariant is preserved. ■

5.1.4 *Status* Restrictions on Counters

We now turn our attention to the *Status* variable. We will show several invariants that restrict the possible enumeration of states for distinct nodes in the tree. In this section, we show that in any state of the execution, there is at most one **Counter** on any path from the root. We show here that in any state of the execution, if a node has a *Status* of a **Counter** or **Balancer** then all of its ancestors are **Balancers**.

Invariant 5.6 $\forall n$, $(n.Status = \mathsf{Counter}) \vee (n.Status = \mathsf{Balancer}) \implies \forall m \in \mathsf{Anc}(n), m.Status = \mathsf{Balancer}$.

Proof: All initial states of the execution satisfy this property by restriction 4.31. We now consider all actions that could possibly violate this invariant. Namely, we look at all changes of *Status* variables from or to **Counter** or **Balancer**. These actions are *fold*(n) and *unfold*(n).

fold(n) only acts if $n.Status = \mathsf{Balancer}$, $\mathsf{Left}(n).Status = \mathsf{Counter}$ and $\mathsf{Right}(n).Status = \mathsf{Counter}$. Now, $n.Status := \mathsf{Counter}$ and it originally satisfied the hypothesis of the invariant, so it still preserves it. Meanwhile, $\mathsf{Left}(n).Status$ and $\mathsf{Right}(n).Status$ change to either **Off** or **Counter_Limit** . They no longer satisfy the hypothesis of the invariant and vacuously preserve the invariant.

unfold(n) only acts if $n.Status = \mathsf{Counter}$, $\mathsf{Left}(n).Status = \mathsf{Off}$ and $\mathsf{Right}(n).Status = \mathsf{Off}$. Now, $n.Status := \mathsf{Balancer}$ and it originally satisfied the hypothesis of the invariant, so it still preserves it. Meanwhile, $\mathsf{Left}(n).Status$ and $\mathsf{Right}(n).Status$ change to **Counter** and since $n.Status := \mathsf{Balancer}$, the invariant is preserved. ■

We now do similar reasoning in the other direction. We show that if a node is not a **Balancer**, then all of its descendants are not **Counters**.

Invariant 5.7 $\forall n$, $n.Status \neq \mathsf{Balancer} \implies \forall m \in \mathsf{Des}(n), m.Status \neq \mathsf{Counter}$.

Proof: This is initially true in all initial states of the tree by restrictions 4.29 and 4.30. We now consider all possible actions that could possibly violate this invariant. Namely, we look at all changes of *Status* variables from or to Counter or Balancer. These actions are *fold*(n) and *unfold*(n).

fold(n) only acts if $n.Status$ = Balancer, Left$(n).Status$ = Counter and Right$(n).Status$ = Counter. Left$(n).Status$ and Right$(n).Status$ change to either Off or Counter_Limit from Counter; they inductively satisfied the invariant, so they satisfy the invariant after the action. $n.Status :=$ Counter and it originally violated the hypothesis of the invariant. Now it satisfies the hypothesis, and its children both satisfy the consequent, so it preserves the invariant.

unfold(n) only acts if $n.Status$ = Counter, Left$(n).Status$ = Off and Right$(n).Status$ = Off. Left$(n).Status$ and Right$(n).Status$ change to Counter from Off, so they still preserve the invariant. $n.Status :=$ Balancer, so it now violates the hypothesis, and vacuously preserves the invariant. ■

5.1.5 *Count* Properties

We are now equipped to show some properties about the *Count* variable.

We first show that if two nodes are Counters in the same state of the execution, their value sets do not cross.

Invariant 5.8 $\forall n, m,\ n \neq m$ if $(n.Status = \text{Counter}) \wedge (m.Status = \text{Counter}) \implies (n.Count \notin \text{ValueSet}(m, m.Count)) \wedge (m.Count \notin \text{ValueSet}(n, n.Count))$.

Proof: By Invariants 5.6 and 5.7, the fact that both n and m have *Status* Counter implies that $m \notin \text{Anc}(n) \cup \text{Des}(n) \cup \{n\}$. This implies by Lemma 5.4 that $\text{Values}(n) \cap \text{Values}(m) = \emptyset$. Finally, by Invariant 5.5, $n.Count \in \text{ValueSet}(n, n.Count) \subseteq \text{Values}(n)$ and $m.Count \in \text{Values}(m, m.Count) \subseteq \text{Values}(m)$, so the lemma holds. ■

We now show that $n.Count$ is always greater than or equal to $n.Change$, a variable that gets set whenever $n.Count$ gets set in a fold or unfold action.

Invariant 5.9 $\forall n,\ n.Count \geq n.Change$.

Proof: This is initially true, since $n.Count = n.Change = \mathsf{Init}(n)$. We now consider all changes of $n.Count$ and $n.Change$.

In $inc_count(n)_p$, $n.Count$ is incremented by $\mathsf{Increment}(n)$, which as a power of 2 is positive. Inductively, $n.Count \geq n.Change \implies n.Count + \mathsf{Increment}(n) \geq n.Change$.

Now, in $fold(n)$ and $unfold(n)$, both $n.Count$ and $n.Change$ are set to the same values. So, the invariant holds. ■

We now introduce the $n.Limit$ value, which is the value that caps the counting of a Counter_Limit from above.

Invariant 5.10 $\forall n$, $n.Limit \in \mathsf{Values}(n)$.

Proof: This is initially true, since $n.Limit = \mathsf{Init}(n)$. We now consider all changes of $n.Limit$.

The only action that changes this value is $fold(n)$. Consider n to be the child node whose $Limit$ gets assigned. $n.Limit := \mathsf{Parent}(n).Count = \mathsf{Fold}(\mathsf{Parent}(n), n.Count, \mathsf{Sibling}(n).Count)$. This occurs if $n.Count < \mathsf{Parent}(n).Count$. However, $\mathsf{Fold}(\mathsf{Parent}(n), vL, vR) = \mathsf{Max}(vL, vR) - 2^{\mathsf{Level}(\mathsf{Parent}(n))}$. Clearly, if $n.Count > \mathsf{Sibling}(n).Count$, then $n.Count > \mathsf{Parent}(n).Count$. So, in order for limit to be assigned, $n.Count < \mathsf{Sibling}(n).Count$. This means $\mathsf{Parent}(n).Count = \mathsf{Sibling}(n).Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))}$. However, from Lemma 5.3, given that $\mathsf{Sibling}(n).Count \in \mathsf{Values}(\mathsf{Sibling}(n))$ and $\mathsf{Sibling}(n).Count \in \mathsf{Values}(\mathsf{Parent}(n))$, $\mathsf{Sibling}(n).Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))} \in \mathsf{Values}(n)$. This means $\mathsf{Parent}(n).Count \in \mathsf{Values}(n)$. If $\mathsf{Parent}(n).Count > n.Count$, then $n.Limit := \mathsf{Parent}(n).Count$, and the invariant holds. ■

We show that a Counter_Limit always has the $n.Count$ lower than its limit value.

Invariant 5.11 $\forall n$, $n.Status = \mathsf{Counter_Limit} \implies n.Count < n.Limit$.

Proof: This is vacuously true for the initial state of the tree, since $\mathsf{Counter_Limit} \notin S_0$. We now look at all places where the $Status$ of n is changed to or from Counter_Limit and where $n.Count$ changes. These actions are $fold(n)$ and $inc_count(n)_p$.

In $fold(n)$, child n's $Status$ variable can only set to Counter_Limit if it is initially Counter and $n.Count < \mathsf{Parent}(n).Count$. However, if this occurs, then $n.Limit := \mathsf{Parent}(n).Count$, which preserves the invariant.

In $inc_count(n)_p$, $n.Count$ is incremented to the next value in $\mathsf{Values}(n)$. However, by Invariant 5.10, $n.Limit \in \mathsf{Values}(n)$ and we know that prior to this action, $n.Count < n.Limit$. This implies that after the assignment, $n.Count \leq n.Limit$. However, if $n.Count = n.Limit$ then $n.Status := \mathsf{Off}$. Either way, the invariant is preserved. ■

5.1.6 *Count* Never Decreases

Now, we would like to prove that $n.Count$ never decreases. To do this, we add to each node a variable *CountMax* that keeps the maximum value that $n.Count$ has ever taken on. Here is its definition:

$n.CountMax \in \mathbb{N}$, initially $\mathsf{Init}(n)$.

Any action that assigns a value to $n.Count$ also performs the following bookkeeping change:

$$n.CountMax := \mathsf{Max}(n.CountMax, n.Count)$$

We have to take into account the changes that occur from an increment, fold, or unfold. The following invariant will set up this proof for the unfold action.

Invariant 5.12 $\forall n$, $n.Status = \mathsf{Off} \implies n.Count \leq \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$ and $n.Status = \mathsf{Counter_Limit} \implies n.Count < \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$ and $n.Status = \mathsf{Counter_Limit} \implies n.Limit \leq \mathsf{Parent}(n).Count$.

Proof: This is initially true since $\forall n \neq Root, \mathsf{UnFold}(n, \mathsf{Init}(\mathsf{Parent}(n))) = \mathsf{Init}(n)$.

First, examine $inc_count(\mathsf{Parent}(n))_p$. This action can only increase the value of UnFold, which preserves the invariant. Now consider $inc_count(n)_p$. It won't change $n.Count$ if $n.Status = \mathsf{Off}$, which preserves the invariant. If $n.Status = \mathsf{Counter_Limit}$, it increments $n.Count$. Now, we know from Invariants 5.11 and 5.10 that the value of $n.Count < n.Limit$ and that $n.Limit \in \mathsf{Values}(n)$, so the new value of $n.Count$ satisfies $n.Count \leq n.Limit$. If $n.Count = n.Limit$, then $n.Status := \mathsf{Off}$, and by the induction, $n.Limit \leq \mathsf{Parent}(n).Count \leq \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$, so the invariant is preserved. If $n.Count < n.Limit$, then by the previous, $n.Count < \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$ and the invariant is preserved.

Now, consider $fold_p$. We only need to consider the case where we call $fold(\mathsf{Parent}(n))$, since

the children become Off or $\mathsf{Counter_Limit}$. The assignment for the *Count* is shown below. Since in most cases $n.Status := \mathsf{Off}$, we need to show that

$$\begin{aligned} \mathsf{Parent}(n).Count &:= \mathsf{Fold}(\mathsf{Parent}(n), n.Count, \mathsf{Sibling}(n).Count) \\ n.Count &\leq \mathsf{UnFold}(n, \mathsf{Parent}(n).Count) \end{aligned}$$

If $n.Count > \mathsf{Sibling}(n).Count$, then $\mathsf{Fold}(\ldots) = n.Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))}$. But $n.Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))} \notin \mathsf{Values}(n)$ by Lemma 5.3, so UnFold adds $2^{\mathsf{Level}(n)-1}$ to it, and out comes $n.Count$. If $n.Count < \mathsf{Sibling}(n).Count$, then $\mathsf{Fold}(\ldots) = \mathsf{Sibling}(n).Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))}$ and it is clear that $n.Count \leq \mathsf{Sibling}(n).Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))}$. So, if $n.Status := \mathsf{Off}$, the invariant is preserved. Now, if $n.Status := \mathsf{Counter_Limit}$, then necessarily $n.Count < \mathsf{Parent}(n).Count$ and $n.Limit = \mathsf{Parent}(n).Count$ and the invariant is preserved.

The only other action to examine is *unfold*(n). Now, if $n.Status := \mathsf{Counter}$, the invariant is vacuously true. However, if $\mathsf{Parent}(n) \neq Root$, consider *unfold*$(\mathsf{Parent}(\mathsf{Parent}(n)))$. Any action implies that $\mathsf{Parent}(n).Status = \mathsf{Off}$, which means $n.Count \leq \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$. So, the assignment $n.Count := \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$ preserves the invariant with respect to n, since $\mathsf{Parent}(n).Count$ did not decrease. ■

We now need to show that $n.Count$ is always less than or equal to the value it would obtain in a fold action.

Invariant 5.13 $\forall n$, $n.Status = \mathsf{Balancer} \implies$
$n.Count \leq \mathsf{Fold}(n, \mathsf{Left}(n).Count, \mathsf{Right}(n).Count)$.

Proof: This is initially true since $\forall n$, $\mathsf{Fold}(n, \mathsf{Init}(\mathsf{Left}(n)), \mathsf{Init}(\mathsf{Right}(n))) = \mathsf{Init}(n)$.

First, examine *inc_count*$(n)_p$. This action is not enabled if $n.Status = \mathsf{Balancer}$. If we call this action on $\mathsf{Left}(n)$ or $\mathsf{Right}(n)$, then incrementing their *Count* can only increase the right hand side of the inequality, which preserves the invariant.

The only other action to consider is *unfold*(n). $n.Status := \mathsf{Balancer}$, and the two children's *Count* get assigned UnFold's. The two UnFold's return the values $n.Count$ and $n.Count + 2^{\mathsf{Level}(n)}$ to the two *Count* variables. But, $\mathsf{Fold}(n, vL, vR)$ returns $\mathsf{Max}(vL, vR) - 2^{\mathsf{Level}(n)} = n.Count$. So, $n.Count \leq n.Count$, and the invariant is preserved. ■

We are now ready to show that $n.Count$ never decreases.

Invariant 5.14 $\forall n$, $n.Count = n.CountMax$.

Proof: This is initially true since they are both equal to $\mathsf{Init}(n)$.

Consider the three actions that change $n.Count$. $inc_count(n)_p$ increases $n.Count$, so by the inductive assumption, $n.CountMax := n.Count$ and the invariant is preserved. $fold(n)$ can only assign $\mathsf{Fold}(n, \mathsf{Left}(n).Count, \mathsf{Right}(n).Count)$ to $n.Count$ if $n.Status = \mathsf{Balancer}$, but by invariant 5.13, this assignment preserves the invariant. Finally, $unfold(\mathsf{Parent}(n))$ can only assign $\mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$ to $n.Count$ if $n.Status = \mathsf{Off}$, but by invariant 5.12, this preserves the invariant. ■

5.1.7 Counter_Limit *Count* properties

We now need to show that if there exists a Counter_Limit in the tree, its *Count* variable is restricted by a Counter ancestor.

Invariant 5.15 $\forall n, m$, $((n.Status = \mathsf{Counter_Limit}) \wedge (m.Status = \mathsf{Counter}$ or $\mathsf{Counter_Limit}) \wedge (m \in \mathsf{Anc}(n)) \implies n.Count < m.Count$.

Proof: Invariant 5.12 provides two inequalities regarding Off and Counter_Limit nodes. If $n.Status = \mathsf{Off}$, $n.Count \leq \mathsf{UnFold}(n, \mathsf{Parent}(n).Count)$. We can restate this as:

$$\mathsf{Parent}(n).Count \geq n.Count - 2^{\mathsf{Level}(\mathsf{Parent}(n))}$$

If $n.Status = \mathsf{Counter_Limit}$, by the same invariant, $n.Limit \leq \mathsf{Parent}(n).Count$, from invariant 5.11, $n.Count < n.Limit$, and from invariant 5.10, $n.Limit \in \mathsf{Values}(n)$, so $n.Count + 2^{\mathsf{Level}(n)} \leq n.Limit$, and we can conclude

$$\mathsf{Parent}(n).Count \geq n.Count + 2^{\mathsf{Level}(n)}$$

Now consider the nearest ancestor m whose *Status* is Counter or Counter_Limit . The pathway from n to m consists of nodes $n_1, n_2, \ldots, n_j$ *s.t.* $\mathsf{Parent}(n) = n_1, \ldots, \mathsf{Parent}(n_j) = m$. By invariant 5.7, $n_i.Status = \mathsf{Off}$. So, $m.Count \geq n_j.Count - 2^{\mathsf{Level}(m)}$ and $\forall i > 1$, $n_i.Count \geq$

$n_{i-1}.Count - 2^{\mathsf{Level}(n_i)}$. So, putting them all together, $m.Count \geq n_{i-1}.Count - (2^{\mathsf{Level}(n_2)} + \ldots + 2^{\mathsf{Level}(m)})$. Finally, since $n.Status$ = Counter_Limit, $n_1.Count \geq n.Count + 2^{\mathsf{Level}(n)}$, and we can finally relate $m.Count$ and $n.Count$ by the inequality $m.Count \geq n.Count + 2^{\mathsf{Level}(n)} - (2^{\mathsf{Level}(n_2)} + \cdots + 2^{\mathsf{Level}(m)})$. However, it is a basic mathematical fact that $2^k > 1 + \cdots + 2^{k-1}$, and Level($n$) is greater than any of the other levels in the expression. So, $2^{\mathsf{Level}(n)} > (2^{\mathsf{Level}(n_2)} + \cdots + 2^{\mathsf{Level}(m)})$, and $2^{\mathsf{Level}(n)} - (2^{\mathsf{Level}(n_2)} + \cdots + 2^{\mathsf{Level}(m)}) > 1$, which makes $m.Count \geq n.Count + 1$, or $m.Count > n.Count$, which satisfies the lemma.

Finally, if $m.Status$ = Counter, then by invariant 5.6, all ancestors of m are Balancers. if $m.Status$ = Counter_Limit, then we can re-apply the lemma to inductively finish the proof of this lemma. ■

5.1.8 Final *Count* Properties

We are now near completion of the distinctness proof. We need to show that the values that any Counter or Counter_Limit can take on do not ever intersect with each other.

First, we show that any value in the value set of a Counter can not be taken on by the *Count* variable of any other Counter or Counter_Limit node.

Invariant 5.16 $\forall n, m,\ n \neq m$, ($n.Status$ = Counter)$\wedge$($m.Status$ = Counter or Counter_Limit) $\Longrightarrow$ $m.Count \notin$ ValueSet($n, n.Count$).

Proof: If m and n have Counter as their *Status*, then this lemma is proved by lemma 5.8. Now consider when $m.Status$ = Counter_Limit. There are two cases, either $n \in$ Anc(m) or $n \notin$ Anc(m). If $n \notin$ Anc(m), we know that $n \notin$ Des(n) by lemma 5.7. By lemma 5.4, Values(m)$\cap$ Values(n) $= \emptyset$, and $m.Count \notin$ ValueSet($n, n.Count$).

Now, consider the case that $n \in$ Anc(m). By invariant 5.15, $m.Count < n.Count$, and $m.Count \notin$ ValueSet($n, n.Count$), so the lemma holds. ■

Then, we show that any value in the value set of a Counter_Limit can not be taken on by the *Count* variable of any other Counter or Counter_Limit node.

Invariant 5.17 $\forall n, m,\ n \neq m$, ($n.Status$ = Counter_Limit)$\wedge$
($m.Status$ = Counter or Counter_Limit) $\Longrightarrow$

$m.Count \notin \mathsf{Values}(n, n.Limit) - \mathsf{ValueSet}(n, n.Count)$.

Proof: There are three cases. Either $n \in \mathsf{Anc}(m)$, $n \in \mathsf{Des}(m)$, or $n \notin \mathsf{Anc}(m) \cup \mathsf{Des}(m)$. Consider the last case. By lemma 5.4, $\mathsf{Values}(m) \cap \mathsf{Values}(n) = \emptyset$, and $m.Count \notin \mathsf{Values}(n, n.Limit) - \mathsf{ValueSet}(n, n.Count)$.

Now, consider the case that $n \in \mathsf{Des}(m)$. By lemma 5.15, $n.Count < m.Count$ This is true for all values of $n.Count < n.Limit$, thus $n.Limit \leq m.Count$. This implies then that $m.Count \notin \mathsf{Values}(n, n.Limit) - \mathsf{ValueSet}(n, n.Count)$.

Finally, consider the case that $n \in \mathsf{Anc}(m)$. We know from lemma 5.7 that $m.Status \neq$ Counter. So, $m.Status =$ Counter_Limit. However, m is a descendant of n, and so $m.Count < n.Count$. This implies that $m.Count \notin \mathsf{Values}(n, n.Limit) - \mathsf{ValueSet}(n, n.Count)$, and the lemma is proved. ■

We now show that any value that a Counter or Counter_Limit may hand out in a future state is not presently in the *Outputs* set.

Invariant 5.18 $\forall n, n.Status =$ Counter, $v \in \mathsf{ValueSet}(n, n.Count) \implies v \notin Outputs$, and $n.Status =$ Counter_Limit, $v \in \mathsf{ValueSet}(n, n.Limit) - \mathsf{ValueSet}(n, n.Count) \implies v \notin Outputs$.

Proof: Initially, this is true since $Outputs = \emptyset$.

Consider the actions that make assignments to $answer_val_p$ or change $n.Count$ or $n.Status$. First, look at $inc_count(n)_p$. $answer_val_p$ is assigned $n.Count$, $Outputs := Outputs \cup n.Count$, and $n.Count$ takes on the next value in $\mathsf{Values}(n)$. This only happens if the *Status* is Counter or Counter_Limit, and by invariants 5.16 and 5.17, this does not invalidate the invariant for any other node, preserving the invariant for this node.

$fold(n)$ makes n a counter and $n.Count := \mathsf{Fold}(n, \mathsf{Left}(n).Count, \mathsf{Right}(n).Count)$. Now, $\forall v \in \mathsf{ValueSet}(\mathsf{Left}(n), \mathsf{Left}(n).Count) \cup \mathsf{ValueSet}(\mathsf{Right}(n), \mathsf{Right}(n).Count), v \notin Outputs$. But assuming the assignment has occurred, we know

$$\mathsf{ValueSet}(n, n.Count) \subseteq \mathsf{ValueSet}(\mathsf{Left}(n), \mathsf{Left}(n).Count) \cup \mathsf{ValueSet}(\mathsf{Right}(n), \mathsf{Right}(n).Count)$$

so the invariant is preserved. When a child m has $m.Status :=$ Counter_Limit, it automatically satisfies the invariant by 5.17.

Finally, consider $unfold(n)$. It creates two child counters. Assuming $n = \mathsf{Parent}(m)$, $m.Count := \mathsf{UnFold}(m, n.Count)$. However, we know $\forall v \in \mathsf{Values}(n, n.Count), v \notin Outputs$, and $\mathsf{UnFold}(m, n.Count) \geq n.Count$, so $\mathsf{Values}(m, m.Count) \subseteq \mathsf{Values}(n, n.Count)$ and the invariant is preserved. ■

We can now prove that the value currently in the *Count* variable of a **Counter** or **Counter_Limit** is never in *Outputs*.

Invariant 5.19 $\forall n \in N$, $(n.Status = \mathsf{Counter}) \vee (n.Status = \mathsf{Counter_Limit}) \implies n.Count \notin Outputs$.

Proof: By definition, $n.Count \in \mathsf{ValueSet}(n, n.Count)$ and $n.Count \in \mathsf{ValueSet}(n, n.Limit) - \mathsf{ValueSet}(n, n.Count)$, so this holds by invariant 5.18. ■

5.1.9 Conclusion

We are now ready to conclude the distinctness proof. We just showed that $n.Count \notin Outputs$ in any reachable state where n is a **Counter** or **Counter_Limit**. Now we show that $n.Count$ is the only source of values for *Outputs*, which results in the lemma that every value that appears in *Outputs* has only 1 occurrence in the multi-set.

Lemma 5.20 $\forall v \in \mathrm{N}$, $Outputs(v) <= 1$

Proof: By definition, a value can only go in *Outputs* if it is assigned to $answer_val_p$. However, this only occurs in $inc_count(n)_p$, the value assigned is $n.Count$, and only if $(n.Status = \mathsf{Counter}) \vee (n.Status = \mathsf{Counter_Limit})$. But, by 5.19, $n.Count \notin Outputs$, and the action concludes by incrementing $n.Count$. Finally, $n.Count$ never decreases by 5.14, so this lemma holds. ■

This last lemma is strong enough to show our main theorem, since values can only be output if they are in *Outputs*.

Theorem 5.21 If $return(V_1)_{p_1}$ and $return(V_2)_{p_2}$ occur, $V_1 \neq V_2$.

Proof: If $return(V_1)_{p_1}$ and $return(V_2)_{p_2}$ occur, then $inc_count(n)_{p_1}$ and $inc_count(m)_{p_2}$ occurred, where $answer_val_{p_1} := V_1$ and $answer_val_{p_2} := V_2$. However, *Outputs* would have

recorded both of these transactions, and $V_1, V_2 \in \mathit{Outputs}$. However, for all v, $\mathit{Outputs}(v) \leq 1$, so $V_1 \neq V_2$. ■

5.2 Output Value Limit

If there have been only k input requests, we will prove that $\mathit{return}(V)_p$ can only occur if $V < k$. To be more precise, we add a history variable k accessible over the entire automata composition, give it an initial value 0, and place $k := k+1$ in $\mathit{request}_p$. We will then show that if $v \in \mathit{Outputs}$, $v < k$. To do this, we will show the following two things: every $v \in \mathit{Outputs}$ is bounded above by some existing $n.\mathit{Count}$ and every $n.\mathit{Count}$ is bounded in some way by k to guarantee that $v < k$.

We immediately show the first condition mentioned above.

Invariant 5.22 $\forall v \in \mathrm{N}$, $v \in \mathit{Outputs} \implies \exists n, v <= n.\mathit{Count} - 2^{\mathsf{Level}(n)}$.

Proof: This is initially true since $\mathit{Outputs} = \emptyset$.

We now consider when v is added to *Outputs*. This only occurs in $\mathit{inc_count}(n)_p$, the value assigned is $n.\mathit{Count}$, and $n.\mathit{Count}$ is incremented by $2^{\mathsf{Level}(n)}$. However, by invariant 5.14, $n.\mathit{Count}$ never decreases. So this invariant always holds. ■

We now have to prove the more difficult second condition. First, however, we present some necessary mathematical formulas.

5.2.1 Mathematical Facts

We must first give some mathematical facts about floors and ceilings, since they abound in the following proofs. In all the equations, $a \in \mathrm{N}$. We leave the proofs to the interested reader.

$$2\left\lceil\frac{a}{2}\right\rceil = \begin{cases} a & a \text{ even} \\ a+1 & a \text{ odd} \end{cases} \tag{5.23}$$

$$2\left\lfloor\frac{a}{2}\right\rfloor = \begin{cases} a & a \text{ even} \\ a-1 & a \text{ odd} \end{cases} \tag{5.24}$$

$$\left\lceil\frac{a+b}{2}\right\rceil = \begin{cases} \lceil\frac{a}{2}\rceil + \lceil\frac{b}{2}\rceil & a \text{ even} \\ \lceil\frac{a}{2}\rceil + \lfloor\frac{b}{2}\rfloor & a \text{ odd} \end{cases} \tag{5.25}$$

$$\left\lfloor\frac{a+b}{2}\right\rfloor = \begin{cases} \lfloor\frac{a}{2}\rfloor + \lfloor\frac{b}{2}\rfloor & a \text{ even} \\ \lfloor\frac{a}{2}\rfloor + \lceil\frac{b}{2}\rceil & a \text{ odd} \end{cases} \tag{5.26}$$

$$\left\lceil\frac{a+1}{2}\right\rceil = \begin{cases} \lceil\frac{a}{2}\rceil + 1 & a \text{ even} \\ \lceil\frac{a}{2}\rceil & a \text{ odd} \end{cases} \tag{5.27}$$

$$\left\lfloor\frac{a+1}{2}\right\rfloor = \begin{cases} \lfloor\frac{a}{2}\rfloor & a \text{ even} \\ \lfloor\frac{a}{2}\rfloor + 1 & a \text{ odd} \end{cases} \tag{5.28}$$

$$\left\lceil\frac{a}{2}\right\rceil - \left\lfloor\frac{a}{2}\right\rfloor = \begin{cases} 0 & a \text{ even} \\ 1 & a \text{ odd} \end{cases} \tag{5.29}$$

5.2.2 Limits and Value Sets Revisited

We now begin by adding an additional value set operator that lets us easily refer to the ith element of the value set. We define Limit functions which will be the necessary bound on $n.Count$ to keep it below k. Finally, we define Half functions which let us avoid the proliferation of floors and ceilings everywhere.

$$\begin{aligned}
\mathsf{Values}(n)[j] &:= \mathsf{Init}(n) + j * 2^{\mathsf{Level}(n)}, j \geq 0 \\
\mathsf{Limit}(n,k) &:= \begin{cases} k & n = \mathit{Root} \\ \left\lceil\frac{\mathsf{Limit}(\mathsf{Parent}(n),k)}{2}\right\rceil & n = \mathsf{Left}(\mathsf{Parent}(n)) \\ \left\lfloor\frac{\mathsf{Limit}(\mathsf{Parent}(n),k)}{2}\right\rfloor & n = \mathsf{Right}(\mathsf{Parent}(n)) \end{cases} \\
\mathsf{Half}_i(j) &:= \begin{cases} \lceil j/2\rceil & i = 0 \\ \lfloor j/2\rfloor & i = 1 \end{cases}
\end{aligned}$$

We now show that this Limit definition will be a suitable bound for $n.Count$ to keep it safely below k.

Lemma 5.30 $\mathsf{Values}(n)[\mathsf{Limit}(n,k)-1] < k$.

Proof: First, we give some equations which clearly come out of the definition of $\mathsf{Values}(n)[j]$.

$$\begin{aligned} \mathsf{Values}(\mathsf{Left}(n))[j] &= \mathsf{Values}(n)[2j] \\ \mathsf{Values}(\mathsf{Right}(n))[j] &= \mathsf{Values}(n)[2j+1] \end{aligned}$$

Now, we will calculate the value of $\mathsf{Values}(n)[\mathsf{Limit}(n,k)-1]$. If $n = Root$, then $\mathsf{Limit}(n,k) = k$ and $\mathsf{Values}(Root)[k-1] = k-1$, so the inequality holds for $Root$. We will now do an inductive proof indexed by the depth into the tree.

Assume $m = \mathsf{Parent}(n)$. There are two cases. Either $n = \mathsf{Left}(m)$ or $n = \mathsf{Right}(m)$. If $n = \mathsf{Left}(m)$, then

$$\begin{aligned} \mathsf{Values}(n)[\mathsf{Limit}(n,k)-1] &= \mathsf{Values}(m)[2(\left\lceil \frac{\mathsf{Limit}(m,k)}{2} \right\rceil - 1)] \\ &= \begin{cases} \mathsf{Values}(m)[\mathsf{Limit}(m,k)-2] & \mathsf{Limit}(m,k) \text{ even} \\ \mathsf{Values}(m)[\mathsf{Limit}(m,k)-1] & \mathsf{Limit}(m,k) \text{ odd} \end{cases} \end{aligned}$$

By induction, the inequality 5.30 holds for m, so it clearly holds for $\mathsf{Left}(m)$. Now, assume $n = \mathsf{Right}(m)$.

$$\begin{aligned} \mathsf{Values}(n)[\mathsf{Limit}(n,k)-1] &= \mathsf{Values}(m)[2(\left\lfloor \frac{\mathsf{Limit}(m,k)}{2} \right\rfloor - 1) + 1] \\ &= \begin{cases} \mathsf{Values}(m)[\mathsf{Limit}(m,k)-1] & \mathsf{Limit}(m,k) \text{ even} \\ \mathsf{Values}(m)[\mathsf{Limit}(m,k)-2] & \mathsf{Limit}(m,k) \text{ odd} \end{cases} \end{aligned}$$

By induction, the inequality 5.30 holds for m, so it holds for $\mathsf{Right}(m)$. ■

We now begin the long journey towards proving that this bound holds. First, some necessary invariants about IDs, states, and balancers.

5.2.3 *ID*, *Status*, and Balancer Properties

We now give several low level invariants that are needed. This first invariant shows that a Balancer can only have Balancers and Counters as children.

Invariant 5.31 $n.Status = \textsf{Balancer} \implies (\textsf{Left}(n).Status \neq \textsf{Off or Counter_Limit}) \wedge (\textsf{Right}(n).Status \neq \textsf{Off or Counter_Limit})$.

Proof: This is initially true from equation 4.31.

The only actions to consider are those that change *Status*. $fold(n)$ changes a Balancer to a Counter, so the invariant is vacuously preserved. $unfold(n)$ changes a Counter to a Balancer, but also changes both children to Counters, which preserves the invariant. $inc_count(n)_p$ can change $n.Status$ to Off, but only if it was previously a Counter_Limit, which by the contrapositive of the invariant, implies that $\textsf{Parent}(n).Status \neq \textsf{Balancer}$. This proves the invariant. ■

Invariant 5.32 $Root.ID = 0$.

Proof: This is initially true, and the only actions that change *ID* are $fold(n)$ and $unfold(n)$, which change children nodes' *ID*s. However, *Root* has no parent, so the invariant is preserved. ■

Invariant 5.33 $Root.Status \neq$ Off or Counter_Limit.

Proof: This is initially true. All actions that change $n.Status$ to Counter_Limit are acting on the child of a node. All actions that change $n.Status$ to Off either operate on the child of a node or change a Counter_Limit to Off. But *Root* has no parent, so the invariant is preserved. ■

This invariant shows that if a processor has the correctly forecasted ID information for a node, then the node cannot be Off.

Invariant 5.34 $ID_p[n] = n.ID \implies n.Status \neq \textsf{Off}$.

Proof: This is initially true since $\forall n, ID_p[n] = \perp$.

Now, $request_p$ sets $ID_p[Root] := Root.ID = 0$ which is always true by 5.32, but the consequent is always true by 5.33, so this preserves the invariant.

$bal_req_{p,<n,i>}$ sets $node_p := n$ and caches the *ID*'s of $\mathsf{Left}(n)$ and $\mathsf{Right}(n)$. But at this time, $n.Status = \mathsf{Balancer}$, and by invariant 5.31, neither child's *Status* is Off.

All other actions to consider add 1 to $n.ID$, but these actions invalidate the hypothesis of the invariant and trivially preserve the invariant. ■

Finally, we show that a processor is inside a balancer iff bal_wait_p is True.

Invariant 5.35 $\exists b, p \in \mathbb{P}_b \iff bal_wait_p = \mathsf{True}$.

Proof: This is initially true since $\forall b, \mathbb{P}_b = \emptyset$ and $\forall p, bal_wait_p = \mathsf{False}$.

The only actions to consider are $bal_req_{p,<n,i>}$ and $bal_ret(V)_{p,<n,i>}$. $bal_req_{p,<n,i>}$ adds p to $\mathbb{P}_{<n,i>}$ and sets $bal_wait_p := \mathsf{True}$. $bal_ret(V)_{p,<n,i>}$ removes p from $\mathbb{P}_{<n,i>}$ and sets $bal_wait_p :=$ False, and is only enabled when $p \in \mathbb{P}_{<n,i>}$. ■

5.2.4 Active Processor Tracking

We need to show that an active processor is either enabled on exactly one action in the main process automaton, or is inside a balancer automaton, waiting to be returned to the main process automaton. To define an active processor, we create a history variable *Active* that is defined as follows:

$Active \subseteq \{1, \dots, P\}$, initially $\emptyset$.

We add assignment 5.36 to $request_p$ and 5.37 to $return(v)_p$.

$$Active := Active \cup \{p\} \tag{5.36}$$

$$Active := Active - \{p\} \tag{5.37}$$

We now show that an active processor not in a balancer has only one action enabled in any one state.

Invariant 5.38 $(p \in Active) \wedge (bal_wait_p = \mathsf{False}) \implies$ exactly one action is enabled, and it is in $\{return(v)_p, bal_req_{p,<n,i>}, inc_count(n)_p : V, i \in \mathrm{N}, n \in N\}$.

Proof: Initially, this is true since $Active = \emptyset$.

Now, we consider every action. $request_p$ adds p to *Active*, sets $node_p$ to *Root* and sets $answer_val_p$ to $\perp$ and bal_wait_p to False. It also sets $ID_p[Root] := Root.ID = 0$ and by 5.32 and 5.33, $Root.ID = 0$ and $Root.Status \neq$ Off. So, p must be enabled on $bal_req_{p,<Root,i>}$ or $inc_count(Root)_p$.

$bal_req_{p,<n,i>}$ sets $bal_wait_p :=$ False, which vacuously preserves the invariant.

$bal_ret(V)_{p,<n,i>}$ is only enabled if $p \in \mathbb{P}_{<n,i>}$. But by invariant 5.35, this is true iff $bal_wait_p :=$ True, when this invariant is vacuously preserved. Now, when this action does occur, it sets $bal_wait_p :=$ False and sets $node_p$ to either Left($node_p$) or Right($node_p$). If the assigned node has a matching *ID*, then invariant 5.34 shows that $bal_req_{p,<node_p,i>}$ or $inc_count(node_p)_p$ will be enabled. If the *ID* is not the same, then a traversal up the tree of $node_p$'s ancestors will eventually reach a node n whose *ID* is the same (by invariant 5.32), and by 5.34, either $bal_req_{p,<n,i>}$ or $inc_count(n)_p$ will be enabled.

$inc_count(n)_p$ sets $answer_val_p$ to a non-$\perp$ value V, so $return(v)_p$ is enabled. All other actions in this set have a $answer_val_p = \perp$ precondition, so the invariant is preserved.

$return(v)_p$ outputs the value V, and removes p from *Active*, preserving the invariant.

$fold(n)$ does several things. It changes $n.Status$ from Balancer to Counter. So, any processors enabled on $bal_req_{p,<n,i>}$ are now enabled on $inc_count(n)_p$. If it changes a child m to a Counter_Limit, it keeps the same *ID* and all the processors enabled on $inc_count(m)_p$ remain enabled there. If it changes a child to Off, it increases the *ID*, and any processors enabled on $inc_count(m)_p$ can seek up the ancestors of m until they find a node n' on which $n'.ID = ID_p[n']$, and by 5.34, either $inc_count(n')_p$ or $bal_req_{p,<n',i>}$ will be enabled.

$unfold(n)$ changes a Counter to a Balancer. So, all processors that had $inc_count(n)_p$ enabled now have $bal_req_{p,<n,i>}$ enabled. There were no processors enabled on either child, since they were Off, and hence had an *ID* distinct from that of any cached copy. ■

5.2.5 *ID* tracking

We now give some properties about *ID*'s and *PID*'s.

Lemma 5.39 $\forall n, n.ID$ and $n.PID$ do not decrease.

Proof: These are trivial, since all changes increment the variables by 1. ■

We now show the correlation between a parent's PID and a child's ID.

Invariant 5.40 $\forall n \neq Root, n.Status = \text{Counter_Limit} \implies n.ID + 1 = \text{Parent}(n).PID$ and $n.Status \neq \text{Counter_Limit} \implies n.ID = \text{Parent}(n).PID$.

Proof: This is initially true, since $\forall n, n.ID = n.PID = 0$.

We now consider all actions that change this. *unfold*(n) increments $n.PID$, $\text{Left}(n).ID$, and $\text{Right}(n).ID$, and Counter_Limit's are not involved, so the invariant is preserved. *unfold*(n) increments $n.PID$, and if a child m becomes Off, increments $m.ID$, preserving the invariant. The alternative is to make the child a Counter_Limit, which preserves the invariant. Finally, $inc_count(n)_p$ increments $n.ID$ only if it is a Counter_Limit, changes it to Off, and the invariant is preserved. ■

5.2.6 *PathID* tracking

We now introduce another history variable that clears up the confusion inherent in the ID structure. Each node has a *PathID* array that indicates what *version* of the tree that node currently belongs to. If a node is a Balancer or Counter, then this history variable contains the current IDs for all of its ancestors. If a node is a Counter_Limit, then this history variable contains the old IDs that were around when the node became a Counter_Limit. This is important, because processors enabled on a node will have their ID_p array equal to this array, and hence, equal to each other. Here is its definition.

$n.PathID \subseteq N \times \mathbb{N}$, initially $\{(n', 0) : n' \in \text{Anc}(n') \cup \{n\}\}$.

We perform assignment 5.41 wherever we increment $m.ID$ (*fold*(n) and $inc_count(n)_p$ have this case). We also perform assignment 5.42 in the end of *unfold*(n), where the code for m is performed for $m = \text{Left}(n)$ and $m = \text{Right}(n)$.

$$m.PathID[m] := m.ID \tag{5.41}$$

$$\forall n' \in \text{Anc}(m), m.PathID[n'] := n.PathID[n'] \tag{5.42}$$

We now show that a Counter or Balancer node's *PathID* variable has ID information consistent with its ancestors IDs in any state.

Invariant 5.43 $\forall n, (n.Status = \text{Counter or Balancer}) \implies \forall n' \in \text{Anc}(n) \cup \{n\}, n'.ID = n.PathID[n']$.

Proof: This is initially true since $\forall n, n.ID = 0$, and $\forall n' \in \mathsf{Anc}(n), n.PathID[n'] = 0$.

We now consider any changes of *ID*'s, the history variable, or *Status*. *fold*(n) turns a **Balancer** into a **Counter** and does not change the parent's *ID*, so it still satisfies the invariant. The two children become **Counter_Limit** or **Off**, so no longer satisfy the hypothesis, preserving the invariant.

unfold(n) turns a **Counter** into a **Balancer** and as above, preserves the invariant. The two children become **Counters**, so they now satisfy the hypothesis. However, each child receives a full copy of n.**PathID** which inductively satisfies the invariant, and then fills in its own value with the new *ID* assigned it, so it satisfies the invariant, and completes this proof. ■

We now show that a processor who has the current *ID* for a **Counter** or **Balancer** has the correct *ID* for its parent node.

Invariant 5.44 $\forall n, (n.Status = \text{Counter or Balancer}) \wedge (ID_p[n] = n.ID) \implies (n = Root) \vee (ID_p[\text{Parent}(n)] = \text{Parent}(n).ID)$.

Proof: This is initially true since $\forall n, ID_p[n] = \perp$.

request$_p$ sets $ID_p[Root] = Root.ID$, and this equality and the rest of the hypothesis is invariant for *Root*, and always preserves the invariant by the first part of the consequent.

We now consider the actions that change *Status* or *ID*. $bal_req_{p,<n,i>}$ sets $ID_p[\text{Left}(n)] := \text{Left}(n).ID$ and $ID_p[\text{Right}(n)] := \text{Right}(n).ID$, so if either child is a **Counter** or **Balancer**, n has $ID_p[n] = n.ID$, and this preserves the invariant.

$inc_count(n)_p$ increments $n.ID$ if $n.Status = \text{Counter_Limit}$, which vacuously satisfies the invariant. By 5.7, no descendants of n are **Counter** or **Balancer**, so there is no danger in violating any node's consequent.

fold(n) changes $n.Status$ from **Balancer** to **Counter**, and remains preserving the invariant. The children turn to **Counter_Limit** or **Off** , and so vacuously preserve the invariant.

unfold(n) changes $n.Status$ from a **Counter** to a **Balancer**, and remains preserving the invariant. Both children become **Counters**, but their *ID*'s increase, and since these values do not decrease (lemma 5.39) and values in $ID_p[n']$ come from $n'.ID$, no processor p could have $ID_p[m] = m.ID$, where $\text{Parent}(m) = n$. ■

We can now show this simple but useful lemma which shows that a processor that is enabled on a Balancer or Counter node has ID records that are in agreement with the node's *PathID*.

Lemma 5.45 $\forall n, (n.Status = \textsf{Counter or Balancer}) \wedge (ID_p[n] = n.ID) \implies$
$\forall n' \in \textsf{Anc}(n), (ID_p[n'] = n.PathID[n'])$

Proof: By invariant 5.43, $n.PathID$ contains the correct *ID*'s for its ancestors if it is a Balancer or Counter. But by invariant 5.44, if a processor p has the correct *ID* for a node, it also has the correct *ID* for it's parent. By invariant 5.6, the ancestors of a Balancer or Counter are all Balancers, so we can recurse this condition up the tree to conclude this proof. ■

Now, all we have left to show in this section is that a variant of the above lemma holds when $n.Status = \textsf{Counter_Limit}$.

Invariant 5.46 If $n.Status = \textsf{Counter}$ or $\textsf{Counter_Limit}$ and $p \in \{p : p \in \mathbb{P}_{<\textsf{Parent}(n), n.ID>}\} \cup \{p : inc_count(n')_p$ is enabled and $ID_p[n] = n.ID, n' \in \{n\} \cup \textsf{Des}(n)\}$, then $\forall n'' \in \textsf{Anc}(n), ID_p[n''] = n.PathID[n'']$.

Proof: This is initially true since there are no processors enabled on any actions.

We now consider all actions. *request* would only affect this if $Root.Status = \textsf{Counter}$, but $Root.ID = 0$ always and so does $Root.PathID[Root]$, so the invariant is preserved.

$bal_req_{p,<n,i>}$ affects this by its output action placing p in $\mathbb{P}_{n,n.PID}$. But, in order for this action to be enabled, $n.Status = \textsf{Balancer}$ which by invariant 5.31 implies that both children are Counters. But, a child m in this situation has $m.ID = n.PID$, and so the invariant needs to be withheld. However, this action sets $ID_p[m] := m.ID$ for both children, and we then know from invariant 5.44 that $\forall n'' \in \textsf{Anc}(m), ID_p[n''] = m.PathID[n'']$, so the invariant is preserved.

$bal_ret(V)_{p,<n,i>}$ assigns $node_p$ to be one of its children. However, the precondition for $bal_ret(V)_{p,<n,i>}$ is that $p \in \mathbb{P}_{<n,i>}$. If a child m is a Counter or Counter_Limit, $ID_p[m] = m.ID$, and $i = m.ID$, then the inductive assumption holds. So, if $node_p := m$, then $inc_count(m)_p$ becomes enabled, the hypothesis is still satisfied, and the consequent does not change, so the invariant is preserved. If $node_p := m$ and $i < m.ID$, then $ID_p[m] = i \neq m.ID$, so $inc_count(m)_p$ is not enabled and the hypothesis is invalidated, so the invariant is preserved.

$inc_count(n)_p$ only affects this invariant by changing $n.Status$ from Counter_Limit to Off.

However, this invalidates the hypothesis and preserves the invariant.

fold(n) changes both children to Off or Counter_Limit. If a child becomes Off, it invalidates the hypothesis. Now, if a child becomes a Counter_Limit, it was a Counter so it preserved the invariant, and the *ID* remains the same, so it still preserves the invariant. The parent node becomes a Counter so it must now uphold the invariant. If $p \in \mathbb{P}_{\mathsf{Parent}(n),n.ID}$, then necessarily $ID_p[n] = n.ID$, and the other two possibilities for the processor require that $ID_p[n] = n.ID$. But, since n becomes a Counter, invariant 5.45 proves the invariant.

unfold(n) changes the parent to a Balancer so it invalidates the invariant. It makes both children Counters, and only if they were both Off. since $n.PID$ increases by 1 to a new value, $\mathbb{P}_{n,n.PID} = \emptyset$. Similarly, since each child takes on a new *ID*, no processor could have this value cached, and so there are no processors that meet the conditions, so the invariant is vacuously preserved. ■

Now, two corollaries.

Corollary 5.47 If $inc_count(n)_p$ is enabled, $\forall n' \in \mathsf{Anc}(n), ID_p[n'] = n.PathID[n']$.

Proof: This follows directly from invariant 5.46. ■

Corollary 5.48 If $bal_req_{p,<n,i>}$ is enabled, $\forall n' \in \mathsf{Anc}(n), ID_p[n'] = n.PathID[n']$.

Proof: This follows directly from the preconditions of $bal_req_{p,<n,i>}$ and invariant 5.45. ■

5.2.7 Processor Travel Plans

We can now discuss the final piece needed before we can start the main proof. A processor enabled on a Balancer or Counter is simply waiting until it can perform. However, if a change occurs in the tree or it is waiting on a Counter_Limit or it is stuck inside a Balancer, it is necessary to define where a processor would go next. We can now give a more precise definition.

$\mathsf{Next}(m) = n$ iff
$\quad n \in \mathsf{Anc}(m)$
$\quad n.ID = m.PathID[n]$
$\quad \forall n' \in \mathsf{Des}(n) \cap \mathsf{Anc}(m)$
$\quad\quad n'.ID \neq m.PathID[n']$

We will immediately show the use of this function.

Invariant 5.49 If $inc_count(n)_p$ is enabled, then in the next state, either $inc_count(n)_p$, $bal_req_{p,<n,i>}$, $inc_count(\mathsf{Next}(n))_p$, $bal_req_{p,<\mathsf{Next}(n),i>}$, or $return(v)_p$ is enabled.

Proof: If $inc_count(n)_p$ is performed, then $return(v)_p$ is enabled. The only other ways that this action would be disabled (by 5.38, the only way another action could be enabled) would be that either $n.Status :=$ **Balancer**, in which case $bal_req_{p,<n,i>}$ would be enabled, or $n.ID$ was increased. However, by 5.47, each processor has the same ancestral *ID* record, equal to $n.PathID$, so each processor *travels* up the tree, and all clearly arrive at $inc_count(\mathsf{Next}(n))_p$ or $bal_req_{p,<\mathsf{Next}(n),i>}$, depending on the *Status* of n. (and by 5.34, $n.Status \neq$ **Off**). ■

Invariant 5.50 If $bal_req_{p,<n,i>}$ is enabled, then in the next state, either $bal_req_{p,<n,i>}$, $inc_count(n)_p$,$inc_count(\mathsf{Next}(n))_p$, or $bal_req_{p,<\mathsf{Next}(n),i>}$ is enabled, or $bal_wait_p :=$ **True**.

Proof: This should be clear from the actions of $bal_req_{p,<n,i>}$ and the above proof. ■

5.2.8 Conservation of Energy

We have now reached the pivotal point of this proof, showing that $n.Count$ is bounded by the Limit function. We need to track where in the tree processors are enabled to act, so that one counter does not get increased too high. We will create an equality which shows that balancing preserves a *Conservation of Energy* which limits the concentrations of processors throughout the tree.

We now define several derived variables that can be derived from other variables in a given state of an execution. These variables are all an attempt to count processors to create equations for the nodes that always hold. Some processors can be counted directly, because it is clear what they will do next. However, processors in a balancer have two possible paths. These processors have to be carefully counted using the balancer properties, and these counts will take into account numbers and not allocated processors.

$V(n) \triangleq j$ where $n.Count = \mathsf{Values}(n)[j]$.

We give an invariant that shows that this is related to the toggle bits.

Invariant 5.51 $\forall n, n.Status = \mathsf{Balancer} \implies V(n) \mod 2 = n.toggle[n.PID]$.

Proof: This is initially true, since $\forall n, V(n) = 0$ and $n.toggle[n.PID] = 0$.

Now, whenever n becomes a **Balancer**, $n.toggle[n.PID] := \mathsf{Toggle_Init}(n, n.Count)$, which has the same parity as $V(n)$. While n is a **Balancer**, $V(n)$ doesn't change. So, the invariant is preserved. ■

This is necessary because new balancers are oriented in different ways. Either the odd processor goes left or right. Many times throughout the proof, we will use this. The automaton easily handles this by assigning the left child the index of y equal to $n.toggle[n.PID]$ and testing if the value returned is this index in order to go left. However, it will clutter up the proof to continually calculate which child gets which value. So, we add a derived variable *tog* to each node that is equal to the index it currently is assigned in its parents balancer. Here is the rule for computing it:

$$m.tog := \begin{cases} n.toggle[n.PID] & m = \mathsf{Left}(n) \\ 1 - n.toggle[n.PID] & m = \mathsf{Right}(n) \end{cases} \tag{5.52}$$

$W(n)$ is equal to the number of processors that are enabled to act on n. If $n.Status =$ **Balancer**, then these processors are enabled to perform $bal_req_{p,<n,i>}$. If $n.Status =$ **Counter** or **Counter_Limit** then these processors are enabled to perform $inc_count(n)_p$.

$S(n)$ is equal to the maximum number of processors that are in a balancer (i.e. $p \in \mathbb{P}_{<n',i>}, i < n'.PID$) whose version i is no longer equal to the original parent node's *PID*. In other words, it can no longer be counted by the next defined variable, which handles normal balancer transactions. In a situation like this, either a processor arrives at a **Counter_Limit**, which it can then safely access, or it will pass up the tree until it finds the correctly reconciled node to access. In either case, $S(n)$ counts the maximum number of processors in the tree in this situation who would eventually reach n. Now, these balancers are old, since $i < n'.PID$. This means that $bal_req_{p,<n,i>}n'$ will never again be enabled. So, $x_{<n',i>}$ is fixed and we can calculate how many of the remaining processors will go to **Left**(n') and **Right**(n').

$$\mathsf{Half}_{n.tog}(x_{<\mathsf{Parent}(n),n.ID>}) - y_{n.tog_{<\mathsf{Parent}(n),n.ID>}}$$

We now get into some slight confusion because the next two definitions depend on each other. However, they apply to different nodes in the tree and will be inductively used in the next invariant.

$I(n)$ represents the maximum number of processors to be inherited from its parent. This uses the Half function to compute the splitting the balancer does. Given that $m = \mathsf{Parent}(n)$,

$$I(n) \triangleq \mathsf{Half}_{n.tog}(I(m) + C(m) + S(m) + W(m) + x_{<m,m.PID>}) - y_{n.tog<m,m.PID>}$$

$C(n)$ counts the excess processors that are enabled at descendant **Counter_Limits** but won't be able to take values because there are more processors than values. Once the **Counter_Limit** turns **Off**, the remaining processes will move up to the next **Counter_Limit**, **Counter**, or **Balancer** that they are enabled to access. We need to count these processors because the excess will eventually float up to the appropriate **Counter** or **Balancer**. Since all processors enabled on $inc_count(m)_p$ share the same ID_p array up the tree from m by invariant 5.47, a particular **Counter_Limit** node will send all of its extra processors to the same node once it turns **Off**. Define $\mathbb{CL}(n)$ to be the set of nodes n' that are **Counter_Limit**'s and $\mathsf{Next}(n') = n$. Define $R(n)$ for a **Counter_Limit** to be the number of values it has remaining to hand out, $(n.Limit - n.Count)/2^{\mathsf{Level}(n)}$. We can then define $C(n)$. Let $n' = \mathsf{Parent}(m)$.

$$C(n) \triangleq \sum_{m \in \mathbb{CL}(n)} C(m) + S(m) + I(m) + W(m) - R(m)$$

We summarize these formulas in table 5.1.

We now state the invariant.

Invariant 5.53 Conservation of Energy

$\forall n,\ n.Status = \mathsf{Balancer} \implies$

$V(n) + C(n) + S(n) + I(n) + W(n) + x_{n,n.PID} = \mathsf{Limit}(n,k)$

$n.Status = \mathsf{Counter} \implies$

$V(n) + C(n) + S(n) + I(n) + W(n) = \mathsf{Limit}(n,k)$

$n.Status = \mathsf{Counter_Limit} \implies$

$C(n) + W(n) + S(n) - R(n) \geq 0.$

Function	Definition	Description
$V(n)$	j where $n.Count = \mathsf{Values}(n)[j]$	values already handed out or held by other **Counter_Limit**.
$W(n)$	$\|\{p \in P : inc_count(n)_p$ or $bal_req_{p,<n,i>}$ are enabled$\}\|$	processors enabled to act on n.
$S(n)$	for **Counter_Limit**, $\mathsf{Half}_{n.tog}(x_{<\mathsf{Parent}(n),n.ID>}) - y_{n.tog_{<\mathsf{Parent}(n),n.ID>}}$	processors waiting in balancer automaton out of date and destined towards n.
$I(n)$	$\mathsf{Half}_{n.tog}(I(m) + C(m) + S(m) + W(m) + x_{<m,m.PID>}) - y_{n.tog_{<m,m.PID>}}$, $m = \mathsf{Parent}(n)$	maximum number of processors coming through parent balancer
$R(n)$	$(n.Limit - n.Count)/2^{\mathsf{Level}(n)}$	number of values **Counter_Limit** has left to give out
$C(n)$	$\sum_{m \in \mathbb{CL}(n)} C(m) + S(m) + I(m) + W(m) - R(m)$	excess number of processors at **Counter_Limit** that are destined to arrive at n.

Table 5.1: Conservation of Energy definitions

This invariant is initially true since $k = 0, \forall n, \mathsf{Limit}(n,k) = 0$, and **Counter_Limit** $\notin S_0$.

Proof: We will now consider each action using an inductive proof.

request$_p$

The effect of executing this action is that $k := k + 1$. We now need to consider every node and make sure that the count went up accordingly. This process is now enabled on $bal_req_{p,<n,i>}$, where $n = Root$, so $W(Root) := W(Root) + 1$, and since $\mathsf{Limit}(Root, k+1) = k + 1$, that terms also goes up by one, but they are on opposite sides of the invariant, so the invariant is preserved.

Now consider any node whose *Status* is **Balancer** or **Counter**. We need to consider how $\mathsf{Limit}(n,k)$ changes when k changes by 1.

$$\mathsf{Limit}(\mathsf{Left}(n), k+1) - \mathsf{Limit}(\mathsf{Left}(n), k) = \begin{cases} 1 & \mathsf{Limit}(n,k) \text{ even} \\ 0 & \mathsf{Limit}(n,k) \text{ odd} \end{cases} \tag{5.54}$$

$$\mathsf{Limit}(\mathsf{Right}(n), k+1) - \mathsf{Limit}(\mathsf{Right}(n), k) = \begin{cases} 1 & \mathsf{Limit}(n,k) \text{ odd} \\ 0 & \mathsf{Limit}(n,k) \text{ even} \end{cases} \tag{5.55}$$

Now, we show that for any of the invariant expressions that contain $\mathsf{Limit}(n,k)$, $I(n)$ only increases by 1 because of the increase in k if and only if $\mathsf{Limit}(n,k)$ increases by 1, and this will

	$\Delta_k I(\mathsf{Left}(m))$	$\Delta_k I(\mathsf{Right}(m))$
$\mathsf{Limit}(m,k)$ even $V(n)$ even $m.toggle[m.ID] = 0$	$i = 0$ Apply 5.27.1 $\Delta_k = 1$	$i = 1$ Apply 5.28.2 $\Delta_k = 0$
$\mathsf{Limit}(m,k)$ even $V(n)$ odd $m.toggle[m.ID] = 1$	$i = 1$ Apply 5.28.2 $\Delta_k = 1$	$i = 0$ Apply 5.27.1 $\Delta_k = 0$
$\mathsf{Limit}(m,k)$ odd $V(n)$ even $m.toggle[m.ID] = 0$	$i = 0$ Apply 5.27.2 $\Delta_k = 0$	$i = 1$ Apply 5.28.1 $\Delta_k = 1$
$\mathsf{Limit}(m,k)$ odd $V(n)$ odd $m.toggle[m.ID] = 1$	$i = 1$ Apply 5.28.1 $\Delta_k = 0$	$i = 0$ Apply 5.27.2 $\Delta_k = 1$

Table 5.2: Proof of Invariant on $request_p$

clear this action for the invariant, because it works inductively down the chain of $I(n)$ and it was proven for $I(Root)$. We can now assume that for n, its parent m is a **Balancer**, because either n is a **Balancer** or n is a **Counter**

We now generate a useful expression for $I(n)$.

$$\begin{aligned}
I(n) &= \mathsf{Half}_{n.tog}(I(m) + C(m) + S(w) + W(m) + x_{<m,m.ID>}) - \\
&\quad y_n.tog_{<m,m.ID>}, m = \mathsf{Parent}(n) \\
\mathsf{Limit}(m,k) &= V(m) + C(m) + S(I) + I(m) + W(m) + x_{m,m.ID} \implies \\
I(n) &= \mathsf{Half}_{n.tog}(\mathsf{Limit}(m,k) - V(m)) - y_n.tog_{<m,m.ID>}
\end{aligned}$$

Now, we know from Invariant 5.51 that the parity of $V(m)$ is equal to the toggle's parity. From this, we can show using a simple case-by-case analysis that $I(n)$ increases by 1 if and only if $\mathsf{Limit}(n,k)$ increases by 1. This analysis is done in Table 5.2. This table and the incremental analysis done in equations 5.54 and 5.55 proves the induction.

bal_request$_{p,<n,i>}$

A processor that is enabled here is counted by $W(n)$. Now, when this action is performed, it is no longer enabled for that processor, so $W(n) := W(n) - 1$. But, $x_{<n,n.PID>} := x_{<n,n.PID>} + 1$ by the composition with the balancer automaton. So, every invariant that contains $W(n)$ also contains $x_{<n,n.PID>}$ as a summand, and all the invariants are still preserved.

bal_return$(V)_{p,<m,i>}$

When this action is called, a processor's $node_p$ gets assigned to one of its children. There are two cases.

If $i = m.PID$, then by 5.40, this processor is now enabled on a child node n of m. So, $W(n) := W(n) + 1$. But, $I(n) = \mathsf{Half}_{n.tog}(I(m) + S(m) + C(m) + W(m) + x_{<m,m.PID>}) - y_{n.tog_{<m,m.ID>}}$. However, $node_p$ is set to n only if $y_{n.tog} := y_{n.tog} + 1$, and so $I(n) := I(n) - 1$. $W(n)$ and $I(n)$ are summands, so they cancel and the invariants are preserved.

The other case is that $i < m.PID$. The processor then inductively belongs to a $S(n')$ for some node n'. It will join $W(n')$ and leave $S(n')$, cancelling out.

inc_count$(n)_p$

When this action occurs, $answer_val_p$ is assigned and *inc_count*$(n)_p$ is no longer enabled for p, so $W(n) := W(n) - 1$. But, the assignment to $answer_val_p$ causes $V(n) := V(n) + 1$ and these effects cancel. Now, consider when it is a Counter_Limit. $R(n) := R(n) - 1$ and they are subtracted so these effects cancel. If the *Status* becomes Off all the processes go to their destined n' and increase $W(n')$, but come out of $C(n')$. They also contribute all their $S(n)$ and $C(n)$ to n'. which were originally counted in $C(n')$, so this cancels out as well. If they find another Counter_Limit at node n', their addition to $W(n')$ is cancelled by their departure from $C(n')$, preserving the Counter_Limit invariant.

fold(n)

Consider a successful *fold*(n) action. Assume $\mathsf{Parent}(m) = n, m' = \mathsf{Sibling}(m)$. We then have the following two equations from the invariant, and do a simple sum.

$$\begin{aligned}
\mathsf{Limit}(m,k) &= I(m) + V(m) + C(m) + S(m) + W(m) \\
\mathsf{Limit}(m',k) &= I(m') + V(m') + C(m') + S(m') + W(m') \\
\mathsf{Limit}(m,k) + \mathsf{Limit}(m',k) &= I(m) + I(m') + V(m) + V(m') + C(m) + C(m') + \\
&\quad S(m) + S(m') + W(m) + W(m')
\end{aligned}$$

Now, we know that $\lfloor a/2 \rfloor + \lceil a/2 \rceil = a$. So, the **Limit** terms and the I terms add up to produce the new equation:

$$\begin{aligned}\text{Limit}(n,k) &= I(n) + W(n) + C(n) + S(n) + x_{<n,n.PID>} - y_{0<n,n.PID>} - y_{1<m,n.ID>} + \\ &\quad V(m) + V(m') + C(m) + C(m') + S(m) + S(m') + W(m) + W(m')\end{aligned}$$

There are two cases. Either both child nodes become **Off**, or one becomes a **Counter_Limit**.

Consider when both children become **Off**. Then, given the definition of $V(n)$ and the assignment of $n.Count$, $V(n) := V(m) + V(m')$. All processors who were stragglers and were being counted by m or m' since that was their eventual destination now have n as a destination. But, $C(n)$ already counted these, so $C(n) = C(m) + C(m')$. All processors in $W(n)$ remain in $W(n)$. Since both nodes become off, all processors that eventually come out of the balancer will also be enabled on n, so and $S(n) := S(n) + S(m) + S(m') + x_{<n,n.PID>} - y_{0<n,n.PID>} - y_{1<m,n.PID>}$. These new values together satisfy the invariant.

If one child becomes a **Counter_Limit**(and only one can), without loss of generality m, then the processors in the balancer join $S(m)$. So, $S(m) := S(m) + \text{Half}_{m.tog}(x_{<n,m.ID>}) - y_{m.tog<n,m.ID>}$. m keeps $W(m)$ and $C(m)$. Now, the parent gets all the processors from the other child. So, $W(n) := W(n) + W(m')$. $C(n) = C(m') + W(m) + C(m) + S(m) - R(m)$. The processors in the balancer allocated towards the **Off** node get added to $S(n) = S(n) + \text{Half}_{m'.tog}(x_{<n,n.PID>}) - y_{m.tog<n,n.PID>}$. Finally, it is clear that $V(n) := V(m) + V(m') + R(m')$. Notice however that $R(m)$ cancels with the $-R(m)$ in the equation for $W(n)$, and the balancer equation sums correctly as above, preserving the invariant.

We now need to show in that last case that $S(m) + W(m) + C(m) - R(m) \geq 0$ to preserve the **Counter_Limit** invariant. We calculate the value of $R(m)$.

$$R(m) = \begin{cases} V(m') - V(m) & m = \text{Left}(\text{Parent}(m)) \\ V(m') - V(m) - 1 & m = \text{Right}(\text{Parent}(m)) \end{cases}$$

So, we can consider the difference $V(m) - V(m')$ since we need $-R(m)$.

$$\begin{aligned}V(m) - V(m') &= I(m') + C(m') + S(m') + W(m') - \text{Limit}(m',k) + \\ &\quad \text{Limit}(m,k) - I(m) - C(m) - S(m) - W(m)\end{aligned}$$

We now add identical terms to both sides:

$$\begin{aligned} V(m) - V(m') + & \quad C(m') + S(m') + W(m') + I(m') - I(m) + \\ C(m) + S(m) + W(m) + & = \mathsf{Limit}(m,k) - \mathsf{Limit}(m',k) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & \quad \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - \\ y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} & \quad y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} \end{aligned}$$

We can finally rewrite this substituting for I.

$$\begin{aligned} V(m) - V(m') + & \quad C(m') + S(m') + W(m') + \\ C(m) + S(m) + W(m) + & = \mathsf{Limit}(m,k) - \mathsf{Limit}(m',k) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & \quad \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - \\ y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} & \quad y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} + \\ & \quad \mathsf{Half}_{m'.tog}(\mathsf{Limit}(n,k) - V(n)) - y_{m'.tog_{<\mathsf{Parent}(m),m.ID>}} - \\ & \quad \mathsf{Half}_{m.tog}(\mathsf{Limit}(n,k) - V(n)) + y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} \end{aligned}$$

We can now cancel two terms and do a final rewrite.

$$\begin{aligned} V(m) - V(m') + & \quad C(m') + S(m') + W(m') + \\ C(m) + S(m) + W(m) + & = \mathsf{Limit}(m,k) - \mathsf{Limit}(m',k) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & \quad \mathsf{Half}_{m'.tog}(\mathsf{Limit}(n,k) - V(n)) - \\ y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} & \quad \mathsf{Half}_{m.tog}(\mathsf{Limit}(n,k) - V(n)) + \\ & \quad \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - \\ & \quad y_{m'.tog_{<\mathsf{Parent}(m),m.ID>}} \end{aligned}$$

We now have the equation on the right side that has to be greater than or equal to 0 when $m = \mathsf{Left}(\mathsf{Parent}(m))$ and we can just subtract one to get the equation for the other child. We now perform a case-by-case analysis.

If $m = \mathsf{Left}(\mathsf{Parent}(m))$, then we have to prove the LHS of 5.56 is greater than or equal to

0. $C(m') + S(m') + W(m') \geq 0$, so we can ignore these. Also, we know that $m.tog$ is 0 when $V(n)$ is even, and 1 otherwise. So, if $V(n)$ is even, we get the following equation:

$$\begin{aligned} V(m) - V(m') + & \quad C(m') + S(m') + W(m') + \\ C(m) + S(m) + W(m) + &= \mathsf{Half}_0(\mathsf{Limit}(n,k)) - \mathsf{Half}_1(\mathsf{Limit}(n,k)) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & \quad \mathsf{Half}_1(\mathsf{Limit}(n,k) - V(n)) - \\ y_{m.tog\,<\mathsf{Parent}(m),m.ID>} & \quad \mathsf{Half}_0(\mathsf{Limit}(n,k) - V(n)) + \\ & \quad \mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>}) - \\ & \quad y_{1<\mathsf{Parent}(m),m.ID>} \end{aligned}$$

Since $V(n)$ is even, $\mathsf{Limit}(n,k)$ and $\mathsf{Limit}(n,k) - V(n)$ have the same parity, so the first two sets of differences always cancel, by 5.29. The maximum value of $y_{1<\mathsf{Parent}(m),m.ID>}$ is $\mathsf{Half}_1(x_{<\mathsf{Parent}(m),m.ID>})$, which by 5.29 is less than or equal to $\mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>})$, so the invariant is preserved.

If $V(n)$ is odd, we get the following equation:

$$\begin{aligned} V(m) - V(m') + & \quad C(m') + S(m') + W(m') + \\ C(m) + S(m) + W(m) + &= \mathsf{Half}_0(\mathsf{Limit}(n,k)) - \mathsf{Half}_1(\mathsf{Limit}(n,k)) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & \quad \mathsf{Half}_0(\mathsf{Limit}(n,k) - V(n)) - \\ y_{m.tog\,<\mathsf{Parent}(m),m.ID>} & \quad \mathsf{Half}_1(\mathsf{Limit}(n,k) - V(n)) + \\ & \quad \mathsf{Half}_1(x_{<\mathsf{Parent}(m),m.ID>}) - \\ & \quad y_{0<\mathsf{Parent}(m),m.ID>} \end{aligned}$$

The first two sets of differences have the same ordering, ceilings minus floors. Since $V(n)$ is odd, either $\mathsf{Limit}(n,k)$ or $\mathsf{Limit}(n,k) - V(n)$ is odd, so by 5.29, these two sets of differences all sum to 1. The maximum value of $y_{0<\mathsf{Parent}(m),m.ID>}$ is $\mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>})$, and $\mathsf{Half}_1(x_{<\mathsf{Parent}(m),m.ID>}) - \mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>})$ has a minimum value of -1, by 5.29. So, the 1 and -1 cancel and the invariant is preserved.

If $m = \mathsf{Right}(\mathsf{Parent}(m))$, then we have to prove the LHS of 5.56 is greater than or equal to -1. $C(m') + S(m') + W(m') \geq 0$, so we can ignore these. Also, we know that $m.tog$ is 1 when

$V(n)$ is even, and 0 otherwise. So, if $V(n)$ is even, we get the following equation:

$$\begin{array}{rcl} V(m) - V(m') & & C(m') + S(m') + W(m') \\ C(m) + S(m) + W(m) + & = & \mathsf{Half}_1(\mathsf{Limit}(n,k)) - \mathsf{Half}_0(\mathsf{Limit}(n,k)) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & & \mathsf{Half}_0(\mathsf{Limit}(n,k) - V(n)) - \\ y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} & & \mathsf{Half}_1(\mathsf{Limit}(n,k) - V(n)) + \\ & & \mathsf{Half}_1(x_{<\mathsf{Parent}(m),m.ID>}) - \\ & & y_{0_{<\mathsf{Parent}(m),m.ID>}} \end{array}$$

Since $V(n)$ is even, $\mathsf{Limit}(n,k)$ and $\mathsf{Limit}(n,k) - V(n)$ have the same parity, so the first two sets of differences always cancel, by 5.29. The maximum value of $y_{0_{<\mathsf{Parent}(m),m.ID>}}$ is $\mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>})$, and $\mathsf{Half}_1(x_{<\mathsf{Parent}(m),m.ID>}) - \mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>})$ has a minimum value of -1, by 5.29, so the invariant is preserved.

If $V(n)$ is odd, we get the following equation:

$$\begin{array}{rcl} V(m) - V(m') + & & C(m') + S(m') + W(m') + \\ C(m) + S(m) + W(m) + & = & \mathsf{Half}_1(\mathsf{Limit}(n,k)) - \mathsf{Half}_0(\mathsf{Limit}(n,k)) + \\ \mathsf{Half}_{m.tog}(x_{<\mathsf{Parent}(m),m.ID>}) - & & \mathsf{Half}_1(\mathsf{Limit}(n,k) - V(n)) - \\ y_{m.tog_{<\mathsf{Parent}(m),m.ID>}} & & \mathsf{Half}_0(\mathsf{Limit}(n,k) - V(n)) + \\ & & \mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>}) - \\ & & y_{1_{<\mathsf{Parent}(m),m.ID>}} \end{array}$$

The first two sets of differences have the same ordering, floors minus ceilings. Since $V(n)$ is odd, either $\mathsf{Limit}(n,k)$ or $\mathsf{Limit}(n,k) - V(n)$ is odd, so by 5.29, these two sets of differences all sum to -1. The maximum value of $y_{1_{<\mathsf{Parent}(m),m.ID>}}$ is $\mathsf{Half}_1(x_{<\mathsf{Parent}(m),m.ID>})$, which by 5.29 is less than or equal to $\mathsf{Half}_0(x_{<\mathsf{Parent}(m),m.ID>})$, so the invariant is preserved.

This shows that the Counter_Limit invariant is preserved.

unfold(n)

This action, if successful, creates two counters, so we must show that both counters and the new balancer all preserve the invariant. First, n becomes a balancer. It's *ID* does not change, so all processes enabled on $inc_count(n)_p$ now are enabled on $bal_req_{p,<n,i>}$, and so $W(n) := I(n)$. $C(n)$ and $I(n)$ do not change. It is a new balancer, because $n.PID$ is changed, so $x_{<n,n.PID>} = 0$. So, the invariant is preserved for the parent.

To see that it is preserved for the children (say $m = \mathsf{Left}(n)$ and $m' = \mathsf{Right}(n)$), we know that $W(m) = W(m') = C(m) = C(m') = S(m) = S(m') = 0$ because these counters were just created with a new *ID* so no processor could be enabled on them. We also know that $V(m) = \mathsf{Half}_0(V(n))$ and $V(m') = \mathsf{Half}_1(V(n))$. Finally, $I(m)$ and $I(m')$ become:

$$\begin{aligned} I(m) &:= \mathsf{Half}_{m.tog}(I(n) + C(n) + S(n) + W(n) + x_{<n,n.PID>}) - y_{m.tog_{<n,n.PID>}} \\ I(m') &:= \mathsf{Half}_{m'.tog}(I(n) + C(n) + S(n) + W(n) + x_{<n,n.PID>}) - y_{m'.tog_{<n,n.PID>}} \end{aligned}$$

We can do a substitution using the invariant, and realizing that as a new balancer, the y_i are both zero, these are equivalent to:

$$\begin{aligned} I(m) &:= \mathsf{Half}_{m.tog}(\mathsf{Limit}(n,k) - V(n)) \\ I(m') &:= \mathsf{Half}_{m'.tog}(\mathsf{Limit}(n,k) - V(n)) \end{aligned}$$

But, $V(n)$, $m.tog$, and $m'.tog$ are related. $m.tog = n.toggle[n.ID], m'.tog \neq n.toggle[n.ID]$. So, we now wish to compute what $V(m) + I(m)$ and $V(m') + I(m')$ sum in all possible cases.

If $V(n)$ is even, then $m.tog = 0$ and $m'.tog = 1$. So, $V(m) = \mathsf{Half}_0(V(n))$ and $I(m) = \mathsf{Half}_0(\mathsf{Limit}(n,k) - V(n))$. But, by 5.25.1, $V(m) + I(m) = \mathsf{Half}_0(\mathsf{Limit}(n,k)) = \mathsf{Limit}(m,k)$. Now, $V(m') = \mathsf{Half}_1(V(n))$ and $I(m') = \mathsf{Half}_1(\mathsf{Limit}(n,k) - V(n))$. But, by 5.26.1, $V(m') + I(m') = \mathsf{Half}_1(\mathsf{Limit}(n,k)) = \mathsf{Limit}(m',k)$.

If $V(n)$ is odd, then $m.tog = 1$ and $m'.tog = 0$. So, $V(m) = \mathsf{Half}_0(V(n))$ and $I(m) = \mathsf{Half}_1(\mathsf{Limit}(n,k) - V(n))$. But, by 5.25.2, $V(m) + I(m) = \mathsf{Half}_0(\mathsf{Limit}(n,k)) = \mathsf{Limit}(m,k)$. Now, $V(m') = \mathsf{Half}_1(V(n))$ and $I(m') = \mathsf{Half}_0(\mathsf{Limit}(n,k) - V(n))$. But, by 5.26.2, $V(m') + I(m') = \mathsf{Half}_1(\mathsf{Limit}(n,k)) = \mathsf{Limit}(m',k)$.

So the invariant is fully proved. ■

5.2.9 Conclusion

We can now finally prove the necessary lemma to finish the output limit theorem.

Lemma 5.56 $\forall n$, $n.Count \leq \mathsf{Values}(n)[\mathsf{Limit}(n,k)]$.

Proof: This just means that $V(n) \leq \mathsf{Limit}(n,k)$. But, we just proved that for all **Counter** and **Balancer**, $V(n) + \ldots = \mathsf{Limit}(n,k)$, and showed that all those excess terms are greater than or equal to 0. So, for all **Counter** and **Balancer**, this invariant holds. If a node n is a **Counter_Limit**, then by invariant 5.15, there is a **Counter** ancestor m of n and $n.Count < m.Count$, so this holds for all **Counter_Limit**'s. Finally, if a node n is **Off**, it does not count. If it has not always been **Off**, then it was last a **Counter** or **Counter_Limit**, this invariant held then, and $\mathsf{Limit}(n,k)$ increases with k, so it holds. This concludes the proof. ■

We are now ready to prove the output limit theorem.

Theorem 5.57 If $v \in Outputs, v < k$.

Proof: Invariant 5.22 from the beginning of the section stated that $\forall v \in \mathbf{N}$, $v \in Outputs \implies \exists n, v <= n.Count - 2^{\mathsf{Level}(n)}$. The lemma above shows that $\forall n$, $n.Count \leq \mathsf{Values}(n)[\mathsf{Limit}(n,k)]$, or just slightly renumbered, $\forall n$, $n.Count - 2^{\mathsf{Level}(n)} \leq \mathsf{Values}(n)[\mathsf{Limit}(n,k) - 1]$. Finally, lemma 5.30 stated $\mathsf{Values}(n)[\mathsf{Limit}(n,k) - 1] < k$. This, however, leads to the following chain of events which proves our theorem.

$$v \in Outputs \implies \exists n, v \leq n.Count - 2^{\mathsf{Level}(n)} \implies$$
$$v \leq n.Count - 2^{\mathsf{Level}(n)} \leq \mathsf{Values}(n)[\mathsf{Limit}(n,k) - 1] < k \implies v < k$$

■

5.3 Safety Property

Property 5.58 (Safety Property) In any state of an execution, the values output are distinct and in the set $\{0, \ldots, k-1\}$, where k is the number of requests.

Proof: Theorems 5.21 and 5.57 together prove the safety property. ■

Chapter 6

Implementation Verification

We now argue that the implementation presented efficiently meets and optimizes the specification presented in 4. We will discuss each action in the automata and show its implementation, arguing its correctness.

- Request
- Increment_Count
- Return
- Balancer_Request
- Balancer_Return
- Fold
- UnFold

6.1 Request

The request is implemented by the initial lines of the main traversal code, found on page 27. The initial balancer is set to *Root*, the ID of *Root* is set to 0, which is always its value, and *answer* is set to INVALID, which is the same as $answer_val_p$.

6.2 Increment_Count

Increment_Count is implemented by the increment_counter code on page 28. The atomicity is created by a mutual exclusion lock. The preconditions are checked after the lock is obtained, and if the preconditions are false, INVALID is returned and the processor climbs up the tree, which corresponds to a processor in the automata being enabled on the ancestor of the node that was accessed here due to versioning. The assignments in the automata action are all clearly here.

6.3 Return

Return only occurs if a valid answer is returned from Increment_Count, which correctly simulates the behavior of the automata.

6.4 Balancer_Request

This is where an optimization comes in. The Bookkeeping array, as clearly described in Chapter 2, is implemented for the same reason that the infinite balancer scheme is designed in the specification. The infinite balancer scheme allows old processors to sit in older incarnations of a Balancer potentially forever. The implementation limits this behavior, restricting changes in the tree until all processors leave the balancer code for a new counter. The bookkeeping scheme also resolves the atomicity of the precondition. If a processor registers its observance of a Balancer in the bookkeeping array, and rechecks to find it still a Balancer, then other processors upon a folding of this action are unable to unfold this node until the processor finishes balancing. Now, technically this processor balanced through the node while it transitioned from a Balancer to a Counter, which increments *PID*, a precondition for balancing. However, the Counter condition is another precondition which limits processors from balancing in the automata, so the balancer actions in the Balance I/O Automaton that correspond to this balancer will remain empty for the Counter. We can then charge the processor safely to the prior incarnation as a Balancer, and the correspondence is preserved.

6.5 Balancer_Return

This is proven in the original work by Shavit and Zemach [20].

6.6 Fold

This action is implemented directly from the specification, with mutual exclusion on all three nodes obtained.

6.7 UnFold

This action is implemented almost directly from the specification, with the added restriction of the bookkeeping entry. Since this was originally a non-deterministic action, this does not affect correctness. The implementation also releases the lock for the parent node early. Since all the changes to the parent occur beforehand, this is clearly a correct optimization.

6.8 Liveness

Safety is normally a property that comes along with showing that an implementation meets the specification. Liveness properties do not have the same luxury. However, in this case, liveness is rather easy to prove for the specification and the implementation. Notice the specification and implementation's fold and unfold actions restrict the changes to be made only if the count variable is distinct from the change variable, which is recorded from the last change made. This is the key reason why liveness is satisfied. Clearly, in a DDT environment where no changes occur, it is clear that every fair execution should have all processors eventually terminate with values. Now, the effect of the conditions on the change code is that we allow a change to occur only once a processor terminates. Once that change occurs, any processors dependent on that change can go up the tree and eventually can rely on getting to the root node who's ID never changes and can always be accessed. We can then run a simple induction on the number of requests made, since the liveness condition specifies a finite number of requests, and the liveness condition is satisfied.

Chapter 7

Summary and Future Work

This thesis bridges the gap between queue-locks and diffracting trees of various depths, providing near-optimal performance at all loads. We develop, implement, and verify the specification of a new reactive shared counter, the Dynamic Diffracting Tree, that performs well at all static levels of contention and can react to better handle new levels of contention. We summarize the work below and then present items for future work.

We begin the design of the DDT by taking a diffracting tree and generalizing it into an irregular diffracting tree, removing the restriction that the tree must be balanced. Merging a counter node and a balancer node into a single node with a state variable allow us the possibility of change. Versioning information to keep the values handed out balanced throughout the tree is described. The folding and unfolding mechanisms are presented in detail, with an informal understanding of why they work. An implementation is given using multiple locks. Finally, scaling policies are discussed and presented, and some warnings are presented about necessary policy restrictions.

We then give experimental results that show that the DDT performs within a constant factor of optimal diffracting trees in throughput and average latency at all levels of contention, and on the Alewife, even surpasses diffracting trees for a range in which an irregular tree is best suited for the task. We also show that it effectively competes with load balancing schemes in some Producer/Consumer applications.

Next, we give a formal description of the shared counter problem and describe safety and liveness properties that our algorithm must satisfy. We present a specification written in Lynch

and Tuttle's I/O Automata [13] format for which we prove the safety property. We then argue that our implementation meets the specification, carrying with it the safety property. Finally, we argue that our implementation also satisfies the liveness property.

In conclusion, the DDT is a new and exciting algorithm that carries with it the best features of diffracting trees while avoiding the latency drawbacks in low contention cases. Using cache-coherence, it pushes agreement into the low-contention levels of the tree, making it faster and easier to reach consensus. It provides a locality measure that allows trees of irregular size to occur based on the memory-layout of the data structure. Overall, it provides a truly fast and scalable implementation of *fetch-and-increment* for a variety of applications, including $k-$exclusion barriers, pools, stacks, and priority queues.

7.1 Continued Work

There is still much work left to do on this project. The most important are listed below.

7.1.1 More Performance Results

It would be important to run this algorithm on larger versions of the MIT Alewife machine when they become available to see whether irregular diffracting trees continue to fill in the gaps between optimal diffracting trees and to bring down the constant factor seen in the Proteus simulations.

7.1.2 Completion of Implementation Proof

It would be valuable to construct a simulation relation between the implementation and the specification, which would formally prove the safety property for the implementation. A good formal model of the Alewife's operation should be designed, to make future distributed design easier to correctly think about.

7.1.3 Wait-Free Version

A single lock version of the algorithm should be feasible, even with difficulties in keeping the tree consistent. Once this step is taken, then the DDT is closer to being wait-free. The state variable still needs to be checked during the counter fetch-and-increment, but a hardware provided conditional fetch-and-increment would be enough to make a single lock version of this algorithm wait-free.

7.1.4 Timing Scheme

In this scheme, a processor in the balancing algorithm would check the state variable each time it failed on the test and set toggle lock. Then, if a timing constant could be produced which was longer than the maximum amount of time necessary for a processor to pass through one pass of the balancing algorithm, this could be used to delay unfolding to guarantee that all processors were out of the balancer before the newest counter version of the node could successfully unfold.

7.2 Future Work

There are a variety of directions that this work should lead. The most important areas of study are listed below.

7.2.1 Message Passing

A message passing version of this algorithm should be implemented, due to its superior performance on the MIT Alewife machine. In a message passing system, different processors act as nodes in the tree, passing messages to other nodes as a substitute for traversing the tree. This allows a processor to solely control a node, providing better understanding of the contention levels and thus it can more accurately decide when to grow or shrink. This algorithm can easily be implemented on a message passing system and should outperform our shared memory results.

7.2.2 Elimination Trees

The folding and unfolding mechanisms in DDTs should be implementable in an elimination tree, a form of diffracting tree presented by Shavit and Touitou [19] where tokens and anti-tokens pass through the tree in different ways, either acting as increments and decrements at leaf counters or as enqueue and dequeue operations on leaf pools, queues, or stacks. This algorithm clearly carries over to the second case, and might be applicable to the first case.

7.2.3 Scaling Policies

We present some simple policies for determining when to change the tree, and some suggestions on when to hold back from changing the tree. However, there seems to be much work to be researched in this area, since it is directly related to on-line algorithms.

7.2.4 Distributed Data Structures

We feel that many of the issues discussed here are applicable to future design of data structures. The key idea of pushing agreement into the low-contention levels of the tree by exploiting cache-coherence is definitely an idea that deserves further attention in current data structure design. The verification done in this paper, although of a slightly difficult nature, is also a necessity in future distributed design. I/O Automata and their proof mechanisms at an early stage greatly helped the algorithm come into shape by centering the focus on the invariants of the algorithm, which is difficult to do when looking at distributed system code. Finally, the reactive nature of this tree and of B.H. Lim's work should lead the way for other data structures to become reactive, which may soon become a necessity in the distributed world.

Bibliography

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication. Also, appears as MIT Technical Report MIT/LCS/TM-454, June 1991.

[2] Anant Agarwal and Mathews Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[3] T. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994. Earlier version in *Proceedings of the 23rd ACM Annual Symposium on Theory of Computing*, pp. 348–358, May 1991. Also, MIT Technical Report MIT/LCS/TM-451, June 1991.

[5] K. E. Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[6] E. A. Brewer, , and C. N. Dellarocas. Proteus user documentation, version 4.0, March 1992.

[7] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A HIGH-PERFORMANCE PARALLEL-ARCHITECTURE SIMULATOR. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[8] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *asplosIV*, pages 224–234, Santa Clara, California, 1991.

[9] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, Massachusetts, April 1989.

[10] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[11] Beng-Hong Lim. *Reactive Synchronization Algorithms for Multiprocessors.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technolgy, February 1995.

[12] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI),*, pages 25–35, October 1994.

[13] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[14] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for on-line problems. volume 20 of *Proceedings of the Symposium on Theory on Computing*, pages 322–333, 1988.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems, TOCS*, 9(1):21–65, February 1991. Earlier version published as TR 342, University of Rochester, Computer Science Department, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice UNIV, May 1990.

[16] L. Rudolph, M. Slivkin, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, July 1991.

[17] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 340–347, June 1984.

[18] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees, October 1995. Unpublished Manuscript.

[19] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. In *SPAA'95: 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 54–63, Santa Barbara, California, July 1995. Also, Tel-Aviv University *Technical Report*, January 1995.

[20] Nir Shavit and Asaph Zemach. Diffracting trees. In *Proceedings of the Sixth Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 167–176, Cape May, New Jersey, June 1994. Submitted for journal publication.

[21] P. C Yew, N. F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.