Reactive Diffracting Trees

(Extended Abstract)

Giovanni Della-Libera
Microsoft Corporation and M.I.T.

Nir Shavit
Tel-Aviv University and M.I.T.

Abstract

Shared counters are concurrent objects which provide a fetch-and-increment operation on a distributed system and can be used to implement a variety of data structures, such as barriers, pools, stacks, and priority queues. Diffracting trees are novel data structures that provide an effective, high throughput and low contention, shared counter construction. Under high loads, their performance has been shown to surpass all known counter implementations. Unfortunately, Diffracting trees of differing depths are optimal for limited load ranges, and a deep tree that performs well under high load performs rather poorly when the load is very low.

To overcome this drawback, we introduce the *Reactive Diffracting Tree*, a novel Diffracting tree construction which can grow and shrink as necessary to better handle the changing access patterns and memory layout of the machine on which it runs. It provides true scalability and locality by dynamically "morphing" itself all the way from a simple queue-lock based counter under low load, through a range of increasingly deeper/shallower Diffracting trees as the load varies.

Empirical evidence, collected on a 32-node Alewife cache-coherent multiprocessor and the Proteus distributed shared-memory simulator, shows that the reactive diffracting tree provides throughput within a constant factor of optimal depth Diffracting trees at all load levels. It also proves to be an effective competitor with known randomized load balancing algorithms in producer/consumer applications.

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 has wide applications in concurrent algorithm design. Since good hardware support is not readily available, there have been a variety of software solutions presented for this problem.

SPAA 97 Newport, Rhode Island USA

1.1 Background

Simple solutions often involve a counter protected by a mutual exclusion using test-and-set locks with exponential backoff (see Agarwal, Anderson, and Graunke [2, 3, 10]) or queue-locks (see Anderson or Mellor-Crummey and Scott [3, 13]). These algorithms are popular because they provide short latencies in low load situations. However, they can not hope to obtain good throughput under high loads due to the bottleneck inherent in mutual exclusion.

Algorithms that offer greater utilization of parallelism under high loads are the combining trees of Yew, Tzeng, and Lawrie [19] and Goodman, Vernon, and Woest [9], the counting networks of Aspnes, Herlihy, and Shavit [4], and the Diffracting trees of Shavit and Zemach [18, 17]. These methods are highly distributed and lower the contention on individual memory locations, allowing for better performance at high loads.

Of the distributed methods, Diffracting trees [18] have proven the most efficient in terms of latency, throughput, and robustness in the face of load fluctuations [18, 17]. Diffracting trees are constructed from simple one-input two-output computing elements called *balancers* [4] that are connected to one another by wires to form a balanced binary tree. Requests arrive at the root balancer of the tree and are evenly divided by the balancers as they trickle down to the leaves. The tree of balancers can thus quickly distribute requests to a collection of lock-based counters at the leaves, which in turn hand out indexes without duplication or omission. For any given load, there is a Diffracting tree of a certain depth that will provide optimal performance. However, a deep tree has unwanted costs for lower loads due to its higher latencies. One set of experiments we conducted revealed that in a low load situation, the throughput over a fixed period of time for a queue-lock based counter was 652 operations while the tree 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

It was proven in a recent paper by Shavit, Upfal, and Zemach [17], that the range in which a Diffracting tree of a given depth can provide optimal performance is rather limited. Our goal is thus 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 over the complete range of loads in the system.

Lim and Agarwal recently came up with an innovative reactive mutual exclusion algorithm [11, 12] that switches between a test-test-and-set lock by Rudolph [14], a queue lock, and a combining tree. They report that the algorithm performed well from the low to mid-load ranges, as the combining tree took over for the queue lock. Unfortunately, their approach relies on the sequential nature of mutual-exclusion, where one processor at a time has unique access, to allow processors to "agree" to switch among algorithms. It will thus not work for highly distributed counter structures like Diffracting trees which may be concurrently traversed by multiple processors. (Nevertheless, Lim and Agarwal's methods have given us valuable insights to reactive policy making.)

1.3 Reactive Diffracting Trees

This paper introduces the *Reactive Diffracting Tree* (RDT), a novel Diffracting tree construction that allows changes on-the-fly to assume near-optimal size and shape for a given request load. It is thus the most effective and scalable software counter implementation known to date.

Two main insights were necessary in order to provide a reactive construction of a highly distributed data structure such as a Diffracting tree.

- Localize decision making – grow and shrink the tree at its leaves instead of trying to switch among trees.
- Allow inconsistent views – let individual processors run optimistically based on possibly inconsistent views while keeping global behavior consistent.

Localized decision making spares processors from continuously deciding on the overall structure of the shared counter, which is what the Lim-Agarwal algorithm requires. A major drawback with global decision making is that processors can get delayed while they wait for a change to occur. Our new algorithm is based on a way of distributing changes among local counters/nodes of the tree, so that 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.

By allowing processes to maintain inconsistent views we are able to minimize the need for expensive access to shared variables. Instead of forcing a processes to maintain a coherent global state, we opt for an optimistic approach. We let each processor run based on a view of the global structure which is locally cached. Though processors' views may be inconsistent, we have developed an algorithm that guarantees that processes will eventually detect inconsistency and repeat the necessary parts of their traversal so that the tree as a whole acts in a coherent fashion. We can thus exploit the performance benefits of caching. Our algorithm can use cache-coherence (implicit or explicit[1]) to keep local copies of shared state, modifying these states infrequently in high load situations and more frequently in the low load ones.

[1] We have implemented a version of our algorithm that has its own specialized coherence code to allow execution on machines that do not provide cache-coherence

We evaluated RDT performance on the MIT Alewife machine of Agarwal et. al [1]. However, since the largest Alewife machine only has 32 nodes, we extended our experiments up to the 256 processor range using the Proteus Parallel Hardware Simulator of Brewer, Dellarocas, Colbrook, and Weihl [5, 6]. As we show, Proteus simulates Alewife well, giving results that are comparable when normalized.

We compared the RDT to simple queue-locks and optimal Diffracting trees of various depths and observed substantially improved scalability. For example, in the same benchmark described earlier, the RDT under low load provided 243 operations and under high load provided 3932 operations. In general our results show that the RDT performs within a constant factor of optimal Diffracting trees at all load levels, and is thus the most scalable counter implementation available to date. We believe that our ongoing efforts at tuning of the reactive policy will allow us to reduce this factor further.

Interestingly enough, on the MIT Alewife machine we observed that the RDT outperforms all regular Diffracting Trees. As we explain in the sequel, this is due to its ability to assume an irregularly-shaped tree structure, taking advantage of locality. We also show that the RDT performs effectively in Producer/Consumer applications.

In summary, we believe that the truly scalable counter offered by RDT will prove to be an effective tool for the design of future data structures and algorithms for multi-scale computing. We are currently busy tuning the algorithm and extending our approach to other data structures such as elimination trees [16, 8]. Code for all algorithms presented in this paper can be found at *http://theory.lcs.mit.edu/~gio/code.html*.

2 Diffracting Trees

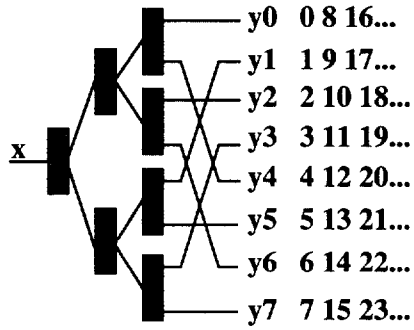

Figure 1: A Diffracting tree

A Diffracting tree [18] 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 continually splits the number of requests on its input wire onto its two output wires. A local counter increments based on its depth into the tree. Figure 1 shows a Diffracting tree of size 8. A processor starts at the root node and balances until it reaches a leaf counter, obtains a value, and exits.

2.1 Balancers

A simple implementation of a balancer would be a memory location with a lock that toggles between the values 0 and 1. A token locks the location, gets the value, stores the inverse value, and exits on the assigned wire bit. However, 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 [18], which exploits large numbers of requests by having a prism array on which processors can meet and "pair off" onto the two output wires, using the toggle-lock bit only as a last resort. (Our implementation actually uses an advanced version of a Diffracting balancer presented in [17]).

2.2 Irregular Diffracting Trees

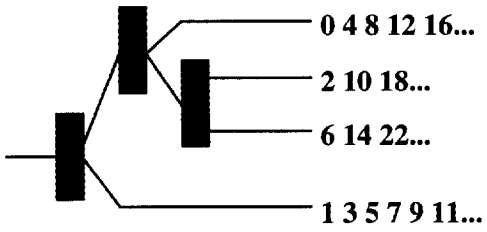

Figure 2: An Irregular Diffracting Tree

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. Figure 2 shows how one would set the counter's increment and initial values to make it count properly.

3 Reactive Diffracting Trees

The irregular diffracting tree moves us towards our goal of making a flexible structure. It would be expensive to change an entire tree to a different size. However, by allowing irregular diffracting trees, we can make local changes that would expectedly be replicated throughout the tree, assuming that loads on subtrees of the same depth are similar. However, 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.

We now introduce the additional state and algorithms necessary for a Diffracting tree to be reactive.

3.1 State and Versioning Information

Once we let the tree size change, a processor needs to be able to distinguish what state a node is presently in. We do this by adding a state variable. This state variable initially only takes three values, Counter, Balancer, or Off. If a processor visits an Off node, then it would know that the tree has changed, so it would go back up the tree until it finds a node it can successfully visit. We start with an arbitrary-sized tree, assign an initial configuration of balancers and counters from the root and leave the rest of the tree Off. The designer can decide whether a real leaf of this tree can have an allocation call to expand or if there is a static limit to the tree. We add a Counter_Limit state for the case where the tree shrinks, which will be explained later on.

Checking the state variable could be potentially expensive. 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 keep the tree larger than a base counter under high loads, which would keep the root state value unchanged. In doing so, processors can utilize cache-coherence to keep accesses to the top level state variables relatively inexpensive. This keeps global agreement inexpensive.

The state variable, along with an appropriate folding and unfolding mechanism 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. The key change needed 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[2] 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.[3] But, for simplicity, it is kept separate here.

3.2 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 *unfolding* a counter into a balancer with two counter children.

3.2.1 Folding

A processor, upon deciding that a balancer and its children counters need to be folded will attempt to obtain locks for all three nodes. If it is successful, then it tests whether the 3 nodes are still in the proper states to be folded.

[2] 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.

[3] This versioning scheme can be folded into the state variable in an implementation, to reduce size and complexity, since versioning really is additional state.

Then, the two child counters' values are compared. The values these counters hand out are somewhat related. 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. 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 3.

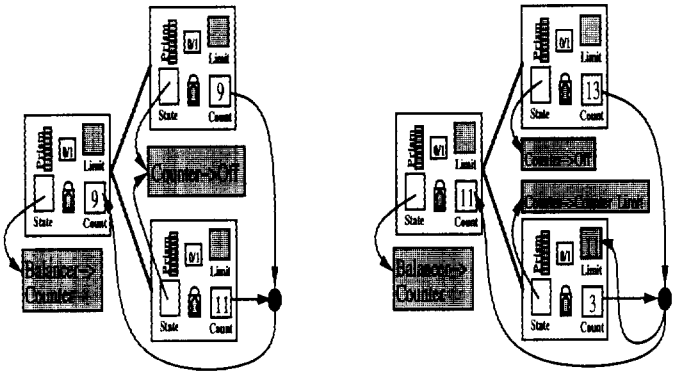

Figure 3: Two cases of Folding, Parent node at Level 1

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 the second picture in Figure 3 gives an example.

3.2.2 Unfolding

Unfolding is a bit easier to understand, but has its own challenges. 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 4.

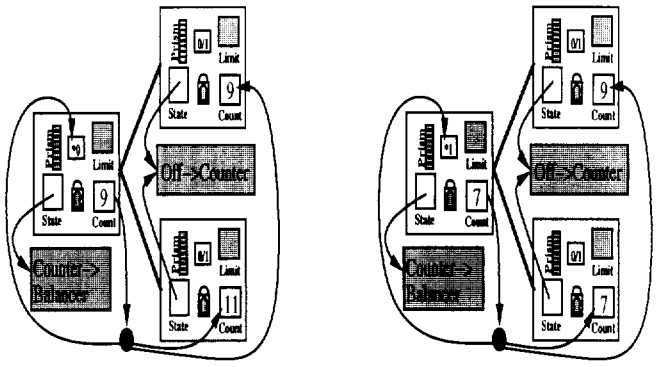

Figure 4: Two cases of Unfolding, Parent node at Level 1

4 Folding and Unfolding Policy

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 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.

5 Experimental Results

We evaluated a Reactive Diffracting Tree by running a collection of benchmarks on both a multiprocessor machine and a simulator. 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*[4] simulator, developed by Brewer et. al. [6], where we were able to extend our results to 256 processes. We ran correlation benchmarks to show that the results were comparable, and present those results in Appendix A. Our benchmarks include index-distribution, reaction time to spikes in load levels, and producer/consumer runs, all of which demonstrate the advantages (and disadvantages) of the RDT structure.

[4] Version 3.00, dated February 18, 1993

5.1 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 [7]. 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.

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.

5.2 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 better distribute the requests over time, providing a lower-contention environment. We ran this benchmark for a fixed amount of time on the Alewife machine (10^7 cycles), varying the number of processors[5] 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 the timing begins. This brings into question the fact that the RDT 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 RDT, a substantial startup period will be allowed before timing begins to allow the RDT to best match the input load.

We mainly collected throughput data. The throughput is the total number of `get_next_index()` operations that returned in the time allowed. We also examined latency, the average amount of time between the call to `get_next_index()` and its return, but these numbers are clearly related and one can be calculated from another.

The algorithms we ran were the Reactive 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.

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

5.2.1 Alewife Results

We have the first published performance results for Diffracting Trees on the Alewife machine. For the Reactive Diffracting Tree, 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 5 shows throughputs for a queue-lock based counter, Diffracting trees of depth 1, 2, and 3, and the RDT. The most interesting result is that the RDT 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.

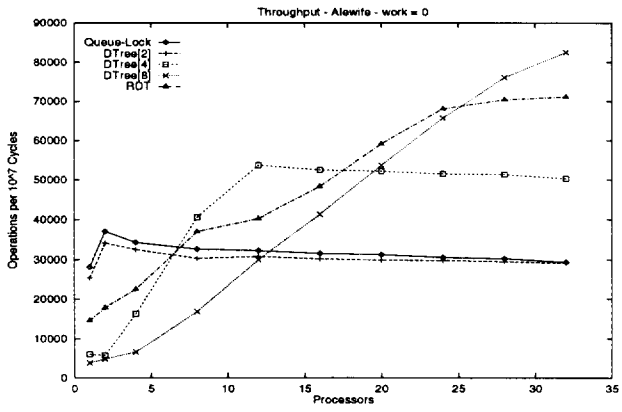

Figure 5: Alewife Throughputs of Diffracting Trees, Queue-Lock Based Counter, and RDT

Since the Reactive 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 also construct a worst-case composite graph for comparison. We show the optimal composite, RDT, and worst-case composite for the Alewife in Figure 6 under high contention. The throughput and latency of the RDT appear to stay within a factor of the optimal composite throughout its performance, and stays well ahead of the worst-case composite which levels off. The average factor between the throughput of the RDT and the optimal composite is 1.27.

5.2.2 Proteus Results

We turn to the Proteus simulator, which simulates Alewife's hardware, although it does not fully implement Alewife's LimitLESS [7] cache-

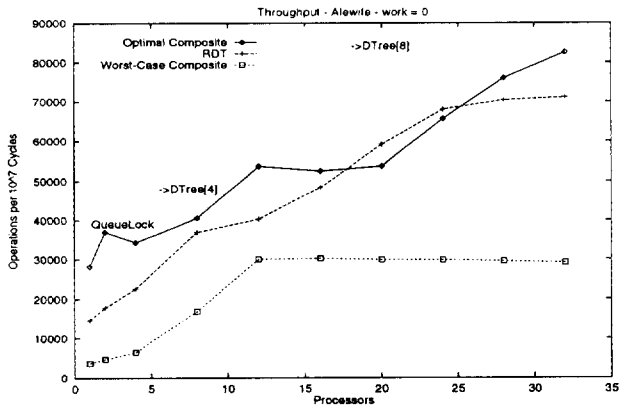

Figure 6: Alewife Throughputs of Optimal Composite, RDT, and Worst-Case Composite

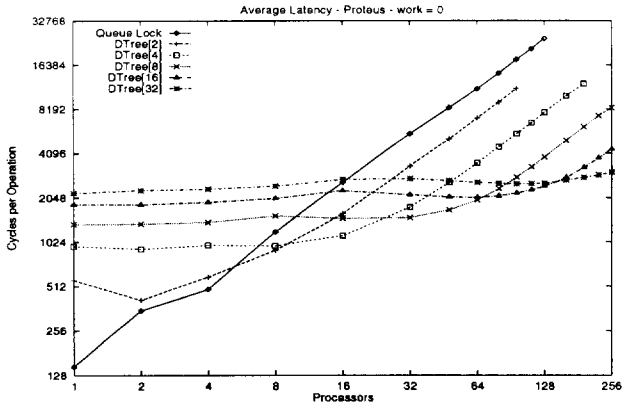

Figure 8: Latencies of Diffracting Trees and Queue Lock on Proteus

coherence policy. Appendix A shows our experiment which correlates the results from Alewife with the results from Proteus.

We now show our Proteus results, running up to 256 processes and adding Diffracting Trees of depths 4 and 5. Figure 7 shows the throughputs and latencies of Diffracting Trees of depth 0 (queue-lock based counter) through 5.

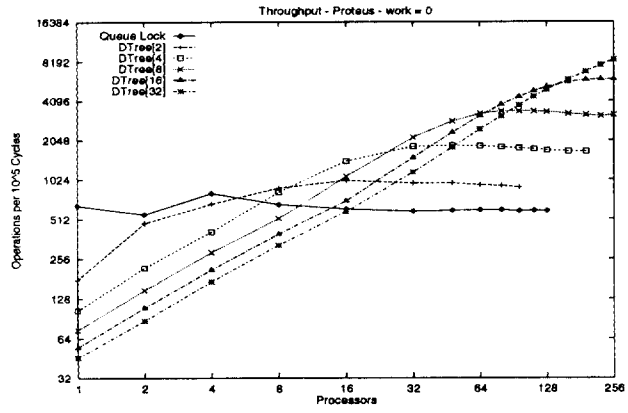

Figure 7: Throughputs of Diffracting Trees and Queue Lock on Proteus

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, RDT, and the worst-case composite in Figure 9 for high contention ($work = 0$) and Figure 10 for low 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 RDT and the optimal composite was 1.56, and in the low contention case, the average factor was 1.41. In both cases, the worst-case composite leveled off quickly and couldn't even be calculated after 128 nodes due to inability of proteus to handle such contention.

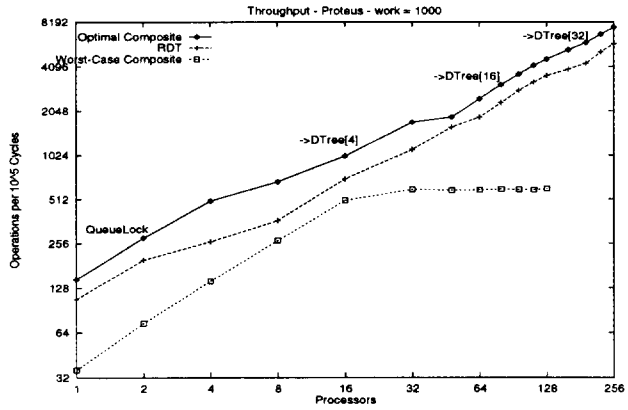

Figure 9: Throughputs of Optimal Composite, RDT, and Worst-Case Composite on Proteus under High Contention

5.3 Large Contention Change Benchmark

We measured the response of a RDT to a sudden spike in contention levels, measuring the average latency of the RDT 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.

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

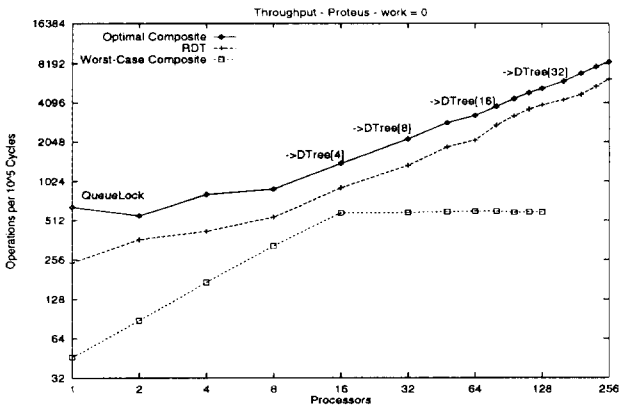

Figure 10: Throughputs of Optimal Composite, RDT, and Worst-Case Composite on Proteus under Low Contention

number of processors to 256, timing for 400,000 additional cycles. The tree grew to depth 5. Figure 11 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.

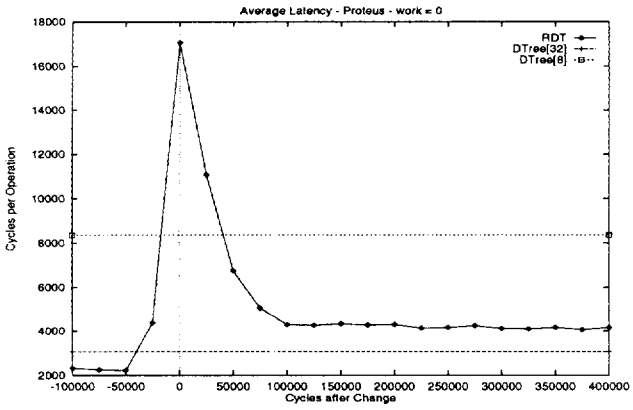

Figure 11: Average latency of RDT over time in response to sudden surge

The plot also contains Diffracting Trees of depth 3 and 5 with their average latency at 256 processors, which are what the RDT emulates before and after the change. Here is a good example of the tradeoff that a developer must consider in choosing to use the RDT. Imagine that the developer initially used the Diffracting tree of depth 3. The triangle on the left formed by the RDT 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 RDT and the Diffracting Tree of depth 3 to the right of the triangle is the region that a developer gains in using the RDT. Of course, the developer could choose to use the depth 5 tree all along, but the RDT outperforms this tree in the lower load case, which may usually be the common case.

5.4 Producer/Consumer Benchmarks

Job pools are shared data structures that store a changing set of jobs that need to be performed by 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) [15]. 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, load balancing algorithms, and specifically RSU, are known to outperform Diffracting and Elimination trees [18, 16]. The lock-based counters do well against RSU in the low load levels, and Diffracting and Elimination trees seem to come close to RSU's level of performance, but overall, no shared counter method has been able to effectively compete with RSU. We now show that the RDT has become an effective competitor.

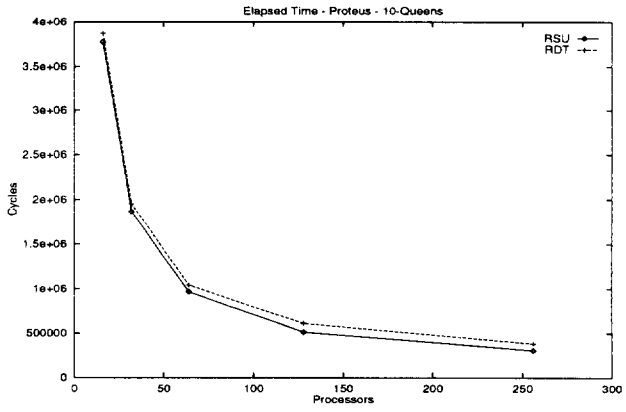

Figure 12: Producer/Consumer Performance

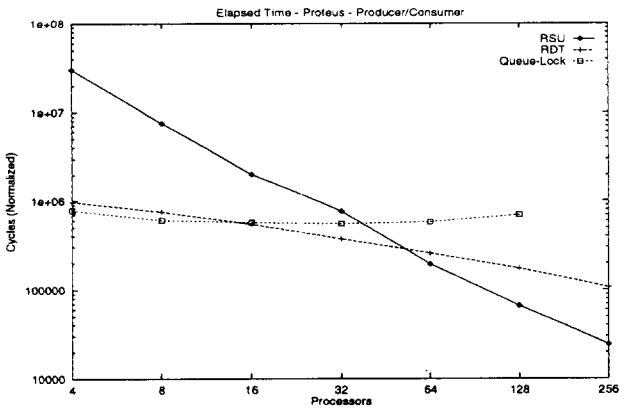

Figure 13: 10-Queens Performance

5.4.1 10-Queens

The n-Queens problem is a good problem to test the RDT 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 13 shows how close the Diffracting tree comes to RSU in total time elapsed throughout the differing load levels.

5.4.2 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 RDT 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 12. As one can see, the RDT provides near queue-lock performance in low-loads, and approaches the performance of RSU in higher loads.

6 Acknowledgments

This work was supported by NSF grants CCR-9520298 and 9225124-CCR, ARPA grant F19628-95-C-0118, and AFOSR-ONR grant F49620-94-1-0199.

References

[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] A. Agarwal and M. 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] J. Aspnes, M. P. Herlihy, and N. 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] E. A. Brewer, , and C. N. Dellarocas. Proteus user documentation, version 4.0, March 1992.

[6] 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.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *asplosIV*, pages 224–234, Santa Clara, California, 1991.

[8] Giovanni Della-Libera. Dynamic diffracting trees. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, July 1996.

[9] J. R. Goodman, M. K. Vernon, and P. 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] B. Lim. *Reactive Synchronization Algorithms for Multiprocessors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technolgy, February 1995.

[12] B. Lim and A. 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] 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.

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

[15] 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.

[16] N. Shavit and D. 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.

[17] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 33–41, Padua, Italy, June 1996.

[18] N. Shavit and A. 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. Also in ACM Transactions on Computer Systems, November 1996.

[19] 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.

A Alewife/Proteus Correlation

It is important to compare the results gathered on the Alewife with the Proteus, to make sure that the results can be extended over. Figure 14 is the counterpart to Figure 5. 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 15. The results show that the Alewife trees have a higher optimal load level, but the graphs still look comparable, a good result for Proteus.

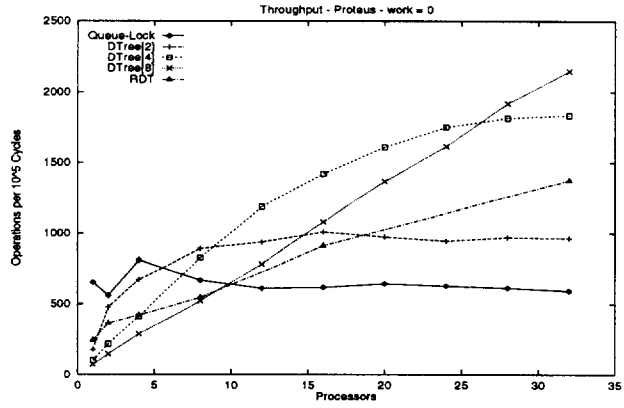

Figure 14: Diffracting Trees, Queue-Lock Based Counter, and RDT on Proteus

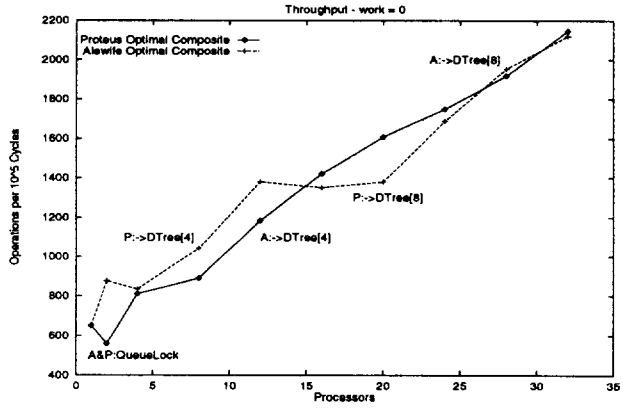

Figure 15: Optimal Composite on Proteus and normalized Alewife