

The TickerTAIP Parallel RAID Architecture

PEI CAO, SWEE BOON LIM, SHIVAKUMAR VENKATARAMAN, and
JOHN WILKES
Hewlett-Packard Laboratories

Traditional disk arrays have a centralized architecture, with a single controller through which all requests flow. Such a controller is a single point of failure, and its performance limits the maximum number of disks to which the array can scale. We describe TickerTAIP, a parallel architecture for disk arrays that distributes the controller functions across several loosely coupled processors. The result is better scalability, fault tolerance, and flexibility.

This article presents the TickerTAIP architecture and an evaluation of its behavior. We demonstrate the feasibility by a working example, describe a family of distributed algorithms for calculating RAID parity, discuss techniques for establishing request atomicity, sequencing, and recovery, and evaluate the performance of the TickerTAIP design in both absolute terms and by comparison to a centralized RAID implementation. We also analyze the effects of including disk-level request-scheduling algorithms inside the array. We conclude that the Ticker TAIP architectural approach is feasible, useful, and effective.

Categories and Subject Descriptors B.4.2 [Input / Output and Data Communications]: Input / Output Devices—*channels and controllers*; D.1.3 [Programming Techniques]: Concurrent Programming—*parallel programming*; D.4.2 [Operating Systems]: Storage Management—*secondary storage*; D.4.7 [Operating Systems]: Organization and Design—*distributed systems*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Decentralized parity calculation, disk scheduling, distributed controller, fault tolerance, parallel controller, performance simulation, RAID disk array

1. INTRODUCTION

A disk array is a structure that connects several disks together to extend the cost, power, and space advantages of small disks to higher-capacity configurations. By providing partial redundancy such as parity, availability can be

An earlier version of this article was presented at the 1993 International Symposium on Computer Architecture

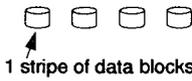
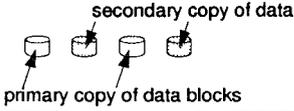
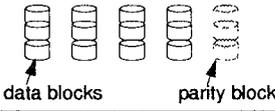
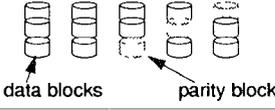
Authors' addresses: P. Cao, Princeton University, Department of Computer Science, Princeton, NJ 08540; email: pc@cs.princeton.edu; S. B. Lim, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, IL 61801; email: sbllm@cs.uiuc.edu; S. Venkataraman, University of Wisconsin, Department of Computer Science, 1210 West Dayton Street, Madison, WI 53706; email: venkatar@cs.wisc.edu; J. Wilkes, Hewlett-Packard Laboratories, PO Box 10490, 1U13, Palo Alto, CA 94304-0969; email: wilkes@hpl.hp.com.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1994 ACM 0734-2071/94/0800-0236\$03.50

ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994, Pages 236–269

Table I. Some Common RAID Levels

Level	redundancy technique	placement of redundant data	diagrammatic rendition
0 <i>striping</i>	none	none	 1 stripe of data blocks
1 <i>mirroring</i>	complete	complete disks	 secondary copy of data primary copy of data blocks
3	parity across a stripe of data	one disk dedicated to parity	 data blocks parity block
5	parity across a stripe of data	parity rotates round-robin across all disks	 data blocks parity block

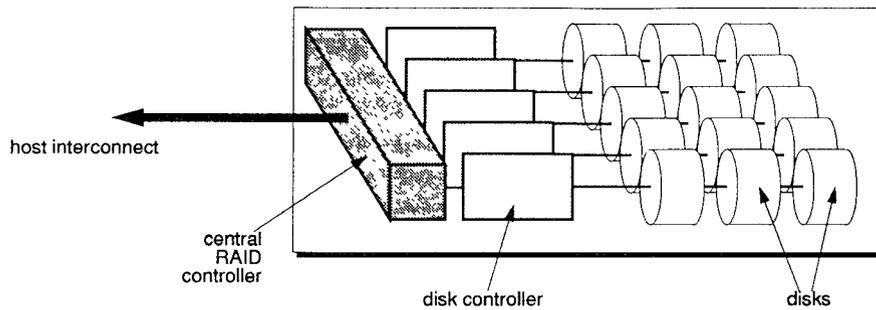


Fig. 1. Traditional RAID array architecture.

increased as well. Such *RAIDs* (for *Redundant Arrays of Inexpensive Disks*) were first described in the early 1980s [Lawlor 1981; Park and Balasubramanian 1986], and popularized by the work of a group at UC Berkeley [Patterson et al. 1988; 1989]. The RAID terminology encompasses a number of different levels, corresponding to different amounts of redundancy and placement of the redundant data. The most commonly encountered of these are summarized in Table I.

To implement a disk array that provides one or more of the RAID levels, the traditional RAID array architecture, shown in Figure 1, has a central controller, one or more disks, and multiple head-of-string disk interfaces. The RAID controller interfaces to the host, processes read and write requests, and

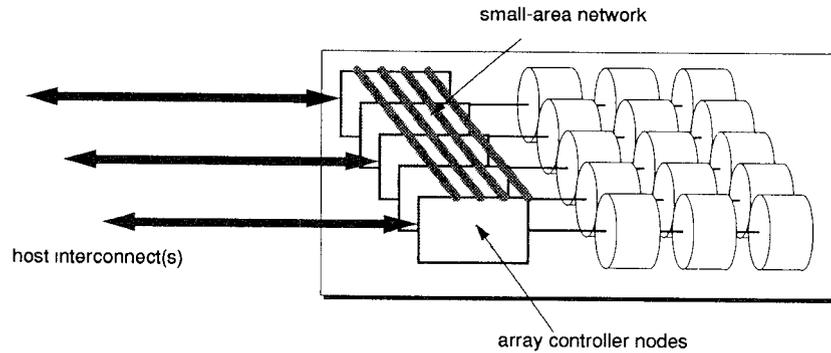


Fig. 2 TickerTAIP array architecture.

carries out parity calculations, block placement, and data recovery after a disk failure. The disk interfaces pass on the commands from the controller to the disks via a disk interconnect of some sort—these days, most often a variety of the *Small Computer System Interface*, or *SCSI* bus [SCSI 1991].

Obviously, the capabilities of the RAID controller are crucial to the performance and availability of the system. If the controller's bandwidth, processing power, or capacity are inadequate, the performance of the array as a whole will suffer. (This is increasingly likely to happen: for example, parity calculation is memory bound, and memory speeds have not kept pace with recent CPU performance improvements [Ousterhout 1990].) A high latency through the controller can reduce the performance of small requests. The single point of failure that the controller represents can also be a concern: failure rates for disk drives and packaged electronics are now similar, and one of the primary motivations for RAID arrays is to survive the failure rates that result from having many disks in a system. Although some commercial RAID array products include spare RAID controllers, they are not normally simultaneously active: one typically acts as a backup for the other, and is held in reserve until it is needed because of failure of the primary. (For example, this technique was suggested in Gray et al. [1990].) This is expensive: the backup has to have all the capacity of the primary controller, but it provides no useful services in normal operation. Alternatively, both controllers are active simultaneously, but over disjoint sets of disks. This limits the performance available from the array, even though the controllers can be fully utilized.

To address these concerns, we have developed the *TickerTAIP architecture for parallel RAIDs*. In this architecture (Figure 2), there is no central controller: it has been replaced by a cooperating set of *array controller nodes* that together provide all the functions needed by operating in parallel. The TickerTAIP architecture offers several benefits, including: fault tolerance (no central controller to break), performance scalability (no central bottleneck), smooth incremental growth (by simply adding another node), and flexibility (it is easy to mix and match components).

This article provides an evaluation of the TickerTAIP architecture. Its main emphasis is on the techniques used to provide parallel, distributed controller functions and their effectiveness.

1.1 Outline

We begin this article by presenting an overview of the TickerTAIP architecture and related work, and follow it with a detailed description of several design issues, including descriptions and evaluations of algorithms for parity calculation, recovery from controller failure, and extensions to provide sequencing of concurrent requests from multiple hosts.

To evaluate TickerTAIP we constructed a working prototype as a functional testbed, and then built a detailed event-based simulation that we calibrated against this prototype. These tools are presented as background material for the simulation-based performance analysis of TickerTAIP that follows, with particular emphasis on comparing it against a centralized RAID implementation. We conclude with a summary of our results.

2. THE TICKERTAIP ARCHITECTURE

A TickerTAIP array is composed of a number of *worker nodes*, which are nodes with one or more local disks connected through a bus. *Originator nodes* provide connections to host computer clients. The nodes are connected to one another by a high-performance, small-area network with sufficient internal redundancy to survive single failures.

Mesh-based switching fabrics can achieve the bandwidth, latency, and availability needs with reasonable costs and complexity across a reasonable scale of array sizes. A design that would meet the performance, scalability, and fault tolerance needs of a TickerTAIP array is described in Wilkes [1991]. A similar scheme has been described in Shin [1991]. For smaller arrays, the interconnect could even be a pair of backplanes. (For example, PCI is capable of running at well over 100MB/s [PCI 1994], which would support 15–20 disks for bandwidth-limited applications, or many hundred disks if the workload was small, random I/Os.) Multiple, independent arrays will become more cost effective at some sufficiently large scale: no interconnect scales perfectly. However, TickerTAIP's requirements on the switching fabric are relatively light, and this point will probably only be reached with arrays that are so large that such a split will probably be desirable for other reasons.

In Figure 2, the nodes are shown as being both workers and originators: that is, they have both host and disk connections. As a result, designing a node requires that both the host interface and the disk interface be designed together, and adding a disk node requires paying for another host connection.

A second design that avoids these problems is shown in Figure 3. It uses separate disk-controller (*worker*) nodes and host-interface (*originator*) nodes. This allows arbitrary mixing and matching of node types (such as SCSI-originator, FDDI-originator, IPI-worker, SCSI-worker), which makes building a TickerTAIP array with several different kinds of host interface simply a configuration-time question, not a design-time one. Since each node is plug

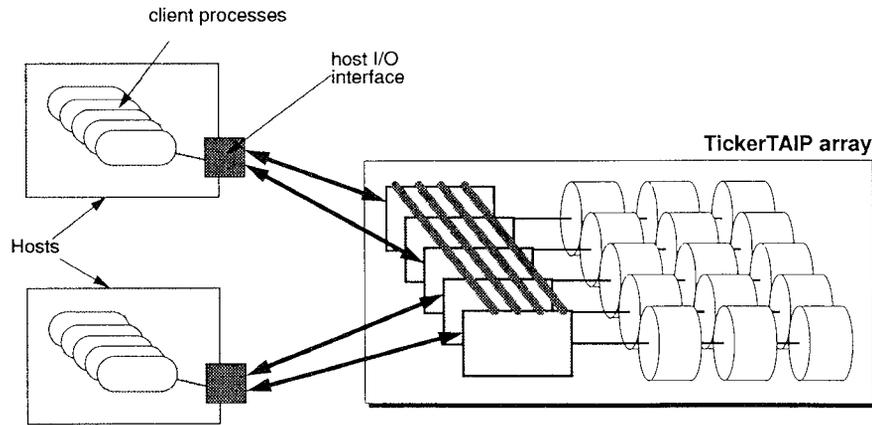


Fig. 3. TickerTAIP system environment

compatible from the point of view of the internal interconnect, it is easy to configure an array with any desired ratio of worker and originator nodes, a flexibility less easily achieved in the traditional centralized architecture.

Figure 3 shows the environment in which we envision a TickerTAIP array operating. The array provides disk services to one or more *host* computers through the originator nodes. There may be several originator nodes, each connected to a different host; alternatively, a single host can be connected to multiple originators for higher performance and greater failure resilience. For simplicity, we require that all data for a request be returned to the host along the path used to issue the request.

In the context of this model, a traditional RAID array looks like a TickerTAIP array with several unintelligent worker nodes, a single originator node on which all the parity calculations take place, and shared-memory communication between the components.

One assumption we made in this study is that parity calculation is a driving factor in determining the performance of a RAID array. The TickerTAIP architecture does these calculations in a decentralized fashion; other high-speed array controller designs (e.g., RAID-II [Drapeau et al. 1994] use a central parity calculation engine. Our approach is predicated on two beliefs: (1) that processors are cost-effective engines for calculating parity and (2) that memory bandwidth, rather than processor cycles, is the determining cost factor in providing this functionality. (By way of example, the bandwidth and functionality requirements of the RAID-II engine required a controller card nearly two feet on a side.) The TickerTAIP architecture reduces the per-processor parity calculation requirements sufficiently far that the cheap commodity microprocessors it uses for the control functions can also be used as the parity calculation engines. At this point, the diseconomies of scale associated with providing high-bandwidth data paths to a hardware parity calculation engine will overwhelm any intrinsic simplicity in the use of specialized logic to perform the exclusive-OR calculation. As the performance

of commodity microprocessors continues to improve at its current rate, the range of array sizes over which this argument holds will only increase.

2.1 Related Work

Many papers have been published on RAID reliability, performance, and on design variations for parity placement and recovery schemes [Clark et al. 1988; Gibson et al. 1989; Menon and Kasson 1992; Schulze et al. 1989; Dunphy et al 1990; Gray et al 1990; Lee 1990; Muntz and Lui 1990; Holland and Gibson 1992]. Our work builds on these studies: we concentrate here on the architectural issues of parallelizing the techniques used in a centralized RAID array, so we take such work as a given—and assume familiarity with basic RAID concepts in the following discussion.

The HP7937 family of disks realizes a physical architecture similar to that of TickerTAIP [Hewlett-Packard 1988]. These disks can be connected together by a 10MB/s bus, which allows access to “remote” disks as well as fast switch-over between attached hosts in the event of system failure. No multi-disk functions (such as a disk array) were provided, however.

Several “shared-nothing” database systems use a hardware architecture similar to that adopted for TickerTAIP, including Bubba [Boral 1988; Copeland et al. 1988], Gamma [DeWitt et al. 1986; 1988], Teradata [Neches 1984; Sloan 1992], and Tandem [Bartlett et al. 1990; Siewiorek and Swarz 1992]. However, none appears to use a distributed RAID implementation across multiple nodes, and all are intended as database engines rather than parallel implementations of RAID. On the other hand, TickerTAIP makes extensive use of well-known techniques such as two-phase commit and partial-write ordering from the database community [Gray 1978].

A proposal was made to connect networks of processors to form a widely distributed RAID controller in Stonebraker [1989]. This approach was called *RADD—Redundant Arrays of Distributed Disks*. It proposed using disks spread across a wide-area network to improve availability in the face of a site failure. In contrast to the RADD study, we emphasize the use of parallelism inside a single RAID server; we assume the kind of fast, reliable interconnect that is easily constructed inside a single-server cabinet; we couple processors and disks closely, so that a node failure is treated as (one or more) disk failures; and we provide much improved performance analyses—Stonebraker used “all disk operations take 30 ms.” The result is a new, detailed characterization of the parallel RAID design approach in a significantly different environment.

3. DESIGN ISSUES

This section describes the TickerTAIP design issues in some detail. It begins with an examination of normal mode operation (i.e., in the absence of faults) and then examines the support needed to cope with failures. Table II may prove helpful in understanding the data layout used for the RAID5 array we are describing.

Table II. Data Layout for a 5-Disk Left-Symmetric RAID 5 Array [Lee 1990]

<i>stripe</i>	<i>logical block number</i>				
0	0	1	2	3	P
1	5	6	7	P	4
2	10	11	P	8	9
3	15	P	12	13	14
4	P	16	17	18	19

Each column represent a disk. The shaded, outlined area represents one possible request that spans most of stripe 1 (a “large stripe”), all of stripes 2 and 3 (“full stripes”), and a small amount of stripe 4 (a “small stripe”). Parity blocks have darker shading and are marked with a P.

3.1 Normal-Mode Reads

In normal mode, no parity computation is required for reads, so they are quite straightforward. All the necessary data is read at the workers and forwarded to the originator, where it is assembled, and transmitted to the host in the correct order. The main performance issue that arises has to do with skipping over the parity blocks: we found it beneficial to perform sequential reads of both data and parity, and then to discard the parity blocks inside the worker nodes, rather than to generate separate requests that omitted reading the parity blocks.

3.2 Normal-Mode Writes

In a RAID array, writes require calculation or modification of stored parity to maintain the partial data redundancy. Each stripe is considered separately in determining the method and site for parity computation, since this is the unit across which the partial redundancy is maintained. The discussion that follows describes the algorithms executed on each of the stripes that a single request spans.

3.2.1 How to Calculate New Parity. The first design choice is how to calculate the new parity. There are three alternatives, depending upon how much of the stripe is being updated (Figure 4):

- full stripe*: all of the data blocks in the stripes have to be written, and parity can be calculated entirely from the new data;
- small stripe*: less than half of the data blocks in a stripe are to be written, and parity is calculated by first reading the old data of the blocks that will be written, XORing them with the new data, and then XORing the results with the old parity block data;
- large stripe*: more than half of the data blocks in the stripe are to be written; the new parity block can be computed by reading the data blocks in the stripe that are not being written and XORing them with the new data (i.e., reducing this to the full-stripe case) [Chen et al. 1990].

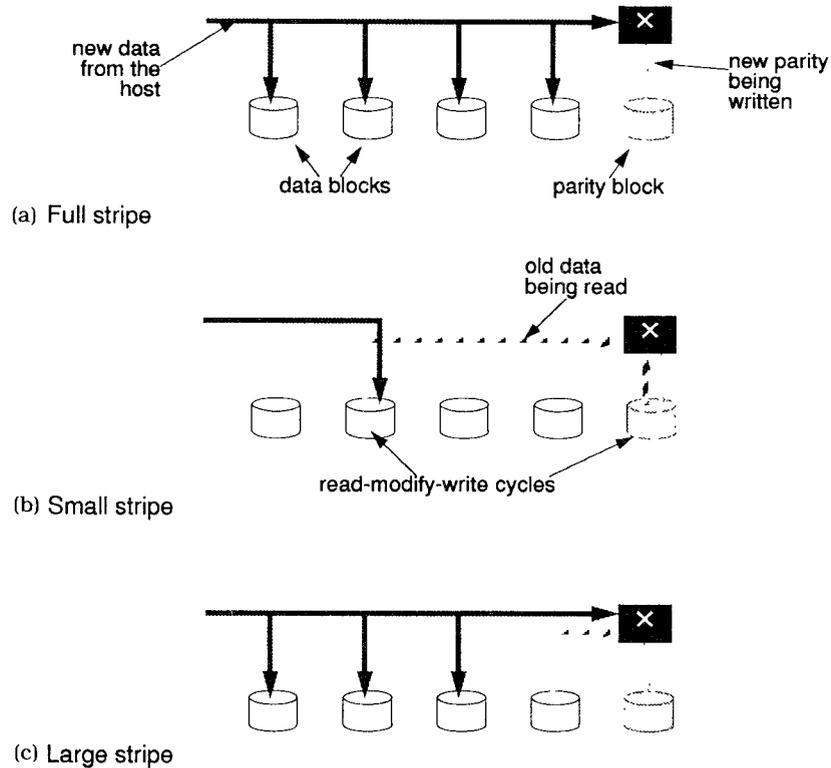


Fig. 4. Three different stripe update size policies. \times indicates where parity calculations occur.

Notice that a single request might span all three kinds of stripes, although all but the first and last stripe will always be full ones.

The large-stripe mode is just a (potential) performance optimization, since the right behavior can be obtained from using just the small- and full-stripe cases. We discuss whether it is beneficial in practice later.

3.2.2 Where to Calculate New Parity. The second design consideration is where the parity is to be calculated. Traditional centralized RAID architectures calculate all parity at the originator node, since only it has the necessary processing capability. In TickerTAIP, every node has a processor, so there are several choices. The key design goal is to load balance the work among the nodes—in particular, to spread out the parity calculations over as many nodes as possible. Here are three possibilities (shown in Figure 5):

- at originator*: all parity calculations are done at the originator;
- solely-parity*: all parity calculations for a stripe take place at the parity node for that stripe;
- at-parity*: same as for solely-parity, except that partial results during a small-stripe write are calculated at the worker nodes and shipped to the parity node.

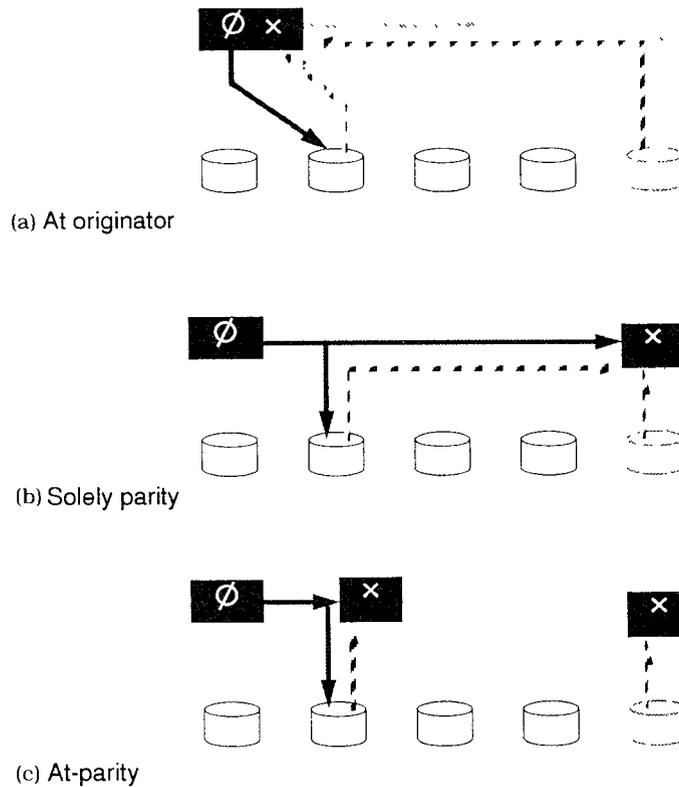


Fig. 5. Three different places to calculate parity \emptyset indicates the originator node; \times indicates the node where parity calculations occur.

The solely-parity scheme always uses more messages than the at-parity one, so we did not pursue it further. We provide performance comparisons between the other two, later in the article.

3.3 Single Failures — Request Atomicity

We begin with a discussion of single-point failures. Notice that a primary goal of a regular RAID array is to survive single-disk failures.¹ The TickerTAIP architecture extends this to include failure of a part of its distributed controller: we do not make the simplifying assumption that the controller is not a possible failure point. The way in which TickerTAIP is intended to be used provides duplex paths to its host (see Figure 3), and since there are several techniques for doing so, we have legislated that the internal interconnect fabric is itself single-fault resilient. As a result the overall architecture is

¹There are variants of the parity calculation schemes that can compensate for multiple disk failures [Gibson et al 1989]. The extension of the Ticker TAIP architecture to cover these cases is straightforward, and not discussed further here.

Table III. Algorithms used to perform a write in failure mode, as a function of the kind of block being written to, and the amount of the stripe being updated

<i>stripe size</i>	<i>physical block type on failed disk</i>		
	<i>updated</i>	<i>parity</i>	<i>not updated</i>
small	large stripe strategy	none	small stripe strategy
large	large stripe strategy	none	small stripe strategy
full	full stripe strategy	none	—

capable of surviving a fault in *any* single system component. However, there are certain requirements on the software algorithms used at the nodes in a TickerTAIP system to ensure correct operation in the presence of faults. This section discusses the first of them: the need to provide request atomicity.

Just as with a regular RAID array, packaging and power-supply issues are very important if the system availability is to be maximized. Some of these decisions are discussed in Schulze [1988] and Schulze et al. [1989]; the design approach used for these questions in a TickerTAIP-based array is identical to that used for a regular disk array.

3.3.1 Disk Failure. In TickerTAIP, a disk failure is treated in just the same way as in a traditional RAID array: the array continues operation in degraded (failed) mode until the disk is repaired or replaced; the contents of the new disk are reconstructed; and execution resumes in normal mode. From the outside of the array, the effect is as if nothing has happened. Inside, appropriate data reconstructions occur on reads, and I/O operations to the failed disk are suppressed. Exactly the same algorithm is used if an entire disk string goes bad for some reason. The algorithms are summarized in Table III.

3.3.2 Worker Failure. A TickerTAIP worker failure is treated just like a disk failure, and is masked in just the same way. (Just as with a regular RAID controller with multiple disks per head-of-string controller, a failing worker means that an entire column of disks is lost at once, but the same recovery algorithms apply.) We assume fail-silent nodes so that we can significantly simplify the fault-isolation and normal-case protocols we use between the nodes. In practice, the isolation offered by the networking protocols used to communicate between nodes is likely to make this assumption realistic in practice for all but the most extreme cases—for which RAID arrays are probably not appropriate choices. (In support of our position, Gray [1988] explains that the complexities of handling the more complicated Byzantine failure modes are rarely deemed worthwhile in practice.)

A node is suspected to have failed if it does not respond to an “are you alive” request within a reasonable time. (This is the only place that such time-outs occur, to simplify the maintenance of other portions of the system.) The node that detects a failure of another node initiates a distributed consensus protocol much like two-phase commit, taking the role of coordina-

tor of the consensus protocol. All the remaining nodes reach agreement by this means on the number and identity of the failed node(s). This protocol ensures that all the remaining nodes enter failure mode at the same time. Multiple failures cause the array to shut itself down safely to prevent possible data corruption.

3.3.3 Originator Failure and Request Atomicity. Failure of a node with an originator on it brings new concerns: a channel to a host is lost; any worker on the same node will be lost as well; and the fate of requests that arrived through this node needs to be determined since the failed originator was responsible for coordinating their execution.

Originator failures during reads are fairly simple: the read operation is aborted since there is no longer a route to communicate its results back to the host.

Failures during write operations are more complicated, because different portions of the write could be at different stages unless extra steps are taken to avoid compromising the consistency of the stripes involved in the write. Worst is failure of a node that is both a worker and an originator, since it will be the only one with a copy of the data destined for its own disk. (For example, if such a node fails during a partial-stripe write after some of the blocks in the stripe have been written, it may not be possible to reconstruct the state of the entire stripe, violating the rule that a single failure must be masked.)

Our solution to both these concerns is to ensure *write atomicity*: that is, either a write operation completes successfully, or it makes no changes. Notice that this is a much stronger guarantee than provided by single disk drives or non-parity-protected disk arrays. With these, the content of a range of logical blocks being written to is indeterminate until the write completes successfully. If a write request is aborted or fails, the contents of the targeted range will be in an indeterminate state. To achieve this guarantee, we added a two-phase commit protocol to write operations. Before a write can proceed, sufficient data must be replicated in more than one node's memory to let the operation restart and complete—even if an arbitrary node fails. If this cannot be achieved, the request is aborted before it can make any changes. (A similar problem occurs in RAID controllers that have two-part nonvolatile write caches that must tolerate failure of either cache half, possibly in conjunction with concurrent disk failures. This issue is discussed in Menon and Courtney [1993]; similar solutions to the one we adopted serve there as well.)

We identified two approaches to implementing the two-phase commit: *early commit* tries to make the decision as quickly as possible; *late commit* delays its commit decision until all that is left to do are the writes. We describe them in reverse order, since late commit is the simpler of the two.

In *late commit*, the commit point (when it decides whether to continue or not) is reached only after the parity has been computed. The reason for this choice is that the computed parity data, suitably distributed, provides exactly the partial redundancy needed. In late commit, all that remains to be done after the commit decision is to perform the writes.

Table IV. Data needed for recovering a stripe during a write, and the stripe-size strategy used to do so

<i>failed node</i>	<i>block type at failed node</i>	<i>stripe size</i>		
		<i>small stripe</i>	<i>large stripe</i>	<i>full stripe</i>
<i>originator</i>	<i>updated</i>	parity node has copy; large-stripe strategy	parity node has copy; large-stripe strategy	updated and parity nodes have copy, full-stripe strategy
	<i>parity</i>	parity not computed	parity not computed	parity not computed
	<i>not-updated</i>	—	parity node has copy; large-stripe strategy	—
<i>worker</i>	<i>updated</i>	originator has copy; large-stripe strategy	originator has copy; large-stripe strategy	originator has copy; full-stripe strategy
	<i>parity</i>	parity not computed	parity not computed	parity not computed
	<i>not updated</i>	—	parity node has copy; large-stripe strategy	—

In *early commit*, the goal is for the array to get to its commit point as quickly as possible during the execution of the request. This requires that the new data destined or the originator/worker node has to be replicated elsewhere, in case the originator fails after commit. The same must be done for old data being read as part of a large-stripe write, in case the reading node fails before it can provide the data. We duplicate this data on the parity nodes of the affected stripes—this involves sending no additional data in the case of parity calculations at the parity node (which we will see below is the preferred policy). The commit point is reached as soon as the necessary redundancy has been achieved.

Late commit is much easier to implement, but has somewhat lower concurrency and higher request latency. We explore the magnitude of this cost later.

A write operation is aborted if any involved worker does not reach its commit point. When a worker node fails, the originator is responsible for restarting the operation. In the event of an originator failure, a temporary originator, chosen from among those nodes that were already participating in the request, is elected to complete or abort it processing. Choosing one of the nodes that was already participating in the request minimizes data and control traffic interchanges, since these nodes already have the necessary information about the request itself.

Table IV summarizes the different cases that need to be accommodated. For each combination of node role and block type that has been lost, it shows which node has a copy of the data required for recovery, and the write policy that must be applied to the stripe.

3.4 Multiple Failures — Request Sequences

This section discusses measures designed to help limit the effects of multiple concurrent failures. The RAID architecture tolerates any single disk failure. However, it provides no behavior guarantees in the event of multiple failures

(especially power-fail), and it does not ensure the independence of overlapping requests that are executing simultaneously. In the terminology of Lampson and Sturgis [1981], multiple failures are disasters: events outside the covered fault set for RAID. TickerTAIP introduces coverage for partial controller failures; and it goes beyond this by using *request sequencing* to limit the effects of multiple failures in a way that is useful to file system designers.

As with a regular RAID, a power-fail during a write can corrupt the stripe being written to unless more extensive recover techniques (such as intentions logging) are used—in this respect, TickerTAIP is exactly emulating the RAID failure model. Power failures can be handled by the use of an uninterruptible power supply for both TickerTAIP and a regular RAID array, but TickerTAIP's request sequencing also provides improved performance to hosts wishing to tolerate crashes and other failures. Strengthening the regular RAID failure guarantees in the controller follows naturally as a consequence of wanting to maximize performance in the array; in turn, doing so at the lower level allows the host to simplify its own failure recover mechanisms.

3.4.1 Requirements. File system designers rely typically on the presence of ordering invariants to allow them to recover from crashes or power failure. For example, in 4.2BSD-based file systems, metadata (inode and directory) writes must occur before the data to which they refer is allowed to reach the disk [McKusick et al. 1984]. The simplest way to achieve this is to defer queueing the data write until the metadata write has completed. Unfortunately, this can severely limit concurrency: for example, parity calculations can no longer be overlapped with the execution of the previous request. This is unfortunate, and becoming more so, as the technology of disk drives improves to include command queueing, immediate reporting, and more nearly optimal request sequencing that exploits position information available only at the disk itself [Seltzer et al. 1990; Jacobson and Wilkes 1991; Ruemmler and Wilkes 1993].

A better way to achieve the desired invariant is to provide—and preserve—partial write orderings in the I/O subsystem. This technique can significantly improve file system performance. From our perspective as RAID array designers, it also allows the RAID array to make more intelligent decisions about request scheduling. We discuss the effects of some of these scheduling decisions later in the article.

A TickerTAIP array can be configured to support multiple hosts. As a result, some mechanism needs to be provided to let requests from different hosts be serialized without recourse to either sending all requests through a single host or requiring one request to complete before the next can be issued.

Finally, multiple overlapping requests from a single or multiple hosts can be in flight simultaneously. This could lead to parts of one write replacing parts of another in a nonserializable fashion, which clearly should be prevented. (Our write commit protocols provide atomicity for each request, but no serializability guarantees.)

3.4.2 Request Sequencing. To address these requirements, we introduced a request-sequencing mechanism using partial orderings for both reads and

writes. Internally, these are represented in the form of *directed acyclic graphs* (DAGs): each request is represented by a node in the DAG, while the edges of the DAG represent dependencies between requests.

To express the DAG, each request is given a unique identifier. A request is allowed to list one or more requests on which it depends *explicitly*; TickerTAIP guarantees that the effect is *as if* no request begins until the requests on which it depends complete (this allows the implementation the freedom to perform eager evaluation, some of which we exploited in our testbed prototype). If a request is aborted, all requests that depend explicitly on it are also aborted (and so on, transitively).

If a host later wishes to reissue any of the aborted dependent requests, it is free to do so, of course. Having TickerTAIP itself propagate the abort to dependent requests preserves sequencing guarantees without requiring a handshake with the host on every operation in the normal (error-free) case. An alternative design² would have TickerTAIP enter a special mode once it detects any abort, during which it would execute no requests until all the hosts had acknowledged that they had aborted any and all requests that depended on the failed one. This would push the dependency-handling back up into the hosts, but at the cost of a more complicated and fragile recovery protocol. Additionally, giving the dependency data to TickerTAIP allows it to determine which requests can be executed in parallel, thereby improving performance in the normal case.

Also, TickerTAIP will arbitrarily assign sufficient *implicit* dependencies to prevent overlapping requests from executing concurrently. Aborts are not propagated across implicit dependencies. In the absence of explicit dependencies, the order in which requests are serviced is some arbitrary serializable schedule.

3.4.3 Sequencer States. The management of sequencing is performed through a high-level state table, with the following states (diagrammed, with their transitions, in Figure 6):

- NotIssued*: the request itself has not yet reached TickerTAIP, but another request has referred to this request in its dependency list.
- Unresolved*: the request has been issued, but it depends on a request that has not yet reached the TickerTAIP array.
- Resolved*: all of the requests that this one depends on have arrived at the array, but at least one has yet to complete.
- InProgress*: all of a request's dependencies have been satisfied, so it has begun executing.
- Completed*: a request has successfully finished.
- Aborted*: a request was aborted, or a request on which this request depended explicitly has been aborted.

²Due to one of the anonymous reviewers.

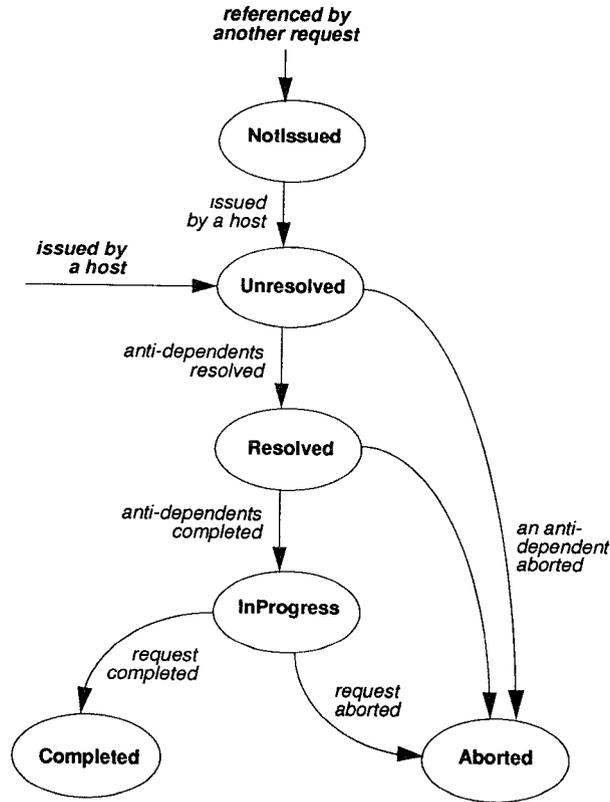


Fig. 6. States of a request. An “antidependent” is a request that this request is waiting for.

Old request state has to be garbage-collected. We do this by requiring that the hosts number their requests sequentially and by keeping track of the oldest outstanding incomplete request from each host. When this request completes, any older completed requests can be deleted. Any request that depends on an older request than the oldest recorded one can consider the dependency immediately satisfied.

Aborts are an important exception to this mechanism since a request that depends on an aborted request should itself be aborted, whenever the original request was aborted—even if this was some considerable time in the past. The simplest solution is to require that a host never issue a request that depends on one that has been aborted, but this would require an unnecessary serialization at the host. As a result, we decided to propagate aborts to other requests already in the TickerTAIP array. Unfortunately, this is not enough: there is a potential race condition between the request being aborted and the host being told about it, and the host ceasing to emit further requests that may depend on the aborted request. Our solution is to maintain state about aborted requests for a guaranteed minimum time—10 seconds in our prototype. (This is not ideal: in the presence of a large number of cascaded aborts,

we may have to delay accepting new commands until the 10 seconds are up. However, we believe this situation is likely to be extremely rare in practice.) Similarly, a time-out on the *NotIssued* state can be used to detect errors such as a host that never issues a request for which other requests are waiting.

3.4.4 Sequencer Design Alternatives. We considered four designs for the sequencer mechanism:

- (1) *Fully centralized*: a single, central sequencer manages the state table and its transitions. (A primary and a backup are used to eliminate a single point of failure.) In the absence of contention, each request suffers two additional round-trip message latency times: between the originator and the sequencer, and between the sequencer and its backup. One of these trips is not needed if the originator is co-located with the sequencer.
- (2) *Partially centralized*: a centralized sequencer handles the state table until all the dependencies have been resolved, at which point the responsibility is shifted to the worker nodes involved in the request. This requires that the status of every request be sent to all the workers, to allow them to do the resolution of subsequent transitions. This has more concurrency, but requires a broadcast on every request completion.
- (3) *Originator driven*: in place of a central sequencer, the originator nodes (since there will typically be fewer of these than the workers) conduct a distributed-consensus protocol to determine the overlaps and sequence constraints, after which the partially centralized approach is used. This always generates more messages than the centralized schemes.
- (4) *Worker driven*: the workers are responsible for all the states and their transitions. This widens the distributed-consensus protocol to every node in the array, and still requires the end-of-request broadcast.

Although the higher-numbered of the above designs may increase concurrency, they do so at the cost of increased message traffic complexity.

We chose to implement the fully centralized model in our prototype, largely because of the complexity of the failure recovery protocols required for the alternatives. As expected, we measured the resulting latency to be that of two round-trip messages (i.e., $440\mu s$ in our prototype) plus a few tens of microseconds of state table management. We believe this additional overhead acceptable for the benefits that request sequencing provides. Nonetheless, we made request sequencing optional for those cases where it is not needed.

3.5 The RAIDmap

Previously sections have presented the policy issues; this one discusses an implementation technique we found useful. Our first design for sequencing and coordinating requests retained a great deal of centralized authority: the originator tried to coordinate the actions taking place at each of the different nodes (reads and writes, parity calculations). We soon found ourselves faced with the messy problem of coping with a complex set of interdependent actions taking place on multiple remote nodes, and coordinating these proved

Stripe	node 0	node 1	node 2	node 3	Type
0	-,-,- unused	-,-,- unused	2,0,3 data	-,0,- parity	small stripe
1	4,1,2 data	5,1,2 data	-,1,- parity	3,1,2 data	full stripe
2	-,-,- unused	-,-,- parity	6,2,1 data	7,2,1 data	large stripe

Fig 7. RAIDmap example for a write request spanning logical blocks 2 through 7. Each cell in the figure represents a physical block on a disk, and contains a four-part tuple. The parts are: the logical block number in the array; the physical block number on this disk (which equates to the stripe number if there is only one disk on each node); the node number to send parity data to, and a block type. The rightmost column is discussed in Section 3.2.1

exceedingly complex—especially so when potential failure modes were taken into consideration.

To avoid this complexity, we developed a new approach: rather than having the originator tell each worker what it had to do, and coordinate the stream of asynchronous events that resulted, we delegated management of its own work to each worker, and then coded everything to *assume that all the nodes were doing what they were supposed to without any further prompting*. So, once the workers are told about the original request (its starting point and length, and whether it is a read or a write) and given any data they needed, they can proceed on their own. For example, if node A needs data from node B, A can rely on B to generate and ship the data to A with no further prompting. We call this approach *collaborative execution*: it is characterized by each node assuming that other nodes are *already* doing their part of the request. It proved to be an enormous simplification.

To orchestrate all the work, we developed a structure known as a *RAIDmap*; a two-dimensional array with an entry for each column (worker) and each stripe.³ Each worker builds its column of the RAIDmap, filling in the blanks as a function of the operation (read or write), the layout policy, and the execution policy. The *layout policy* determines where data and parity blocks are placed in the RAID array (e.g., mirroring, or RAID 4 or 5). The *execution policy* determines the algorithm used to service the request (e.g., where parity is to be calculated). A simplified RAIDmap is shown in Figure 7.

One component of the RAIDmap is a state table for each block (the states are described in Table V). It is this table that the layout and execution policies fill out. A request enters the states in order, leaving each one when the associated function has been completed, or immediately if the state is marked as “not needed” for this request. For example, a read request will

³Although the idea of the RAIDmap is more simply described as if the full array was present, in practice there is no need to actually generate all the rows of the array for a very long request, since the inner full-strip descriptions all look pretty much alike.

Table V. State-Space for each Block at a Worker Node

<i>State</i>	<i>Function</i>	<i>Other information</i>
1	Read old data	disk address
2	XOR old with new data	
3	Send old data (or XORed old data) to another node	node to send it to
4	Await incoming data (or XORed data)	number of blocks to wait for
5	XOR incoming data with local old data/parity	
6	Write new data or parity	disk address

enter state 1 (to read the data), skip through state 2 to state 3 (to send it to the originator), and then skip through the remaining states.

The RAIDmap proved to be a flexible mechanism, allowing us to test out several different policy alternatives (e.g., whether to calculate partial parity results locally, or whether to send the data to the originator or parity nodes). Additionally, the same techniques is used in failure mode: the RAIDmap indicates to each node how it is to behave, but now it is filled out in such a way as to reflect the failure mode operations. Finally, the RAIDmap simplified the configuring of a centralized implementation, using the same policies and assumptions as in the distributed case.

The goal of any RAID implementation is to maximize disk utilization and minimize request latency. To help achieve this, the RAIDmap computation is overlapped with other operations, such as moving data or accessing disks. For the same reason, workers send data needed elsewhere before servicing their own parity computations or local disk transfers.

It also proved important to optimize the disk accesses themselves. When we delayed writes in our prototype implementation until the parity data was available, throughput increased by 25–30% because the data and parity writes were coalesced together, reducing the number of disk seeks needed.

In implement the two-phase commit protocols described in Section 3.3, additional states were added to the worker and originator node state tables. The placement of these additional states determines the commit policy: for early commit, as soon as possible; for late commit, just before the writes.

3.6 Scheduling Disk Accesses

Unlike traditional centralized RAID designs, TickerTAIP provides request atomicity and sequencing to support multiple outstanding requests. As a result, more than one disk access request can be queued at a worker node at one time, which means that it is beneficial to consider more sophisticated request-scheduling policies inside the array (preserving the write-order invariants determined by the sequencing algorithms, of course). In theory, a worker node could use any of the algorithms proposed in the (fairly extensive) literature on disk scheduling. In practice, we are mostly interested in those

that are inexpensive and yet give good performance. We report here on our experiments with four such algorithms.

- first come first served* (FCFS): that is, no request reordering—this is what is implemented in the working prototype described below, and which is used for the majority of the results we present;
- shortest seek time first* (SSTF): the request that has the shortest seek time from the current disk head position is served first;
- shortest access time first* (SATF): the request that has the shortest access time (seek time + rotation time) from the current disk head position is served first [Seltzer et al. 1990; Jacobson and Wilkes 1991];
- batched nearest neighbor* (BNN): like SATF, except that requests are batched—each time it runs, the scheduler takes all the requests currently in the queue as a batch, and runs the SATF algorithm over them; it does not attempt to serve any new requests until the current batch is finished.

Among these algorithms, SATF gives generally the best throughput when applied to Unix system-like workloads, but can potentially starve requests. BNN remedies this at the cost of a small reduction in throughput.

We found that scheduling improved both the throughput and average response time of requests. The improvement depended on the workload and load condition of the array, and (as expected) was largest under heavy loads. The results are reported in Section 4.7.

3.7 Memory Management

The main functionality issue we did not address explicitly in this work is buffer management at the originator nodes. In a real system, memory limitations would complicate some of the algorithms presented here. For example, additional flow control might be needed to ensure that the originator memory would not get swamped if the array was presented with many large requests. However, these costs will be small: by definition, they only show up if the requests are larger than would fit comfortably into the memory of an originator node, so the cost of the flow control will be largely hidden by the cost of moving the data.

Alternatively, the originator node might choose to break up very large requests up into chunks with some maximum size. This approach is commonly used in disk drive controllers today; the main difference would be the use of much larger chunk sizes to ensure that the array could deliver data at close to its full potential bandwidth.

4. EVALUATING TICKERTAIP

This section presents the vehicles we used to evaluate the design choices and performance of the TickerTAIP architecture.

4.1 The Prototype

We first constructed a working *prototype implementation* of the TickerTAIP design, including all the fault tolerance features described above. The intent

Table VI. Characteristics of the HP97560 Disk Drive

<i>property</i>	<i>value</i>
diameter	5.25"
surfaces/heads	19 data, 1 servo
formatted capacity	1.3GB
track size	72 sectors
sector size	512 bytes
rotation speed	4002 RPM
disk transfer rate	2.2MB/s
SCSI bus transfer rate	5MB/s
controller overhead	1ms
track-to-track switch time	1.67ms
seeks \leq 12 cylinders	$1.28 + 1.15\sqrt{d}$ ms
seeks $>$ 12 cylinders	$4.84 + 0.193\sqrt{d} + 0.00494d$ ms

of this implementation was a functional testbed, to help ensure that we had made our design complete. (For example, we were able to test our fault recovery code by telling nodes to “die.”) The prototype also let us measure path lengths and obtain early performance data.

We implemented the design on an array of seven storage nodes, each comprised of a Parsytec MSC card with a T800 transputer, 4MB of RAM, and a local SCSI interface, connected to a local SCSI disk drive. The disks were spin-synchronized for these experiments. A stripe unit (the block-level interleave unit) was 4KB. Each node had a local HP97560 SCSI disk [Hewlett-Packard 1991] with the properties shown in Table VI.

The prototype was built in C to run on Helios [Perihelion 1991]: a small, lightweight operating system nucleus. We measured the one-way message latency for short messages between directly connected nodes to be $110\mu\text{s}$, and the peak internode bandwidth to be 1.6MB/s. Performance was limited by the relatively slow processors, and because the design of the Parsytec cards means that they cannot overlap computation and data transfer across their SCSI bus. Nevertheless, the prototype provided useful comparative performance data for design choices, and served as the calibration point of our simulator.

Our prototype comprised a total of 13.3k lines of code, including comments and test routines. About 12k lines of this was directly associated with the RAID functionality.

4.2 The Simulator

We also built a detailed event-driven *simulator* using the AT & T C++ tasking library [AT & T 1989]. This enabled us to explore the effects of

changing link and processor speeds, and to experiment with larger configurations than our prototype. Our model encompassed the following components:

- Workloads*: both *fixed* (all requests of a single type) and *imitative* (patterns that simulate existing workload patterns); we used a closed queueing model, and the method of independent replications [Pawlikowski 1990] to obtain steady-state measurements.
- Host*: a collection of workloads sharing an access port to the TickerTAIP array; disk driver path lengths were estimated from measurements made on our local HP-UX systems.
- TickerTAIP nodes (workers and originators)*: code path lengths were derived from measurements of the algorithms running on the working prototype and our HP-UX workstations (we assumed that the Parsytec MSC limitations would not occur in a real design).
- Disk*: we modeled the HP97560 disks as used on the prototype implementation, using data taken from measurements of the real hardware. The disk model was fairly detailed, and included:
 - the seek time profile from Table VI;
 - longer settling times for writes than reads (the disk can afford to be optimistic about head positioning for reads, but not for writes);
 - track- and cylinder-skews, including track- and cylinder-switch times incurred during a data transfer;
 - rotation position;
 - SCSI bus and controller overheads, including overlapped data transfers from the mechanism into a disk track buffer and transmissions across the SCSI bus (the granularity used was 4KB).
- Links*: represent communication channels such as the small-area network and the SCSI buses. We report here data from a complete point-to-point interconnect design with a DMA engine per link, since this is both the simplest topology and the one from which it is easiest to extrapolate to the effects of other designs. Our preliminary studies suggest that similar results would be obtained from mesh-based switching fabrics. We did not assume multicast capabilities.

Under the same design choice and performance parameters, our simulation results agreed with the prototype (real) implementation within 3% most of the time, and always within 6%. This gave us confidence in the predictive abilities of the simulator.

The system we evaluate here is a RAID5 disk array with left-symmetric parity [Lee 1990] (the same data layout shown in Table II and Figure 7), stripes composed from a 4KB block on each disk, spin-synchronized disks, FIFO disk scheduling (except where noted), and without any data replication, spare blocks, floating parity, or indirect-writes for data or parity [English and Stepanov 1992; Menon and Kasson 1992]. The configuration simulated had 4 hosts and 11 worker nodes, with each worker node having a single HP97560 disk attached to it via a 5MB/s SCSI bus. Four of the nodes were both

Table VII Read performance for fixed-size workloads, with varying link speeds
(all relative standard deviations were less than 2%)

<i>Request size</i>	<i>throughput MB/s</i>	<i>latency (in ms)</i>		
		<i>1MB/s</i>	<i>10MB/s</i>	<i>100MB/s</i>
4KB	0.94	33	31	30
40KB	1.79	38	34	33
1MB	15.2	178	86	76
10MB	21.1	1520	610	520

originators and workers; for simplicity, and since we were most concerned about exploring the effects of the internal design choices, we used only a single infinite-speed connection between each host and the array.

Except for the results in Section 4.7, the throughput numbers reported here were obtained when the system was driven to saturation; response times with only one request in the system at a time. For the throughput measurements we timed 10,000 requests in each run; for latency we timed 2000 requests. Each data point on a graph represents the average of two independent runs, with all relative standard deviations less than 1.5%, and most less than 0.5%. Each value in a table is the average of five such runs; the relative standard deviations are reported with each table.

In section 4.7, our throughput and response time numbers are means of 5 simulations, each consisting of 10,000 requests. Nearly all relative standard deviations for the data points in Section 4.7 are less than 1.0%, although a few (on the OLTP workload) were as high as 6.0%.

In all cases, 100 requests were run to completion through the simulator before any measurements were taken, to minimize startup effects.

4.3 Read Performance

Table VII shows the performance of our simulated 11-disk array for random read requests across a range of link speeds. The data show no significant difference in throughput for any link speed above 1MB/s, but 10MB/s or more are needed to minimize request latencies for the larger transfers.

4.4 Write Performance: An Exploration of the Design Alternatives

We first consider the effect of the large-stripe policy. Figure 8 shows the result: the difference is small, but enabling the large-write policy resulted always in a slight improvement in throughput at the expense of a slight increase in latency. We chose to enable the large-stripe mode for the remainder of the experiments.

Next, we compared the at-originator and at-parity policies for parity calculation. Figure 9 gives the results: at-parity is significantly better than at-originator, with the differences largest (as expected) at larger write sizes and with lower processor speeds. This is due to the at-parity algorithm spreading

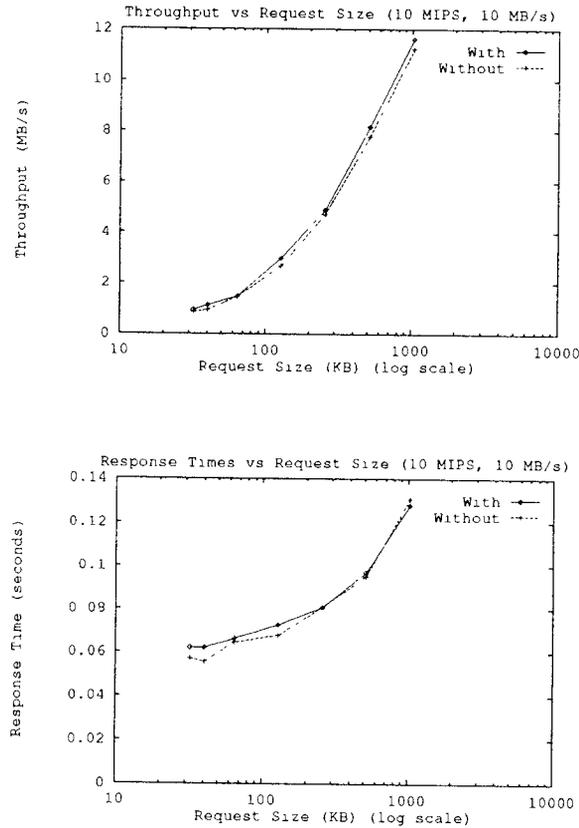


Fig. 8. Effect of enabling the large-stripe parity computation policy for writes larger than one half the stripe.

the parity calculation across several processors more evenly, so we used it for the remainder of our experiments.

The effect of the late-commit protocol on performance is shown in Figure 10: the effect of the commit protocol on throughput is small ($< 2\%$), but the effect on response time is more marked, with the late commit increasing response time by up to 20%. This is because the commit point is acting as a synchronization barrier, which prevents some of the usual overlaps between disk accesses and other operations. For example, a disk that is only doing writes for a request will not start seeking until the commit message is given. The delay that results could presumably be reduced by sending the disk a seek command to position its head while the parity computation was occurring, although we have not performed this experiment because the effect will only show up on an otherwise-idle disk array.

Although not shown, the performance of early commit is slightly better than that of late commit, but not as good as no commit. As a result, we recommend late commit as the preferred design choice: its throughput is

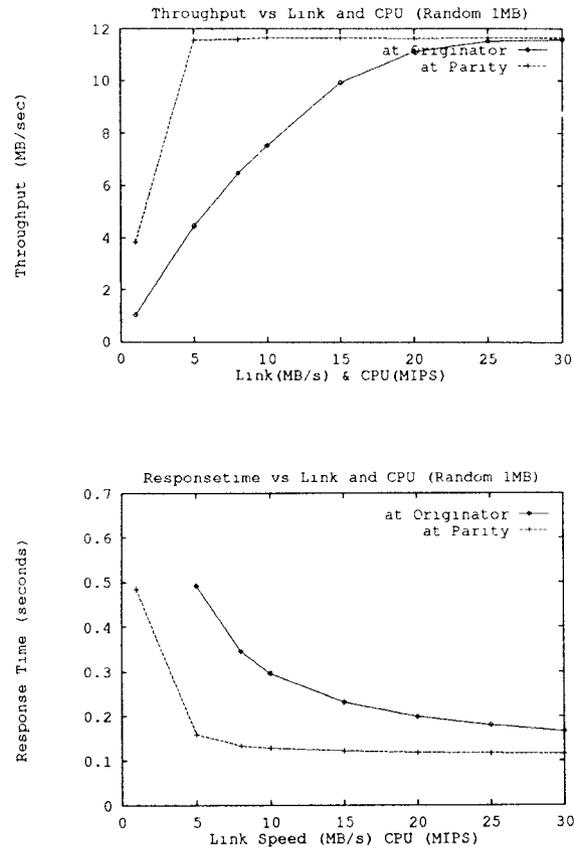


Fig. 9. Effect of parity calculation policy on throughput and response times for 1MB random writes.

almost as good as no commit protocol at all, and it is much easier to implement than early commit.

4.5 Comparison with a Centralized RAID Array

How would a TickerTAIP array compare with a traditional centralized RAID array? This section answers that question. We simulated both the same 11-node TickerTAIP system as before and an 11-disk centralized RAID. The simulation components and algorithms used in the two cases were the same: our goal was to provide a direct comparison of the two architectures, uncontaminated by other factors. The centralized array was modeled as a single, dedicated originator node, together with a set of worker nodes that did read and write operations only when directed to do so.

For amusement, we also provide data for a 10-disk nonfault-tolerant striping array implemented using the TickerTAIP architecture.

The results for 10MIPS processors with 10MB/s links are shown in Figure 11: clearly a nondisk bottleneck is limiting the throughput of the centralized

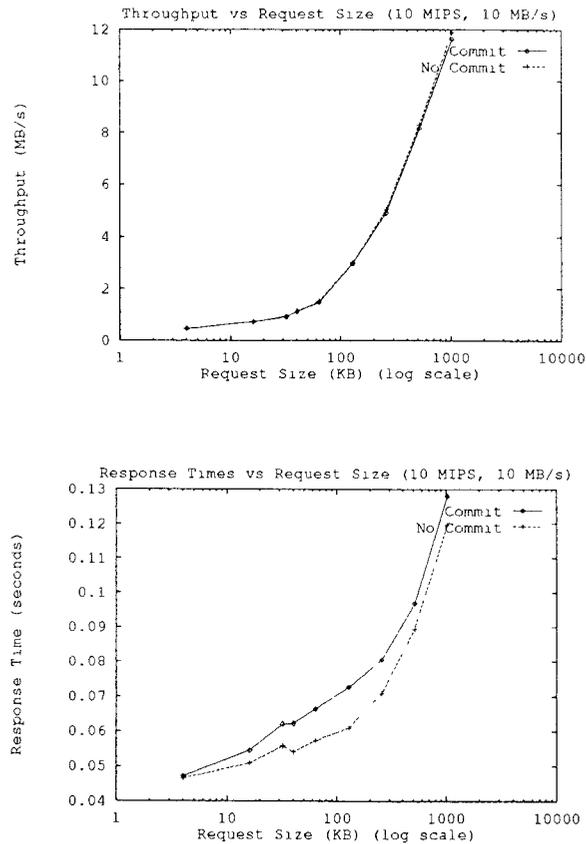


Fig 10. Effect of the late-commit protocol on write throughput and response time

system for request sizes larger than 256KB, and its response time for requests larger than 32KB. The obvious candidate is a CPU bottleneck from parity calculations, and this is indeed what we found. To show this, we plot performance as a function of CPU and link speed (Figure 12), and both varying together (Figure 13)—these graphs show that changing the CPU speed has a marked effect on the performance of the centralized case for 1MB writes, but a much smaller effect on TickerTAIP.

These graphs show that the TickerTAIP architecture is successfully exploiting load balancing to achieve similar (or better) throughput and response times with less powerful processors than the centralized architecture. For 1MB write requests, TickerTAIP's 5MIPS processors and 2–5MB/s' links give comparable throughput to 25MIPS and 5MB/s for the centralized array. The centralized array needs a 50MIPS processor to get similar response times as TickerTAIP.

Finally, we looked at the effect of scaling the number of workers in the array, with both constant request size (400KB) and a varying one with a fixed amount of data per disk (ten full stripes, however large a stripe becomes). In

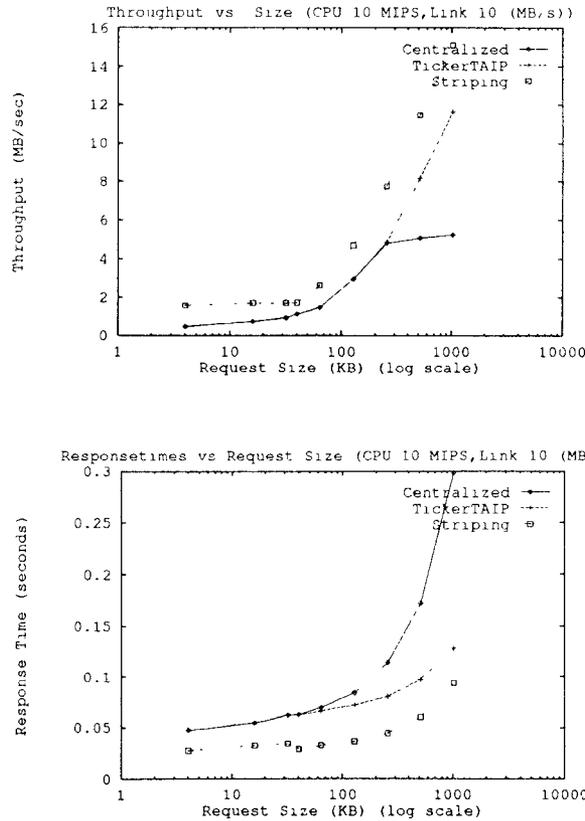


Fig. 11. Write throughput and response time for three different array architectures

these experiments, four of the worker nodes were also originators. The results are seen in Figure 14. With constant request size, the performance grows only slightly with larger number of disks. This is exactly as expected: as the number of disks increases, the fixed-size 400KB request touches a smaller fraction of the stripe size, so the disks get to do less useful work. On the other hand, the performance improvement shown as the request size is scaled up with the number of disks shows almost perfect linearity. (In practice, at some point the host links would become a bottleneck.) We believe these data are a strong vindication of our scalability claims for TickerTAIP.

4.6 Synthetic Workloads

The results reported so far have been from fixed, constant-sized workloads. To test our hypothesis that TickerTAIP performance would scale as well over some other workloads, we tested a number of additional workload mixtures, designed to model “real-world” applications:

—*OLTP*: based on the TPC-A database benchmark [Dietrich et al. 1992];

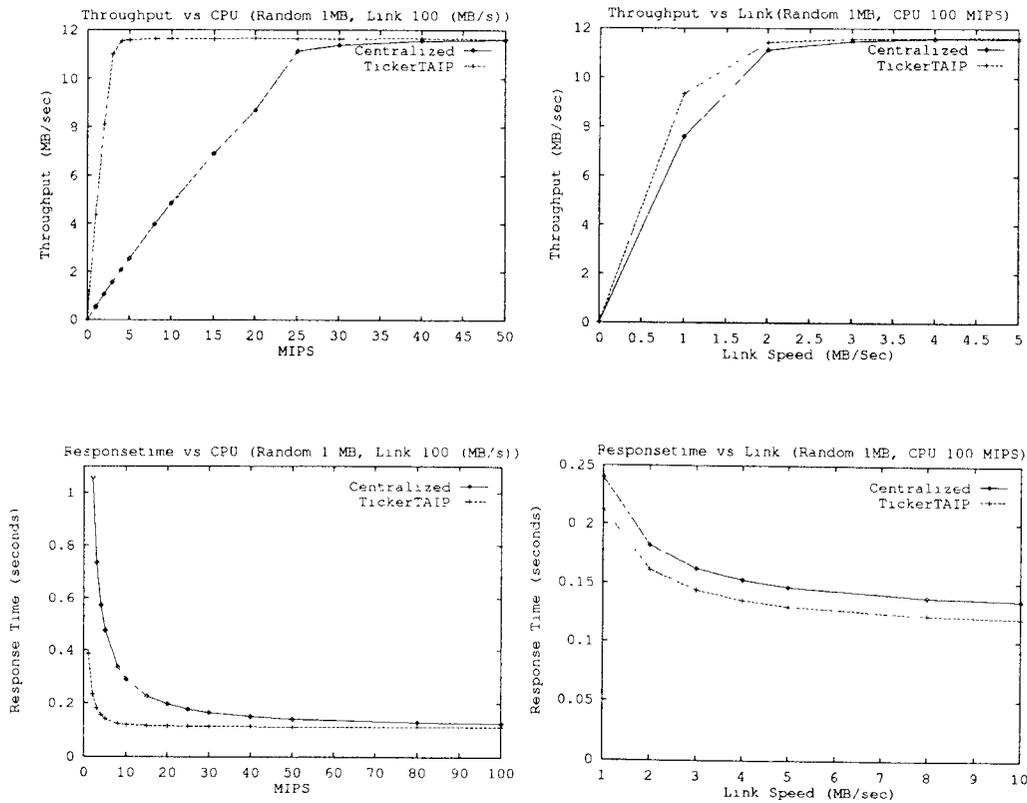


Fig. 12. Throughput and response time as a function of both CPU and link speed. 1MB random writes.

- timeshare*: based on measurements of a local Unix timesharing system [Ruemmler and Wilkes 1993];
- scientific*: based on measurements taken from supercomputer applications running on a Cray [Miller and Katz 1991]; “large” has a mean request size of about 0.3MB; “small” has a mean around 30KB.

Table VIII gives the throughputs of the disk arrays under these workloads for a range of processor and link speeds. As expected, TickerTAIP outperforms the centralized architecture at lower CPU speeds, although both are eventually able to drive the disks to saturation—mostly because the request sizes are quite small. TickerTAIP’s availability is still higher, of course.

4.7 The Effect of Scheduling Individual Disk Accesses

Our previous results used the simplest possible request-scheduling algorithm, FCFS, at the disk device drivers in the worker nodes. In this section we explore the effects of changing this scheduling algorithm. Clearly, this will have little effect when the queue sizes seen at the disk are small, but our early experiments led us to believe that they can sometimes get quite large

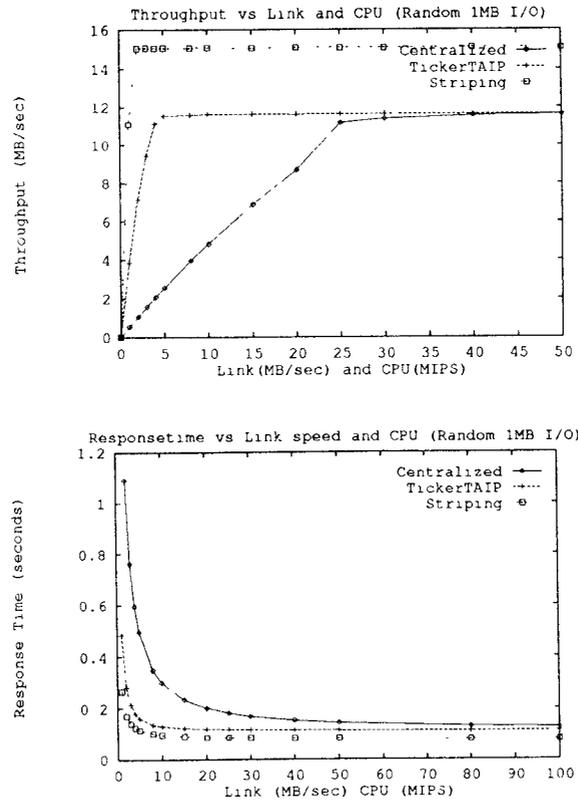


Fig. 13. Throughput and response time as a function of both CPU and link speeds. 1MB random writes.

(especially when operating near saturation)—at which point, a better scheduling algorithm is quite likely to produce a marked improvement in performance. This is indeed what we found.

To demonstrate this, Figure 15 shows the results of applying SSTF, SATF, and BNN scheduling algorithms on workloads comprised of fixed-sized 40KB writes and 1MB writes, as well as the OLTP synthetic workload. The graphs show that scheduling can nearly double the throughput under OLTP workloads and random 40KB writes. The smaller improvement shown for 1MB writes results because the individual I/Os are larger, so the effect of improving the gaps between them (which is the effect of the better scheduling algorithms) is less noticeable. Similar effects are seen on the response time graphs.

Our initial results suggest that both SATF and BNN are good candidates for scheduling algorithms. Currently, we prefer BNN because of its inherent starvation-resistant properties.

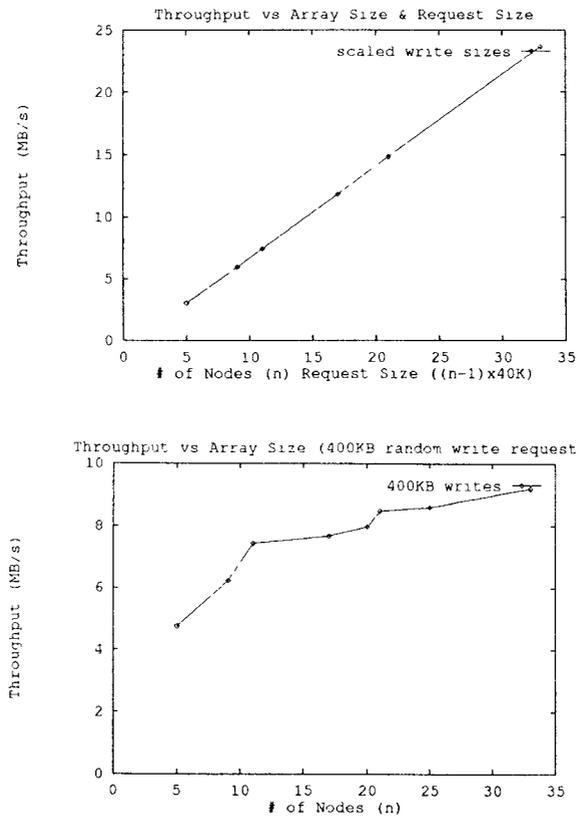
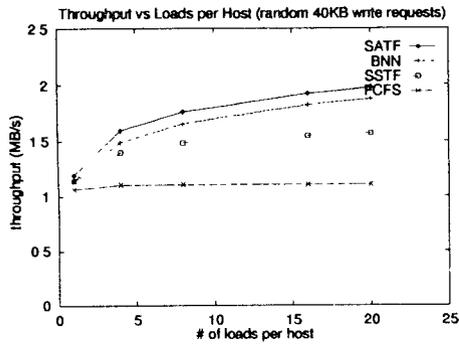


Fig. 14. Effect of TickerTAIP array size on performance.

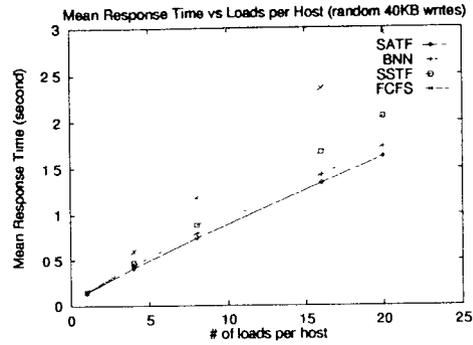
Table VIII. Throughputs, in MB/s, of the three array architectures under different workloads

Workload	Speeds		Throughput, in MB/s		
	link MB/s	cpu MIPS	Central RAID	TickerTAIP	Striping
OLTP	10	10	0.59 (1.7%)	0.59 (1.4%)	1.63 (1.0%)
time-share	1	1	0.43 (0.9%)	0.76 (0.8%)	1.69 (1.3%)
	10	10	0.76 (2.5%)	0.76 (1.7%)	1.69 (1.4%)
small scientific	1	1	0.71 (4.2%)	1.20 (1.2%)	1.73 (0.4%)
	10	10	1.20 (2.1%)	1.20 (1.9%)	1.73 (0.2%)
large scientific	10	10	8.23 (4.8%)	8.39 (3.3%)	9.81 (2.1%)

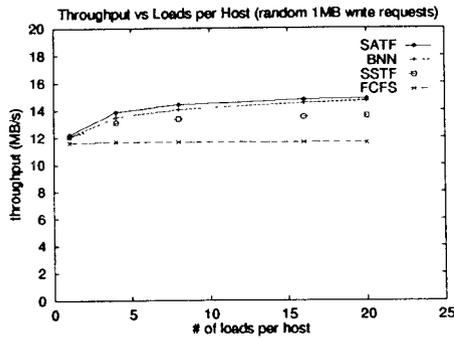
The shading highlights comparable total-MIPS configurations (Relative standard deviations are shown in parentheses)



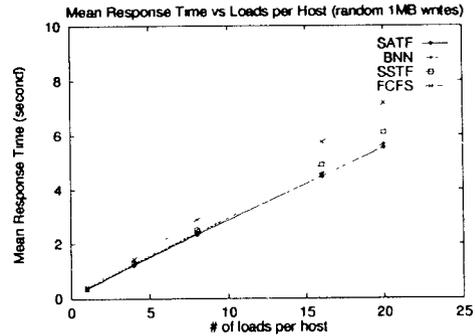
(a) Fixed-size 40KB writes.



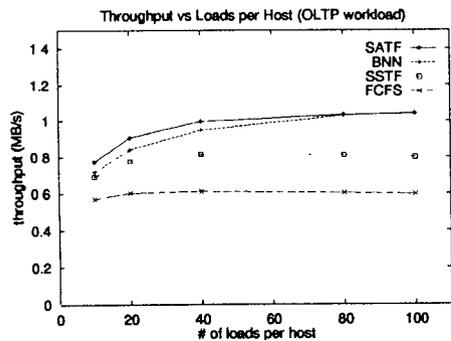
(b) Fixed-size 40KB writes.



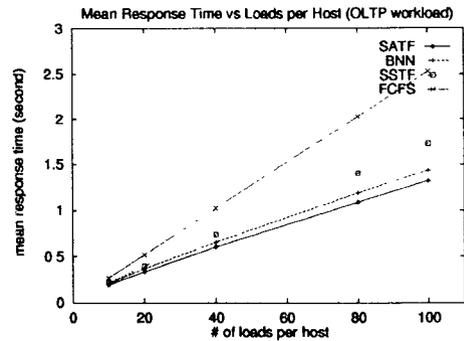
(c) Fixed-size 1MB writes.



(d) Fixed-size 1MB writes.



(e) Synthetic OLTP workload.



(f) Synthetic OLTP workload.

Fig. 15. Effect of different disk-level request-scheduling algorithms on TickerTAIP performance. The left-hand graphs display throughput as a function of load and scheduling policy, the right-hand ones the response time.

5 CONCLUSIONS

TickerTAIP is a new parallel architecture for RAID arrays. Our experience is that it is eminently practical (for example, our prototype implementation took only 12k lines of commented code). The TickerTAIP architecture exploits its physical redundancy to tolerate any single point of failure, including single

failures in its distributed controller; it is scalable across a range of system sizes with smooth incremental growth; and its worker/originator node model provides considerable configuration flexibility. Further, we showed how to provide—and prototyped—partial-write ordering to support clean semantics in the face of multiple outstanding requests and multiple faults, such as power failures. We have also demonstrated that—at least in this application—eleven 5MIPS processors are just as good as a single 50MIPS one, and provided quantitative data on how central and parallel RAID array implementations compare. Finally, we have demonstrated the performance improvements available from more sophisticated request-scheduling algorithms.

Most of the performance differences between the TickerTAIP and centralized designs result from the cost of doing parity calculations, and this turns out to be the main thing that changes with the processor speed: most of the other CPU-intensive work is hidden by disk delays. One suggestion that has been made is to improve the lackluster performance of these XOR calculations in the centralized case by constructing dedicated XOR engines. Unfortunately, it seems that the resulting systems can be unwieldy, in part because of the high speeds they have to operate at. Because the cost of a processor system increases faster than linearly with its performance—much of the cost is in the memory system rather than the processor itself—tackling the XOR-speed problem in this way is unproductive. It is our contention that off-the-shelf microprocessors are, in fact, cost-effective XOR engines, and they bring with them all their advantages of economies of scale in manufacturing cost, design time, and reliability. Thus it is better, we believe, to use an approach like TickerTAIP to divide up and parallelize the work to the point where such microprocessors can be used.

With small request sizes, it is easy for either architecture to saturate the disks. With larger requests, the difference becomes more marked as parity calculations become more significant. The difference is also likely to increase as multiple disks are added to each worker, and as the cost of performing smarter disk-scheduling algorithms is included. Both improvements are obvious upgrade paths for TickerTAIP (indeed, we have demonstrated one of them here); both will make the TickerTAIP architecture even more attractive than the centralized model.

We recommend the TickerTAIP parallel RAID architecture to future disk array implementers. Additionally the TickerTAIP architecture is well suited for use in multicomputers with locally attached disks. In this case, it can provide multinode RAID resilience without any dedicated or specialized hardware beyond that already provided for the multicomputer itself.

ACKNOWLEDGMENTS

The TickerTAIP work was done as part of the DataMesh research project at Hewlett-Packard Laboratories [Wilkes 1992]. The prototype implementation is based loosely on a centralized version written by Janet Wiener, and uses the SCSI disk driver developed by Chia Chao. David Jacobson improved the

AT & T tasking library to use a double for its time value. Federico Malucelli provided significant input into our understanding of the sequencing and request-scheduling options. Chris Ruemmler helped us improve our disk models.

We also thank the IEEE for allowing us permission to publish this revision, and the ACM anonymous reviewers for helping us to improve it.

Finally, whence the name? Because tickerTAIP is used in all the best *pa(rallel)RAIDs!*

REFERENCES

- AT & T. 1989. In *Unix System V AT & T C++ language system release 2.0 selected readings*. Select Code 307-144. AT & T, Indianapolis, In.
- BARTLETT, J., BARTLETT, W., CARR, R., GARCIA, D., GRAY, J., HORST, R., JARDINE, R., LENOSKI, D., AND MCGUIRE, D. 1990. Fault tolerance in Tandem computer systems Tech. Rep. 90.5, Tandem Computers, Cupertino, Calif.
- BORAL, H. 1988. Parallelism and data management. Tech. Rep. ACA-ST-156-88, Microelectronics and Computer Technology Corporation, Austin, Tex.
- CHEN, P. M., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. 1990. An evaluation of redundant arrays of disks using an Amdahl 5890. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 74-85.
- CLARK, B. E., LAWLOR, F. D., SCHMIDT-STUMPF, W. E., STEWART, T. J., AND TIMMS, G. D. JR. 1988. Parity spreading to enhance storage access. U.S. Patent 4,761,785; filed 12 June 1986; granted 2 August 1988.
- COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. 1988. Data placement in Bubba. In *Proceedings of 1988 SIGMOD International Conference on Management of Data*. ACM, New York, 99-108.
- DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. 1986. GAMMA—a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB Endowment, 228-237.
- DEWITT, D. J., GHANDEHARIZADEH, S., AND SCHNEIDER, D. 1988. A performance analysis of the Gamma database machine. In *Proceedings of 1988 SIGMOD International Conference on Management of Data*. ACM, New York, 350-360.
- DIETRICH, S. W., BROWN, M., CORTES-RELLO, E., AND WUNDERLIN, S. 1992. A practitioner's introduction to database performance benchmarks and measurements. *Comput. J.* 35, (Aug.) 322-331.
- DRAPEAU, A. L., SHIRRIFF, K. W., HARTMAN, J. H., MILLER E. L., SESHAN, S., KATZ, R. H., LUTZ, K., PATTERSON, D. A., LEE, E. K., CHEN, P. M., AND GIBSON, G. A. 1994. RAID-II: A high-bandwidth network file server. In *Proceedings of 21st International Symposium on Computer Architecture*. IEEE, New York, 234-244.
- DUNPHY, R. H. JR., WALSH, R., AND BOWERS, J. H. 1990. Disk drive memory. U.S. patent 4, 914, 656; filed 28 June 1988, granted 3 April 1990.
- ENGLISH, R. M. AND STEPANOV, A. A. 1992. Loge: A self-organizing storage device. In *Proceedings of USENIX Winter'92 Technical Conference*. USENIX Assoc., Berkeley Calif., 237-251.
- GIBSON, G. A., HELLERSTEIN, L., KARP, R. M., KATZ, R. H., AND PATTERSON, D. A. 1989. Failure correction techniques for large disk arrays. In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. *Oper. Syst. Rev.* 23, Apr., 123-132.
- GRAY, J. 1988. A comparison of the Byzantine agreement problem and the transaction commit problem. Tech. Rep 88.6 Tandem Computers, Cupertino, Calif.
- GRAY, J. N. 1978. Notes on data base operating systems. In *Operating Systems: An Advanced Course*. Lecture Notes in Computer Science, vol 60. Springer-Verlag, Berlin, 393-481.
- GRAY, J., HORST, B., AND WALKER, M. 1990. Parity striping of disc arrays: Low-cost reliable storage with acceptable throughput In *Proceedings of 16 International Conference on Very Large Data Bases*. VLDB Endowment, 148-159

- HEWLETT-PACKARD. 1991. *HP 97556, 97558, and 97560 5.25-inch SCSI Disk Drives: Technical Manual*. Part No. 5960-0115. Hewlett-Packard Company, Boise, Idaho
- HEWLETT-PACKARD. 1988b. *HP 7936 and HP 7937 Disc Drives Operating and Installation Manual*. Part. No 07937-90902. Hewlett-Packard Company, Boise, Idaho.
- HOLLAND, M., AND GIBSON, G. A. 1992. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems Comput. Arch. News*, 20, 23-35.
- JACOBSON, D. M. AND WILKES, J. 1991. Disk scheduling algorithms based on rotational position. Tech. Rep. HPL-CSP-91-7, Hewlett-Packard Laboratories, Palo Alto, Calif.
- LAMPSON, B. W. AND STURGIS, H. E. 1981. Atomic transactions. In *Distributed Systems—Architecture and Implementation: An Advanced Course*. Lecture Notes in Computer Science, vol. 105 Springer-Verlag, New York, 246-265
- LAWLOR, F. D. 1981. Efficient mass storage parity recovery mechanism. In *IBM Tech. Disclos. Bull* 24, 2 (July), 986-987.
- LEE, E. K. 1990. Software and performance issues in the implementation of a RAID prototype. UCB/CSD 90/573. Computer Science Div., Dept of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif
- MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. 1984. A fast file system for UNIX. In *ACM Trans. Comput. Syst.* 2, 3 (Aug.), 181-197.
- MENON, J. AND COURTNEY, J. 1993. The architecture of a fault-tolerant cached RAID controller. In *Proceedings of 20th International Symposium on Computer Architecture*. IEEE, New York, 76-86.
- MENON, J. AND KASSON, J. 1992. Methods for improved update performance of disk arrays. In *Proceedings of 25th International Conference on System Sciences*. Vol. 1. IEEE, New York, 74-83.
- MILLER, E. L. AND KATZ, R. H. 1991. Analyzing the I/O behavior of supercomputer applications. In *Digest of Papers, 11th IEEE Symposium on Mass Storage Systems*. IEEE, New York, 51-59
- MUNTZ, R. R. AND LUI, J. C. S. 1990. Performance analysis of disk arrays under failure. In *Proceedings of 16th International Conference on Very Large Data Bases*. VLDB Endowment, 162-173.
- NECHES, P. M. 1984. Hardware support for advanced data management systems. In *IEEE Comput* 17, 11 (Nov.), 29-40.
- OUSTERHOUT, J. K. 1990. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer'90 Technical Conference*. USENIX Assoc., Berkeley, Calif., 247-256
- PARK, A. AND BALASUBRAMANIAN, K. 1986. Providing fault tolerance in parallel secondary storage systems. Tech. Rep. CS-TR-057-86. Dept. of Computer Science, Princeton Univ., Princeton, NJ
- PATTERSON, D. A., CHEN, P., GIBSON, G. AND KATZ, R. H. 1989. Introduction to redundant arrays of inexpensive disks (RAID). In *Spring COMPCON'89*. IEEE, New York, 112-117
- PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of 1988 SIGMOD International Conference on Management of Data*. ACM, New York.
- PAWLIKOWSKI, K. 1990. Steady-state simulation of queueing processes: A survey of problems and solutions. In *ACM Comput. Surv.* 22, 2 (June), 123-170
- PCI. 1994. In *PCI Specification*. Intel Corporation, Hillsboro, Or.
- PERIHELION. 1991. *The Helios Parallel Operating System*. Prentice-Hall International, London.
- RUEMMLER, C. AND WILKES, J. 1993. UNIX disk access patterns. In *Proceedings of Winter 1993 USENIX*. USENIX Assoc., Berkeley, Calif., 405-420
- SCHULZE, M. E. 1988. Considerations in the design of a RAID prototype. Tech. Rep. UCB CSD 88-448, Computer Science Div., Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif.
- SCHULZE, M., GIBSON, G., KATZ, R., AND PATTERSON, D. 1989. How reliable is a RAID? In *Spring COMPCON'89*. IEEE, New York, 118-123.

- SCSI. 1991. Secretariat, Computer and Business Equipment Manufacturers Association. Draft proposed American National Standard for information systems—Small Computer System Interface-2 (SCSI-2), Draft ANSI standard X3T9.2/86-109, 2 February 1991 (revision 10d).
- SELTZER, M., CHEN, P., AND OUSTERHOUT, J. 1990. Disk scheduling revisited. In *Proceedings of Winter 1990 USENIX Conference*. USENIX Assoc., Berkeley, Calif., 313–323.
- SHIN, K. G. 1991. HARTS: A distributed real-time architecture. In *IEEE Comput.* 24, 5 (May), 25–35.
- SIEWIOREK, D. P. AND SWARZ, R. S. 1992. *Reliable Computer Systems: Design and Evaluation*. 2nd ed. Digital Press, Bedford, Mass.
- SLOAN, R. D. 1992. A practical implementation of the database machine—Teradata DBC/1012. In *Proceedings of 25th International Conference on System Sciences*. Vol. 1. IEEE, New York, 320–327.
- STONEBRAKER, M. 1989. Distributed RAID—a new multiple copy algorithm. Tech. Rep. UCB/ERL M89/56, Electronics Research Lab., Univ. of California, Berkeley, Calif.
- WILKES, J. 1992. DataMesh research project, phase 1. In *USENIX Workshop on File Systems*. USENIX Assoc., Berkeley, Calif., 63–69.
- WILKES, J. 1991. The DataMesh research project. In *Transputing'91*. Vol. 2. IOS Press, Amsterdam, 547–553.

Received October 1993; revised May 1994; accepted June 1994