Implementation of RAID

* Harat B. Bhangava
  John Riedl
  Department of Computer Sciences
  Purdue University
  West Lafayette, IN 47907

Abstract

RAID is a robust and adaptable distributed system for transaction processing. RAID is a message-passing system, with server processes on each site. A high-level, layered communications package provides a clean, location independent interface between servers. RAID processes concurrent updates and retrievals on multiple sites. The servers manage concurrent processing, consistent replicated copies during site failures or network partitionings, and atomic distributed commitment. The latest version of the communications package is able to deliver messages in a high-performance configuration in which several servers are linked into a single process. RAID provides the infrastructure to experimentally investigate various methods for supporting reliable distributed transaction processing. Experiments on handling site failure with partial replication, checkpointing, and alternative communications methods have been performed. Measurements on various aspects of RAID transaction processing performance are presented.

Implementation Objectives

RAID has been implemented to conduct experiments to provide empirical evaluation of the algorithms for replicated copy control, quorum based methods, dynamic reconfiguration techniques, communications software, and transaction support in operating systems. These experiments identify principles that are necessary for reliable, high-performance transaction processing. In this paper we discuss the implementation of the RAID system, including the site structure, the communications package, the flow of transaction processing, some measurements that we have performed, and briefly our ongoing experimental work. RAID has evolved over the last five years. During this time period the RAID design has gone through many changes, and many of the servers have been implemented more than once. The server design has facilitated the implementation effort by providing for flexibility, and by explicitly defining the interfaces between servers. Servers can be modified or replaced without affecting other servers. Unfortunately, such a design performance problems, both because of the high cost of communicating between processes and because of the increased operating system overhead from context switching between multiple processes. Our current design provides a mechanism through which multiple servers can reside in the same Unix process. Our approach is to modify the communications system to support delivery of messages to servers within the same process via a procedure call. More details are in sections 2 and 3.2. In such a design, servers are tested as separate processes communicating via messages, but experiments are performed on a high-performance version of the system in which most of the communication becomes procedure calls. Performance measurements are being done to compare the relative merits of various algorithms, and to demonstrate the feasibility of new implementation techniques. Performance data is included to give examples of the types of instrumentation available in RAID. The current version of RAID supports distributed transactions on a replicated database. An implementation of partial replication is in the testing phase. A replicated copy control protocol maintains consistency and availability despite site and network failures. The distributed concurrency controller can select one of four different concurrency control algorithms. The RAID system is implemented in 20K lines of C code, and can run on either Vaxen or Suns under 4.3 BSD Unix. We also have Mini-RAID, an implementation of RAID for minicomputers.
abstraction of RAID, that is an additional 6K lines of C code, and runs in the same environment. Mini-RAID has been used to prototype site failure and partial replication algorithms before their incorporation into the RAID system. Mini-RAID is being used for studying load balancing and deadlock detection at two other institutions.

1.1 Distinguishing Features

- RAID is designed as a modular, message-passing system to support easy extensions and modifications. Servers can be relocated, and new implementations of servers can be substituted. The RAID communication sub-system provides location transparent addressing, and supports multiple virtual sites on a single physical host. Servers can each be separate processes for testing, or multiple servers can be linked into a single process for actual experiments. The name-server provides a notifier service that automatically informs interested servers of failures or recoveries of other servers.

- RAID supports the implementation of distributed control algorithms so that they can be replaced with other algorithms. Such adaptability permits RAID to respond to environmental or administrative demands. For instance, if at certain times it is crucial that long transactions be able to commit, RAID can switch to methods that comply. Certain servers can adapt between algorithms while transactions are executing [6]. For instance, the concurrency controller can convert from a timestamp based approach to locking if too many transactions are being aborted, or the network partition manager can convert from an optimistic approach to a primary copy based approach if partitions are expected to be of long duration. The RAID approach is to formally model the distributed sub-systems as history sequencers, which are functions that take as input a series of actions (a history) and produce as output the same actions, possibly in a different order. Switching between two implementations of a sequencer requires transferring the state information from the running implementation to the new implementation. This can be done by developing a generic state common to the implementations, by developing methods for converting the state information between the two implementations, or by introducing an intermediate period of processing during which the new implementation absorbs enough state information to be able to correctly sequence the existing transactions. More details are given in [6].

- RAID provides replicated copy management that can handle multiple site and network failures. A Read One/Write All Available algorithm is used to ensure data availability to transactions while maintaining consistency. Out-of-date data on a recovering site are identified by maintaining fail-locks [1].

- RAIDTool is a window-based front-end to the RAID system. RAIDTool has a separate window for each site showing the status of the site. A control panel is available to create new sites, cause old sites to fail or recover, and monitor system performance. RAIDTool permits an operator to configure a RAID system on multiple workstations, test the system with randomly generated transactions, and communicate reconfiguration and adaptation decisions to the servers.

- We have designed PUSH, a system with which user programs can safely and simply specify algorithms to be run within the kernel.

1.2 Experiments in RAID

The RAID infrastructure includes a fully functioning system for processing transactions, measurement tools, a user-friendly interface for monitoring experiments, and support for modifying the system to test new ideas. There is a need for concrete empirical evidence of how operating system support for communications, shared memory, lightweight processes, etc., impacts transaction processing performance. Furthermore, we need experimental comparisons of different techniques for distributed transaction processing in areas such as concurrency control, network partition management, and distributed commitment. Finally, techniques for developing distributed systems that are adaptable to changing conditions – in both the short and the long term – must be studied and measured. RAID is a platform on which these experiments can be performed. The following experiments have already been performed on RAID:

1. We studied the performance of replicated copy control protocols during site failure and recovery [5]. We found that if the refreshing of out-dated copies using the expensive copyer transactions was delayed for a brief period after recovery, most of the data items were refreshed for free by ongoing transactions on other sites that write to the out-dated copy. After about 80% of the out-dated copies were refreshed this way, however, the remaining copies can be explicitly brought up-to-date to improve availability.

In a separate study [2] we examined the effects of partial replication on the results from the above study. We learned that the increased availability provided by a high degree of replication can be achieved more efficiently with a lower degree of replication along with a threshold policy. The threshold policy maintains
a minimum number of copies of each data item by creating new copies when necessary.

2. Our communications experiments [4] demonstrated that a substantial part of datagram latency can be attributed to getting into and out of the kernel. In [3] we investigate two methods for improving multicast performance. Simulating multicast in the kernel software cuts the cost of one round of messages by about 30%. Hardware multicast cuts the cost of one round by almost 50%. We plan to measure the improvement on transaction throughput provided by utilizing these lower-level multicast mechanism.

We describe experiments [3] with using special-purpose communications methods for local communications. The four methods we studied were one queue message passing, two queue message passing, named pipes, and shared memory with semaphores for synchronization. One queue message passing, the best of the four methods, is 75% faster than UDP for transferring 1000 byte messages between two processes on the same host. We are currently investigating the use of these local communications methods in RAID.

3. We investigated the performance of concurrent checkpointing techniques for recovering the state of a RAID server that fails due to a transient hardware or software error [2]. Servers with which the failed server communicated with since it took its last checkpoint may also need to be rolled back. Our experiments reveal that the overheads due to synchronization in concurrent checkpointing and rollbacks are of the order of a single checkpoint on the stable storage. The smart overlapping of checkpoint instances can terminate such processes with a small elapsed time.

1.3 Related Work

RAID is similar to CAMELOT [13], ARGUS [12], and $R^*$ [10] in its support for distributed transactions. CAMELOT and ARGUS encapsulate each data object in a single server process, with actions on the data performed by remote procedure calls. RAID and $R^*$ have a data server for each user. Having a single data server is a performance advantage for transactions that access many data items, and amortizes session connection and authentication over several transactions. On the other hand, the virtual site abstraction provided by a separate data server per data item is elegant, and provides better modularity. CLOUDS [9] also supports distributed transaction processing. CLOUDS provides a minimal kernel that provides fast communication and support for objects. The Encompass distributed data management system is a commercial product of Tandem Corporation [8]. Encompass provides continuous, fault-tolerant processing in a decentralized distributed environment. For reliability it uses a mechanism called process-pair, in which two cooperating processes run on separate processors physically connected to a device.

2 RAID Structure

We designed the modules that implement transaction processing as separate server processes to provide for reconfiguration. The servers communicate among themselves using the RAID communication package, which is layered on UDP/IP. The current design provides for two versions of the RAID system. The first version runs, with each server in an asynchronous process communicating via messages. The second version combines the servers that do not need to be asynchronous into a single process. This approach reduces the communication cost. Transmitting a message between two servers takes tens of milliseconds in the multiple server model, but only tens of microseconds in the single server model.

The second version is implemented as a change in the communication library. Compilation options specify which servers are to be loaded together, and at run-time the communications system chooses either datagram communication or a procedure call to the destination server routine. In the first version of RAID, each server has a main program that consists of a loop that receives messages and dispatches them to the appropriate handler using a procedure call. In the new version, a single main loop replaces each server's main loop. This loop receives messages and uses a table to dispatch them to the appropriate handler in the appropriate server. Since all of the servers were implemented in a multi-threaded manner the main loop is the only place in which messages are received. Note that this new design makes the servers that are linked together run synchronously. An alternative would be to implement lightweight processes such as are used in Camelot [13] or Argus [11]. Asynchronous lightweight processes are harder to implement, and both Camelot and Argus report that synchronizing multiple processes that share an address space can be difficult. On the other hand, multiple processes are able to make use of a shared memory multiprocessor architecture and permit asynchrony between the servers. Asynchrony is an advantage since without it a separate address space must be kept for each user to permit concurrency on a single site.
2.1 RAID Site Structure

Figure 1 depicts the organization of a RAID virtual site. The site is virtual since its servers can reside on one or more hosts (machines) and since the site is not tied to any particular host on the network. Multiple sites can run on a single host, so we can run a ten-site RAID instance on the five workstations currently available in the laboratory. Furthermore, two or more separate instances of RAID can run independently. Each site implements facilities for query parsing and execution as a transaction, access management with stable storage, concurrency control, replicated copy management, site failure and network partitioning management, naming, etc. The following describes the role of each of the RAID servers in the system.

The communications package design supports arbitrary grouping of RAID servers into physical processes. During testing, each of the servers is a separate process to isolate errors. When performance measurements are being taken, the following two servers are linked into a single process for each user:

- **User Interface (UI)** is a front-end invoked by a user to process relational calculus queries.
- **Action Driver (AD)** accepts a parsed query in the form of a tree of actions from its UI and executes the transaction, reading data from the local copy of the database. It formats the query as a transaction (read and write actions).

The remaining servers are linked together to form the transaction management process during performance experiments:

- **Access Manager (AM)** provides write access to the local database, and works with AC and AD to ensure that updates are posted atomically to stable storage.
- **Atomicity Controller (AC)** manages two commit phases of transaction processing to ensure that a transaction commits or aborts globally. When AC receives a transaction, it sets a commit-lock on each data item in the transaction's write-set. The commit-locks provide a localized critical section during the two-phase commit. They are short-lived, since they are only needed during commitment, rather than during transaction processing.
- **Concurrency Controller (CC)** checks whether a transaction history is locally serializable at a given site. RAID performs concurrency control using the timestamp validation approach. After a transaction finishes executing, its history information is passed to all sites in the system, each of which validates

A site is up (operational) if all of its servers are operational. Some servers may be operational on a down (failed) site, but other sites will consider the entire site failed when they observe that any server is failed.

---

Legend

AC = Atomicity Controller
CC = Concurrency Controller
AD = Action Driver
AM = Access Manager
RC = Replication Controller
whether the history is locally serializable with respect to previously positively validated transactions on that site. The CC has been implemented to perform the validation with one of many different implementations. Currently implementations of simple locking, read/write locking, timestamping, and conflict graph cycle detection are available. The type of concurrency control may be chosen at run-time, but all transactions must be completed before it can be switched to a different type.

2.2 RAID Communications

This section describes the high-level services provided by the RAID communications package, including the RAID name space, the oracle (name-server), and the available communications services.

The Name Space. We can uniquely identify each RAID server with the tuple (RAID instance number, RAID virtual site number, server type, server instance). The last tuple element is necessary because some server types can have multiple instances active simultaneously.

To send a message to a server, UDP needs a (machine name, port number) pair. The RAID oracle maps between RAID 4-tuples and UDP addresses. The communications software at each server automatically caches the address of servers with which it must communicate. Thus, the oracle is only used at system start-up, when a server is moved, and during failure and recovery.

The RAID Oracle. An oracle is a server process listening on a well-known port for requests from other servers. The two major functions it provides are registration and lookup. A server registers by telling the oracle its name (i.e., the tuple that uniquely identifies it). Any server can ask the oracle for the lookup service to determine the location of any other server.

To permit other servers to locate it, a server must perform the RegisterSelf() call. RegisterSelf takes a single argument, called a notifier set. The notifier set is a list of regular expressions describing the RAID addresses of servers with which the new server must communicate. Whenever a server changes status (e.g., moves, fails, or recovers) the oracle sends a notifier message to all other servers that have specified the changing server in their notifier set. Notifier messages are handled automatically by the communications software, which caches the address of the new server. In many cases the higher-level code is not even aware of the reconfiguration.

The performance of the RAID Oracle only affects the start up and reconfiguration delays of RAID. The RegisterSelf(), FindPartner(), and FindAll() functions require just a few packet round-trips. FindOracle() is an order of magnitude more expensive, since it must check for the oracle on all possible hosts on the network.

RAID Communications Facilities. The RAID servers communicate with each other using high-level operations such as SendAcC(). The high-level facilities are implemented on top of the low-level RAID transport protocol. We call this protocol LDG, for Long DataGram. This protocol has no restriction on packet sizes. LDG is currently built on top of UDP. Each LDG packet is fragmented if necessary, and then sent using UDP. At the destination, fragments are collected and reassembled. The fragment size is an important parameter to LDG. Normally we use fragments of 8000 bytes, which is the largest possible on our Suns. Since IP gateways usually fragment messages into 512 byte packets we also have a version of LDG with 512 byte fragments.

Table I compares UDP, LDG, and RAID round-trip communication times for datagrams of various lengths. LDG-512 is the version of LDG with 512 byte fragments, while LDG-8000 is the version with 8000 byte fragments. LDG is about three milliseconds more expensive than UDP for packets that do not need to be fragmented. LDG-512 becomes much more expensive for larger packets, since UDP and LDG-8000 are only transmitting a single packet. The difference between the two versions of LDG shows the advantage of kernel-level fragmenting over user-level fragmenting. The numbers given for the RAID layer are based on LDG-8000. The current implementation copies the buffer several times while building the header. A new implementation of the RAID layer is expected to perform almost as well as LDG, because of changes that completely avoid buffer copying. Also, the LDG layers now invoke UDP with a sendmsg() system call that copies from multiple source buffers directly into the kernel buffer from.

<table>
<thead>
<tr>
<th>Bytes:</th>
<th>64</th>
<th>512</th>
<th>2048</th>
<th>8192</th>
<th>32768</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP</td>
<td>7.2</td>
<td>10.6</td>
<td>16.5</td>
<td>48.8</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>LDG-512</td>
<td>10.2</td>
<td>17.2</td>
<td>41.9</td>
<td>147.4</td>
<td>550.0</td>
<td>8,630</td>
</tr>
<tr>
<td>LDG-8000</td>
<td>9.6</td>
<td>12.8</td>
<td>19.2</td>
<td>65.1</td>
<td>224.8</td>
<td>3,200</td>
</tr>
<tr>
<td>RAID</td>
<td>11.9</td>
<td>20.1</td>
<td>46.7</td>
<td>153.3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: RAID Communication Time by Packet Length (in milliseconds)
which the data is to be transmitted. The packet header is built separately from the data, and the sendmsg() call appends them. In the old implementation the data was copied multiple times to make room for new header fields. In the new implementation the data is in a separate buffer and does not need to be copied.

3 Transaction Processing

Transaction processing in RAID is separated into one execution phase and two commit phases. In the execution phase the transaction executes on the site to which it was submitted, using only the local copy of the database. During this phase no concurrency control is done, and no messages are exchanged. The transaction maintains timestamps for its reads, and writes to a copy of the data in volatile memory. During the first commit phase, the executing site communicates with other sites to determine global commitment. The entire read/write set of the transaction is distributed in a single round of messages. Phase 2 of commitment communicates the commit decision to all of the sites. Sections 3.1 and 3.2 describe RAID transaction processing in detail.

In order that the AC can manage multiple transactions simultaneously it is implemented in a multi-threaded manner. A multi-threaded server maintains a queue of the requests for which it is waiting for replies. Whenever the server receives a reply message, it locates the request in the queue, and updates the state information in the queue element. Whenever a server receives a request message, it immediately processes it and returns a reply.

3.1 Logical Transaction Execution Flow

Figure 2 depicts the logical relationships between the RAID servers during transaction processing. Transactions are processed in this manner during integration testing of RAID modules. Section 3.2 describes the actual flow of messages between processes. The numeric labels in the figure refer to the phases in the life of a transaction as follows:

0. UI gets a query from the user, parses it, and passes the parsed query on to AD.
1. AD assigns a globally unique transaction ID, and records the read-set and write-set as it processes the transaction using the local database. Updates are preserved in a log or a differential file. When using locking concurrency control reads from the AD must go through the AM.
2. AD forms a commit request and sends it to the local AC. This request contains the transaction ID, a list of identifiers of items read, along with the time at which the read occurred, and the list of identifiers of items written. No timestamps are available for the writes since they have not yet taken place.
3. AC sends transaction history to RC for read-set validation if AC considers this site to still be recovering (i.e., fail-locks are still set for copies on this site). RC checks for a fail-lock on each data item in transaction's read-set. Copier transactions are generated for any out-of-date items that are found in the read-set.
4. RC responds to AC with indication of read-set validity after completion of necessary copier transactions.
5. If read-set is valid (no fail-locks), the AC acquires special commit-locks for the items in the write-set. If some commit-locks are already set, it may choose to wait for them to be released, in which case it uses a method for avoiding or breaking deadlocks. AC then sends the transaction history to CC and remote ACs. If the read-set is invalid, the AC aborts this transaction.
6. CC and remote ACs reply to AC with a commit/abort decision for the transaction.
7. Once all votes are recorded from the local CC and the remote ACs, AC informs AD of the commit/abort decision.
8. AD sends the log or differential file to all AMs and tells them to commit the transaction, if the transaction was deemed globally serializable by AC.
9. AM writes all data of the committed transaction to the database.
10. AM informs AC that the transaction's data was successfully written.
11. AC releases the commit-locks belonging to the transaction, and informs the local CC and all other ACs. The remote ACs release their commit-locks and inform their CCs. The CCs move the transaction to their commit lists.
12. AC sends the transaction write-set to RC. RC clears fail-locks for items in the write-set. Fail-locks are set for any sites that are perceived to be down.
13. AM tells AD that write was successful. AD informs user that the transaction has committed.

3.2 Physical Transaction Execution Flow

Figure 3 shows the physical communication paths between the RAID servers in the current design. The numeric labels in the figure refer to the phases in the life of a transaction as follows:
Remote AMs

Long-lived servers are denoted by italic type

Remote ACs

Figure 2: Transaction processing on a RAID site.
0. AD records the read-set and write-set as it processes the transaction using the local database. When using locking concurrency control, reads from the AD must go through the AM.

1. AD forms a commit request and sends it to the RAID server.

2. The RAID server transmits the commit request to remote RAID servers and invokes the local concurrency controller.

3. The remote RAID servers reply.

4. The RAID server makes a commit/abort decision and returns it to the AD.

5. The AD acknowledges the commit/abort decision.

6. The RAID server updates the database.

7. The RAID server informs the remote RAID servers that the transaction is committed.

8. The AD tells the RAID server to update the database. This step is not necessary in the single process version of RAID, but is needed when running with the AM and AC separate.

### 3.3 Performance Measurements in RAID

This section describes measurements of the performance of the transaction processing protocols in the first version of RAID. The new version is expected to reduce these numbers by several hundred milliseconds. The servers involved in commitment are the Atomicity Controller (AC) which manages the commit algorithm, and the Concurrency Controller (CC) which determines whether transactions are serializable.

The following series of performance measurements were done on Sun 3/50s (approximately 1 MIPS machines) connected by a 10 megabit/second ethernet. The database for the experiment is 100 tuples from a truncated version of the thousand relation used in [7]. The relation has columns named unique, two, twenty, hundred, and thousand. The first column of the relation is a unique key for the tuple. The other columns are random numbers selected from the range specified by the column name. For instance, the range for column twenty is 0 to 19. These columns provide for a wide range of selectivity in queries. For example, the query `database thousand, get thousand : thousand.ten = 8;` can be used to select approximately 10% of the tuples.

RAID has built-in support for measurements of both elapsed time and CPU time for each of the phases of distributed transaction processing. The server that executes transactions measures wall clock time for each of the phases of transaction execution. A special control message causes the communication package to print the current resource utilization for the server, including the number of packets it has sent and received, and its user and system CPU time. These mechanisms were used to collect the following performance data.

#### 3.3.1 Elapsed Time for Transaction Processing

Table 2 shows the time taken by transaction processing for several different database queries on RAID systems with varying numbers of sites. The times do not include the cost of interpreting the database query or the cost of translating the query to a transaction.

Both select queries use a simple predicate that only examines one field in each tuple. The insert query inserts twenty tuples. The update query updates one field of a selected tuple. The processing time is higher

<table>
<thead>
<tr>
<th>transaction</th>
<th>1 site</th>
<th>2 sites</th>
<th>3 sites</th>
<th>4 sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>select one tuples</td>
<td>0.3</td>
<td>0.3</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>select eleven tuples</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>insert twenty tuples</td>
<td>0.6</td>
<td>0.6</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>update one tuple</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 2: Execution time for transactions in RAID systems with various numbers of sites (in seconds).

Figure 3: Physical transaction processing on a RAID site.
for the insert query because its write set is larger than any of the other read or write sets.

The fact that the processing time is fairly constant as the number of sites increase is due to the use of built-in multicast in the RAID layer of the communications package [3,4]. This lower level multicast only has to format the packet once regardless of the number of sites. Hence, the execution occurs in parallel on each site. Our estimate is that this time will remain constant up to around ten sites if we continue to use UDP as our transport mechanism. We are currently preparing a kernel-level multicast [3] that will help maintain this property for even larger numbers of sites.

3.3.2 Server CPU Time

We have measured the CPU time for the various servers. A significant fraction of the CPU time is spent processing messages. For instance, the CC spends 40 milliseconds of CPU time processing a simple transaction, about 10 milliseconds of which are spent in a single message round-trip. The total CPU time for all of the servers is a small fraction of the elapsed time for each transaction. This suggests that multiple queries executing at the same time would be able to overlap significantly. We are considering ways by which the experiments may be extended to multiple simultaneous transactions.

3.3.3 CPU Time for Atomicity Control

CPU times for most of the servers are constant as the number of sites increases, but the AC does some additional processing for each new site. Table 3 shows the CPU time taken by the AC for various numbers of sites. The times show a slight tendency to increase with the number of sites, but the variance in the measurements is too large to permit stronger statements. The variance is probably caused by the granularity of the hardware clock, since individual time slices are likely to be smaller than twenty milliseconds. For instance, sometimes the AC averages almost one context switch per millisecond. Since these numbers are collected directly by the kernel we do not have any means to further control them. In any case, it is again clear that most of the wall clock processing time for an individual transaction is not CPU time.

4 Brief Retrospect

RAID has been successful in promoting experimentation. Five separate experiments are underway [2,3,4,5]. Central to this capability is the server philosophy, which clearly defines the interfaces to the functional components of the system. New implementations need only match this interface to run correctly as RAID servers. Current performance problems are caused both by the additional operating system overhead of managing multiple processes, and by the expensive communication primitives available. To correct these problems we have modified the design to provide for a high-performance single-process version of RAID that can be compiled from the same source code as the test versions that use separate processes for each server. The principal change needed to produce the high-performance version was to change the communication system to check for compilation flags that indicated that certain servers are grouped in the same process. In the future we will report on experiments that measure the performance benefits obtained with this new approach.

We made an early decision that the communications system would be datagram based. This works well for small control messages, but for messages containing large amounts of data it uses excessive resources. For instance, the log or differential file that is transmitted to the Access Manager to be merged with the database must be placed in a single large buffer, fragmented by LDG, transmitted in pieces by UDP/IP, placed into a single large buffer, and finally passed to the AM. Our future implementation will transmit the log as a stream or as a series of data, and the AM will process and discard each data as it receives it. Incremental execution preserves resources.

The need for concurrent execution in several server types has been a problem. The AD must be able to process several user transactions concurrently. We provide for this by creating a different AD for each user. This approach has the disadvantages that it creates many new processes between which Unix must context-switch, that multiple copies of the code for executing transactions must be loaded into memory, and that database caches can only be kept on a per-user basis. The AC supervises distributed commitment for multiple transactions simultaneously. Currently it maintains the state for each transaction internally, looking up the corresponding transaction for each message it receives. Since this approach does not provide for any form of preemptive scheduling, it is essential that the AC process each message quickly, so that other incoming packets are not discarded. The AC implementation would be simplified if the operating system provided preemptive scheduling within the AC address space. In addition, the CC could be modified to process multiple transactions simultane-
<table>
<thead>
<tr>
<th>transaction</th>
<th>1 site</th>
<th>2 sites</th>
<th>3 sites</th>
<th>4 sites</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>user</td>
<td>sys</td>
<td>user</td>
<td>sys</td>
</tr>
<tr>
<td>select one tuples</td>
<td>0.04</td>
<td>0.14</td>
<td>0.06</td>
<td>0.14</td>
</tr>
<tr>
<td>select eleven tuples</td>
<td>0.04</td>
<td>0.08</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>insert twenty tuples</td>
<td>0.20</td>
<td>0.16</td>
<td>0.08</td>
<td>0.14</td>
</tr>
<tr>
<td>update one tuple</td>
<td>0.04</td>
<td>0.10</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Table 3: CPU time used by RAID Atomicity Controller (AC) in executing transactions on varying numbers of sites (in seconds).

5 Acknowledgments

The following students have contributed to the RAID project at the University of Pittsburgh and at Purdue University: T. Chen, J. Dilley, A. Helal, F. Lamaa, H. Lee, P. Leu, S. Leung, E. Mafla, T. Mueller, P. Ng, P. Noll, A. Royappa, D. Sabo, and B. Sauder.

We would like to thank Dr. Jim Gray of Tandem Corporation for suggestions that led to the new design in which multiple servers share a single operating system process.

References