Resource Conservation in a Mobile Transaction System

Qi Lu and Mahadev Satyanarayanan, *Senior Member, IEEE*

Abstract—This paper addresses the problem of providing transactional support for improved data consistency in mobile file access, while paying careful attention to the resource constraints of mobile clients. We present data on resource consumption from an implementation of the isolation-only transaction (IOT) mechanism of the Coda File System. The data shows that the resource conservation techniques used by the IOT mechanism do indeed result in modest demands on the three critical resources on a mobile client: CPU and I/O usage, disk space, and RVM space. Overall, our measurements confirm that even a severely resource-constrained mobile client can benefit from the improved consistency offered by the IOT mechanism.

Index Terms—Mobile computing, transaction, resource management, distributed file systems, disconnected operation, performance evaluation, Coda File System.

1 Introduction

Can mobile clients in a distributed file system preserve a high degree of data consistency in spite of the meager processing and storage resources available to them? If a mechanism to attain this goal is feasible, is it sufficiently frugal in resource consumption to enable a mobile client to operate fully disconnected for extended periods lasting many days? Two factors make a positive answer to these and related questions important.

First, mobile clients face wide variance in network quality, including frequent periods of disconnection. This makes it a challenge to provide users with satisfactory performance and availability while preserving an acceptable level of consistency for shared data. Second, mobility as a requirement generates relentless pressure on clients to be lighter, smaller, and consume less power. Though resource-rich mobile clients can be built, they will always be at a competitive disadvantage with respect to smaller and lighter rivals. Hence, any viable solution to the problem of mobile data consistency must strive to keep its resource costs low.

In this paper, we report on experimental data from the *isolation-only transaction (IOT)* mechanism [9], [10] of the Coda File System. This mechanism is an extension of the original Coda design [7], [15] and improves the data consistency of disconnected operation in three ways. First, the IOT mechanism permits sequences of file operations from a process tree to be grouped together for mutual consistency. Second, it detects read-write consistency violations, not just write-write violations as in the original Coda design. Third, it supports a number of flexible resolution strategies to recover from consistency violations. To the best of our knowledge, this is the first paper to present detailed measurements of resource consumption in a transaction system for mobile computing.

Conceptual simplicity is the keystone of the techniques for resource conservation in IOT. But in spite of their simplicity, our measurements confirm that these techniques are highly effective in keeping resource demands modest. For instance, executing a typical Unix application as an IOT on a disconnected client only increases its elapsed time by 3%. As another example, the average disk space overhead of supporting an entire week of active disconnected IOT activities is only about 5MB—well within the capacity of even the humblest mobile client today.

Since performance measurements and analysis are the central focus of this paper, we only provide a minimal overview of the IOT abstraction and implementation in Section 2. More background material and a broader discussion of IOT-related issues can be found elsewhere [9]. We identify the key resources impacted by the IOT implementation in Section 3, and then devote an entire section to each resource. These sections (4 through 6) are identical in structure: a discussion of how the IOT mechanism impacts a particular resource, a description of the techniques we use to minimize this impact, and the presentation of measurements confirming that these techniques are indeed successful. We conclude the paper with a brief review of related work and a summary of the results.

2 Background

2.1 Coda File System

Coda is an experimental distributed Unix file system providing a single location-transparent name space to a large collection of clients. Support for disconnected operation [6], [7], [16] is primarily the responsibility of the user-level cache manager, *Venus*, at each client. Venus allows applications to continue accessing cached data during periods of disconnection; it also records updates performed while disconnected and replays them upon reconnection to servers

• *Q. Lu is with IBM Corporation, Almaden Research Center, K57/B3, 650 Harry Road, San Jose, CA 95120-6099. E-mail: qilu@almaden.ibm.com.*
• *M. Satyanarayanan is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: satya@cs.cmu.edu.*

For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C97020.

0018-9340/97/$10.00 ©1997 IEEE

in a process known as *reintegration*. Disconnected operation is fully transparent to applications unless one of two events occur: Uncached data is referenced while disconnected, or write-write conflicts are detected upon reintegration. Since Coda is intended for a usage environment dominated by educational, research, software development, and document processing workloads, its design goals specifically exclude database workloads with high degrees of write-sharing across users.

2.2 IOT Motivation and Rationale

The need for IOTs can be seen from a simple example. Suppose an executive, who is disconnected, is writing a business forecast in `report.doc` based on data in a spreadsheet `budget.xls`. Concurrently, his subordinate updates the server copy of `budget.xls`. When the executive reintegrates, Coda will transparently propagate his edits to `report.doc` since there are no write-write conflicts. But there is clearly a consistency problem because the forecast is now obsolete with respect to the new budget figures. This is an instance of a read-write conflict, and epitomizes the kind of situations that IOTs are intended to handle. Conceptually, IOTs can be viewed as a realization of Kung and Robinson's *optimistic concurrency control* (OCC) model [8], customized for a mobile computing environment and applied to a distributed Unix file system rather than a database system.

The name "IOT" stems from the fact that the IOT model focuses only on the *isolation* aspect of the classical *ACID* transactional properties [3], [4]. This restriction of scope is made for two reasons. First, the resource constraints of mobile clients are likely to render support for full ACID properties too expensive. Second, it is inappropriate to offer atomicity and durability guarantees on individual files stronger than those provided by a client's local Unix file system in which Coda stores the contents of cached files.

Although IOT intrafile guarantees are only those of a local Unix file, an IOT does provide stronger guarantees for its metadata. This implies, for example, that the identity of files involved in an IOT are robustly preserved across client crashes even if updates to the contents of those files are lost due to Unix's write-behind policy. To provide these stronger guarantees, IOT stores all its persistent meta-data using *Recoverable Virtual Memory* (RVM) [17], a software package that endows regions of Venus's address space with transactional atomicity and durability properties. To benefit from these properties, updates to recoverable memory must be performed within the scope of an RVM transaction. When such a transaction is committed, RVM flushes data pertinent to the updates to a write-ahead log on disk. The log is asynchronously truncated by RVM to reclaim disk space.

2.3 IOT Implementation

A working IOT extension to Coda has been implemented and operational since early 1995. From a user's viewpoint, an IOT is just a flat sequence of file system calls bracketed by the **`begin_iot()`** and **`end_iot()`** calls of the IOT API. A new transaction[1] is started whenever a nonnested **`begin_iot()`** call is issued by an application. All operations on Coda objects performed by the process that initiated the transaction or its descendant processes are included in the scope of the transaction; operations on non-Coda objects are outside its scope. A special Unix shell, called an *IOT shell*, allows shrink-wrapped applications to be executed within IOTs.

IOT execution occurs entirely on the client. Execution results are held within the client cache, and are not visible on the servers until the execution finishes. A completed transaction remains in the *pending* state until the client is reconnected to servers. Upon reconnection, pending transactions are propagated to the servers one at a time, in a serial order consistent with their local read-write dependencies. The first step in transaction propagation is *consistency validation*, which verifies that the local results of a pending transaction are consistent with the current global server state using serializability-based criteria. The transaction will be immediately committed if it passes such consistency validation. Otherwise, it is called an *invalidated transaction* and must be *resolved* to restore data consistency. The objects involved in the loss of data consistency are referred to as *inconsistent objects.*

The IOT mechanism provides four options for resolving an invalidated transaction: automatic reexecution, automatic invocation of an application-specific resolver (ASR) supplied by the user, automatic abort, and manual repair using an interactive tool. To assist in resolution and manual repair, the IOT mechanism uses a *dual-replica representation* (DRR) mechanism to provide simultaneous access to both the local (i.e., client) and global (i.e., server) states of objects involved in the invalidated transaction. A local replica represents the object value that was last seen by the transaction. By comparing this value with that of the global replica, a resolver can construct an appropriate compensating action. In the earlier example, when the read-write conflict between `budget.xls` and `report.doc` is detected, the DRR mechanism will enable the executive to compare the client and server contents of these files, and to devise an appropriate fix.

The IOT implementation has three major components in Venus: the *execution monitor,* which keeps track of every file operation and maintains data structures such as the readsets and writesets of IOTs; the *concurrency controller,* which performs optimistic concurrency control across clients and two phase locking (2PL) within a client; and the *consistency maintainer,* which is responsible for validating pending transactions and resolving invalidated transactions.

3 Critical Resources on a Disconnected Coda Client

Since the focus of this paper is resource conservation, we restrict our attention to disconnected clients. While resource consumption by the IOT mechanism on servers and connected clients is relevant, it is far less critical. Enhancing the CPU, memory, or disk resources on a server is usually a much simpler proposition than on a client whose design has been optimized for mobility. Further, a connected client has the option of reclaiming disk space by flushing part of

1. We will use the terms *transaction* and *IOT* interchangeably in the rest of this paper as long as there is no ambiguity in the context.

its cache; if necessary, it can also offload computation to servers. In contrast, a disconnected client cannot resort to these strategies. Flushing the cache may eliminate data that is needed later in the disconnected session; it may also cause loss of work if modified files are flushed before they are propagated to servers. While disconnected, a client obviously has no accessible server to which it can offload computation—it has to rely entirely on its own resources.

In effect, disconnected clients offer a worst-case scenario for the IOT mechanism. The longer the period of isolation, the more critical resource conservation becomes. Empirical data from Coda indicates that voluntary disconnections of hours or days are common [13]. Hence, a key design goal of the IOT mechanism is to be able to support extended periods of disconnection, lasting up to a week.

The IOT mechanism consumes three kinds of client resources: CPU cycles and I/O operations togther contributing to increased elapsed time for applications, disk space, and RVM space. The extra CPU cycles are needed to perform the housekeeping activities for IOT management. The increased I/O operations arise from the recording of IOT information in persistent storage. Disk space is mostly used by the transaction system to record data needed by the DRR mechanism. This overhead has to be paid for each IOT, even though the infrequency of IOT invalidation means that most of this data is discarded without being used. As mentioned earlier, RVM space is primarily used by the IOT mechanism for storing its metadata in a persistent and fault-tolerant manner.

Each of the next three sections is devoted to one of these three resources. For each resource, we first examine its use in greater detail. We then describe the conservation strategies for that resource in the current IOT implementation. Finally, we present measurements from controlled experiments, trace-driven simulation, or symbolic analysis to confirm that these techniques are indeed effective in keeping resource consumption low.

Conspicuous by its absence from our discussion is another important resource on a disconnected mobile client: battery power. Minimizing its usage should indeed be a goal relevant to the IOT mechanism. Unfortunately, only crude tools are available to us to instrument client software and hardware for measuring and analyzing power consumption. We are therefore forced to exclude this important resource from our purview. We are also unable to provide any data on the power savings possible by using storage media such as flash or NVRAM rather than disk to store persistent IOT data structures. Incorporating considerations of power consumption would certainly be a valuable extension of the work presented here.

4 CPU AND I/O USAGE

In this section, *CPU and I/O usage* refer to the extra CPU and I/O resources spent on IOT-incurred internal activities. We use the term *performance overhead* to refer to the increase in elapsed time caused by such CPU and I/O usage in executing a particular task. Performance overhead is the metric we use to characterize IOT-incurred CPU and I/O usage.

4.1 Sources of CPU and I/O Usage

IOT-incurred internal activities vary over a broad range from simple bookkeeping to synchronizing concurrent transactions. However, *transaction management* and *file access monitoring* stand out as the primary sources of CPU and I/O usage.

Transaction Management. Three kinds of activities are performed by the transaction system on behalf of a transaction. The first kind of activity is maintaining internal transaction representations, particularly the transaction readset and writeset. Whenever an object is read or written by an ongoing transaction, it needs to be immediately inserted into that transaction's readset or writeset. Information in an element of the readset or writeset includes the internal identifier of the object and its local version number. The second kind of activity is maintaining dependencies on a client among its *live transactions* (that is, those ongoing or pending), using a data structure called a *serialization graph* (SG). Each node in SG corresponds to a live transaction; an edge between two nodes means that the corresponding two transactions performed a pair of conflicting operations on a common object. Note that transaction propagation must follow the partial order dictated by the edges of SG. The third kind of activity is performing local concurrency control using 2PL and maintaining a *wait-for graph* for periodic deadlock detection [1].

File Access Monitoring. The CPU and I/O overheads described in the previous section only occur on *transactional file operations* (that is, those within the scope of some ongoing transaction). But even *normal file operations* (that is, those outside the scope of any transaction) incur some CPU overhead due to the IOT mechanism. This is because the transaction system needs to determine, for every file operation, whether it has been issued within the scope of an IOT. We use the term *file access monitoring* to refer to the two main steps necessary for distinguishing transactional file operations from normal file operations. For a file operation, `op`, the kernel-resident part of the Coda client must obtain information (such as process group id) specific to the process that issued `op`. This information is passed to the user-level Venus, which then uses it to determine whether `op` belongs to an ongoing transaction. Although these steps are simple and cheap, the cumulative impact on CPU usage can be significant because the cost is incurred on every file operation.

4.2 Reducing Performance Overhead

Transaction Management. Because the current IOT implementation represents sets and graphs using simple linked lists, the predominant transaction management activities are searching and updating linked lists stored in RVM. Although list-searching is far more frequent, and though its cost grows quadratically, RVM disk flushes triggered by updates to persistent memory are orders of magnitude slower. Only when the size of linked lists becomes extremely large will the quadratic growth be able to outweigh persistent updates. Thus, our efforts focus on minimizing the negative impact of persistent updates on performance overhead.

We adopt the asynchronous update strategy that has proven effective in a variety of contexts such as buffered

Phase	Objects Read	Objects Updated	Single Transaction		Transaction per Phase	
			Elapsed Time (seconds)	Overhead (%)	Elapsed Time (seconds)	Overhead (%)
MakeDir	8	4	1.6 (0.5)	23.1	1.7 (0.7)	30.8
Copy	150	74	17.9 (1.0)	34.6	19.5 (1.1)	46.6
ScanDir	23	0	17.3 (0.8)	10.9	18.5 (0.8)	18.6
ReadAll	169	0	27.5 (1.3)	11.8	28.2 (0.9)	14.6
Make	76	23	87.7 (1.6)	2.7	87.9 (2.3)	2.9
Total	426	101	152.0 (2.2)	8.4	155.8 (2.8)	11.13

This table displays the elapsed time of executing the Andrew Benchmark both as a single transaction and with each phase encapsulated in its own transaction. The time values represent the mean over ten runs, and the numbers in parentheses are standard deviations. The listed performance overhead is with respect to the elapsed time of executing the benchmark as a normal application using an IOT-Venus shown in the table in Fig. 4.

Fig. 1. Transaction performance for Andrew Benchmark.

I/O in Unix file systems. The basic idea is to defer updates to persistent transaction data structures, at the risk of exposing a small window of vulnerability for losing persistent data. This offers two major performance benefits. First, since RVM disk flushes are now asynchronous with respect to updates, the flush latency is no longer directly borne by the process that triggers those updates. Second, updates to identical or overlapping ranges of RVM addresses can be coalesced to reduce total RVM log traffic.

This strategy is used in two ways in the current IOT implementation. First, we use RVM in the *asynchronous flush* mode. In this mode, a separate daemon thread is dedicated to performing asynchronous disk flushes for buffered RVM updates, with a maximum delay bound of 30 seconds [6]. The second technique applies the same principle at a higher level: Deferring updates to the persistent data structures of `SG` in such a manner that a clean `SG` can still be reconstructed upon restart after a crash.

File Access Monitoring. Minimizing the performance overhead of file access monitoring is crucial because any significant slowdown when IOTs are not being used will seriously undermine the usefulness of the system. Most of the performance cost here comes from gathering and passing the process-specific information from the kernel to the user-level Venus. There is no simple way to avoid this overhead.

Another source of overhead is searching through the data structures of ongoing transactions. Since we do not anticipate a large number of concurrent transactions on a client, this overhead is likely to be small. We use a simple scheme to completely avoid this search when possible: A flag indicates whether there is any transaction currently active in the system. If this flag is off, the IOT system can immediately conclude that it is servicing a normal file operation rather than a transactional file operation.

4.3 Evaluating Performance Overhead

Our evaluation of IOT-incurred CPU and I/O usage addresses the following questions. What is the performance overhead of executing a typical Unix application as a transaction on a disconnected client? What is the corresponding overhead without transactional encapsulation? What are the dominant factors in transaction management performance?

4.3.1 Methodology

We rely on controlled experiments in which a broad range of workloads are executed on a disconnected client. The client is a DEC 425sl laptop with a 25MHz Intel 486 processor and a 400MB disk, running the Mach 2.6 operating system. The workloads include the Andrew Benchmark [5], replay of file reference traces [12], and four typical software development and document processing tasks. To measure the performance overhead of transaction management, we compare the elapsed times of executing the selected workload with and without transactional encapsulation. To measure the performance overhead of file access monitoring, the comparison is between the elapsed time of executing the workloads as a normal application using a Venus with the IOT extension (denoted IOT-Venus) and that using a Venus without the IOT extension (denoted plain-Venus).

4.3.2 Performance Overhead of Transaction Management

Andrew Benchmark. The Andrew Benchmark is widely used for comparing file system performance. It has five phases: `MakeDir`, which creates a few directories in a test area; `Copy`, which copies files from a source tree into the test area; `ScanDir`, which examines all the directories and files in the test tree; `ReadAll`, which reads all the files; and `Make`, which compiles a number of C programs. Fig. 1 presents the measurement results of executing the benchmark as a single transaction as well as one transaction per phase.

The performance overhead of the entire benchmark is about 10%, but the degradation across different phases varies considerably. For example, the performance overhead of the Copy phase is quite high, while that of the Make phase is rather low. This is because the Copy phase is very I/O intensive while the Make phase is not. I/O intensity, defined as the frequency of file operations on Coda objects, proves to be the dominant factor in determining IOT performance overhead. The more I/O intensive a transaction, the more frequent the need to extend its readset or writeset. This, in turn, implies more persistent updates within a given time period. As a result, there are fewer opportunities for RVM disk flushes to overlap with transaction computation, thus leading to higher performance overhead.

Trace Replay. Our experiments using trace replay workloads further confirm the negative impact of I/O intensity on transaction performance. The workloads involve replaying four segments of file reference traces, each capturing 30 minutes of heavy user activity. Characteristics of these trace segments and more details of their use in replay

Traces	I/O Intensive (λ = 60)					Non-I/O Intensive (λ = 1)				
	Normal Replay (seconds)		Transactional Replay (seconds)		Overhead (%)	Normal Replay (seconds)		Transactional Replay (seconds)		Overhead (%)
Segment #1	225.0	(8.3)	250.0	(10.8)	11.1	1,659.8	(16.5)	1,687.4	(20.8)	1.7
Segment #2	278.6	(8.7)	309.4	(7.4)	11.1	1,572.4	(5.4)	1,597.8	(9.0)	1.6
Segment #3	125.4	(4.7)	149.6	(2.9)	19.3	1,537.4	(8.8)	1,564.0	(19.5)	1.7
Segment #4	24.8	(0.8)	29.2	(1.6)	17.7	1,570.2	(3.9)	1,575.4	(5.0)	0.3

This table shows the elapsed time of running trace replay both as a normal application and as a transaction. The time values represent the mean over five runs, and the numbers in parentheses are standard deviations.

Fig. 2. Transaction performance overhead for trace replay.

Typical Tasks	Normal Execution (seconds)		Transactional Execution (seconds)		Overhead (%)
Compile a Coda Client	3,679.2	(50.7)	3,738.8	(34.5)	1.6
Compile a Coda Server	998.6	(6.3)	1,018.6	(3.6)	2.0
Typeset a PhD Dissertation	146.0	(1.9)	150.8	(1.5)	3.3
Typeset a Thesis Proposal	34.2	(0.5)	35.6	(0.6)	4.1

This table shows the elapsed time of executing four typical tasks in the Coda environment as transactions and as normal applications: compiling a Coda client, compiling a Coda server, typesetting a 204-page PhD dissertation, and typesetting a 52-page thesis proposal. The time values represent the mean over five runs, and the numbers in parentheses are standard deviations.

Fig. 3. Transaction performance overhead for common tasks.

experiments have been described by Mummert [11]. The I/O intensity of a trace replay experiment can be adjusted by a parameter, λ, referred to as *think threshold.* Delays less than λ seconds in the original trace are suppressed during replay. When λ = 1, most of the original delays are preserved so that replay proceeds at a speed close to that of the original trace. When λ = 60, few delays are preserved, thus resulting in much higher I/O intensity.

The measurement results in Fig. 2 clearly demonstrate the strong negative effect of I/O intensity on transaction performance. For each of the four trace segments, the performance overhead of transactional replay is less than 2% when λ is 1, but rises to between 10% to 20% when λ is 60. The overhead is still significantly lower than that of the first two phases of the Andrew Benchmark. This is because of longer execution duration, making RVM disk flushes less influential on total performance. However, the overhead is higher than that of the entire Andrew Benchmark due to the higher overall I/O intensity.

Typical Tasks. To obtain another data point on the performance overhead of IOT, we examine four software development and document processing tasks representative of common workloads in our environment. As shown in Fig. 3, the observed performance overhead for these tasks ranges between 1.5% and 4.5%. The two compilation tasks have lower performance overhead mainly due to their much longer execution duration. This offers transaction-triggered RVM disk flushes more opportunities to overlap with computation.

4.3.3 Performance Overhead of File Access Monitoring

We measured the performance overhead of file access monitoring by comparing the elapsed time of executing the set of workloads described in the previous section as normal applications using plain-Venus and IOT-Venus. The results shown in Figs. 4, 5, and 6 indicate that the performance overhead is well under 1% most of the time. This matches our qualitative experience that there is no noticeable performance degradation when transactions are not being used.

4.3.4 Summary

Our decision to focus on transaction-triggered RVM disk flushes turns out to be well justified. The strategy of deferring persistent updates is effective in keeping IOT performance overhead acceptable in most circumstances. Generally speaking, given the same volume of I/O activity, the longer it takes to run a transaction, the less the performance penalty it suffers. Since normal Unix applications interleave file access operations with computation and/or user think time, observed performance degradation for transaction execution is typically around 3%.

Careful engineering and the simple optimization employed in file access monitoring are adequate to render its performance overhead negligible under most circumstances. It should be noted that our evaluation does not include concurrent transaction executions, where transaction performance can vary widely depending on the patterns of read-write sharing. A meaningful performance study of concurrent transactions needs to wait until a significant amount of empirical data on IOT usage has been gathered.

5 Disk Space Usage

5.1 Sources of Disk Space Usage

There are two forms of disk space overhead in the IOT mechanism, both arising from the need to support conflict resolution. First, in preparation for resolution, shadow files have to be preserved in the cache between the time a transaction is completed locally and the time it is validated at a server. Second, when resolution occurs, the DRR representation requires additional disk space to support multiple

	Plain-Venus (seconds)		IOT-Venus (seconds)		Overhead (%)
MakeDir	1.3	(0.5)	1.3	(0.5)	0.0
Copy	12.8	(0.9)	13.3	(0.9)	3.9
ScanDir	14.7	(0.7)	15.6	(0.7)	6.1
ReadAll	23.6	(0.8)	24.6	(0.8)	4.2
Make	85.0	(1.2)	85.4	(1.0)	0.5
Total	137.4	(0.7)	140.2	(0.9)	2.0

This table shows the elapsed time of executing the Andrew Benchmark as a normal application using both plain-Venus and IOT-Venus. The time values represent the mean over ten runs, and the numbers in parentheses are standard deviations.

Fig. 4. Nontransactional performance for Andrew Benchmark.

Traces	I/O Intensive (λ = 60)					Non-I/O Intensive (λ = 1)				
	Plain Venus (seconds)		IOT Venus (seconds)		Overhead (%)	Plain Venus (seconds)		IOT Venus (seconds)		Overhead (%)
Segment #1	224.2	(9.0)	225.0	(8.3)	0.4	1,655.8	(5.3)	1,659.8	(16.5)	0.2
segment #2	277.2	(5.0)	278.6	(8.7)	0.5	1,564.8	(7.8)	1,572.4	(5.4)	0.5
Segment #3	125.0	(1.6)	125.4	(4.7)	0.3	1,523.4	(8.1)	1,537.4	(8.8)	0.9
Segment #4	24.6	(0.6)	24.8	(0.8)	0.8	1,564.6	(5.1)	1,570.2	(3.9)	0.4

This table shows the elapsed time of running trace replay as a normal application using both plain-Venus and IOT-Venus. The time values represent the mean over five runs, and the numbers in parentheses are standard deviations.

Fig. 5. Nontransactional performance overhead for trace replay.

Typical Tasks	Plain-Venus (seconds)		IOT-Venus (seconds)		Overhead (%)
Compile a Coda Client	3,662.0	(37.0)	3,679.2	(50.7)	0.5
Compile a Coda Server	992.0	(8.3)	998.6	(6.3)	0.6
Typeset a PhD Dissertation	145.4	(0.6)	146.0	(1.9)	0.4
Typeset a Thesis Proposal	33.8	(0.5)	34.2	(0.5)	1.2

This table shows the elapsed time of executing the workloads described in Fig. 3 as normal applications using both plain-Venus and IOT-Venus. The time values represent the mean over five runs, and the numbers in parentheses are standard deviations.

Fig. 6. Nontransactional performance overhead for typical tasks.

views of inconsistent objects. It should be noted that although RVM indirectly uses disk space, we treat this usage as a distinct resource and discuss it separately in Section 6.

Shadow Cache Files. The need to maintain shadow cache files arises because the IOT mechanism presents a last-access-snapshot view to resolvers. This view provides a resolver with the access to the state of objects as seen by the invalidated transaction at the end of its execution. Since these last-access values may be locally updated by subsequent transactions, the IOT mechanism has to preserve shadow copies in the cache. These shadow copies are created lazily.

When an object `obj` is about to be updated and the current value of `obj` was last accessed by at least one pending transaction, `T`, the transaction system will immediately create a shadow cache file for `obj` and `T`, denoted `shd(T,obj)`, and use it to record the current value of `obj`. In case `T` fails consistency validation, the value recorded in `shd(obj,T)` will be used to present the last-access-snapshot view of `obj` to `T`'s resolver. `shd(T,obj)` will be garbage collected as soon as `T` is committed or resolved. We use the term *shadow space* to refer to the disk space consumed by shadow cache files.

Conflict Representation. The second major source of disk space usage is the DRR representation of inconsistent objects. Since an inconsistent object can be a directory, the local or global replica of an inconsistent object is, in the most general case, a subtree of objects rooted at that directory and representing its local or global state respectively. The disk space for the subtree corresponding to the local replica is supplied by the shadow cache files discussed in the previous section. But additional disk space is required for the global replica. It should be noted that this additional disk usage occurs only infrequently, because conflicts are rare. Further, this space is only used for a short period of time since conflicts are typically resolved soon after their detection.

5.2 Reducing Disk Space Usage

Shadow Cache Files. Our approach to keeping shadow space consumption low involves a technique called *transaction cancellation.* The intuition behind this technique is that file updates are often repetitive. For example, the same set of object files are often repeatedly regenerated in a typical edit-compile-debug cycle. If each compilation is executed as a transaction, a later transaction will completely wipe out the results of the previous one. Therefore, there is no need to retain the results of the *redundant* older transaction, because it no longer has any impact on the file system state. Once a transaction has been declared redundant, its persistent resources such as shadow cache files can be reclaimed immediately.

Trace Identifier	Machine Name	Machine Type	Simulation Start	Records
Full-Week #1	concord.nectar.cs.cmu.edu	Sun 4/330	26-Jul-91, 11:41	4,008,084
Full-Week #2	holst.coda.cs.cmu.edu	DEC station 3100	18-Aug-91, 23:31	2,303,306
Full-Week #3	ives.coda.cs.cmu.edu	DEC station 3100	03-May-91, 23:21	4,233,151
Full-Week #4	messiaen.coda.cs.cmu.edu	DEC station 3100	27-Sep-91, 00:15	1,634,789
Full-Week #5	purcell.coda.cs.cmu.edu	DEC station 3100	21-Aug-91, 14:47	2,193,320

This table shows key characteristics of the five week-long traces used in the trace-driven simulations reported in Section 5.3.1. The `Records` column refers to the number of trace records that are actually processed by the trace simulator during the simulated period of 168 hours. These traces have been used previously by Kistler [6] and Mummert [11] for analyses of other aspects of Coda.

Fig. 7. Characteristics of file reference traces.

Application	Executable Pathname	Frequency
`awk`	/bin/awk	9.16%
`cc`	/usr/cs/bin/cc	5.31%
`cp`	/bin/cp	1.48%
`cpp`	/usr/cs/lib/cpp	0.02%
`emacs`	/usr/cs/bin/emacs	0.77%
`find`	/usr/cs/bin/find	1.45%
`make`	/usr/cs/bin/make	6.25%
`rcsci`	/usr/misc/bin/rcsci	0.01%
`rcsco`	/usr/misc/bin/rcsco	0.30%
`scribe`	/usr/misc/bin/scribe	0.02%
`sed`	/bin/sed	8.40%
`sh`	/bin/sh	66.00%
`vi`	/usr/ucb/vi	0.83%

This table displays the name and pathname of the applications that are simulated as transactions. The `Frequency` column shows the percentage of each application among the total number of simulated transactions.

Fig. 8. Transactional applications in trace-driven simulation.

In practice, detecting redundant transactions is more complex than implied by the simple example above. The transaction cancellation algorithm has to ensure that the cancellation of a transaction, `T`, preserves strong semantic equivalence. Our algorithm consists of three parts. First, we maintain a transaction mutation log and use log-record tagging to detect whether all the mutations performed by `T` have been completely overwritten or offset[2] by subsequent transactions. This guarantees that `T` no longer has any direct effect on the file system state. Second, we traverse `SG` and analyze transaction read-write dependencies to ensure that the elimination of `T` will not affect the consistency validation outcome of any other live transactions. Third, we employ a graph-based method to make sure that all redundant transactions containing offsetting mutations are canceled together. Further details of the transaction cancellation algorithm can be found elsewhere [9].

Conflict Representation. Since resolution is usually of short duration, its transient disk space usage is far less critical than the long-term space usage by shadow files. Rather than striving for minimal disk space consumption, we have therefore chosen to bias the DRR representation toward greater flexibility and effectiveness from the viewpoint of writing resolvers. The ideal DRR representation would provide a mutable *workspace* replica in addition to read-only local and global replicas. However, concerns of implementation complexity as well as space overhead have led to the current design which has a read-only local replica and a mutable global replica. Our experience with writing resolvers confirms that this is indeed a good compromise, striking the right balance between conserving disk space and enhancing usability.

2. For example, `rmdir foo/bar` offsets a previous `mkdir foo/bar`.

5.3 Evaluating Disk Space Usage

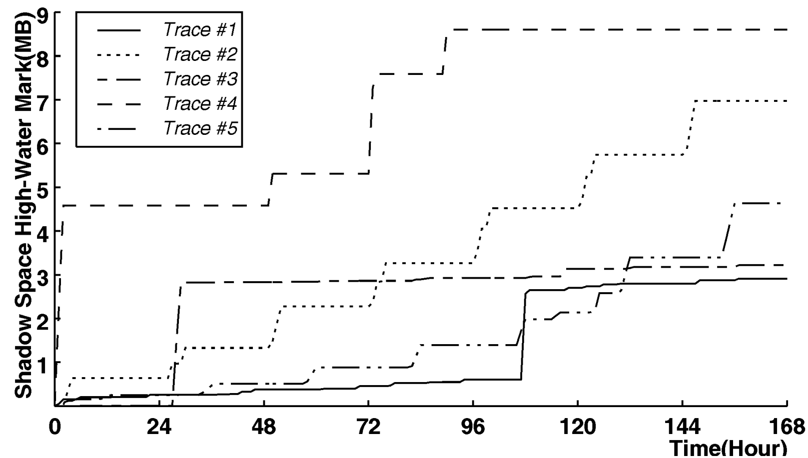
For shadow space, our evaluation concentrates on measuring long term accumulated cost. For conflict representation, the focus is on typical cases. Specifically, we address two questions. What is the amount of shadow space needed for disconnected transaction processing over an extended period of time, up to a week? When a transaction is invalidated, what is the amount of disk space typically needed by the DRR representation of an inconsistent object?

5.3.1 Shadow Space Cost

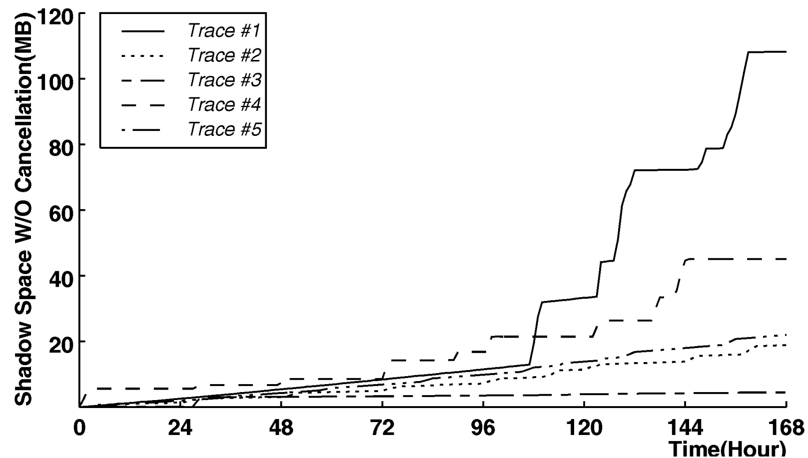
Methodology. We use trace-driven simulation to obtain a realistic estimate of how much shadow space is needed to support a full week of disconnected transaction processing. Our analysis is based on five week-long file reference traces from our environment, carefully screened to ensure sustained high levels of activity. Fig. 7 displays salient characteristics of these traces.

We have developed a transaction simulator that reads a sequence of trace records, emulates disconnected transaction processing activities, and records resource usage. A list of pathnames representing applications to be executed as transactions are provided as input to the simulator. Based on manual examination of the traces, we have chosen the 13 frequently encountered applications shown in Fig. 8 as candidates for transactional execution.

Results. Fig. 9a shows the high-water marks of shadow space cost for the five traces. The highest cost is less than 9MB and the average cost is about 5MB. This is quite small



(a)



(b)

The graph in part (a) shows the high-water marks of shadow space cost recorded by the transaction simulator for the five week-long traces shown in Fig. 7. *The graph in part (b) shows the same results except that transaction cancellation is not performed during the simulation. Note that the scales of the Y axis are different in the two graphs.*

Fig. 9. High-water marks of shadow space cost.

Trace	Total Tran. Count	Live Transaction Count	Read Only Transaction Count	Canceled Transaction Count	Total File Reference Count	Transactional File Reference Count
#1	1,028	51 (5.0%)	277 (26.9%)	700 (68.1%)	4,008,084	2,596,980 (64.8%)
#2	781	86 (11.0%)	182 (23.3%)	513 (65.7%)	2,303,306	1,267,155 (55.0%)
#3	495	57 (11.5%)	231 (46.7%)	207 (41.8%)	4,233,151	396,427 (9.4%)
#4	142	40 (28.2%)	21 (14.8%)	81 (57.0%)	1,634,789	442,855 (27.1%)
#5	952	63 (6.6%)	514 (54.0%)	375 (39.4%)	2,193,320	642,647 (29.3%)
Avg	679.6	59.4 (8.7%)	245 (36.1%)	375.2 (55.2%)	2,874,530	1,069,212.8 (37.2%)

This table presents transaction and file reference statistics of the trace-driven simulations discussed in Section 5.3.1.

Fig. 10. Transaction statistics in trace-driven simulation.

considering the growth of disk capacity on mobile clients, the long duration of disconnection, and the large volume of transaction activities shown in Fig. 10. On average, about 680 transactions are executed during a week, and over 37% of the file operations occur within the scope of some transaction.

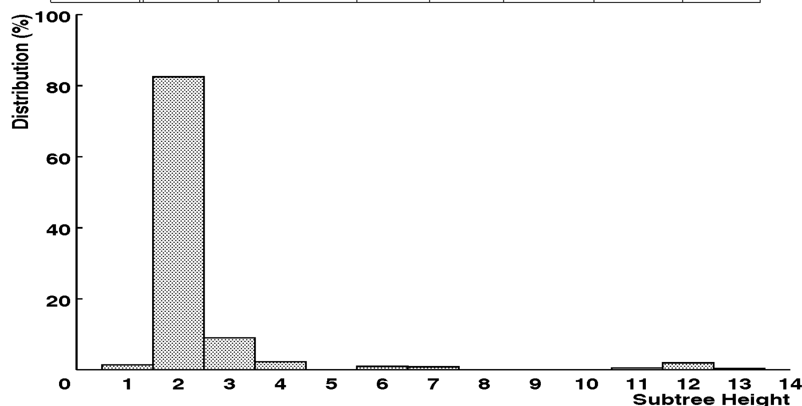
The effectiveness of transaction cancellation in keeping shadow space cost low is evident from Fig. 10, which shows that more than 55% of the transactions are redundant. Fig. 9b shows what shadow space cost would have been, had transaction cancellation been turned off. This graph shows that the highest shadow space cost for a week would have been close to 115 MB, and the average cost over 40 MB. The reduction in shadow space cost by transaction cancellation is thus about one order of magnitude!

Characteristic	Volume Type									
	User		Project		System		BBoard		All	
Total Number of Volumes		786		121		72		71		1,050
Total Number of Directories		13,955		33,642		9,150		2,286		59,033
Total Number of Files		152,111		313,890		113,029		144,525		723,555
Total Size of File Data (MB)		1,700		7,000		1,500		560		11,000
File Size (KB)	10.3	(65.0)	24.0	(145.7)	16.4	(72.6)	2.6	(7.0)	19.1	(118.0)
Directories/Directory	3.6	(13.4)	3.0.	(4.5)	3.6	(10.4)	6.8	(19.4)	3.2	(8.3)
Files/Directory	14.6	(30.6)	16.2	(35.6)	15.9	(36.9)	66.9	(142.4)	15.7	(34.5)
Hard Links/Directory	3.7	(12.4)	2.0	(1.5)	4.0	(3.9)	0.0	(0.0)	3.4	(5.7)
Symbolic Links/Directory	4.1	(10.1)	3.4	(7.5)	13.6	(45.3)	6.0	(25.9)	6.3	(24.9)

This table summarizes various characteristics of system, user, project, and bulletin board volumes in AFS at Carnegie Mellon University in early 1990. The data was obtained via static analysis by Ebling [2]. The numbers in parentheses are standard deviations.

Fig. 11. File and directory characteristics of AFS.

Subtree Height	Subtree Height Distribution Over Mutation Operations (%)							
	Create	Link	Unlink	Mkdir	Rmdir	Symlink	Rename	Total
1	0.00	0.00	1.45	0.00	0.00	0.00	0.00	1.46
2	36.53	1.05	37.14	0.15	0.00	1.94	5.67	82.50
3	2.49	0.13	4.16	0.07	0.05	1.26	0.82	8.99
4	0.77	0.17	1.03	0.01	0.00	0.14	0.16	2.28
5	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.04
6	0.06	0.00	0.06	0.00	0.00	0.00	0.88	0.99
7	0.00	0.00	0.00	0.00	0.00	0.00	0.86	0.86
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
11	0.13	0.13	0.27	0.00	0.00	0.00	0.00	0.53
12	0.67	0.32	0.98	0.00	0.00	0.00	0.00	1.98
13	0.03	0.03	0.31	0.00	0.00	0.00	0.00	0.36
total	40.69	1.83	45.40	0.24	0.07	3.36	8.41	100.00



The table and graph in this figure show the distribution of replica subtree height over mutation operations obtained via trace-driven analysis on the traces listed in Fig. 7.

Fig. 12. Height distribution for replica subtree.

5.3.2 Disk Space Cost for Conflict Representation

Methodology. The amount of disk space consumed by the DRR of an inconsistent object is given by the sum of the sizes of the cache representations of the objects in the global replica subtree of the DRR.[3] The ideal way to obtain this quantity would be to collect empirical data from resolutions in a large-scale deployment of the IOT system. In the absence of such a deployment, we provide an estimate obtained by a combination of trace-driven and symbolic analysis. As input to this analysis, we use the file and directory characteristics from AFS [5] shown in Fig. 11. The similarity in workloads and user populations of Coda and AFS make this a reasonable extrapolation.

The first step in this evaluation is to estimate the height of a typical replica subtree using trace analysis. Every replica subtree corresponds to a conflict, and every conflict involves at least one update. By examining all the mutation operations in a set of file reference traces, we can determine the distribution of locations within the hierarchical file name space referenced by those mutations. From this, we can obtain the distribution of the heights of the replica subtrees rooted at the mutated locations.

The next step is to use symbolic analysis to deduce the number of files in a typical replica subtree. This analysis is

3. For brevity, we refer to "global replica subtree" as "replica subtree" in the rest of this paper.

based on the height distribution obtained from the previous step, and the file system statistics shown in Fig. 11. Finally, an estimate of disk space usage for a typical replica subtree can be obtained by multiplying the number of files in the subtree and their average size.

Results and Analysis. The results of trace-driven analysis of the week-long traces listed in Fig. 7 are shown in Fig. 12. They confirm our intuition that most mutations are performed close to the leaves of a hierarchical file name space: A typical replica subtree has a height of two.

To obtain the relationship between the height of a subtree and the number of files in that subtree, we need to solve the following recurrences:

$$P(1) = 0 \tag{1}$$

$$P(H) = F + D * P(H-1) \tag{2}$$

where $P(H)$ is the number of files in a subtree of height H, F is the number of files per directory, and D is the number of directories per directory. The solution to the above equations is:

$$P(H) = F * \left(D^{H-1} - 1\right) \Big/ (D-1)$$

Using the numbers from Fig. 11, we have:

$$P(H) = 15.7 * \left(3.2^{H-1} - 1\right) \Big/ 2.2$$

Therefore, a typical replica subtree of height 2 will have 16 files in it. Since the average file size is 19.1 KB from Fig. 11, a typical replica subtree costs 306 KB in disk space. In other words, each inconsistent object involved in an invalidated transaction will require roughly 300 KB for its DRR representation during resolution. Even if the transaction involves 100 inconsistent objects, the space cost still remains a modest 30 MB—entirely acceptable as short-term space usage.

6 RVM Space Usage

The persistent, fault-tolerant virtual memory abstration provided by RVM is a key element in keeping the implementation complexity of the IOT mechanism tractable. However, this benefit of RVM comes at a price: Since RVM is not integrated with the virtual memory component of the operating system, its performance degrades rapidly with the onset of paging [17]. To avoid paging, it is necessary to keep the portion of Venus' address space backed by RVM to a small fraction of available physical memory. On today's laptops with 20 MB or so of memory, this implies total RVM space usage of no more than a few MB.

6.1 Sources of RVM Space Usage

Persistent IOT Data Structures. The primary reason we store IOT data structures in recoverable memory is because important information about a transaction such as readset, writeset, and the execution environment must be able to survive crashes and shutdowns. Such events tend to be significantly more frequent on mobile clients than on stationary workstations. The recoverable memory allocated to a transaction can be reclaimed as soon as it is committed, resolved, or canceled.

Conflict Representation. During conflict resolution, objects belonging to the local or global replicas of a DRR must be accessible to the resolver in the same way as a normal cached object. This requires the IOT system to maintain an internal persistent representation of such objects. This representation primarily consists of file system metadata such as link count, version information, size, and modification time [6].

6.2 Reducing RVM Space Usage

Persistent IOT Data Structures. Since the amount of RVM space used by transaction data structures is proportional to the number of live transactions, the transaction cancellation mechanism described in Section 5.2 also contributes to reducing RVM space usage. Early experience with the IOT mechanism revealed another significant opportunity for reducing RVM space usage. Part of the meta-data associated with an IOT is the set of values of Unix environment variables accessible to the IOT when it was initiated. Since these variables are rarely modified, the simple optimization of sharing a copy of this state across IOTs is highly effective. If a few of these variables are modified, only those variable need to be explicitly represented in the state of IOTs initiated subsequently—effectively a simple form of copy-on-write.

Conflict Representation. Because of the strong similarity between the ways persistent memory and disk space are consumed by the DRR representation, the same resource management strategy discussed in Section 5.2 is applicable here.

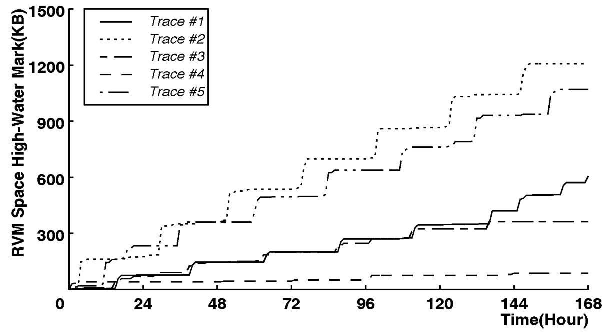
6.3 Evaluating RVM Space Usage

Our evaluation of RVM space usage parallels that of disk space usage presented in Section 5.3. We address two questions: What is the amount of RVM space needed for disconnected transaction processing over sustained periods, up to a week? When a transaction is invalidated, what is the amount of RVM space needed for representing an inconsistent object?

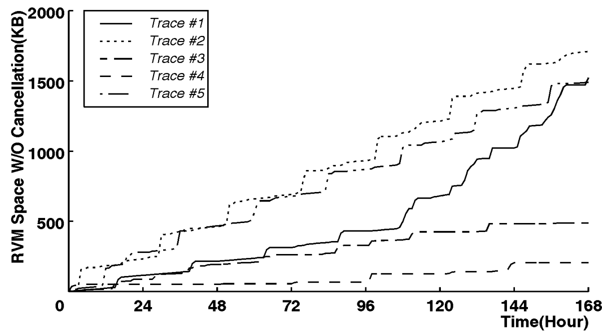
6.3.1 RVM Space Cost for Transaction Data Structures

Methodology. We employ the same trace-driven simulation experiment described in Section 5.3.1 to measure the accumulated RVM cost for maintaining persistent transaction data structures during a full week of disconnected transaction processing. We also use controlled experiments to measure the RVM cost for individual transactions using common workloads such as software development and document processing. In addition, we execute three different *SynRGen micro-models* [2] as workloads representative of interactive transactions containing repetitive editing and compiling activities.

Results. Fig. 13a displays the high-water marks of RVM space cost recorded during trace-driven simulation. The simulation assumes that the total RVM space cost for maintaining the shared pool of environment variables is 3 KB, which is ample in our experience. The highest RVM



(a) With Transaction Cancellation



(b) Without Transaction Cancellation

The graph in part (a) shows the high-water marks of RVM space cost recorded by the transaction simulator for the five week-long traces shown in Fig. 7. The graph in part (b) shows same results except that transaction cancellation is not performed during the simulation.

Fig. 13. High-water marks of RVM space cost.

Application	RVM Space Cost (B)		
	Total	Readset	Writeset
Latex Dissertation (204 pages)	3,162	2,340	612
Latex Proposal (52 pages)	1,052	756	108
Latex Short Paper (6 pages)	830	540	108
Build Coda Venus	13,278	8,280	4,794
Build Coda Server	7,493	6,012	1,247
Build Coda Repair Tool	2,146	1,584	340
Synrgen Codahacker	6,071	2,700	3,205
Synrgen Programmer	5,933	2,664	3,103
Synrgen Synrgenhacker	5,641	2,664	2,808

This table shows the RVM space cost for executing common applications as transactions. The `Readset` and `Writeset` columns display the RVM space cost for storing the transaction readset and writeset respectively. Note that the total cost here does not include the RVM space for environment variables, which are stored in a shared pool. The average RVM space cost of this pool is 2.4 KB in these workloads.

Fig. 14. RVM space cost for common transactions.

Application Name	Average Readset Size	Average Writeset Size	Average RVM Space Cost (B)
`awk`	0.02	0.01	188.8
`cc`	18.1	2.7	862.7
`cp`	30.5	15.3	2,126.3
`cpp`	51.5	1.0	2,093.5
`emacs`	35.6	4.0	1,543.3
`find`	62.5	0.3	2,512.5
`make`	108.7	13.6	4,726.4
`rcsci`	13.0	7.0	779.0
`rcsco`	9.0	4.6	614.4
`scribe`	13.7	3.0	745.7
`sed`	0.1	0.02	193.1
`sh`	50.9	2.5	2,108.1
`vi`	5.5	2.4	390.2

This table shows the average RVM space cost and the average readset and writeset sizes of applications executed as transactions in trace-driven simulations using the traces listed in Fig. 7. Note that the RVM cost here does not include the space for environment variables.

Fig. 15. RVM usage statistics from trace-driven simulation.

space cost among the five traces in Fig. 13a is about 1.2 MB, and the average cost is about 0.5 MB. This falls well below the limit of a few MB of RVM space, mentioned earlier in this section as necessary to obtain good RVM performance on current mobile hardware. Fig. 13b shows the important role of transaction cancellation in reducing RVM space usage.

Further data on RVM space usage by typical applications is provided by Fig. 14, which reports on the results of controlled experiments, and Fig. 15, which presents results from trace-driven simulation. The data shows that RVM space cost per transaction for most applications is small, ranging from a few KB to about 13 KB.

6.3.2 RVM Space Cost for Conflict Representation

Methodology. Our approach to estimating the RVM space cost for a typical replica subtree during resolution is identical to that discussed earlier in Section 5.3.2 for disk space usage. We use symbolic analysis to derive a formula for the number of nodes in a typical replica subtree, and then apply the empirical data presented in Fig. 11 to this formula.

Results and Analysis. The relationship between the number of nodes and the height of a subtree is the solution of the following recurrences:

$$N(1) = 1 \tag{3}$$

$$N(H) = (1 + F + L + S) + D * N(H - 1) \tag{4}$$

where N is the number of nodes; H is the subtree height; F is the number of files per directory; L is the number of links per directory; S is the number of symbolic links per directory, and D is the number of directories per directory. The solution is the following formula:

$$N(H) = \left(D^H + (F + L + S) * D^{H-1} - F - L - S - 1\right) \Big/ (D - 1)$$

Using the numbers from Fig. 11, we have:

$$N(H) = \left(3.2^H + 25.4 * 3.2^{H-1} - 26.4\right) \Big/ 2.2$$

According to this formula, a typical replica subtree with a height of 2 contains 30 nodes. In the current Venus implementation, each node costs about 392 bytes of RVM space. Hence, a typical replica subtree costs around 11.7 KB of RVM space. Even if a transaction being resolved has 100 inconsistent objects, the RVM space used by its DRR representation will be less than 1.2 MB—certainly acceptable as short-term RVM usage on today's hardware.

7 Conclusion

Preserving data consistency in mobile computing environments is a challenge that has attracted growing interest in the recent past. For example, the Bayou system [18], [19] is exploring techniques for managing databases that are replicated on mobile clients. This work differs from Coda in its use of a peer-to-peer model rather than a client-server model, and in its focus on application-specific techniques rather than the system-wide, transparent infrastructure provided by Venus. As another example, Pitoura and Bhargava [14] are investigating weakened consistency models for classical database systems operating in mobile environments.

In contrast to these efforts, Coda extends the widely-used Unix file system in a manner that improves consistency yet is minimally demanding of applications. New or existing applications can easily use the IOT extensions to the Unix API. Even unmodified applications can benefit from improved consistency if they are executed inside an IOT shell. Nontransactional and transactional execution of applications can be interleaved. Overall, the IOT mechanism of Coda strikes a good balance between the conflicting demands of consistency, upward compatibility, and resource conservation. To the best of our knowledge, this work represents the first attempt to provide a transactional capability for mobile file access that pays serious attention to these pragmatic concerns.

The goal of this paper is to show that the benefits of Coda's IOT mechanism come at an acceptable price. Specifically, our goal is to demonstrate that resource consumption on mobile clients by the IOT mechanism is acceptable. We have presented measurements from the IOT implementation to show that it is indeed minimally demanding of all three critical resources on a mobile client: CPU and I/O usage, disk space, and RVM space. Our measurements confirm that even a severely resource-constrained mobile client can benefit from the improved consistency offered by IOT.

The next step is to obtain broader validation of the IOT concept. In particular, this paper does not provide evidence on the usability of the IOT model, or of its applicability to a wide range of applications. Only extended usage experience by a large and diverse user community can provide the empirical data necessary to critically evaluate IOT along these dimensions. We look forward to this challenge.

Acknowledgments

This research was supported by the U.S. Air Force Materiel Command (AFMC) and Defense Advanced Projects Agency (DARPA) under contract number F19628-93-C-0193. Additional support was provided by the IBM, Digital Equipment, and Intel Corporations. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, IBM, DEC, Intel, CMU, or the U.S. Government. This work builds upon the contributions of many past and present members of the Coda project. We wish to especially thank Jay Kistler, for his early advocacy of this line of research, and Lily Mummert for her file tracing and replay tools that have played a central role in the evaluations presented here.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
[2] M.R. Ebling and M. Satyanarayanan, "SynRGen: An Extensible File Reference Generator," *Proc. 1994 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 108-117, Nashville, Tenn., May 1994.
[3] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The Notions of Consistency and Predicate Locks in a Distributed Database System," *Comm. ACM*, vol. 19, no. 11, pp. 624-633, 1976.
[4] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.

[5] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Trans. Computer Systems*, vol. 6, no. 1, pp. 51-81, Feb. 1988.
[6] J.J. Kistler, "Disconnected Operation in a Distributed File System," PhD thesis, Carnegie Mellon Univ., Pittsburgh, May 1993.
[7] J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 3-25, Feb. 1992.
[8] H.T. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems*, vol. 6, no. 2, pp. 213-226, 1981.
[9] Q. Lu, "Improving Data Consistency in Mobile File Access Using Isolation-Only Transactions," PhD thesis, Carnegie Mellon Univ., Pittsburgh, May 1996.
[10] Q. Lu and M. Satyanarayanan, "Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions," *Proc. Fifth Hot Topics in Operating Systems*, pp. 124-128, Orcas Island, Wash., May 1995.
[11] L.B. Mummert, M.R. Ebling, and M. Satyanarayanan, "Exploiting Weak Connectivity in Mobile File Access," *Proc. 15th ACM Symp. Operating Systems Principles*, pp. 143-155, Copper Mountain, Colo., Dec. 1995.
[12] L.B. Mummert and M. Satyanarayanan, "Long Term Distributed File Reference Tracing: Implementation and Experience," *Software Practice and Experience*, vol. 26, no. 6, pp. 705-736, June 1996.
[13] B. Noble and M. Satyanarayanan, "An Empirical Study of a Highly Available File System," *Proc. 1994 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 138-149, Nashville, Tenn., May 1994.
[14] E. Pitoura and B. Bhargava, "Maintaining Consistency of Data in Mobile Distributed Environments," *Proc. 15th Int'l Conf. Distributed Computing Systems*, pp. 404-413, May 1995.
[15] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 447-459, Apr. 1990.
[16] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebling, P. Kumar, and Q. Lu, "Experience with Disconnected Operation in a Mobile Environment," *Proc. USENIX Symp. Mobile Location-Independent Computing*, pp. 11-28, Cambridge, Mass., Aug. 1993.
[17] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler, "Lightweight Recoverable Virtual Memory," *ACM Trans. Computer Systems*, vol. 12, no. 1, pp. 33-57, Feb. 1994. Corrigendum: vol. 12, no. 2, May 1994.
[18] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, "Session Guarantees for Weakly Consistent Replicated Data," *Proc. Third Int'l Conf. Parallel and Distributed Information Systems*, pp. 140-149, Sept. 1994.
[19] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing Update Conflicts in a Weakly Connected Replicated Storage System," *Proc. 15th ACM Symp. Operating Systems Principles*, pp. 172-183, Copper Mountain, Colo., Dec. 1995.

Qi Lu received the BS and MS degrees in computer science from Fudan University, Shanghai, China, in 1984 and 1987, respectively. He received the PhD degree in computer science from Carnegie Mellon University in 1996. He is currently a research staff member at IBM Almaden Research Center.

Dr. Lu's research interests include information systems for the internet and intranets, mobile computing systems, distributed file systems, advanced transaction systems, operating systems, object-oriented database systems, software development tools, and environments.

Mahadev Satyanarayanan received the Bachelor's and Master's degrees from the Indian Institute of Technology, Madras, India. He received the PhD degree in computer science from Carnegie Mellon University. Prof. Satyanarayanan is currently a professor of computer science at Carnegie Mellon University. He has been a consultant and advisor to many industrial and governmental organizations.

Prof. Satyanarayanan is an experimental computer scientist who has pioneered research on mobile information access. An outcome of this work is the Coda File System, which provides application-transparent support for disconnected and weakly-connected operation. A complementary approach, application-aware adaptation, is being explored in the context of Odyssey, a platform for mobile computing. Prior to his work on Coda and Odyssey, Prof. Satyanarayanan was a principal architect and implementor of the Andrew File System, a location-transparent distributed Unix file system that addressed issues of scale and security. Later versions of this system have been commercialized and incorporated into the Open Software Foundation's DCE offering.