Speculative Execution in a Distributed File System

EDMUND B. NIGHTINGALE, PETER M. CHEN, and JASON FLINN
University of Michigan

Speculator provides Linux kernel support for speculative execution. It allows multiple processes to share speculative state by tracking causal dependencies propagated through interprocess communication. It guarantees correct execution by preventing speculative processes from externalizing output, for example, sending a network message or writing to the screen, until the speculations on which that output depends have proven to be correct. Speculator improves the performance of distributed file systems by masking I/O latency and increasing I/O throughput. Rather than block during a remote operation, a file system predicts the operation's result, then uses Speculator to checkpoint the state of the calling process and speculatively continue its execution based on the predicted result. If the prediction is correct, the checkpoint is discarded; if it is incorrect, the calling process is restored to the checkpoint, and the operation is retried. We have modified the client, server, and network protocol of two distributed file systems to use Speculator. For PostMark and Andrew-style benchmarks, speculative execution results in a factor of 2 performance improvement for NFS over local area networks and an order of magnitude improvement over wide area networks. For the same benchmarks, Speculator enables the Blue File System to provide the consistency of single-copy file semantics and the safety of synchronous I/O, yet still outperform current distributed file systems with weaker consistency and safety.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

General Terms: Performance, Design

Additional Key Words and Phrases: Distributed file systems, speculative execution, causality

1. INTRODUCTION

Distributed file systems often perform substantially worse than local file systems because they perform synchronous I/O operations for cache coherence and

The work is supported by the National Science Foundation under award CNS-0509093. NSF has also supported development of the Blue File System through award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. Intel Corp and Motorola Corp have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

Authors' address: Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122; email: enightin@eecs.umich.edu; {pmchen,jflinn}@umich.edu.

data safety. File systems such as AFS [Howard et al. 1988], and NFS [Callaghan et al. 1995] present users with the abstraction of a single, coherent namespace shared across multiple clients. Although caching data on local clients improves performance, many file operations still use synchronous message exchanges between client and server to maintain cache consistency and protect against client or server failure. Even over a local area network, the performance impact of this communication is substantial. As latency increases due to physical distance, middleboxes, and routing delays, the performance cost may become prohibitive.

Many distributed file systems weaken consistency and safety to improve performance. Whereas local file systems typically guarantee that a process that reads data from a file will see all modifications previously completed by other processes; distributed file systems such as AFS and NFS provide no such guarantee. For example, most NFS implementations provide *close-to-open* consistency, which guarantees only that a client that opens a file will see modifications made by other clients that have previously closed the file. Weaker consistency semantics improve performance by reducing the number of synchronous messages that are exchanged. Nevertheless, as our results show, even these weaker semantics are time consuming.

We demonstrate that, with operating system support for lightweight checkpointing, speculative execution, and tracking of causal interdependencies between processes, distributed file systems can be fast, safe, and consistent. Rather than block a process while waiting for the result of a remote communication with a file server, the operating system checkpoints its state, predicts the result of the communication, and continues to execute the process speculatively. If the prediction is correct, the checkpoint is discarded; if it is false, the application is rolled back to the checkpoint.

Our solution relies on three observations. First, file system clients can correctly predict the result of many operations. For instance, consistency checks seldom fail since concurrent file updates are rare. Second, the time to take a lightweight checkpoint is often much less than network round trip time to the server, so substantial work can be done while waiting for a remote request to complete. Finally, modern computers often have spare resources that can be used to execute processes speculatively. Encouraged by these observations, and by the many prior successful applications of speculation in processor design, we have added support for speculative execution, which we call Speculator, to the Linux kernel.

In our work, the distributed file system controls when speculations start, succeed, and fail. Speculator provides a mechanism for correct execution of speculative code. It does not allow a process that is executing speculatively to externalize output, for example, make network transmissions or display output to the screen, until the speculations on which that output depends prove to be correct. If a speculative process tries to execute a potentially unrecoverable operation, for example, it calls the `reboot` system call, it is blocked until its speculations are resolved. Speculator tracks causal dependencies between kernel objects in order to share speculative state among multiple processes. For instance, if a speculative process sends a signal to its nonspeculative parent, Speculator checkpoints the parent and marks it as speculative before it delivers the signal. If a speculation on which the child depends fails, both the child and parent are restored to their checkpoints (since the parent might not receive the signal on the correct execution path). Speculator tracks dependencies passed through fork, exit, signals, pipes, fifos, Unix sockets, and files in local and distributed file systems. All other forms of IPC currently block the speculative process until the speculations on which it depends are resolved.

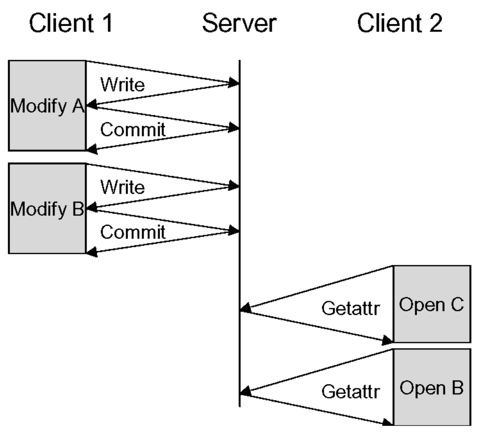
Since speculation is implemented entirely in the operating system, no application modification is required. Speculative state is never externally visible. In other words, the semantics of the speculative version of a file system are identical to the semantics of the nonspeculative version; however, the performance of the speculative version is better.

Results from PostMark and Andrew-style benchmarks show that Speculator improves the performance of NFS by more than a factor of 2 over local area networks; over networks with 30 ms of round-trip latency, speculation makes NFS more than 14 times faster. We have also created a version of the Blue File System [Nightingale and Flinn 2004], that uses speculator to provide *single-copy* semantics, in which the file consistency seen by two processes sharing a file and running on two different file clients is identical to the consistency that they would see if they were running on the same client. In addition, our version of BlueFS provides *synchronous I/O* in which all file modifications are safe on the server's disk before an operation is observed to complete. Despite providing these strong guarantees, BlueFS is 66% faster than nonspeculative NFS over a LAN and more than 11 times faster with a 30 ms delay.

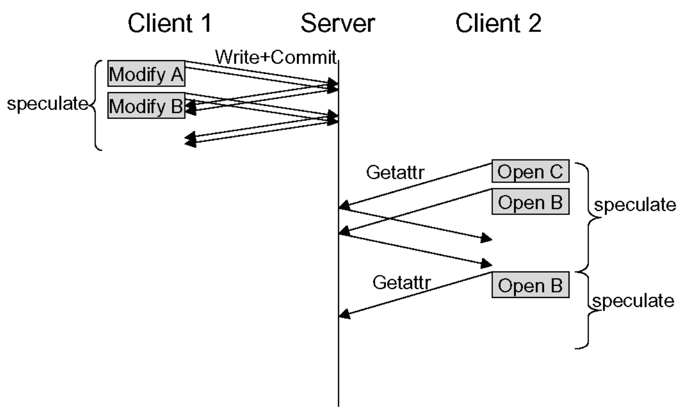
2. MOTIVATION: SPECULATION IN NFS

Figure 1 illustrates how Speculator improves distributed file system performance. Two NFS version 3 clients collaborate on a shared project that consists of three files: A, B, and C. At the start of the scenario, each client has up-to-date copies of all files cached. Client 1 modifies A and B; client 2 then opens C and B. Client 2 should see the modified version of B since that file was closed by client 1 before it was opened by client 2.

When an application closes a file, the Linux 2.4.21 NFSv3 client first sends asynchronous `write` remote procedure calls (RPCs) to the server to write back any data for that file that is dirty in its file cache—these RPCs are necessary to provide close-to-open consistency. After receiving replies for all `write` RPCs, the client sends a synchronous `commit` RPC to the server. The server replies only after it has committed all modifications for that file to disk. The NFS client returns from the `close` system call after receiving the `commit` reply. The `commit` RPC provides a safety guarantee, namely that no file modifications will be lost due to a server crash after the file has been closed. Thus, a Linux application that modifies a file in NFS incurs a performance penalty on the close of at least two network round-trips, and one synchronous disk access. Some other operating systems have NFS clients that do not wait for a `commit` reply before returning from `close`—these clients sacrifice safety, but improve performance since they block only until replies for all outstanding `write` RPCs have been received.



(a) Unmodified NFS



(b) Speculative NFS

Fig. 1. Example of speculative execution for NFS.

When an NFS client opens a file that it has previously cached, it issues a `getattr` RPC to the server. The file attributes returned by the server indicate whether the file has been modified since it was cached (in which case the cached copy is discarded and a new copy is fetched). Since the NFS server is a single point of synchronization, the `getattr` RPC guarantees that the cached copy is fresh; if another client had modified and closed the file, the returned attributes would show the modification. For instance, in Figure 1(a), when client 2 reads file B, the attributes returned by `getattr` indicate that the file was modified. Hence, client 2 discards its cached copy of file B.

Cache coherence in NFS is time consuming because a process blocks each time a file is closed after being modified or opened, as well as on directory lookups, permission checks, and modifications. This cost is magnified many times during activities such as listing a directory or compiling a program, because each activity invokes several file system operations; for instance, most applications that show directory listings fetch the attributes of all files within the directory to display file types, sizes, or other metadata.

Our speculative version of NFS is shown in Figure 1(b). Client 1 asynchronously executes `write` and `commit` RPCs, speculating that all modifications will succeed at the server. Client 2 asynchronously executes the `getattr` RPCs, speculating that its cached copy of C and B are up-to-date. When the latter speculation fails, the calling process is rolled back to the start of the system call that opened B. This system call is re-executed and a new speculation begins.

Speculation improves file system performance because it hides latency: multiple file system operations can be performed concurrently, and computation can be overlapped with I/O. Speculation also improves write throughput. Because speculation transforms sequential operations issued by a single thread of control into concurrent operations, it allows the server to group commit such operations. Without OS support for speculative execution, the system call interface prevents these optimizations. The file system cannot return to the application until it receives the results of a remote operation, since that operation might fail.

3. CONDITIONS FOR SUCCESS

We believe that adding support for speculative execution to a commodity OS kernel will eventually benefit many applications. However, we have targeted distributed file systems as the first clients of Speculator because they exhibit three ideal characteristics:

(1) **The results of speculative operations are highly predictable**. File system clients cache data in memory and on disk to improve performance. Due to ever increasing memory and disk capacities, today's workstations can cache data for long periods of time. In the absence of concurrent modifications, cached data remains valid and can be used to successfully predict the outcome of remote operations. Speculation is a form of optimistic concurrency, since a client detects a conflicting update by another client when it next accesses the modified file. Since concurrent updates are rare in distributed file systems [Howard et al. 1988], we expect the vast majority of speculations to succeed. However, our results show that even when many speculations fail, speculative file systems still substantially outperform nonspeculative ones.

(2) **Checkpointing is often faster than remote I/O**. Speculator checkpoints are essentially copy-on-write forks where the forked child is typically never executed. The time to take and discard a checkpoint of a small process is 52 μs; considerably less than the cost of a disk or network I/O. Although this time is greater for larger processes (6.3 ms for a 64 MB process), checkpoint cost can be amortized across several speculations by having those speculations share a single process checkpoint. Thus, there is considerable time available to execute applications speculatively when they would normally block on file system operations.

```
create_speculation   (OUT spec_id, OUT dependencies);
commit_speculation   (IN spec_id);
fail_speculation     (IN spec_id);
```

Fig. 2. Speculator interface.

(3) **Modern computers often have spare resources**. Speculative execution requires CPU cycles, and checkpoint storage requires memory. Fortunately, modern workstations typically have these resources in abundance. Since I/O is increasingly the performance bottleneck in modern computers [Rosenblum et al. 1995], we can improve application performance by using these spare resources to hide I/O latency and improve I/O throughput.

4. AN INTERFACE FOR SPECULATION

Our design for speculative execution exhibits a separation of concerns between policy and mechanism. The distributed file system determines when speculations begin, succeed, and fail. Speculator provides a lightweight checkpoint and rollback mechanism that allows speculative process execution. Speculator ensures that speculative state is never externalized or directly observed by nonspeculative processes.

Speculator is implemented as part of the core Linux 2.4.21 kernel and consists of roughly 7,500 lines of C source code. Figure 2 shows Speculator's interface. A process must be executing in kernel mode (e.g., within a system call) to use this interface. To initiate speculative execution, a process calls `create_speculation`. This function returns a *spec_id* that uniquely identifies the particular speculation and a list of prior speculations on which the new speculation depends. Any process may later declare whether that speculation succeeds or fails by calling `commit_speculation` or `fail_speculation` with that spec_id. This design enables Speculator to remain ignorant of the particular hypothesis that underlies each speculation, as well as the semantics for success and failure. In turn, a Speculator client, for example, a distributed file system, need not concern itself with the details of how speculative execution is performed.

The next section describes our basic implementation of speculative execution in the Linux kernel, which allows processes to execute speculatively in isolation. Section 6 extends this implementation by allowing multiple processes to share speculative state. Section 7 describes how distributed file systems use Speculator.

5. IMPLEMENTING SPECULATION

5.1 Process Checkpoint and Rollback

Speculator implements checkpointing by performing a copy-on-write fork of the currently running process. It also saves the state of any open file descriptors and copies any signals pending for the checkpointed process. In contrast to a normal fork, the child process is not placed on the run queue—it is simply a vessel for storing state. If all speculations on which a checkpoint depends prove to be correct, Speculator discards the checkpoint by reclaiming the kernel data structures associated with the child process.

If one of the speculations on which a checkpoint depends fails, Speculator restores the process to the state captured during the checkpoint. The process that is currently executing speculatively, called the *failed process*, is marked for termination by setting a flag in its task structure. When the failed process is next scheduled, Speculator forces it to exit. Speculator ensures that the failed process performs no externally visible operations prior to termination.

The process that was forked during the checkpoint, called the *checkpoint process*, assumes the identity of the failed process. Speculator gives the checkpoint process the process identifier, thread group identifier, and other distinguishing characteristics of the failed process. It also changes its file descriptors and pending signals to match the values saved during the checkpoint. The program counter of the checkpoint process is set to the system call entry point, and its kernel stack pointer is set to the initial kernel stack frame. Thus, after Speculator places the checkpoint process on the run queue, the process re-executes the system call that was being executed when the checkpoint was taken. Since the checkpoint process steals the identity of the failed process, this manipulation is hidden from observers outside the kernel; to the user, it appears that the speculative execution never happened.

5.2 Speculation

Speculator adds two new data structures to the kernel to track speculative state. A *speculation* structure is created during `create_speculation` to track the set of kernel objects that depend on the new speculation. An *undo log* is associated with each kernel object that has speculative state—this log is an ordered list of speculative operations that have modified the object. Each entry in the log contains sufficient information to undo the modifying operation, as well as references to all new speculative dependencies that were introduced by the operation. The presence of an entry in an object's undo log indicates that the object will be rolled back to the state associated with that entry if any of the referenced speculations fail. We say that a kernel object *depends* on all speculations for which references exist in its undo log. The speculations on which an object depends are *resolved* if all speculations commit (in which case, the object becomes nonspeculative) or if any speculation fails (in which case, the object is rolled back to a prior state).

When a previously nonspeculative process calls `create_speculation`, Speculator creates a new speculation structure and an undo log for the calling process. It checkpoints the calling process as described in the previous section and inserts an entry containing the checkpoint into its undo log. The new entry and the new speculation structure refer to each other. If the speculation fails, the checkpoint is used to restore the process to its prior state. If the process calls `create_speculation` again, Speculator creates a new speculation structure and appends a new entry to the process undo log. If the previous speculation was caused by a read-only operation such as a `stat` on a file in the distributed file system, no new checkpoint is needed. In this case, the new undo log entry shares a reference to the checkpoint contained in the previous log entry. If either speculation fails, the process is rolled back to the common checkpoint. Allowing speculations to share checkpoints improves performance in the common case where speculations succeed. Of course, in the uncommon case where the second speculation fails, the application must re-execute more work than if separate checkpoints had been taken.

Speculator caps the amount of work unnecessarily re-executed by taking a new checkpoint if the prior checkpoint for the process is more than 500 ms old. Speculator also caps the number of outstanding speculations at 2000 to prevent speculation from consuming too many system resources. As we gain more experience with the system, it may also prove useful to limit specific resources such as the amount of physical memory used for speculation.

Currently, two operations do not share a common checkpoint if the first operation modifies state. For example, if the first operation is a `mkdir` in the distributed file system, a common checkpoint cannot be used since the file server might make the effects of the `mkdir` visible to other clients, then fail the second operation. When the second operation fails, the client must roll back to the common checkpoint. Any clients that view the new directory would see incorrect state if the process that performed the `mkdir` does not recreate the directory when it re-executes. If necessary, Speculator could allow two mutating operations to share a checkpoint by modifying the file server to atomically perform all operations that share a checkpoint. However, this further optimization requires substantial server modifications, and our performance results indicate it is not needed.

5.3 Ensuring Correct Speculative Execution

We define the speculative execution of a process to be *correct* if two invariants hold. First, speculative state should never be visible to the user or any external device. Enforcing this invariant requires that Speculator prevent a speculative process from externalizing output to the screen, network, or other interfaces. Second, a process should never view speculative state unless it is already speculatively dependent upon that state (because it could produce output that depends on that state). If a nonspeculative process tries to view speculative state, Speculator either blocks the process until the state becomes nonspeculative, or it makes the process speculative and rolls it back if a speculation on which that state depends fails.

Since interactions between a process and its external environment pass through the operating system, Speculator can prevent a speculative process from performing potentially incorrect operations by blocking that process until the speculations on which it depends are resolved. If all of those speculations prove successful, the process is unblocked and allowed to execute the operation (which is correct since the process is no longer speculative). If a speculation fails, the process is terminated.

We observe that blocking a speculative process is always correct, but that blocking limits the amount of work that can be done while waiting for remote operations to complete. Our approach to developing Speculator was to first create a correct but slow implementation that blocked whenever a speculative process performed a system call. We created a new system call jump table and modified the Linux system call entry point to use this table if the task structure of the currently executing process is marked as speculative. Initially, we set all entries in this jump table to `sys_spec_deny`, a function we created to block the calling process until its speculations are resolved.

Next, we observed that system calls that do not modify state (e.g., `getpid`) are correct if performed by a speculative process. We let speculative processes perform these calls by replacing `sys_spec_deny` entries with the addresses of the functions that implement these syscalls. We also found that several system calls modify only state that is private to the calling process; these calls are correct to perform while speculative since their effects are not observed by other processes and, on speculation failure, their effects are undone as a side effect of restoring the checkpoint process. For example, `dup2` creates a new file descriptor that is a copy of an existing descriptor. This state is private to a process since file descriptors are not shared (except on fork, which is handled in Section 6.8). Further, when Speculator restores the checkpoint process, the effect of `dup2` is undone since the restored checkpoint contains the descriptor state of the failed process prior to calling `dup2`.

We next allowed speculative processes to perform operations on files in speculative file systems. For these system calls, it is insufficient to replace `sys_spec_deny` with syscalls such as `mkdir` in the speculative jump table because the OS may mount some file systems that support speculative execution and some that do not. For instance, one might use a nonspeculative version of ext3 along with a speculative version of NFS—in this case, it is correct to allow speculative processes to modify state in NFS but not in ext3.

When a speculative process performs a file system operation, Speculator inspects the file system type to determine whether to block the calling process or allow speculative execution. On mount, a file system may set a flag in its superblock that indicates that speculative processes are allowed to read and write the files and directories it contains. Another flag allows just speculative read-only operations. All file system syscalls check these flags if the current process is speculative, to decide whether to block or permit the operation. For example, `mkdir` blocks speculative processes unless the superblock of the parent directory indicates that mutating operations are allowed, and `stat` blocks unless read-only operations are allowed. Our speculative versions of NFS and BlueFS set the read/write flag. Because file systems that we have not modified do not set either flag, operations in those file systems block speculative processes until their speculations are resolved.

Speculator uses a similar strategy for the `read` and `write` system calls. When a file descriptor is opened, the type-specific VFS `open` function can set flags in the Linux file structure to indicate that speculative reads and/or writes are permitted. If a speculative process tries to read from, or write to, a file descriptor for which the appropriate flag is not set, that process is blocked until its speculations are resolved. The `read` flag is needed because read is a mutating operation for some inode types—for instance, reading from a Unix socket consumes the data that is read. For this inode type, speculative reads are incorrect. For other inode types, such as files in local file systems that do not update access times, reads are correct since they do not change object state.

If a speculative process writes to a tty or other external device, its action is incorrect since it is externalizing output. Yet, blocking such writes greatly limits the amount of work that can be done while speculative. This led us to support a third behavior for `write`: the data being written can be buffered in the kernel until the speculations on which it depends have been resolved. Speculator first validates such output operations to ensure that they can be performed. It then stores the output in a queue associated with the last checkpoint of the current process. After all speculations associated with that checkpoint commit, the buffered output is delivered to the device for which it was destined. If a speculation fails, the output is discarded. Currently, Speculator uses this strategy for output to the screen and network.

Output from a speculative process can appear before all of the speculations for that process are resolved. Consider a process that speculates on a remote operation, outputs a message, and then performs another speculative operation. Since the output depends only on the first speculation, Speculator can deliver it to the output device once the first speculation succeeds—it need not wait for the second speculation to succeed or fail. In a nonspeculative system, the output would be delayed while the process blocked on the remote I/O operation. Thus, the condition on which the output awaits, the completion of the remote I/O, is the same in nonspeculative and speculative systems. In fact, output in the speculative system often appears *faster* than in the nonspeculative system since the remote operations on which that output waits complete faster. An exception occurs if two speculations share a checkpoint since output that depends on only one speculation must wait for both to complete—this is another reason Speculator limits the maximum time difference between speculations that depend on the same checkpoint.

6. MULTIPROCESS SPECULATION

We next allowed speculative processes to participate in interprocess communication. This significantly extends the amount of speculative work done by applications composed of multiple cooperating processes. For example, `make` forks children to perform compilation and linking; these processes communicate via pipes, signals, and files. If we limit speculation to a single process, `make` would often block waiting for a signal to be delivered or for data to arrive on a pipe.

6.1 Generic Strategy

Speculator's strategy for IPC allows selected data structures within the kernel to have speculative state. Figure 3 illustrates how speculative state is propagated. In Figure 3(a), processes 8000 and 8001 both `stat` different files in BlueFS—BlueFS calls `create_speculation`, sends an asynchronous RPC to check cache consistency, and continues execution assuming that the cached attributes are up-to-date. Speculations 1 and 2 track the state associated with each speculation. Each process is marked as speculative, associated with an

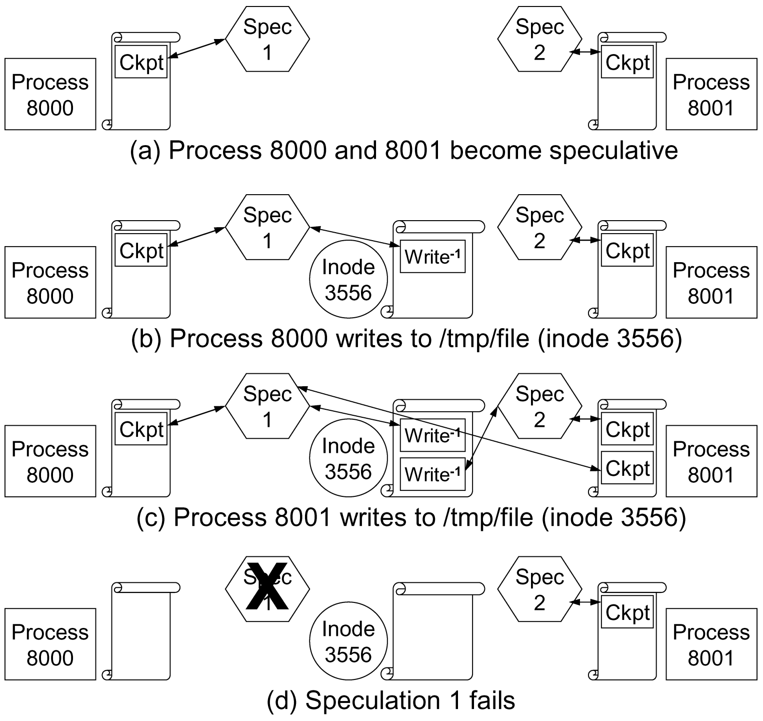

Spec 1
Spec 2
Ckpt
Process 8000
Process 8001
(a) Process 8000 and 8001 become speculative
Inode 3556
Write-1
(b) Process 8000 writes to /tmp/file (inode 3556)
(c) Process 8001 writes to /tmp/file (inode 3556)
(d) Speculation 1 fails

Fig. 3. Propagating causal dependencies.

undo log, and checkpointed. Each undo log entry contains the process checkpoint and a reference to the speculation on which the process depends.

In Figure 3(b), process 8000 writes data to a file in /tmp. This causes Speculator to *propagate dependencies* from process 8000 to inode 3556. After this operation, the file contains speculative state; for example, if speculation 1 fails, process 8000 may write different data to the file on re-execution. Speculator therefore creates an undo log for inode 3556 and marks it as speculative. The entry in the file's undo log describes how to restore it to its previous state; in this case, the entry describes how to reverse the write operation that modified the file. The association between speculation 1 and this entry indicates that the write operation will be undone if speculation 1 fails.

In general, Speculator propagates dependencies from a process P to an object X whenever P modifies X and P depends on speculations that X does not. The entry in X's undo log for that operation is associated with all speculations on which P depended but X did not.

In Figure 3(c), process 8001 writes data to the same file. This operation creates another entry in the inode's undo log—this entry is associated with speculation 2. This operation also creates an entry in the process 8001 undo log since the process may have observed speculative state as a result of writing to the file. For example, if a speculative operation changes file permissions, the return value of a later write might depend on whether that speculation succeeds.

In general, Speculator propagates dependencies from an object X to a process P whenever P observes X's state and X depends on speculations that P does not. The entry in P's undo log for that operation is associated with all speculations on which X depended but P did not.

In our experience, almost all operations that modify kernel objects also observe the object being modified (in fact, the only exception we have seen is a signal sent by an exiting process). Thus, mutating operations normally propagate dependencies bi-directionally between the mutating process and the mutated object.

Process undo entries are not needed for many operations. For instance, an undo entry is not created for process 8000 when it modifies the file because the file depended on no speculations. Similarly, if process 8001 modifies the file again after step (c), an entry is not put in its undo log since the file and process 8001 depend on the same set of speculations.

During `commit_speculation`, Speculator deletes the speculation structure and removes its association with any undo log entries. Entries at the front of an undo log are deleted once they depend on no more speculations. When an undo log has no more entries, it is deleted, and its associated object becomes nonspeculative.

Figure 3(d) shows what happens when a speculation fails. The failed speculation, speculation 1, is deleted. Each kernel object that depends on that speculation is rolled back to the state captured by the undo entry with which the failed speculation is associated. In this example, process 8000 is restored to its checkpoint and will retry the failed operation. Inode 3556 is restored to the state that existed before it was modified by process 8000—this is done by applying the two inverse operations in its undo log. Process 8001 is rolled back to its second checkpoint (because it could have observed incorrect speculative state in inode 3556). When process 8001 is restarted, it will attempt to write to inode 3556 again. This rollback is performed atomically during `fail_speculation`.

The undo log, undo entries, and speculations are generic kernel data structures. However, each undo log entry contains pointers to type-specific state and functions that implement type-specific rollback and roll forward processing. This design allows a common implementation for the type-independent logic associated with propagating dependencies, rolling state forward, and rolling state back. We next describe the type-specific logic for each form of IPC that Speculator currently supports.

6.2 Objects in a Distributed File System

In our design, the file server always knows the correct state of each object (file, directory, etc.). If a speculation on which an object depends fails, Speculator simply invalidates the cached copy. When the object is next accessed, its correct state is retrieved from the server. Thus, undo log entries do not contain type-specific data. However, the undo log must still track each object's speculative dependencies so as to propagate those dependencies when one process reads a locally-cached object previously modified by another speculative process.

6.3 Objects in a Local Memory File System

We have modified the RAMFS memory-only file system to support speculative state. When a VFS operation modifies a speculative RAMFS object, Speculator inserts an entry that describes the inverse operation needed during rollback in the inode's undo log. For instance, an `rmdir` entry contains a reference to the directory being deleted as well as the name of the directory. The reference prevents the OS from reclaiming the directory object until the speculation is resolved. Thus, if a speculation on which the `rmdir` depends fails, Speculator can reinsert the directory into its original parent.

6.4 Objects in a Local Disk File System

We also modified the ext3 file system to support speculation. While Speculator uses the same strategy for managing ext3 inode undo logs that it uses for RAMFS, the presence of persistent state in ext3 presents many challenges.

One challenge is dealing with the possibility of system crashes. For instance, it would be incorrect to write speculative state to disk while maintaining undo information only in memory, since uncommitted speculative state would be visible after a crash. While Speculator could potentially write undo logs to disk to deal with this possibility, we felt a simpler solution is to never write speculative data to disk (i.e., use a *no-steal* policy [Haerder and Reuter 1983]). Speculator attaches undo logs to kernel buffer head objects. When a speculative process modifies a buffer, Speculator marks the buffer as speculative and inserts a record in its undo log. Kernel daemons that asynchronously write buffers to disk, skip any buffer marked as speculative. A process that explicitly forces a speculative buffer to disk, for example, by calling `fdatasync`, is blocked until the buffer's speculations are resolved. A substantial advantage of using a no-steal policy is that Speculator often does not need to shadow data overwritten by a speculative operation in memory—if the speculation fails, it invalidates the buffer, which will cause the prior state to be re-read from disk.

A second challenge is presented by shared on-disk metadata structures such as the superblock and allocation bitmaps. Ext3 operations often read and modify these structures, potentially creating too many speculative dependencies. For instance, when a speculative process allocates a new disk block for a file, it modifies the count of free blocks in the superblock. Under our previously described policy, any nonspeculative process performing an ext3 operation would become speculative since it would observe speculative state in the superblock.

We felt that this policy was too aggressive as these metadata objects are seldom observed outside the kernel. Even in the case of inode numbers, which are externally visible, the impact of observing speculative state is limited (e.g., a new file might receive a different inode number than it would have received in the absence of speculation, yet the particular number received is typically unimportant as long as that number is unique). Speculator therefore allows processes to observe speculative metadata in ext3 superblocks, bitmaps, and group descriptors, without propagating causal dependencies.

Speculator must allow these ext3 metadata structures to be occasionally written to disk. For instance, the ext3 superblock could remain speculative indefinitely if it is continuously modified by speculative processes. In this case, writing the superblock to disk is incorrect since it contains speculative data, yet if it is not written out periodically, substantial data could be lost after a crash. Speculator addresses this issue by maintaining shadow buffers for the superblock, bitmaps, and group descriptors. The actual buffer contains the current nonspeculative state, while the shadow buffer contains the speculative state. When a speculative operation modifies one of these structures, it supplies redo and undo functions for the modification. The redo function is initially used to modify the speculative state. If the speculative operation commits, the redo function is applied to the nonspeculative state. If the operation rolls back, the undo function is applied to the speculative state. Only the actual buffer containing nonspeculative state is written to disk. This design relies on metadata operations being commutative (which is the case in Speculator).

A final challenge is presented by the ext3 journal, which groups multiple file system operations into compound transactions. A compound transaction may contain both speculative and nonspeculative operations. To preserve the invariant that no speculative state is written to disk, we initially decided to block the commit of a compound transaction until all of its operations became nonspeculative. However, this proved too restrictive since there are many instances in the ext3 code that wait for the current transaction to commit. We therefore adopted an approach that leverages Speculator's tracking of causal dependencies. Any file system operation that is causally dependent upon a speculative operation must itself be speculative. Thus, any nonspeculative operation within a compound transaction cannot depend on a speculative operation within that transaction. Prior to committing a compound transaction, Speculator moves any speculative buffers in that transaction to the next compound transaction. For shared metadata buffers such as the superblock, Speculator commits the nonspeculative version of the buffer as part of the current transaction. This approach lets the journal immediately commit nonspeculative operations without writing speculative data to disk.

6.5 Pipes and Fifos

Speculator handles pipes and fifos with a strategy similar to that used for local file systems. Since the `read` and `write` system calls both modify and observe pipe state, they propagate dependencies bi-directionally between the pipe and the calling process. The undo entry for `read` saves the data read so that it can be restored to the pipe if a speculation fails. The `write` entry saves the amount and location of data written so that it can be deleted on failure. Undo entries are also created when a pipe is created or deleted—the latter entry prevents the pipe from being destroyed until its speculations are resolved.

6.6 Unix Sockets

Speculator tracks speculative dependencies propagated through Unix sockets by allowing kernel socket structures to contain speculative state. When data is written to a Unix socket, it is placed in a queue associated with the destination socket. Thus, any speculative dependencies are propagated bi-directionally between the sending process and the destination socket. Additionally, since the process sending data observes the state of the source socket, any speculative dependencies associated with that socket are propagated to both the sending process and destination socket. When a process reads data from the destination socket, dependencies are propagated bi-directionally between the process and the socket.

6.7 Signals

Signals proved difficult to handle. Our initial design had the sending process propagate dependencies directly to the process that receives the signal. However, this design requires Speculator to be able to checkpoint the receiving process at arbitrary points in its execution. The receiving process could receive a signal while it is in the middle of a system call. If the process has already performed some nonidempotent operation during the system call, restarting from the beginning of the system call is incorrect. If the checkpoint process is instead restarted from the location at which the signal was received, it would not have any kernel resources such as locks that were previously acquired during the system call by the process that received the signal.

Speculator solves this problem by requiring that all checkpoints be taken by the process being checkpointed. Checkpoints are only taken at well-known code locations where the checkpoint process will behave correctly if a rollback occurs.

Speculator delivers signals using a two-step process. First, when the signal is sent, it creates an undo log for that signal—the log's only entry depends on the same speculations as the sending process. The signal is put in a *speculative sigqueue* associated with the receiving process. Signals in this queue are not yet considered delivered; thus, these signals are not visible to the receiving process when it checks its pending signals. During this step, Speculator propagates dependencies from the receiving process to the sending process, since the sender observes that it is *allowed* to send a signal to the destination process.

The second step occurs immediately before a process returns from kernel mode. At this time, Linux delivers any pending signals. We modified this code to first move any signals from the speculative sigqueue to the list of pending signals. If necessary, the receiving process is checkpointed and dependencies are propagated from the signals being delivered to the process. A flag is set in the checkpoint so that the checkpoint process will exit the kernel instead of restarting the system call if a rollback occurs. This preserves the invariant that checkpoints are only taken by the checkpointed process.

When a signal is moved from the speculative sigqueue to the list of pending signals, a new entry is inserted into the signal undo log. This ensures that if the receiving process rolls back to a point before the signal was delivered, that signal will be redelivered. On the other hand, if a speculation on which the signal depends fails, the signal is destroyed and the receiving process rolls back to the checkpoint taken before delivery.

If a signal becomes nonspeculative while it waits in the speculative sigqueue (because all speculations on which it depends commit), it is immediately delivered to the receiving process. This ensures that a process will eventually proceed if it blocks in the kernel waiting for a speculative signal to be delivered. However, to maximize performance, such signals should be delivered immediately. Speculator therefore interrupts `wait` and `select` when a signal arrives in the speculative sigqueue (just as they are interrupted for a pending signal). In these instances, we have ensured that these system calls can be restarted correctly if a speculation on which the delivered signal depends fails.

6.8 Fork

When a process forks a child, that child inherits all dependencies of its parent. If one of these speculations fails, the forked process is simply destroyed since it may never exist on the correct execution path.

6.9 Exit

When a speculative process exits, its `pid` is not deallocated until all of its dependencies are resolved. This ensures that checkpoints can be restarted with that `pid` if a speculation fails. The rest of the Linux exit code is executed, including the sending of signals to the parent of the exiting process. Speculating through exit is important for supporting applications such as `make` that fork children to perform subtasks. Without this support, the parent process would block until its child's speculations are resolved.

6.10 Other Forms of IPC

Speculator does not currently let speculative processes communicate via System V IPC, futexes, or shared memory. It does allow speculative execution of processes with read-only access to shared memory segments, or write access to private segments. However, it blocks processes that have write access to one or more shared memory segments—this precludes multithreaded applications from speculating. Rather than track individual memory accesses, our planned approach to support multi-threaded processes is to consistently checkpoint all threads within a single thread group whenever a checkpoint is needed for any thread.

7. USING SPECULATION

Modifying a distributed file system to use Speculator typically involves changes to the client, server, and network protocol. We first inspect the client code and identify instances in which the client can accurately predict the outcome of remote operations. Predicting the outcome of read-only operations typically requires that the client memory or disk cache contain some information about the state of the file system. For instance, an NFS client can predict the result of a `getattr` RPC only when the inode attributes are cached. In this case, the client is speculating that the file was not modified after it was cached.

We change the synchronous RPCs in these instances to be asynchronous, and have the client create a new speculation before sending each request. When a reply to the asynchronous RPC arrives, the client calls `commit_speculation` if it correctly predicted the result of the RPC and `fail_speculation` if it did not.

Normally, if a prediction fails, the file system invalidates any cached values on which that prediction depended—this forces a nonspeculative, synchronous RPC when the operation is re-executed. However, if the reply contains updated state, the file system instead caches the new state—in this case, a new speculation based on the updated values is made when the operation is re-executed.

In the next two subsections, we address two generic issues that occur when modifying a file system to use Speculator: correctly supporting mutating operations and implementing group commit. Sections 7.3 and 7.4 describe our speculative implementations of NFS and BlueFS in more detail. Section 7.5 discusses how other file systems might benefit from using Speculator.

7.1 Handling Mutating Operations

Operations that mutate file system state at the server are challenging, since a speculative mutation could potentially be viewed by a client other than the one that made the mutation. For example, a process might read the contents of a cached directory and write them to a file (e.g., `ls /dfs/foo > /dfs/bar`). The `readdir` operation creates a speculation that the cached version of directory `foo` is up-to-date. After the process writes to and closes `bar`, the contents of this file depend on that speculation. If a process running on another file system client reads `/dfs/bar`, it views speculative state. Thus, if the original speculation made during the `readdir` fails, both clients and the server must roll back state.

Potentially, Speculator could allow multiple clients to share speculative state using a mechanism such as Time Warp's virtual time [Jefferson 1985]. Alternatively, it could allow each client to share its speculative state with the server, but not with other clients, by having the server block any clients that attempt to read or write data that has been speculatively modified by another client. However, we realized that the restricted nature of communication in a server-based distributed file system enables a much simpler solution.

The file server always knows the true state of the file system; thus, when it receives a speculative mutating operation, it can immediately evaluate whether the hypothesis that underlies that speculation is true or false. Based on this observation, we modify the server to only perform a mutation if its hypothesis is valid. If the hypothesis is false, the server fails the mutation. Thus, the server effectively controls whether a speculation succeeds or fails. A client that receives a failure response cannot commit that speculation.

In each speculative RPC, the client includes the hypothesis underlying that speculation. In the above example, a BlueFS client would send a `check_version` RPC containing the version number of its cached copy of `foo` when it executes a speculative `readdir`. The server checks this version number against the current file version and fails the speculation if the two differ. Part of the hypothesis of any mutating operation is that all prior speculations on which it depends have succeeded. In the example, the modification to `bar` depends on the success of the `readdir` speculation. These causal dependencies are precisely the set of speculations that are associated with the undo log of the process prior to transmitting the mutating operation. This list is returned by `create_speculation` and included in any speculative RPC sent to the server. If the server has previously failed any of the listed speculations, it fails the mutation.

This design requires that the server keep a per-client list of failed speculations. It also requires that the server process messages in the same order that a client generates them (otherwise the server could erroneously allow a mutation because it has not yet processed an invalid speculation on which that mutation depends). We enforce this ordering using client-generated sequence numbers for all RPCs. This sequence number also limits the amount of state that the server must keep. Each client appends the sequence number of the last reply that it received from the server to each new RPC. Once the server learns that the reply associated with a failed speculation has been processed, it discards that state (since the client will not allow any process that depends on that speculation to execute further).

A substantial advantage of this approach is simplicity: the server never stores speculative file data. Because all file modifications must pass through the server, other clients never view speculative state in the file system. Thus, when a speculation fails, only a single client must roll back state. To enable this simple design we had to modify each file system's underlying RPC protocol to support dependencies and asynchronous messages. While we have chosen to explicitly modify each file server, all server modifications (with the exception of group commit, described below) could be implemented with a server-side proxy.

Speculator reduces the size of the list sent to the server by representing speculations as <pid,spec_id> tuples and generating the low-order spec_id with a per-process counter that is incremented on each new speculation. The dependency list is encoded as a list of extents of consecutive identifiers. Typically, this produces a list size proportional to the number of processes sharing speculative state, rather than the number of speculations (it is not quite proportional because rollbacks can create gaps in the spec_id sequence).

Speculator assumes that a client communicates with only one file server at a time. In the future, we plan to block any process that attempts to write to a server while it has uncommitted speculations that depend on another server.

7.2 Speculative Group Commit

Our original intent in using speculative execution was to hide the latency associated with remote communication. However, we soon realized that speculative execution can also improve throughput. Without speculation, synchronous operations performed by a single thread of control must be done sequentially. Speculative execution allows such operations to be performed concurrently. This creates opportunities to amortize expensive activities across multiple operations. Group commit is a prime example of this.

In most distributed file systems (e.g., Coda and NFS), the file server commits synchronous mutating operations to disk before replying to clients. Commits limit throughput because each requires a synchronous disk write. A well known optimization in the presence of concurrent operations is to group commit multiple operations with a single disk write. However, opportunities for group commit in a file server are limited because each thread of control has at most one inflight synchronous mutating operation (although the Linux NFS server groups the commit of asynchronous writes to the same file). In contrast, speculative execution lets a single thread of control have many synchronous mutating operations in-flight—these can be committed with a single disk command.

We modify the file server to delay committing and replying to operations until either (a) no operations are in its network buffer, or (b) it has processed 100 operations. The server then group commits all operations and replies to clients.

7.3 Network File System

In order to explore how much speculative execution can improve the performance of existing distributed file systems, we modified NFS version 3 to use Speculator. Our modified version, called SpecNFS, preserves existing NFS semantics, including close-to-open consistency. It issues the same RPCs that the nonspeculative version issues; however, in SpecNFS, many of these RPCs are speculative and contain the additional hypothesis data described in Section 7.1.

The NFS version 3 specification requires data to be committed to stable storage before returning from a synchronous mutating RPC such as `mkdir` (the Linux 2.4.21 implementation meets this specification unless the `async` mount flag is used). Implicit in this specification is the assumption that the operation should not be observed to complete until its modifications are safe on stable storage. In SpecNFS, although a process may continue to execute speculatively while it waits for a reply, it cannot externalize any output that depends on the speculation. Thus, any operation that is observed to complete has already been committed to stable storage at the server.

The SpecNFS client speculatively executes `getattr` RPCs when it has a locally cached copy of the inode being queried. If the inode is not cached, a blocking, nonspeculative `getattr` is issued. The client also speculatively executes `lookup` and `access` RPCs if cached data is available. Mutating RPCs such as `setattr`, `create`, `commit`, and `unlink` always speculate.

One complication with the NFS protocol is that the server generates new file handles and fileids on RPCs such as `create` and `mkdir`. These values are chosen nondeterministically, making it difficult for the client to predict which values the server will choose. The SpecNFS client chooses aliases for those values that it can use until it receives the reply to its original RPC. After the client receives the reply, it discards the aliases and uses the real values. When the server receives a message that uses an alias, it replaces the alias with the actual value before performing the request. It discards an alias once it learns that the client has received the reply that contains the actual value.

The Linux NFS server does not use a write-ahead log; instead, it updates files in place. The server syncs data to disk whenever it processes `commit` RPCs and RPCs that modify directory data. Rather than sync data during the processing of these individual RPCs, the SpecNFS server syncs its file cache as a whole when no more incoming operations are in its network buffer, or when its commit limit of 100 operations has been reached. After syncing the file cache, the server replies to the RPCs sent by its clients—these clients then commit the speculations associated with those RPCs. This implementation of group commit is less efficient than one that uses a write-ahead log; however, it still improves disk throughput because the kernel orders disk writes during syncs and coalesces writes to the same disk block.

For example, consider a client that quickly modifies and closes a small file twice. The Linux NFS client generates a `write` RPC, followed by a `commit` RPC for the first close. It then generates additional `write` and `commit` RPCs for the second close. Given the Linux client implementation that waits for replies to all `write` RPCs before sending the `commit`, this activity results in four network round-trip delays. Furthermore, the client must wait twice for file data to be committed to disk—once for each `commit` RPC. In comparison, SpecNFS creates four speculations and sends all four RPCs asynchronously. The SpecNFS server delays committing data to disk and replying to the client until it processes all four messages. Committing the file data requires only a single disk write if the two writes are to the same file location.

7.4 Blue File System

We next explored whether speculative execution enables a distributed file system to provide strong consistency and safety, yet still have reasonable performance. Since we are currently developing a new distributed file system, BlueFS [Nightingale and Flinn 2004], we had the opportunity to develop a clean-sheet design that used Speculator. We exploited this opportunity to provide the consistency of single-copy file semantics and the safety of synchronous I/O.

Single-copy file semantics are equivalent to the consistency of a shared local disk: any read operation sees the effects of all preceding modifications. In contrast, the weaker close-to-open consistency of NFS guarantees only that a process that opens a file on one client will see the modifications of any client that previously closed that file. NFS makes only loose, timeout-based guarantees about data written before a file is closed, file attributes, and directory operations. This creates a window during which clients may view stale data or create inconsistent versions, leading to erroneous results.

In BlueFS, each file system object has a version number that is incremented when it is modified. All file system operations that read data from the client cache speculate and issue an asynchronous RPC that verifies that the version of the cached object matches the version at the server. All operations that modify an object verify that the cached version prior to modification matches the server's version prior to modification. In the event of a write/write or read/write conflict, the version numbers do not match—in this case, the speculation fails and the calling process is restored to a prior checkpoint.

Clearly, an implementation that performs these version checks synchronously provides single-copy semantics since all file system operations are synchronized at the server. Our speculative version (which performs version checks asynchronously) also provides single-copy semantics since no operation is observed to complete until the server's reply is received. Until some output is observed, the status of the operation is unknown (similar to the fate of Schrödinger's cat). In order for a write operation on one client to precede a read (or write) operation on another, the read must begin after the write has been observed to complete. Since that observation can only be made after the write reply has been received from the server, the incrementing of the version number at the server also precedes the beginning of the read operation. Thus, if the read operation uses stale data in its cache that does not include the modification made during the write operation, the version check fails. In this event, a rollback occurs, and the read is re-executed with the modified version.

Our second goal was to provide the safety of synchronous I/O. Before the server replies to a client RPC, it commits any modifications associated with that RPC to a write-ahead log. Since the client does not externalize any output that depends on that RPC until it receives the reply, any operation that is observed to complete is already safe on the server's disk. In NFS, this safety guarantee is provided only when a file is closed; our modified version of BlueFS provides safety for *all* file system operations, including each `write`. Since commits occur frequently, BlueFS uses group commit to improve throughput. After a group commit, the server reduces network traffic by summarizing the outcome of all operations committed on behalf of a client in a single reply message

7.5 Discussion: Other File Systems

While we have modified only NFS and BlueFS to use speculation, it is useful to consider how Speculator could benefit other distributed file systems. Since speculation improves performance by eliminating synchronous communication, the performance improvement seen by a particular file system will depend on how often it performs synchronous operations.

Both NFS and BlueFS implement cache coherence by polling the file server to verify that cached files are up-to-date. Since polling requires frequent synchronous RPCs to the server, these file systems see substantial benefit from Speculator. Other file systems use cache coherence schemes that reduce the number of synchronous RPCs performed in common usage scenarios. AFS [Howard et al. 1988] and Coda [Kistler and Satyanarayanan 1992] grant clients *callbacks*, which are promises by the server to notify the client before a file system object is modified. A client holding a callback on an object does not need to poll the server before executing a read-only operation on that object. File delegations in NFSv4 [Shepler et al. 2003] provide similar functionality. SFS [Mazières et al. 1999] also extends the NFS protocol with callbacks and leases on file attributes. Echo [Birrell et al. 1993] uses leases to provide single-copy consistency; a client granted an exclusive lease on an object can read or modify that object without contacting the server.

While the use of callbacks or leases reduces the number of synchronous RPCs, it can increase the latency of individual RPCs. Before a server can accept a file modification or grant a client exclusive access to a file, it must first synchronously revoke any callbacks or leases held by other clients. Potentially, a well-connected client must wait on one or more poorly-connected clients. Speculator would help hide the latency of these expensive operations. Thus, we expect that file systems that use leases or callbacks would see substantial benefit from Speculator, even though the relative benefit would be less than that seen by file systems that use polling.

Speculator also reduces the cost of providing safety guarantees. AFS, Coda (in its strongly-connected mode), and NFS, write file modifications synchronously to the server on close. Directory caching in these file systems is write-through. Speculator would substantially improve write performance in these file systems by hiding the latency of these synchronous operations. Since file modifications may not be written back to the server until the file is closed, data can be lost in the event of a system crash. Echo caches modifications for longer periods of time and writes back modifications asynchronously (unless a lease is revoked). This improves performance by reducing the number of synchronous RPCs, but increases the size of the window during which data can be lost due to system crashes.

To date, file system designers have had to choose whether to provide strong consistency guarantees, strong safety guarantees, or good performance. Speculative execution changes this equation by eliminating synchronous communication. As our BlueFS results in the next section demonstrate: a distributed file system using Speculator can provide the safety of synchronous I/O, as well as single-copy semantics, and still perform better than current file systems.

8. EVALUATION

Our evaluation answers two questions:

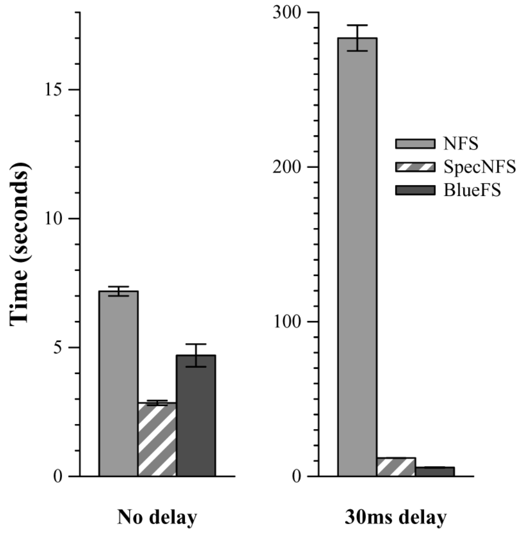
- How much does Speculator improve the performance of an existing file system (NFS)?
- With Speculator, what is the performance impact of providing single-copy file semantics and synchronous I/O (in BlueFS)?

8.1 Methodology

We use two Dell Precision 370 desktops as the client and file server. Each machine has a 3 GHz Pentium 4 processor, 2 GB DRAM, and a 160 GB disk. We run RedHat Enterprise Linux release 3 with kernel version 2.4.21. To insert delays, we route packets through a Dell Optiplex GX270 desktop running the NISTnet [Carson http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm] network emulator. All computers communicate via 100 Mb/s Ethernet switches—the measured ping time between client and server is 229 μs.

SpecNFS mounts with the -o tcp option to use TCP as the transport protocol. For comparison, we run the nonspeculative version of NFS with both UDP and TCP. Although results were roughly equivalent, we always report the best of the two results for nonspeculative NFS. While BlueFS can cache data on local disk and portable storage, it uses only the Linux file cache in these experiments—this provides a fair comparison with NFS, which uses only the file cache. The client/tmp directory is a RAMFS memory-only file system for all tests.

We ran each experiment in two configurations: one with no latency, and the other with 15 ms of latency added between client and server (for a 30 ms round-trip time). The former configuration represents the LAN environments in which current distributed file systems perform relatively well, and the latter configuration represents a wide-area link over which current distributed file systems perform poorly.



This figure shows the time to run the PostMark benchmark. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Fig. 4. PostMark file system benchmark.

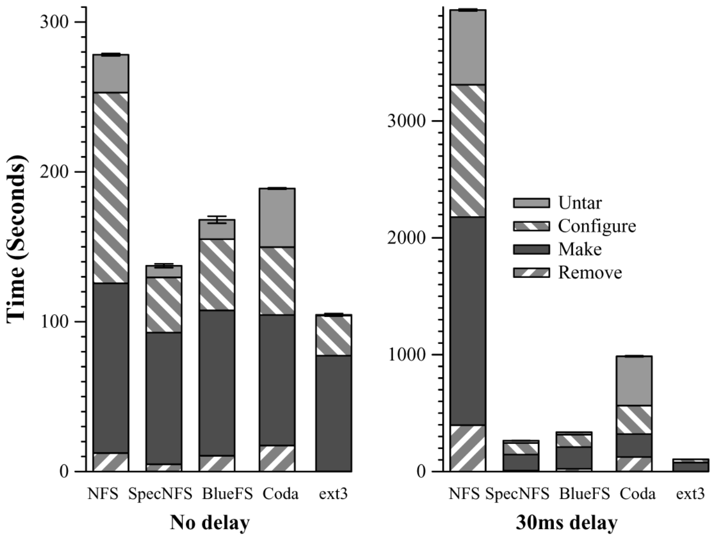
8.2 PostMark

We first ran the PostMark benchmark, which was designed to replicate the small-file workloads seen in electronic mail, netnews, and web-based commerce [Katcher 1997]. We used PostMark version 1.5, running in its default configuration, which creates 500 files, performs 500 transactions consisting of file reads, writes, creates, and deletes, and then removes all files.

The left graph in Figure 4 shows benchmark results with no additional delay inserted between the file client and server. The difference between the first two bars shows that NFS is 2.5 times faster with speculation. This speedup is a result of using speculative group commit and the ability to pipeline previously sequential file system operations. Because PostMark is a single process that performs little computation, this benchmark does not show the benefit of propagating speculative state within the OS, or the benefit of overlapping communication and computation.

The right graph in Figure 4 shows results with a 30 ms delay. The adverse impact of latency on NFS is apparent by the difference in scales between the two graphs: NFS without speculation is 41 times slower with 30 ms round-trip time than in a LAN environment. In contrast, SpecNFS is much less affected by network latency since it does not block on most remote operations. Thus, it runs the PostMark benchmark 24 times faster than NFS without speculation.

The benefits of speculative execution are even more apparent for BlueFS. BlueFS runs the PostMark benchmark 53% faster than the nonspeculative version of NFS with no delay, and BlueFS is 49 times faster with a 30 ms delay. This performance improvement is realized even though BlueFS provides



This figure shows the time to untar, configure, make and remove the Apache 2.0.28 source tree. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Fig. 5. Apache build benchmark.

single-copy file semantics and synchronous I/O. Interestingly, BlueFS outperforms the speculative version of NFS with a 30 ms delay. This is attributable to two factors: BlueFS uses write-ahead logging to achieve better disk throughput, and NFS network throughput is limited by the legacy sunrpc package. The BlueFS server achieves better disk throughput than NFS because its use of a write-ahead log reduces the number of disk seeks compared to the Linux NFS server, which updates file data in place. The sunrpc package used by NFS has some synchronous behavior that we were unable to eliminate; for example, we could only have a maximum of 160 asynchronous RPCs in flight.

8.3 Apache Build Benchmark

We next ran a benchmark in which we untar the Apache 2.0.48 source tree into a file system, run `configure` in an object directory within that file system, run `make` in the object directory, and remove all files. During the benchmark, `make` and `configure` fork many processes to perform subtasks—these processes communicate via signals, pipes, and files in the /tmp directory. Thus, propagating speculative kernel state is important during this benchmark.

In Figure 5, each bar shows the total time to perform the benchmark, and shadings within each bar show the time to perform each stage. With no delay inserted between client and server, speculative execution improves NFS performance by a factor of 2. The largest speedup comes during the `configure` stage, which is 3.4 times faster. The bar on the far right shows the time to perform the benchmark on the local ext3 file system—this gives a lower bound for performance of a distributed file system. The potential speedup for `make` is limited since computation represents a large portion of the execution of that stage. However, speculative execution still improves `make` time by 30%. With 30 ms of delay, NFS is 14 times faster with speculation. The cost of cache coherence is quite high without speculation since every RPC incurs a 30 ms round-trip delay.

SpecNFS typically has less than 10 rollbacks during the Apache benchmark. When a file is moved to a new directory, the Linux NFS server assigns it a new file handle; when the client next tries to access the file by the old handle, the server returns ESTALE if the entry with the old file handle has been evicted from its file cache. The client then issues a `lookup` RPC to learn the new file handle. Since SpecNFS does not anticipate the file handle changing, an ESTALE response causes a rollback. Like the nonspeculative version of NFS, SpecNFS learns the correct file handle when it issues a nonspeculative `lookup` on re-execution.

BlueFS is 66% faster than nonspeculative NFS with no delay; with a 30 ms delay, BlueFS is 12 times faster. These results demonstrate that, with speculative execution, it is possible for a distributed file system to be safe, consistent, *and* fast. BlueFS does not experience any rollbacks during this benchmark.

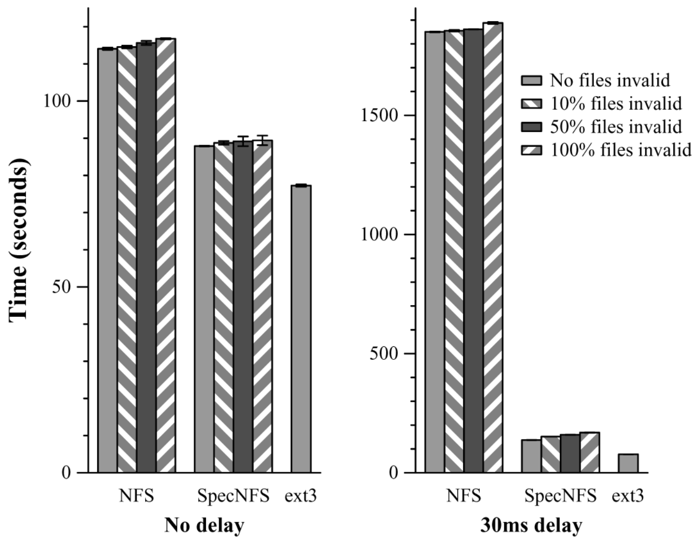
To provide a point of comparison with file systems that use callbacks, we ran the benchmark using version 6 of the Coda distributed file system; we executed the benchmark with Coda in strongly-connected mode (which provides AFS-like consistency guarantees). BlueFS, which provides stronger consistency, outperforms Coda by 12%. With 30ms of delay, BlueFS is 3 times faster than Coda. While we would need to implement a speculative version of Coda to determine precisely how much Speculator would improve its performance, our existing results provide some insight. Since Coda's use of callbacks results in fewer synchronous RPCs than NFS, we suspect that a speculative version of Coda would perform better than SpecNFS.

8.4 The Cost of Rollback

In the previous two benchmarks, almost all speculations made by the distributed file system were correct. We were therefore curious to measure the performance impact of having speculations fail. We quantified this cost by performing the `make` stage of the Apache benchmark while varying the percentage of out-of-date files in the NFS client cache.

To create out-of-date files, we append garbage characters to a randomly selected subset of the files in the Apache source directory—if one of these modified files were to be erroneously used during the `make`, the Apache build would fail. Before the experiment begins, the client caches all files in the source directory—this includes both the unmodified and modified files. We then replace each modified file with its correct version at the server. Thus, before the experiment begins, the client cache contains a certain percentage of out-of-date files that contain garbage characters.

Nonspeculative NFS detects that the out-of-date files have changed when it issues synchronous `getattr` RPCs during file open. The client discards its cached copies of these files and fetches new versions from the server.



This figure shows the time to make Apache 2.0.28 with different percentages of files out-of-date in the client cache. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Fig. 6. Measuring the cost of rollback.

SpecNFS continues to execute with the out-of-date copies and issues asynchronous `getattr` RPCs. When an RPC fails, it invalidates its cached copy and rolls back to the beginning of the `open` system call. In all cases, the compile completes successfully.

Figure 6 shows `make` time when approximately 10%, 50%, and 100% of the files are initially out-of-date. As the number of out-of-date files grows, the time to complete the benchmark increases slightly due to the time needed to fetch up-to-date file versions from the server. This delay is small since network bandwidth is high—however, this cost must be incurred by both speculative and nonspeculative versions of NFS. Even when all files are initially out-of-date, SpecNFS still substantially outperforms NFS without speculative execution: SpecNFS is 31% faster with no delay and over 11 times faster with 30 ms delay. These improvements are comparable to results with no files out-of-date (30% and 13 times faster, respectively). Thus, the impact of failed speculations is almost imperceptible in this benchmark. However, the impact could be greater on a loaded system with less spare CPU cycles for speculation, or when the network bandwidth between client and server is limited.

Figure 7 shows more detail about these experiments. The first column shows the number of out-of-date files in the source directory actually accessed by `make`. The remaining columns show the average number of speculations that roll back and commit during execution of the benchmark. We were initially surprised to see that the number of rollbacks is substantially less than the number of out-of-date files accessed by the benchmark, meaning that several out-of-date files

Files out of date	No delay		30 ms delay	
	Rollbacks	Commits	Rollbacks	Commits
0	0	25973	0	25739
80	40	25870	40	26446
374	183	25869	183	26803
758	362	25831	358	27451

This figure shows the average number of speculations that roll back and commit during the `make` stage of the Apache benchmark when different numbers of source files are initially out-of-date in the client cache.

Fig. 7. Rollback benchmark detail.

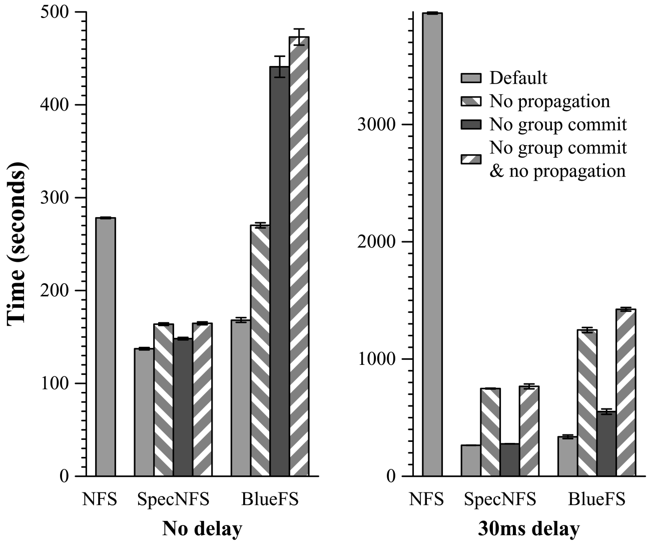
do not cause a rollback. This disparity is caused by `make` accessing several out-of-date files in short succession. For example, consider a case where `make` reads the attributes of two out-of-date files before it receives the `getattr` response for the first file. The first response causes a rollback since the attributes in the client cache were incorrect. Since this rollback eliminates all state modified after the first request was issued, a rollback is not needed when the second `getattr` response is received. SpecNFS updates file attributes in the client cache based upon both responses. Thus, on re-execution, the new speculations for *both* files are correct. In this example, although two files are out-of-date, the client only rolls back once. In effect, this is a type of prefetching of out-of-date file attributes, somewhat similar to the prefetching of file data proposed by Chang and Gibson [1999].

8.5 Impact of Group Commit and Sharing State

Figure 8 isolates the impact of two ideas presented in this article. The left bar of each graph shows the time needed by the nonspeculative version of NFS to perform all four stages of the Apache benchmark described in Section 8.3. The next two datasets show the performance of SpecNFS and BlueFS. For each file system, we show the default configuration (with full support for speculative execution), and three configurations that omit various parts of Speculator.

The second bar in each data set shows performance when Speculator does not let processes share speculative state through kernel data structures. This implementation blocks speculative processes that read or write to pipes and files in RAMFS, send signals to other processes, fork, or exit. Only speculative state in the distributed file system is shared between processes. The third bar in each data set shows the performance of speculative execution when the server does not allow speculative group commit. The last bar shows performance when both speculative group commit and sharing of speculative state are disallowed.

The benefit of propagation is most clearly seen with 30 ms delay. Blocking until speculations are resolved is more costly in this scenario since sever replies take longer to arrive. The benefit of speculative group commit is less dependent on latency. Without propagation and speculative group commit, the improved consistency and safety of BlueFS are much more costly. Especially on low-latency networks, both improvements are needed for good performance.



This figure shows how speculative group commit and propagating state within the kernel impact the performance of speculative file systems for the Apache benchmark. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Fig. 8. Measuring components of speculation.

9. RELATED WORK

To the best of our knowledge, Speculator is the first support for multiprocess speculative execution in a commodity operating system and the first use of speculative execution to improve cache coherence and write throughput in distributed file systems.

Chang and Gibson [1999] and Fraser and Chang [2003] use speculative execution to generate I/O prefetch hints for a local file system. In their work, the speculative thread executes only when the foreground process blocks on disk I/O. When the speculative thread attempts to read from disk, a prefetch hint is generated and fake data is returned so that the thread can continue to execute. Their work improves read performance through prefetching, whereas Speculator improves read performance by reducing the cost of cache coherence. Speculator also allows write-path optimizations such as group commit. In Speculator, speculative processes commit their work in the common case where speculations are correct. However, since Chang's speculative threads do not see correct file data, any computation done by a speculative thread must be re-done later by a nonspeculative thread. Speculator also allows multiple processes to participate in a speculation; Chang's speculative threads are not allowed to communicate with other processes.

Time Warp [Jefferson et al. 1987] explored how speculative execution can improve the performance of distributed simulations. Time Warp is specialized for executing simulations, and imposes certain restrictions that may be onerous to developers of general applications: processes must be deterministic, cannot use heap storage, and must communicate via asynchronous messages. Speculator is targeted for a general purpose operating system and does not impose these restrictions. Time Warp's abstraction of virtual time [Jefferson 1985] tracks causal dependencies using Lamport's [1978] clock conditions. Speculator's dependency lists are more general than a one-dimensional value. This generality is most useful with concurrent speculations where a one-dimensional time value creates an unnecessary dependency between two independent events. In contrast to Time Warp, the client-server nature of distributed file systems allows Speculator to simplify its handling of speculative state: failed speculative state created by a client is never committed at the server, nor is it visible to other clients. Speculator's design for buffering output was inspired by Time Warp's handling of external output.

Common applications of speculative execution in processor design range from branch prediction to cache coherence in multiprocessors. Steffan et al. [2000] have investigated the use of speculation to extract greater parallelism from application code. Similar proposals advocating hardware speculation are Franklin and Sohi [1996], Hammond et al. [1998], and Zhang et al. [1999]. The difference between implementing speculation in the processor and in the OS is the level of granularity: OS speculation can be applied at much higher levels of abstraction where processor-level speculation is inappropriate. Speculative execution has recently been used for dynamic deadlock detection [Li et al. 2005], surviving software failures [Qin et al. 2005], and improving the performance of distributed shared memory [Tapus et al. 2005].

Speculative execution provides a limited subset of the functionality of a transaction. Thus, transactional systems such as QuickSilver [Schmuck and Wylie 1991], Locus [Weinstein et al. 1985], or TABS [Spector et al. 1985] could potentially supply the needed functionality. However, the overhead associated with providing atomicity and durability might preclude transactions from achieving the same performance gains as speculative execution.

Checkpointing and rollback-recovery are well-studied [Elnozahy et al. 2002]. However, most prior work in checkpointing focuses on fault tolerance; a key difference in Speculator is that checkpoints capture only volatile and not persistent state. In this manner, they are similar to the lightweight checkpoints used for debugging by Srinivasan et al. [2004]. Previous work in logging virtual memory [Cheriton and Duda 1995] might offer performance optimizations over Speculator's current fork-based checkpoint strategy. Causal dependency tracking has been applied in several other projects, including QuickSilver [Schmuck and Wylie 1991], BackTracker [King and Chen 2003], and the Repairable File Service [Zhu and Chiueh 2003].

Prior distributed file systems have provided single-copy file semantics—examples include Sprite [Nelson et al. 1988] and Spritely NFS [Srinivasan and Mogul 1989]. However, these semantics came at a performance cost. On the other hand, Liskov and Rodrigues [2004] demonstrated that read-only transactions in a file system can be fast, but only by allowing clients to read slightly stale (but consistent) data. A major contribution of Speculator is showing that strong safety and cache coherence guarantees can be provided with minimal performance impact. In this manner, it is possible that speculative execution might improve the performance of distributed file systems such as BFS [Castro and Liskov 2000], SUNDR [Li et al. 2004], and FARSITE [Adya et al. 2002] that require additional communication to deal with faulty or untrusted servers. Finally, Satyanarayanan [1996] observed that semantic callbacks could reduce the cost of cache coherence by expressing predicates over cache state. The hypotheses used by our speculative file systems can be viewed as predicates expressed over speculative state.

10. CONCLUSION

Speculator supports multiprocess speculative execution within a commodity OS kernel. In this article, we have shown that Speculator substantially improves the performance of existing distributed file systems. We have also shown how speculation enables the development of new file systems that are safe, consistent, *and* fast, even over high-latency links. Our modified version of BlueFS provides single-copy file semantics and synchronous I/O, yet still performs better than current file systems. While our investigation to date has focused on distributed file systems, we believe that generic OS support for speculative execution and causal dependency tracking will prove useful in many other domains. Our future plans therefore include investigating what other applications can benefit from Speculator.

ACKNOWLEDGMENTS

We thank Manish Anand, Dan Peek, Ya-Yunn Su, the anonymous reviewers, and our SOSP shepherd, Franz Kaashoek, for suggestions that improved the quality of this article. Jan Harkes and Mahadev Satyanarayanan helped explain details of the Coda file system. We used David A. Wheeler's SLOCCount to estimate the lines of code for our implementation.

REFERENCES

Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. Boston, MA, 1–14.

Birrell, A. D., Hisgen, A., Jerian, C., Mann, T., and Swart, G. 1993. The Echo distributed file system. Tech. Rep. 111, Digital Equipment Corporation, Palo Alto, CA. October.

Callaghan, B., Pavlowski, B., and Staubach, P. 1995. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF. June.

Carson, M. http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm. *Adaptation and Protocol Testing thorugh Network Emulation*. NIST.

Castro, M. and Liskov, B. 2000. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*. San Diego, CA.

Chang, F. and Gibson, G. 1999. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. New Orleans, LA, 1–14.

Cheriton, D. and Duda, K. 1995. Logged virtual memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, 26–39.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (Sept.), 375–408.

FRANKLIN, M. AND SOHI, G. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput. 45*, 5 (May), 552–571.

FRASER, K. AND CHANG, F. 2003. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference*. San Antonio, TX, 325–338.

HAERDER, T. AND REUTER, A. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv. 15*, 4 (Dec.), 287–317.

HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International ACM Conference on Architecture. Support for Programming Languages and Operating Systems*. San Jose, CA, 58–69.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.).

JEFFERSON, D. 1985. Virtual time. *ACM Trans. Prog. Lang. Syst. 7*, 3 (July), 404–425.

JEFFERSON, D., BECKMAN, B., WIELAND, F., BLUME, L., DILORETO, M., HONTALAS, P., LAROCHE, P., STURDEVANT, K., TUPMAN, J., WARREN, V., WEIDEL, J., YOUNGER, H., AND BELLENOT, S. 1987. Time Warp operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. Austin, TX, 77–93.

KATCHER, J. 1997. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance.

KING, S. T. AND CHEN, P. M. 2003. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 223–236.

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.).

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7, 558–565.

LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. 2004. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. San Francisco, CA, 121–136.

LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. 2005. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 1005 USENIX Technical Conference*. 31–44.

LISKOV, B. AND RODRIGUES, R. 2004. Transactional file systems can be fast. In *Proceedings of the 11th SIGOPS European Workshop*. Leuven, Belgium.

MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. 1999. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Kiawah Island, SC, 124–139.

NELSON, M. N., WELSH, B. B., AND OUSTERHOUT, J. K. 1988. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6*, 1, 134–154.

NIGHTINGALE, E. B. AND FLINN, J. 2004. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. San Francisco, CA, 363–378.

QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. 2005. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. Brighton, United Kingdom, 235–248.

ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. The impact of architectural trends on operating system performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, 285–298.

SATYANARAYANAN, M. 1996. Fundamental challenges in mobile computing. In *Proceedings of the 15th Symposium on Principles of Distributed Computing*. Philadelphia, PA, 1–7.

SCHMUCK, F. AND WYLIE, J. 1991. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. 239–253.

SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., AND NOVECK, D. 2003. Network File System (NFS) version 4 Protocol. Tech. Rep. RFC 3530, IETF. April.

SPECTOR, A. Z., DANIELS, D., DUCHAMP, D., EPPINGER, J. L., AND PAUSCH, R. 1985. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*. Orcas Island, WA, 127–146.

Srinivasan, S., Andrews, C., Kandula, S., and Zhou, Y. 2004. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*. Boston, MA.

Srinivasan, V. and Mogul, J. 1989. Spritely NFS: Experiments with cache consistency protocols. In *Proceedings of the 12th ACM Symposium on Operating System Principles*. 45–57.

Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. Vancouver, Canada, 1–24.

Tapus, C., Smith, J. D., and Hickey, J. 2003. Kernel level speculative DSM. *Workshop on Distributed Shared Memory on Clusters*. Tokyo, Japan.

Weinstein, M. J., Page, T. W. Jr., Livezey, B. K., and Popek, G. J. 1985. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*. Orcas Island, WA, 115–126.

Zhang, Y., Rauchwerger, L., and Torrellas, J. 1999. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. Orlando, FL, 135.

Zhu, N. and Chiueh, T. 2003. Design, implementation and evaluation of the Repairable File Service. In *Proceedings of the International Conference on Dependable Systems and Networks*. San Francisco, CA.

Received January 2006; revised April 2006; accepted April 2006