Speculative Execution Within A Commodity Operating System

by

Edmund B. Nightingale

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:
Assistant Professor Jason N. Flinn, Chair
Professor Peter M. Chen
Associate Professor Ella M. Atkins
Associate Professor Brian D. Noble

For EJN, without whom the journey would never have begun, and for RJN, who is a great part of the reason the journey has completed

ACKNOWLEDGEMENTS

When I arrived at the University of Michigan, Jason Flinn was just getting started as a professor, and I was fortunate enough to become his second graduate student. During our five years working together, Jason's guidance has been an invaluable asset in conceiving and executing my research. From Jason I learned the art of systems building, and the craft of technical writing. I am grateful to have had him as a mentor.

From Peter Chen, with whom I collaborated, I learned the art of refining the rough hewn details of my work into finely tuned principles of research. I am thankful for the opportunity to work with him. I thank my other two committee members, Brian Noble and Ella Atkins. Our discussions about my dissertation and defense provided me with many insights, and helped greatly to improve the final document. I also thank Miguel Castro and Ant Rowstron, whose guidance during my internship has left an enduring impact on my approach to research.

Thank you to my EECS office-mates with whom I shared long days and late nights hacking, writing, and working on the next big idea, especially Manish Anand, Dan Peek, Kaushik Veeraraghavan and Ya-Yunn Su. They made my work better, acting as both critic and ally when I needed it most. Most importantly, they made the office an enjoyable place to work, especially when progress seemed to be at its slowest. Thanks also goes to Yuan Lin, Mona Attariyan, Sushant Sinha, and Evan Cooke for listening to my crazy ideas and keeping me on my toes.

A special thanks to everyone in the Friday lunch club. Getting away from North Campus to explore new cuisine was an event I looked forward to every week.

To my oldest friends, Gino Canessa, Kyle Maynor, Ben Chastek, Pat Merdan, and Tony Sauro, I owe many thanks. I would especially like to thank my father for his thoughtful encouragement, which spurred my efforts towards a doctoral degree. He has always provided patient and wise council to me as I navigated the waters of doctoral research.

Finally, I would like to thank my wife, Rebecca, for all that she has endured as the spouse of a graduate student. She has made this journey worthwhile. Her love, encouragement, generous spirit, and infinite patience were always present to help me on my way to completing this dissertation.

TABLE OF CONTENTS

CHAPTERS

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

Commodity operating systems are traditionally tuned to improve performance, but in the pursuit of faster operation, reliability is sacrificed. Application developers are currently provided with "either/or" abstractions that force them to choose between a fast and unreliable abstraction (which requires them to implement a guarantee within an application) or an abstraction that is slow but safe. Therefore, common abstractions, such as asynchronous file system I/O, explicitly relax reliability guarantees (e.g., delay writes to disk by 30 seconds) in favor of better performance.

The main concern of this thesis is to mitigate the performance/reliability tradeoff through the use of a general purpose mechanism, called Speculator, which implements multi-process speculative execution within the Linux kernel. Speculator is a tool for masking the latency of predictable, but slow, operations. It allows a critical, synchronous operation to be executed asynchronously. Thus, additional work is completed without sacrificing the benefits (i.e., the guarantees) of synchronous execution. Speculator accomplishes this feat by providing facilities that track and propagate the causal dependencies of a process, defer the release of speculative output, and roll back dependent speculative kernel objects and processes should a speculation fail.

This dissertation shows that Speculator can be used to create new abstractions, which replace synchronous abstractions while providing equivalent guarantees, within local file systems, distributed file systems and application level runtime security checks. Abstractions built using Speculator allow a programmer to develop applications for a reliable, synchronous abstraction while obtaining the performance of one that is asynchronous. The speculative abstractions that I have built do not require application modification, the benefits of speculation are transparent.

This chapter begins with a brief overview of prior work regarding speculative execution. I discuss why a speculative version of a synchronous abstraction provides an equivalent guarantee to a synchronous abstraction and what invariants must be maintained to ensure the execution of a speculative process is correct. I describe what conditions must be present for speculative execution to be successful and provide a brief introduction to the two types of speculation that Speculator supports. Finally, I describe the inherent tradeoffs of implementing Speculator at the granularity of a process, rather than implementing it at a more coarse (virtual machine) or fine (instruction level) granularity.

1.1 Speculative execution

In general, speculative execution is a method to hide the latency of an event by predicting its result, and overlapping work assuming the prediction is correct. If the speculation is incorrect, any state that depends upon the speculation must be undone. If an output commit occurs, (an operation that cannot be undone) than speculative progress must be stopped until the speculation is resolved. For a more detailed treatment of the output commit problem and different techniques to manage rollback I refer the reader to Elnozahy's survey of message passing protocols in distributed systems [17].

In the past, speculative execution has been used primarily as either a prefetching mechanism (e.g., branch prediction, disk prefetching), or as a way to implement optimistic concurrency control (e.g., discrete event simulators, database transaction processing). Speculator has the benefit of a much wider scope within the operating system than prior systems built either at the architecture or application layers.

First and foremost, branch prediction leverages speculative execution to prefetch instructions streams into a pipeline while resolving a branch asynchronously. With a high success rate, branch prediction has proven to be very useful within modern processors with deep instruction pipelines. Speculative execution has also been used to automatically extract parallelism from processes [74], and to run speculative operations within a parallel thread on simultaneous multithreading (SMT) machines [24, 33, 68, 74]. The limited visibility of process and kernel state at the architecture layer hinders efforts to use these techniques with higher level abstractions such as processes, kernel objects, external devices, and distributed systems.

At the application layer, two models of speculative execution have been used. In the first, a speculative dummy-thread is spawned to do some work in parallel to its non-speculative counterpart. This has been used to prefetch disk blocks [10, 21] and detect deadlock dynamically [37]. Since the dummy-thread is speculative, its interaction with kernel objects, other processes and devices must be tightly controlled. When the dummy-thread attempts an output commit it is terminated. The system call interface is the boundary of an output commit, since the operating system has no knowledge of speculation.

In the second form of speculative execution, a process is marked as speculative, and a checkpoint is taken. The speculative process can be rolled back using the checkpoint should the speculation fail. This model has been used to avoid bugs at runtime [58]. Again, because the kernel is not aware of speculation, system calls are often output commits, and the speculative process must block. The lack of support within the kernel for speculation limits the amount of work that can be accomplished while speculative.

1.2 Speculator

Speculator is the first system to weave support for speculative execution throughout an operating system, allowing speculative processes to cross process boundaries and affect kernel state. The kernel is often a participant in inter-process communication; therefore, Speculator, which resides within the kernel, intercepts inter-process communication to track and propagate the dependencies of speculative processes as they execute. In prior work, the system call interface was the boundary for an output commit; by making the kernel aware of speculation, Speculator bridges this gap to track and propagate dependencies when a speculative process enters the kernel. Other processes and kernel objects inherit dependencies as they interact, and Speculator provides mechanisms for rolling back changes if a speculation fails. The expansive scope and flexibility of Speculator allows it to be used in a diverse set of abstractions, ranging from storage systems to application security.

Two questions of correctness must be addressed when using speculative execution. First, when a synchronous abstraction is converted to a speculative abstraction, do the two abstractions provide an equivalent guarantee? Second, how do I ensure a process executing speculatively executes correctly (i.e. does there exist some execution of a non-speculative version of the process that obtains the same result)?

1.2.1 A guarantee provided

This thesis claims that a speculative abstraction provides the same guarantees as its synchronous counterpart. I validate this claim by formalizing what guarantees are provided by asynchronous and synchronous abstractions. Using this framework of guarantees, I then show that an abstraction built around Speculator provides an equivalent guarantee to that of its synchronous counterpart.

A *synchronous abstraction* guarantees that when a process invokes some action by calling into the abstraction (e.g., writing data to disk) that the action is performed before the call completes. A synchronous abstraction guarantees the ordering of all actions will match the order in which they were invoked by the application (since they are completed one by one); in addition, any other actions taken by the application via other abstractions will be ordered with respect to the prior calls to the synchronous abstraction. Finally, a synchronous abstraction provides the guarantee to the application that the action has been taken.

An *asynchronous abstraction* provides no such guarantees. The action is moved out of the critical path, but no ordering guarantees are necessarily provided. After the return of the call the application has no guarantee that the action has completed.

Often, the interface to an asynchronous abstraction offers additional calls that a developer may use to obtain the guarantees provided by a synchronous version of the abstraction. It is more work for the programmer to acquire these guarantees (since they must be implemented within every program that requires them), and more difficult for the programmer to reason about the correctness of the program. One advantage of asynchrony is the opportunity to implement optimizations by waiting to see what the future holds. Work that is initially delayed may be amortized (e.g., group committing data to disk) or eliminated completely (e.g., data written and then later deleted). These types of optimizations cannot be used within a synchronous abstraction.

A synchronous abstraction has many benefits, yet time and again the software systems we use opt for an asynchronous abstraction instead. This choice is not an accident, but rather, a reflection of the slowness of a synchronous abstraction – the benefits of synchrony are obtained only by forcing the application to wait for the action to complete. An asynchronous interface hides the expense of an action (and sacrifices any guarantee) by moving it out of the critical path. When the expense is great, systems designers have been forced to settle for an asynchronous abstraction in exchange for faster execution time.

By building operating system support for speculative execution, I have been able to build new abstractions that provide the guarantees of a synchronous abstraction and that often perform nearly as well as their asynchronous counterparts. These abstractions can be built with the benefit of a key insight: the guarantees provided by a synchronous abstraction need only be provided to users and not applications. When guarantees are provided to applications, the application must block and wait for confirmation that a guarantee has been provided. Yet, the user is the ultimate recipient of a guarantee; it is the user who may incur financial liability if data becomes corrupted, and it is the user who may sue a developer whose application claims to provide a guarantee and then fails.

A user reasons about whether a guarantee has been provided by the external output (or output commits) generated by the application. For example, an application that prints "`data written to device`" on the console let's the user know a guarantee has been provided; without external output, the user cannot know the progress of the application, or what (if any) guarantee has been provided. Speculator frees the application from acting as proxy for the user. Speculator guarantees that the ordering of each external output respects the ordering that would have been provided by a synchronous abstraction.

1.2.1.1 Defining external output

To be useful, a precise definition of external output is required. Traditionally, the operating system takes an *application-centric* view of the computer system, in which it considers applications to be external entities observing its behavior. This view divides the computer system into two partitions: the kernel, which is considered internal state, and the user level, which is considered external state. Using this view, the return from a system call is considered an externally visible event.

However, users, not applications, are the true observers of the computer system. Application state is only visible through output sent to external devices such as the screen and network. By regarding application state as internal to the computer system, the operating system can take a *user-centric* view in which only output sent to an external device is considered externally visible. This view divides the computer system into three partitions, the kernel and applications, both of which are considered internal state, and the external interfaces, which are considered externally visible. Using this view, changes to application state, such as the return from a system call, are not considered externally visible events.

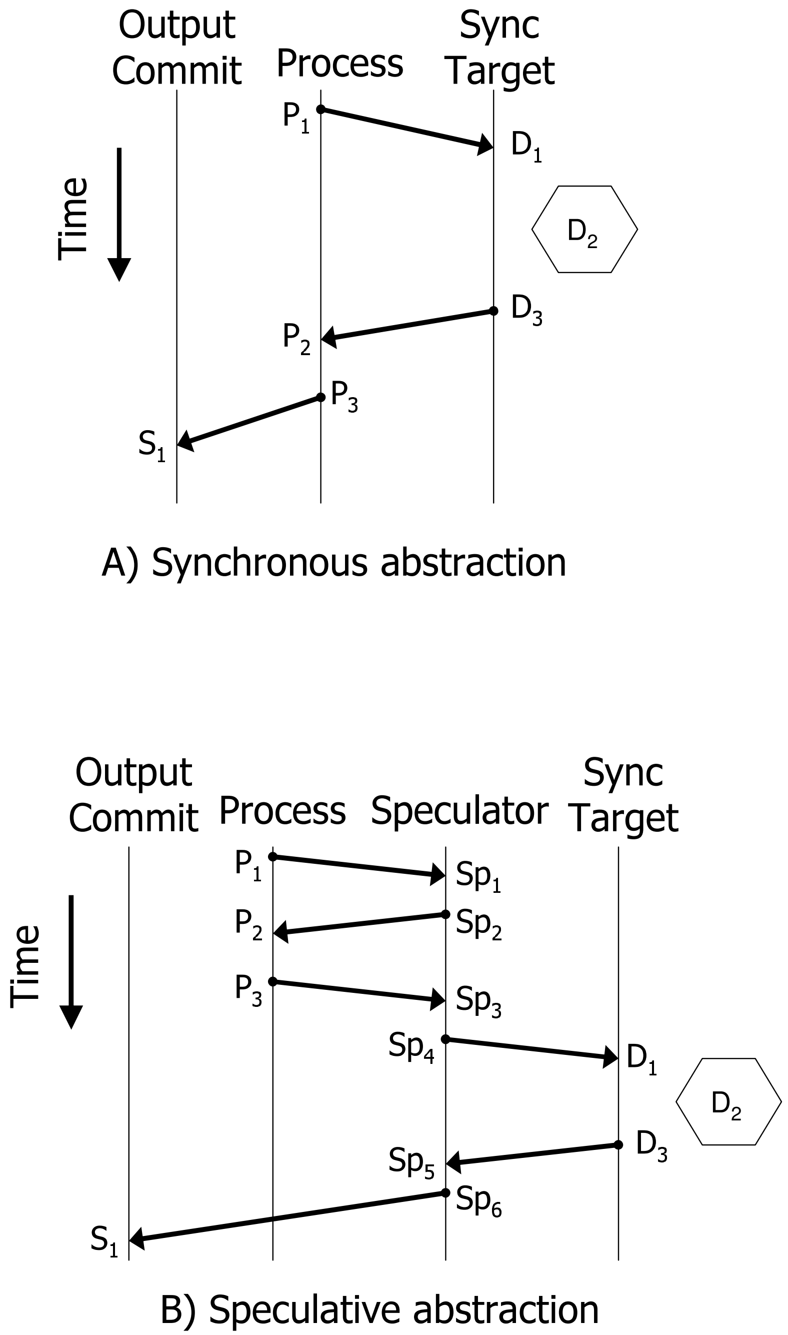
More precisely, an entity in a system *must* be considered external only if it (and its affects on other entities) cannot be rolled back. Conversely, if a system considers an entity internal, then the system must be able to roll back the entity and its affects on other entities. More concretely, Speculator considers the network, the screen, and the disk to be external entities. Of these, only the screen cannot be rolled back (ignoring the possibility of rolling back a user who observed the screen). The network interface and the disk are entities that could be internal. For example, writing data to disk could be internal as long as a commit never occurs (i.e., the durable changes can be undone) and the disk is not removed. Making the disk and network interface external lessened the complexity of the system.

The operating system can implement user-centric guarantees because it controls access to external devices. Applications can only generate external events with the cooperation of the operating system. Applications must invoke this cooperation either directly by making a system call or indirectly by mapping an externally visible device.

1.2.1.2 Equivalence of speculative and synchronous abstractions

Speculator is a general purpose mechanism for multi-process speculative execution; it provides an interface that allows a process, when in kernel mode, to speculate on some event and then later commit or fail that speculation. There is a separation of policy and mechanism in my design of Speculator. Speculator provides the mechanism for a process to execute speculatively, while a process determines when a speculation should begin, and whether it should succeed or fail. Details about the internals of Speculator are covered in Chapters 2 and 3. This thesis demonstrates the utility of implementing synchronous abstractions using speculative execution, thereby moving the synchronous operation out of the critical path and minimizing its impact on performance. Therefore, it is necessary to show that a speculative abstraction built using Speculator does provide a user-centric guarantee equivalent to its synchronous counterpart.

Speculator tracks the potential causal dependencies of a speculative process (with respect to each now speculative synchronous operation) as defined by Lamport's [35] "happens before" relation. Figure 1.1(A) shows an example of a process interacting with a generic synchronous abstraction. The process invokes some action to a device synchronously and then subsequently generates some external output. I model the devices and objects within the operating system as entities within a distributed



A) Synchronous abstraction

B) Speculative abstraction

The upper figure shows a generic synchronous abstraction. A process interacts synchronously with a device. Once the operation completes, the process releases an output commit. The lower figure shows a speculative version of the abstraction. Speculator is used to mask the latency of the operation to the synchronous device while providing an equivalent guarantee to an external observer of the system.

Figure 1.1: A generic synchronous abstraction and its speculative counterpart

system. During event P1, the process invokes an action to the synchronous target; event D2 represents the completion of the action (e.g., a disk write or return of a remote procedure call) and during event P2 control is returned to the process and it continues executing. Between events P1 and P2 the process is blocked (it makes no progress) until the action completes. Finally, in event P3, the process creates external output. In this example, a guarantee is provided that the completion of the action D2 "happens before" the return of control to the process in event P2 (D2 → P2). While technically correct, this relation is not very useful within the context of a user-centric guarantee; the return of control to the process is not external output and is therefore not visible to the user. In turn, P2 → S1, meaning that the return of control to the process is guaranteed to happen before the external output, and therefore, D2 → S1 – as defined by Lamport, the external output has a potential causal dependency with the action on the synchronous device, and the action must happen before the output. A synchronous abstraction ensures that if the external output informs the user of the guarantee provided that the synchronous action has been completed. The synchronous protocol maintains this causal ordering by blocking the application during the synchronous action. An asynchronous abstraction ignores such orderings, they must be provided by the programmer. Further, for any event E occurring after P2, we are guaranteed that D2 → E, and the event will be ordered with respect to D2. In this example, to provide an equivalent user-centric guarantee, we need only ensure that D2 → S1; the other relations do not matter. This frees the application from waiting for an event to complete, and it must no longer act as a proxy for the application.

Figure 1.1(B) shows a version of the the generic synchronous abstraction that uses Speculator to mask the latency of the action to the synchronous device. During the event P1, the process invokes the action to the synchronous device. The speculative version of the abstraction invokes Speculator; the process calling into the abstraction is marked as speculative and the data associated with the action is buffered within the kernel. The call to the synchronous abstraction returns immediately. The process then generates external output during event P3; Speculator intercepts the call, recognizes the causal dependency between the external output and the buffered action to the synchronous device and buffers the external output as well. Note that an asynchronous abstraction would buffer the action to the synchronous device, but it would release the external output, thereby reordering the events as they were generated by the application. In the speculative version of the abstraction, D2 ↛ P2, but this has no effect on the guarantee provided, because no output causally dependent on D2 has been released. At some later point, Speculator executes the action to the synchronous device during event Sp4 (deciding when to commit is discussed in detail in Chapter 2). Once event D2 occurs Speculator can then safely release any external output dependent upon D2; therefore, during event Sp6, Speculator releases the external output previously generated by the process. Thus, in the speculative version of the abstraction, D2 → S1, and for any external output E generated by the process after the event P2, D2 → E. The Speculative version of the abstraction has provided an ordering that respects the ordering that would have been generated by the synchronous version of the protocol, and therefore provides an equivalent user-centric guarantee. The speculative version of the abstraction also provides opportunities for optimizations. For example, if a process executes two consecutive actions to a synchronous device, Speculator could use an atomic group commit to improve performance. Group commit would provide a new ordering (the operations would be concurrent), but the ordering would not violate the original (i.e., if A happens before B, then if B completes A must also complete).

More generally, whenever one masks a synchronous operation by using speculative execution (e.g., disk writes, network messages, or security checks), Speculator ensures that all other output commits are ordered with respect to the completion of a speculative event. This ordering is not limited to the process that initiated the speculation; Speculator propagates the dependencies of a speculative process whenever the process interacts with the other processes and kernel objects. When another process inherits a speculative dependency, its output is ordered with respect to the speculative operation as well. A detailed explanation follows in Chapters 2 and 3.

1.2.1.3 Invariants for correct speculative execution

In general, when a process is speculative, I consider its execution to be correct if the following invariants hold:

1. **Speculative state is never externalized.** If speculative state is externalized and the speculation is incorrect, then the output seen (or consumed) by the end user may be incorrect as well. State cannot be visible until it is determined that each speculation that it depends upon is correct.

2. **The effects of speculation must be transparent.** The causal ordering of external output of a speculative process must not violate the causal ordering of some correct execution of the process when non-speculative (as defined by Lamport's "happens before" relation [35]). Otherwise, speculative execution might effect process execution such that a speculative version of a process results in incorrect computation. An external observer should not be able to tell by examining the output of a system whether or not the operating system was taking advantage of speculative execution.

3. **Speculative state must be undone should a speculation fail.** When state becomes speculative, it now depends upon the success of one or more speculations. If a speculation fails, all state that depends upon that speculation must be restored to the point at which the state no longer depends upon the failed speculation.

Speculator enforces each invariant by buffering speculative output, tracking and propagating the dependencies of speculative processes, and by saving state so that speculative changes to processes and kernel objects can be undone.

Therefore, my thesis is:

Many synchronous abstractions provide strong reliability guarantees but are slow. Their asynchronous counterparts relax reliability guarantees to achieve reasonable performance. With operating system support for speculative execution, new abstractions can be built that provide the reliability and simplicity of a synchronous abstraction while often approximating the performance of an asynchronous abstraction.

1.3 Managing Failure

When a speculation fails, a system must find and roll back all state that is dependent upon the failure. In a distributed system (e.g., a distributed file system), multiple clients could share speculative state, but then dependencies must be propagated between clients, which makes tracking dependencies difficult. Deciding when to commit becomes complicated, because all clients must agree that no rollback will occur; thus, two-phase commit may be required. Finally, the commit strategy must also be carefully constructed to avoid cascading rollbacks [17] on failure. When implementing Speculator, two design decisions greatly reduce the complexity of managing failures. First, speculative state is never shared with multiple clients in a distributed system. Restricting speculative state to a single client eliminates problems such as cascading rollbacks, orphan clients, and the need for two-phase commit. Second, speculative state is kept in volatile memory; therefore, after a crash and reboot, all state on stable storage is known to be correct, and there is no need to reason about whether state on stable storage must be rolled back because a speculation failed.

When I was using speculator to convert synchronous abstractions into speculative abstractions, I realized that two different models for failure were needed. First, in the traditional sense, a speculation is a prediction whose success is unknown. For example, predicting whether a version of a cached file is up to date with respect to a version at a remote server depends entirely upon events outside the control of the client, namely whether another client has updated the file. This type of speculation requires that the speculate process to either commit or fail the speculation when the result is determined. Upon failure, Speculator rolls back all the objects and processes within the kernel that depend upon the failed speculation.

A second type of speculation exists; one for which the outcome of the speculation is known to be correct. In this case, speculation moves the event out of the critical path (i.e., the event completes asynchronously) while preserving the ordering of all external output with respect to the speculative event (using the “happens before” relation, as discussed in Section 1.2.1.2). In this case, a speculation is always committed once the operation completes. If the operating system crashes, the speculative state is deleted, which rolls back the state of the machine. When an abstraction is used in this way, Speculator implements *lightweight* speculations, which do not save checkpoint state for kernel objects and processes.

In this thesis, Chapter 2 discusses how lightweight speculations are implemented, while Chapter 3 discusses in detail the architecture for speculations and the changes needed to support checkpoint and rollback.

1.3.1 Conditions for success

Speculator is designed around four conditions for success – whenever an abstraction and a system operate within these conditions speculative execution has the opportunity to show great benefit.

1. **Executing speculatively is faster than executing synchronously.** Synchronous abstractions such as synchronous file system I/O are far slower than the cost of tracking and propagating causal dependencies. In general, when a synchronous abstraction must: interact with a mechanical device, interact over a network, or interrupt the runtime execution of a program, speculative execution may well run faster than executing the operation synchronously.

2. **The common case for a speculative operation is success.** In this thesis, Speculator is used to predict the success of operations whose common case

is success. For example, in Chapter 4, I use Speculator to predict that an application is not compromised. Clearly, there is a tension in the success rate of speculations; each time a speculation fails, all state that is speculative must be undone.

3. **Checkpoint state is cheap.** The time to take 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 a synchronous operation.

4. **Computers have resources to spare.** Speculative execution requires extra CPU cycles and RAM to be effective. This is typically true in today's workstations. If a computer does not have additional resources to put towards speculative execution, then it is possible little or no benefit may be had from speculating.

Clearly, if these conditions do not exist, Speculator will not achieve performance gains as great as those demonstrated in this thesis; but what about dynamic environments where resource usage may change drastically? Speculator currently manages memory exhaustion by limiting the number of active speculations with a counter. If Speculator were used on a system where available RAM and CPU cycles often varied dramatically (from plentiful to scarce), a more sophisticated system could be employed to tune the priority of speculative processes to non-speculative ones. More drastically, the system could turn Speculator off if the additional CPU cycles were deemed precious enough for another critical (non-speculative) process. Such an environment may not be right for speculative execution. Further, outside of real-time systems, CPU cycles and RAM are often under-utilized; this dissertation makes a strong case that speculative execution is a good use of those resources. Finally, the three uses of Speculator in this thesis satisfy each condition for success. As a result, Speculator achieves up to a two orders of magnitude improvement in performance.

1.4 Process-level speculative execution

Speculator implements speculative execution at the granularity of operating system processes and objects. As the granularity increases, dependencies are more easily tracked (because there are fewer of them), but false positives become more common. Accurately tracking dependencies affects how much state must be rolled back when a speculation fails, and when external output can be safely released.

At a very coarse granularity (e.g., a virtual machine), the entire operating system must be rolled back on failure, halting forward progress for processes that do not depend upon a failed speculation. Further, when any external output is generated, it must be deferred until all prior speculations commit or fail. Therefore, the performance of independent workloads may suffer since all output is assumed to be dependent upon every speculation. At a very fine granularity (e.g., instruction level execution), multi-threaded applications could be rolled back piece by piece, and output could be tracked at a per-thread, rather than a per-process granularity. Unfortunately, such accuracy would come with a high price. As I show in Chapter 4, tracking dependencies at the granularity of an instruction is slow. Therefore, Speculator's process-level dependency tracking is a good compromise between the efficiency of tracking dependencies at a very coarse granularity and the better performance of independent work loads at a very fine granularity.

1.5 Thesis road map

The remainder of this thesis is comprised of 6 chapters. Chapter 2 demonstrates a powerful new abstraction for synchronous I/O, called external synchrony, that virtually eliminates the overhead of providing synchronous guarantees for local file systems. External synchrony uses lightweight speculations, which are never explicitly failed.

They only fail when a power failure or crash occurs, which eliminates the need to save state – a system crash undoes all speculative changes to the system.

Chapter 3 shows how to extend speculative execution into the network; I demonstrate that an existing distributed file system (NFS), can achieve dramatic improvements in performance by converting its consistency and safety protocols to ones that are speculative. I also show that a new distributed file system, built with Speculator in mind, provides strong consistency and safety guarantees while out-performing existing distributed file systems that have much weaker guarantees. This chapter uses a more traditional speculation abstraction; a speculation may fail for reasons of correctness, which triggers a rollback of all dependent state.

Chapter 4 demonstrates that Speculator need not be limited to storage systems. I demonstrate Inspector, a system built around Speculator that takes advantage of multi-core environments to parallelize runtime security checks. Inspector creates parallelism in two ways. First, Inspector decouples a security check from an application by continuing the application, using Speculator, while the security check executes in parallel on another core. Second, Inspector creates parallelism between sequential invocations of a security check by running later checks in parallel to earlier ones. Using Inspector, I implemented three parallel security checks that operate on different types of state: system calls, memory contents, and taint propagation; each parallel check dramatically improves its performance by running on as many as 8 cores.

Chapter 5 has a discussion of the related work, and finally Chapter 6 discusses my thesis contributions and concludes.

CHAPTER 2

Rethinking Synchronous I/O

This chapter demonstrates the use of lightweight speculations to create a new, speculative abstraction for synchronous I/O within local file systems.

2.1 Introduction

File systems serve two opposing masters: durability and performance. The tension between these goals has led to two distinct models of file I/O: synchronous and asynchronous.

A synchronous file system (e.g., one mounted with the `sync` option on a Linux system) guarantees durability by blocking the calling application until modifications are committed to disk. Synchronous I/O provides a clean abstraction to users. Any file system operation they observe to complete is durable — data will not be lost due to a subsequent OS crash or power failure. Synchronous I/O also guarantees the ordering of modifications; if one operation causally precedes another, the effects of the second operation are never visible unless the effects of first operation are also visible. Unfortunately, synchronous I/O can be very slow because applications frequently block waiting for mechanical disk operations. In fact, my results show that blocking due to synchronous I/O can degrade the performance of disk-intensive benchmarks by two orders of magnitude.

In contrast, an asynchronous file system does not block the calling application, so modifications are typically committed to disk long after the call completes. This is fast, but not safe. Users view output that depends on uncommitted modifications. If the system crashes or loses power before those modifications commit, the output observed by the user was invalid. Asynchronous I/O also complicates applications that require durability or ordering guarantees. Programmers must insert explicit synchronization operations such as `fsync` to enforce the guarantees required by their applications. They must sometimes implement complex group commit strategies to achieve reasonable performance. Despite the poor guarantees provided to users and programmers, most local file systems provide an asynchronous I/O abstraction by default because synchronous I/O is simply too slow.

The tension between durability and performance leads to surprising behavior. For instance, on most desktop operating systems, even executing an explicit synchronization command such as `fsync` does *not* protect against data loss in the event of a power failure [43]. This behavior is not a bug, but rather a conscious design decision to sacrifice durability for performance [67]. For example, on `fsync`, the Linux 2.4 kernel commits data to the volatile hard drive cache rather than to the disk platter. If a power failure occurs, the data in the drive cache is lost. Because of this behavior, applications that require stronger durability guarantees, such as the MySQL database, recommend disabling the drive cache [46]. While MacOS X and the Linux 2.6 kernel provide mechanisms to explicitly flush the drive cache, these mechanisms are not enabled by default due to the severe performance degradation they can cause.

I show that a new model of local file I/O, which I term *external synchrony*, resolves the tension between durability and performance. External synchrony provides the reliability and simplicity of synchronous I/O, while closely approaching the performance of asynchronous I/O. In external synchrony, I view the abstraction of synchronous I/O as a set of user-centric guarantees (as discussed in Section 1.2.1.1) that are provided to the clients of the file system. In contrast to asynchronous I/O, which improves performance by substantially weakening these guarantees, externally synchronous I/O provides the same guarantees, but it changes the clients to which the guarantees are provided.

My externally synchronous Linux file system, xsyncfs, uses mechanisms that are provided within Speculator to provide user-centric guarantees. When a process performs a synchronous I/O operation, xsyncfs validates the operation, adds the modifications to a file system transaction, and returns control to the calling process without waiting for the transaction to commit. However, using lightweight speculations, xsyncfs also taints the calling process with a *commit dependency*, that specifies that the process is not allowed to externalize any output until the transaction commits. If the process writes to the network, screen, or other external device, its output is buffered by the operating system. The buffered output is released only after all disk transactions on which the output depends commit. If a process with commit dependencies interacts with another process on the same computer through IPC such as pipes, the file cache, or shared memory, the other process inherits those dependencies so that it also cannot externalize output until the transaction commits. The performance of xsyncfs is generally quite good since applications can perform computation and initiate further I/O operations while waiting for a transaction to commit. In most cases, output is delayed by no more than the time to commit a single transaction — this is typically less than the perception threshold of a human user.

Xsyncfs uses *output-triggered commits* to balance throughput and latency. Output-triggered commits track the causal relationship between external output and file system modifications to decide when to commit data. Until some external output is produced that depends upon modified data, xsyncfs may delay committing data to optimize for throughput. However, once some output is buffered that depends upon an uncommitted modification, an immediate commit of that modification is triggered to minimize latency for any external observer.

My results are very positive. For I/O intensive benchmarks such as Postmark and an Andrew-style build, the performance of xsyncfs is within 7% of the default asynchronous implementation of ext3. Compared to current implementations of synchronous I/O in the Linux kernel, external synchrony offers better performance *and* better reliability. Xsyncfs is up to an order of magnitude faster than the default version of ext3 mounted synchronously, which allows data to be lost on power failure because committed data may reside in the volatile hard drive cache. Xsyncfs is up to two orders of magnitude faster than a version of ext3 that guards against losing data on power failure. Xsyncfs sometimes even improves the performance of applications that do their own custom synchronization. Running on top of xsyncfs, the MySQL database executes a modified version of the TPC-C benchmark up to three times faster than when it runs on top of ext3 mounted asynchronously.

2.2 Design overview

2.2.1 Principles

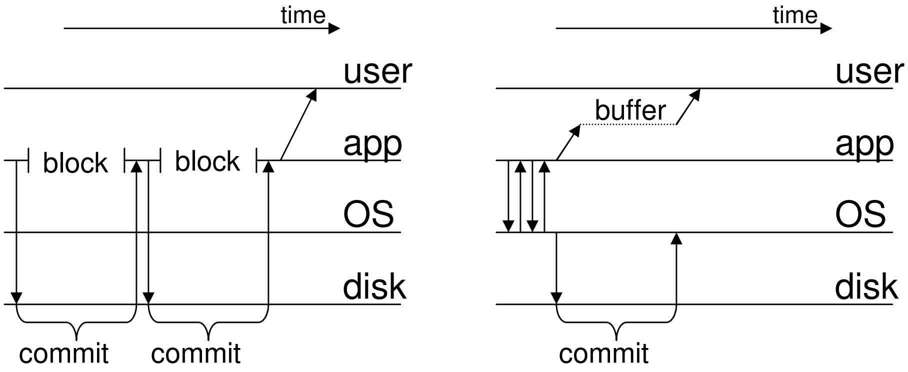
The design of external synchrony is based on two principles. First, as discussed in introduction, I define externally synchronous I/O by its externally observable behavior rather than by its implementation. Second, I note that application state is an internal property of the computer system. Since application state is not directly observable by external entities, the operating system need not treat changes to application state as an external output.

Synchronous I/O is usually defined by its implementation: an I/O is considered synchronous if the calling application is blocked until after the I/O completes [66]. In contrast, I define externally synchronous I/O by its observable behavior: I say that an I/O is externally synchronous if the external output produced by the computer system cannot be distinguished from output that could have been produced if the I/O had been synchronous.

2.2.2 Correctness

While I have already established the general principles that allow a speculative abstraction to provide an equivalent guarantee to its synchronous counterpart, it is useful to provide a correctness argument within the context of synchronous I/O.

Figure 2.1 illustrates these principles as they relate to synchronous I/O by showing an example single-threaded application that makes two file system modifications and



(a) Synchronous I/O

(b) Externally synchronous I/O

This figure shows the behavior of a sample application that makes two file system modifications, then displays output to an external device. The diagram on the left shows how the application executes when its file I/O is synchronous; the diagram on the right shows how it executes when its file I/O is externally synchronous.

Figure 2.1: Example of externally synchronous file I/O

writes some output to the screen. In the diagram on the left, the file modifications made by the application are synchronous. Thus, the application blocks until each modification commits.

I say that external output of an externally synchronous system is equivalent to the output of a synchronous one if (a) the values of the external outputs are the same, and (b) the outputs occur in the same causal order, as defined by Lamport's *happens before* relation [35]. I consider disk commits external output because they change the stable image of the file system. If the system crashes and reboots, the change to the stable image is visible. Since the operating system cannot control when crashes occur, it must treat disk commits as external output. Thus, in Figure 2.1(a), there are three external outputs: the two commits and the message displayed on the screen.

An externally synchronous file I/O returns the same result to applications that would have been returned by a synchronous I/O. The file system does all processing that would be done for a synchronous I/O, including validation and changing the volatile (in-memory) state of the file system, except that it does not actually commit the modification to disk before returning. Because the results that an application sees from an externally synchronous I/O are equivalent to the results it would have seen if the I/O had been synchronous, the external output it produces is the same in both cases.

An operating system that supports external synchrony must ensure that external output occurs in the same causal order that would have occurred had I/O been performed synchronously. Specifically, if an external output causally follows an externally synchronous file I/O, then that output cannot be observed before the file I/O has been committed to disk. In the example, this means that the second file modification made by the application cannot commit before the first, and that the screen output cannot be seen before both modifications commit.

2.2.3 Improving performance

The externally synchronous system in Figure 2.1(b) makes two optimizations to improve performance. First, the two modifications are group committed as a single file system transaction. Because the commit is atomic, the effects of the second modification are never seen unless the effects of the first are also visible. Grouping multiple modifications into one transaction has many benefits: the commit of all modifications is done with a single sequential disk write, writes to the same disk block are coalesced in the log, and no blocks are written to disk at all if data writes are closely followed by deletion. For example, ext3 employs value logging — when a transaction commits, only the latest version of each block is written to the journal. If a temporary file is created and deleted within a single transaction, none of its blocks are written to disk. In contrast, a synchronous file system cannot group multiple modifications for a single-threaded application because the application does not begin the second modification until after the first commits.

The second optimization is buffering screen output. The operating system must delay the externalization of the output until after the commit of the file modifications to obey the causal ordering constraint of externally synchronous I/O. One way to enforce this ordering would be to block the application when it initiates external output. However, the asynchronous nature of the output enables a better solution.

The operating system instead buffers the output and allows the process that generated the output to continue execution. After the modifications are committed to disk, the operating system releases the output to the device for which it was destined.

This design requires that the operating system track the causal relationship between file system modifications and external output. When a process writes to the file system, it inherits a commit dependency on the uncommitted data that it wrote. When a process with commit dependencies modifies another kernel object (process, pipe, file, UNIX socket, etc.) by executing a system call, the operating system marks the modified objects with the same commit dependencies. Similarly, if a process observes the state of another kernel object with commit dependencies, the process inherits those dependencies. If a process with commit dependencies executes a system call for which the operating system cannot track the flow of causality (e.g., an `ioctl`), the process is blocked until its file systems modifications have been committed. Any external output inherits the commit dependencies of the process that generated it — the operating system buffers the output until the last dependency is resolved by committing modifications to disk.

2.2.4 Deciding when to commit

An externally synchronous file system uses the causal relationship between external output and file modifications to trigger commits. There is a well-known tradeoff between throughput and latency for group commit strategies. Delaying a group commit in the hope that more modifications will occur in the near future can improve throughput by amortizing more modifications across a single commit. However, delaying a commit also increases latency — in my system, commit latency is especially important because output cannot be externalized until the commit occurs.

Latency is unimportant if no external entity is observing the result. Specifically, until some output is generated that causally depends on a file system transaction, committing the transaction does not change the observable behavior of the system. Thus, the operating system can improve throughput by delaying a commit until some output that depends on the transaction is buffered (or until some application that depends on the transaction blocks due to an `ioctl` or similar system call). I call this strategy *output-triggered commits* since the attempt to generate output that is causally dependent upon modifications to be written to disk triggers the commit of those modifications.

Output-triggered commits enable an externally synchronous file system to maximize throughput when output is not being displayed (for example, when it is piped to a file). However, when a user could be actively observing the results of a transaction, commit latency is small.

2.2.5 Limitations

One potential limitation of external synchrony is that it complicates application-specific recovery from catastrophic media failure because the application continues execution before such errors are detected. Although the kernel validates each modification before writing it to the file cache, the physical write of the data to disk may subsequently fail. While smaller errors such as a bad disk block are currently handled by the disk or device driver, a catastrophic media failure is rarely masked at these levels. Theoretically, a file system mounted synchronously could propagate such failures to the application. However, a recent survey of common file systems [56] found that write errors are either not detected by the file system (ext3, jbd, and NTFS) or induce a kernel panic (ReiserFS). An externally synchronous file system could propagate failures to applications by using Speculator to checkpoint a process before it modifies the file system. If a catastrophic failure occurs, the process would be rolled back and notified of the failure. I rejected this solution because it would both greatly increase the complexity of external synchrony and severely penalize its performance. Further, it is unclear that catastrophic failures are best handled by applications — it seems best to handle them in the operating system, either by inducing a kernel panic or (preferably) by writing data elsewhere.

Another limitation of external synchrony is that the user may have some temporal expectations about when modifications are committed to disk. As defined so far, an externally synchronous file system could indefinitely delay committing data written by an application with no external output. If the system crashes, a substantial amount of work could be lost. Xsyncfs therefore commits data every 5 seconds, even if no output is produced. The 5 second commit interval is the same value used by ext3 mounted asynchronously.

A final limitation of external synchrony is that modifications to data in two different file systems cannot be easily committed with a single disk transaction. Potentially, I could share a common journal among all local file systems, or I could implement a two-phase commit strategy. However, a simpler solution is to block a process with commit dependencies for one file system before it modifies data in a second. Speculator would map each dependency to a specific file system. When a process writes to a file system, Speculator would verify that the process depends only on the file system it is modifying; if it depends on another file system, Speculator would block it until its previous modifications commit.

2.3 Implementation

2.3.1 External synchrony

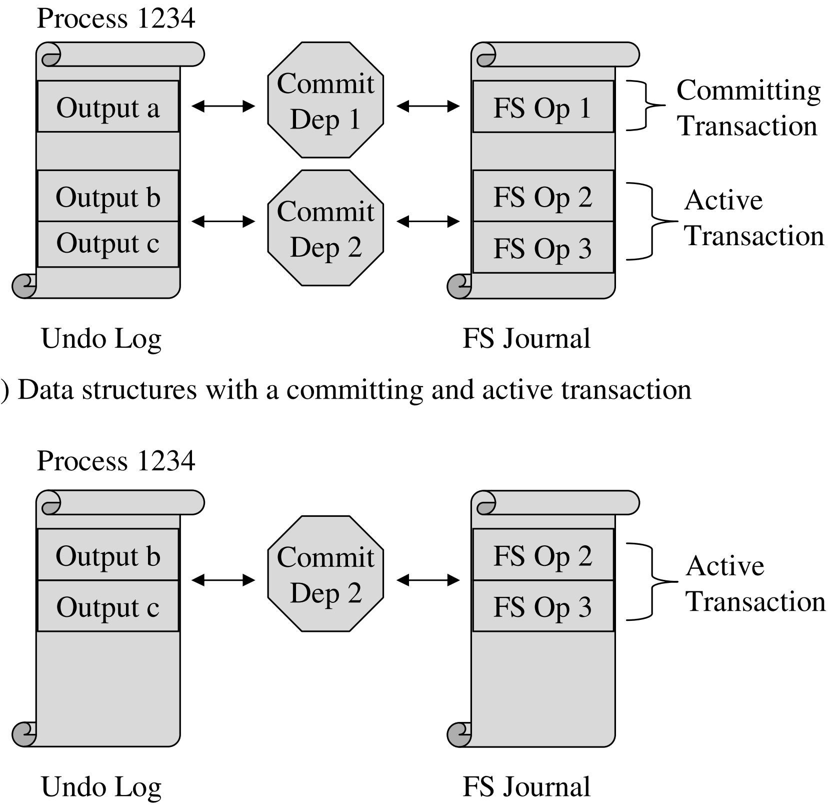
External synchrony is built around Speculator. It uses lightweight speculations to track causal dependencies with respect to disk I/O, and to buffer external output until the data upon which the output depends has been written to disk. In external synchrony, a commit dependency represents the causal relationship between kernel state and an uncommitted file system modification. Any kernel object that has one or more associated commit dependencies is referred to as *uncommitted*. Any external output from a process that is uncommitted is buffered within the kernel (using lightweight speculations) until the modifications on which the output depends have been committed. Thus, uncommitted output is never visible to an external observer.

When a process writes to an externally synchronous file system, Speculator marks the process as uncommitted. Speculator uses a new data structure, called an *undo log*, for both lightweight and regular speculations. When regular speculations are used, undo logs track the state needed to restore the state of a process or object to a point before a failed speculation was created (covered in detail in Chapter 3). Speculator creates an undo log for the uncommitted process and a commit dependency between the process and the uncommitted file system transaction that contains the modification. A link to the commit dependency is placed in the undo log of the process. Speculator maintains a many-to-many relationship between commit dependencies and undo logs. These relationships reveal which buffered output can be safely released when a commit dependency is discarded.

When the file system commits the transaction to disk, the commit dependency is removed. Once all commit dependencies for buffered output have been removed, Speculator releases that output to the external device to which it was written. When the last commit dependency for a process is discarded, Speculator marks the process as committed.

2.3.2 File system support for external synchrony

I modified ext3, a journaling Linux file system, to create xsyncfs. In its default *ordered* mode, ext3 writes only metadata modifications to its journal. In its *journaled* mode, ext3 writes both data and metadata modifications. Modifications from many different file system operations may be grouped into a single compound journal transaction that is committed atomically. Ext3 writes modifications to the *active* transaction — at most one transaction may be active at any given time. A commit of the active transaction is triggered when journal space is exhausted, an application performs an explicit synchronization operation such as `fsync`, or the oldest modification in the transaction is more than 5 seconds old. After the transaction starts to commit, the next modification triggers the creation of a new active transaction. Only one transaction may be committing at any given time, so the next transaction must wait for the commit of the prior transaction to finish before it commits.



(a) Data structures with a committing and active transaction

(b) Data structures after the first transaction commits

Figure 2.2: The external synchrony data structures

Figure 2.2 shows how the external synchrony data structures change when a process interacts with xsyncfs. In Figure 2.2(a), process 1234 has completed three file system operations, sending output to the screen after each one. Since the output after the first operation triggered a transaction commit, the two following operations were placed in a new active transaction. The output is buffered in the undo log; the commit dependencies maintain the relationship between buffered output and uncommitted data. In Figure 2.2(b), the first transaction has been committed to disk. Therefore, the output that depended upon the committed transaction has been released to the screen and the commit dependency has been discarded.

Xsyncfs uses journaled mode rather than the default ordered mode. This change guarantees ordering; specifically, the property that if an operation A causally precedes another operation B, the effects of B should never be visible unless the effects of A are also visible. This guarantee requires that B never be committed to disk before A. Otherwise, a system crash or power failure may occur between the two commits — in this case, after the system is restarted, B will be visible when A is not. Since journaled mode adds all modifications for A to the journal before the operation completes, those modifications must already be in the journal when B begins (since B causally follows A). Thus, either B is part of the same transaction as A (in which case the ordering property holds since A and B are committed atomically), or the transaction containing A is already committed before the transaction containing B starts to commit.

In contrast, the default mode in ext3 does not provide ordering since data modifications are not journaled. The kernel may write the dirty blocks of A and B to disk in any order as long as the data reaches disk before the metadata in the associated journal transaction commits. Thus, the data modifications for B may be visible after a crash without the modifications for A being visible.

Xsyncfs informs Speculator when a new journal transaction is created — this allows Speculator to track state that depends on the uncommitted transaction. Xsyncfs also informs Speculator when a new modification is added to the transaction and when the transaction commits.

As described in Section 2.1, the default behavior of ext3 does not guarantee that modifications are durable after a power failure. In the Linux 2.4 kernel, durability can be ensured only by disabling the drive cache. The Linux 2.6.11 kernel provides the option of using *write barriers* to flush the drive cache before and after writing each transaction commit record. Since Speculator runs on a 2.4 kernel, I ported write barriers to my kernel and modified xsyncfs to use write barriers to guarantee that all committed modifications are preserved, even on power failure.

2.3.3 Output-triggered commits

Xsyncfs uses the causal relationship between disk I/O and external output to balance the competing concerns of throughput and latency. Currently, ext3 commits

its journal every 5 seconds, which typically groups the commit of many file system operations. This strategy optimizes for throughput, a logical behavior when writes are asynchronous. However, latency is an important consideration in xsyncfs since users must wait to view output until the transactions on which that output depends commit. If xsyncfs were to use the default ext3 commit strategy, disk throughput would be high, but the user might be forced to wait up to 5 seconds to see output. This behavior is clearly unacceptable for interactive applications.

I therefore modified Speculator to support output-triggered commits. Speculator provides callbacks to xsyncfs when it buffers output or blocks a process that performed a system call for which it cannot track the propagation of causal dependencies (e.g., an `ioctl`). Xsyncfs uses the ext3 strategy of committing every 5 seconds unless it receives a callback that indicates that Speculator blocked or buffered output from a process that depends on the active transaction. The receipt of a callback triggers a commit of the active transaction.

Output-triggered commits adapt the behavior of the file system according to the observable behavior of the system. For instance, if a user directs output from a running application to the screen, latency is reduced by committing transactions frequently. If the user instead redirects the output to a file, xsyncfs optimizes for throughput by committing every 5 seconds. Optimizing for throughput is correct in this instance since the only event the user can observe is the completion of the application (and the completion would trigger a commit if it is a visible event). Finally, if the user were to observe the contents of the file using a different application, e.g., `tail`, xsyncfs would correctly optimize for latency because Speculator would track the causal relationship through the kernel data structures from `tail` to the transaction and provide callbacks to xsyncfs. When `tail` attempts to output data to the screen, Speculator callbacks will cause xsyncfs to commit the active transaction.

2.3.4 Rethinking sync

Asynchronous file systems provide explicit synchronization operations such as `sync` and `fdatasync` for applications with durability or ordering constraints. In a synchronous file system, such synchronization operations are redundant since ordering and durability are already guaranteed for all file system operations. However, in an externally synchronous file system, some extra support is needed to minimize latency. For instance, a user who types "sync" in a terminal would prefer that the command complete as soon as possible.

When xsyncfs receives a synchronization call such as `sync` from the VFS layer, it creates a commit dependency between the calling process and the active transaction. Since this does not require a disk write, the return from the synchronization call is almost instantaneous. If a visible event occurs, such as the completion of the `sync` process, Speculator will issue a callback that causes xsyncfs to commit the active transaction.

External synchrony simplifies the file system abstraction. Since xsyncfs requires no application modification, programmers can write the same code that they would write if they were using a unmodified file system mounted synchronously. They do not need explicit synchronization calls to provide ordering and durability since xsyncfs provides these guarantees by default for all file system operations. Further, since xsyncfs does not incur the large performance penalty usually associated with synchronous I/O, programmers do not need complicated group commit strategies to achieve acceptable performance. Group commit is provided transparently by xsyncfs.

Of course, a hand-tuned strategy might offer better performance than the default policies provided by xsyncfs. However, as described in Section 2.3.3, there are some instances in which xsyncfs can optimize performance when an application solution cannot. Since xsyncfs uses output-triggered commits, it knows when no external output has been generated that depends on the current transaction; in these instances, xsyncfs uses group commit to optimize throughput. In contrast, an application-specific commit strategy cannot determine the visibility of its actions beyond the scope of the currently executing process; it must therefore conservatively commit modifications before producing external messages.

For example, consider a client that issues two sequential transactions to a database server on the same computer and then produces output. Xsyncfs can safely group the commit of both transactions. However, the database server (which does not use output-triggered commits) must commit each transaction separately since it cannot know whether or not the client will produce output after it is informed of the commit of the first transaction.

2.3.5 Shared memory

Speculator can track causal dependencies because processes can only interact through the operating system. Usually, this interaction involves an explicit system call (e.g., `write`) that Speculator can intercept. However, when processes interact through shared memory regions, only the sharing and unsharing of regions is visible to the operating system. Thus, Speculator cannot readily intercept individual reads and writes to shared memory.

I considered marking a shared memory page inaccessible when a process with write permission inherits a commit dependency that a process with read permission does not have. This would trigger a page fault whenever a process reads or writes the shared page. If a process reads the page after another writes it, any commit dependencies would be transferred from the writer to the reader. Once these processes have the same commit dependencies, the page can be restored to its normal protections. I felt this mechanism would perform poorly because of the time needed to protect and unprotect pages, as well as the extra page faults that would be incurred.

Instead, I decided to use an approach that imposes less overhead but might transfer dependencies when not strictly necessary. I make a conservative assumption that processes with write permission for a shared memory region are continually writing to that region, while processes with read permission are continually reading it. When a process with write permission for a shared region inherits a new commit dependency, any process with read permission for that region atomically inherits the same dependency.

Speculator uses the same mechanism to track commit dependencies transfered through memory-mapped files. Similarly, Speculator is conservative when propagating dependencies for multi-threaded applications — any dependency inherited by one thread is inherited by all. Support for shared memory regions that are write-mapped is limited to lightweight speculations. Details about why Speculator does not yet support regular speculations and write-shared memory regions are provided in 3.5.10.

2.4 Evaluation

My evaluation answers the following questions:

- How does the durability of xsyncfs compare to current file systems?
- How does the performance of xsyncfs compare to current file systems?
- How does xsyncfs affect the performance of applications that synchronize explicitly?
- How much do output-triggered commits improve the performance of xsyncfs?

2.4.1 Methodology

All computers used in my evaluation have a 3.02 GHZ Pentium 4 processor with 1 GB of RAM. Each computer has a single Western Digital WD-XL40 hard drive, which is a 7200 RPM 120 GB ATA 100 drive with a 2 MB on-disk cache. The computers run Red Hat Enterprise Linux version 3 (kernel version 2.4.21). I use a 400 MB journal size for both ext3 and xsyncfs. For each benchmark, I measured ext3 executing in both journaled and ordered mode. Since journaled mode executed faster in every benchmark, I report only journaled mode results in this evaluation. Finally, I measured the performance of ext3 both using write barriers and with the drive cache disabled. In all cases write barriers were faster than disabling the drive cache since the drive cache improves read times and reduces the frequency of writes to the disk platter. Thus, I report only results using write barriers.

2.4.2 Durability

My first benchmark empirically confirms that without write barriers, ext3 does not guarantee durability. This result holds in both journaled and ordered mode, whether ext3 is mounted synchronously or asynchronously, and even if `fsync` commands are issued by the application after every write. Even worse, my results show that, despite the use of journaling in ext3, a loss of power can corrupt data and metadata stored in the file system.

I confirmed these results by running an experiment in which a test computer continuously writes data to its local file system. After each write completes, the test computer sends a UDP message that is logged by a remote computer. During the experiment, I cut power to the test computer. After it reboots, I compare the state of its file system to the log on the remote computer.

My goal was to determine when each file system guarantees durability and ordering. I say a file system fails to provide durability if the remote computer logs a message for a write operation, but the test computer is missing the data written by that operation. In this case, durability is not provided because an external observer (the remote computer) saw output that depended on data that was subsequently lost. I say a file system fails to provide ordering if the state of the file after reboot violates the temporal ordering of writes. Specifically, for each block in the file, ordering is violated if the file does not also contain all previously-written blocks.

For each configuration shown in Figure 2.3, I ran four trials of this experiment: two in journaled mode and two in ordered mode. As expected, my results confirm that ext3 provides durability only when write barriers are used. Without write barriers, synchronous operations ensure only that modifications are written to the hard drive cache. If power fails before the modifications are written to the disk platter, those modifications are lost.

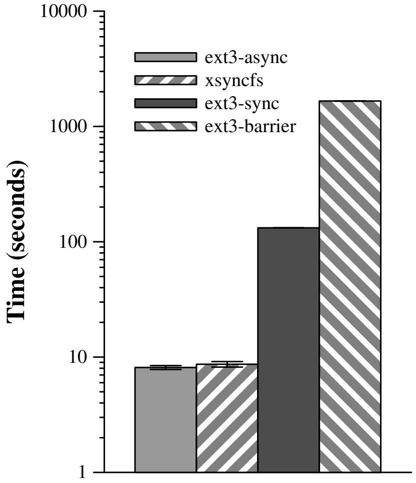
File system configuration	Data durable on `write`	Data durable on `fsync`
Asynchronous	No	Not on power failure
Synchronous	Not on power failure	Not on power failure
Synchronous w/ write barriers	Yes	Yes
External synchrony	Yes	Yes

This figure describes whether each file system provides durability to the user when an application executes a `write` or `fsync` system call. A "Yes" indicates that the file system provides durability if an OS crash or power failure occurs.

Figure 2.3: When is data safe?

Some of my experiments exposed a dangerous behavior in ext3: unless write barriers are used, power failures can corrupt both data and metadata stored on disk. In one experiment, a block in the file being modified was silently overwritten with garbage data. In another, a substantial amount of metadata in the file system, including the superblock, was overwritten with garbage. In the latter case, the test machine failed to reboot until the file system was manually repaired. In both cases, corruption is caused by the commit block for a transaction being written to the disk platter before all data blocks in that transaction are written to disk. Although the operating system wrote the blocks to the drive cache in the correct order, the hard drive reorders the blocks when writing them to the disk platter. After this happens, the transaction is committed during recovery even though several data blocks do not contain valid data. Effectively, this overwrites disk blocks with uninitialized data.

My results also confirm that ext3 without write barriers writes data to disk out of order. Journaled mode alone is insufficient to provide ordering since the order of writing transactions to the disk platter may differ from the order of writing transactions to the drive cache. In contrast, ext3 provides both durability and ordering when write barriers are combined with some form of synchronous operation (either mounting the file system synchronously or calling `fsync` after each modification). If write barriers are not available, the equivalent behavior could also be achieved by disabling the hard drive cache.



This figure shows the time to run the PostMark benchmark — the y-axis is logarithmic. Each value is the mean of 5 trials — the (relatively small) error bars are 90% confidence intervals.

Figure 2.4: The PostMark file system benchmark

The last row of Figure 2.3 shows results for xsyncfs. As expected, xsyncfs provides both durability and ordering.

2.4.3 The PostMark benchmark

I next ran the PostMark benchmark, which was designed to replicate the small file workloads seen in electronic mail, netnews, and web-based commerce [30]. I used PostMark version 1.5, running in a configuration that creates 10,000 files, performs 10,000 transactions consisting of file reads, writes, creates, and deletes, and then removes all files. The PostMark benchmark has a single thread of control that executes file system operations as quickly as possible. PostMark is a good test of file system throughput since it does not generate any output or perform any substantial computation.

Each bar in Figure 2.4 shows the time to complete the PostMark benchmark. The y-axis is logarithmic because of the substantial slowdown of synchronous I/O. The first bar shows results when ext3 is mounted asynchronously. As expected, this offers the best performance since the file system buffers data in memory up to 5 seconds before writing it to disk. The second bar shows results using xsyncfs. Despite the I/O intensive nature of PostMark, the performance of xsyncfs is within 7% of the performance of ext3 mounted asynchronously. After examining the performance of xsyncfs, we determined that the overhead of tracking causal dependencies in the kernel accounts for most of the difference.

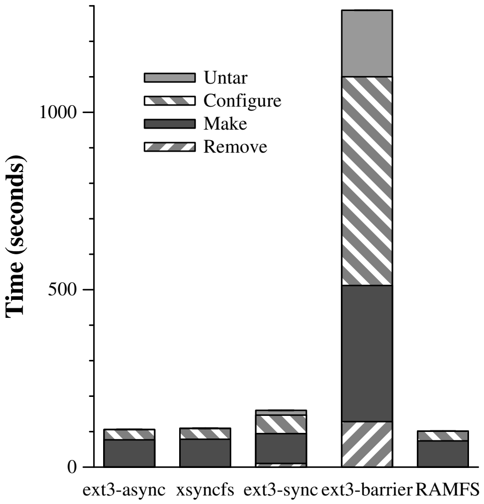
The third bar shows performance when ext3 is mounted synchronously. In this configuration the writing process is blocked until its modifications are committed to the drive cache. Ext3 in synchronous mode is over an order of magnitude slower than xsyncfs, even though xsyncfs provides stronger durability guarantees. Throughput is limited by the size of the drive cache; once the cache fills, subsequent writes block until some data in the cache is written to the disk platter.

The last bar in Figure 2.4 shows the time to complete the benchmark when ext3 is mounted synchronously and write barriers are used to prevent data loss when a power failure occurs. Since write barriers synchronously flush the drive cache twice for each file system transaction, ext3's performance is over two orders of magnitude slower than that of xsyncfs.

Due to the high cost of durability, high end storage systems sometimes use specialized hardware such as a non-volatile cache to improve performance [25]. This eliminates the need for write barriers. However, even with specialized hardware, I expect that the performance of ext3 mounted synchronously would be no better than the third bar in Figure 2.4, which writes data to a volatile cache. Thus, use of xsyncfs should still lead to substantial performance improvements for synchronous operations even when the hard drive has a non-volatile cache of the same size as the volatile cache on the drive.

2.4.4 The Apache build benchmark

I next ran a benchmark in which I 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. The Apache build benchmark requires the file



This figure shows the time to run the Apache build benchmark. Each value is the mean of 5 trials — the (relatively small) error bars are 90% confidence intervals.

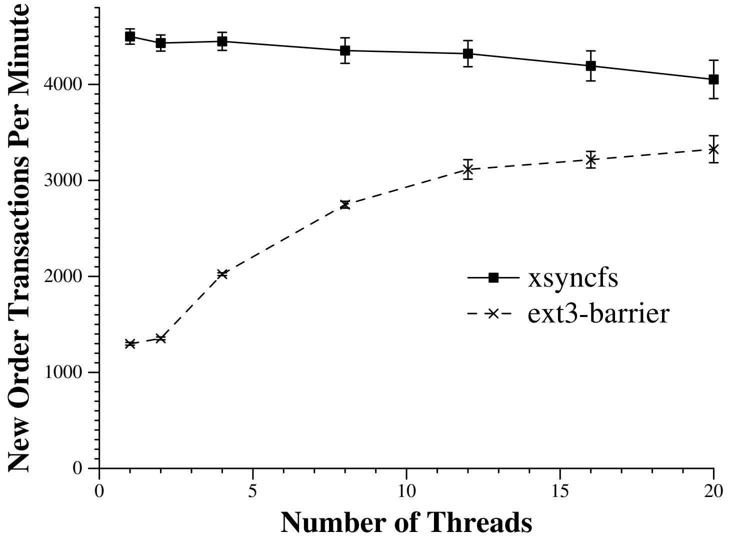
Figure 2.5: The Apache build benchmark

system to balance throughput and latency; it displays large amounts of screen output interleaved with disk I/O and computation.

Figure 2.5 shows the total amount of time to run the benchmark, with shadings within each bar showing the time for each stage. Comparing the first two bars in the graph, xsyncfs performs within 3% of ext3 mounted asynchronously. Since xsyncfs releases output as soon as the data on which it depends commits, output appears promptly during the execution of the benchmark.

For comparison, the bar at the far right of the graph shows the time to execute the benchmark using a memory-only file system, RAMFS. This provides a lower bound on the performance of a local file system, and it isolates the computation requirements of the benchmark. Removing disk I/O by running the benchmark in RAMFS improves performance by only 8% over xsyncfs because the remainder of the benchmark is dominated by computation.

The third bar in Figure 2.5 shows that ext3 mounted in synchronous mode is 46% slower than xsyncfs. Since computation dominates I/O in this benchmark, any difference in I/O performance is a smaller part of overall performance. The fourth bar shows that ext3 mounted synchronously with write barriers is over 11 times slower



This figure shows the New Order Transactions Per Minute when running a modified TPC-C benchmark on MySQL with varying numbers of clients. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 2.6: The MySQL benchmark

than xsyncfs. If I isolate the cost of I/O by subtracting the cost of computation (calculated using the RAMFS result), ext3 mounted synchronously is 7.5 times slower than xsyncfs while ext3 mounted synchronously with write barriers is over two orders of magnitude slower than xsyncfs. These isolated results are similar to the values that I saw for the PostMark experiments.

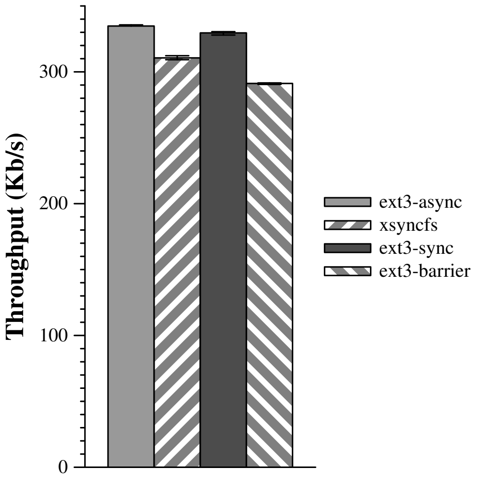
2.4.5 The MySQL benchmark

I was curious to see how xsyncfs would perform with an application that implements its own group commit strategy. I therefore ran a modified version of the OSDL TPC-C benchmark [55] using MySQL version 5.0.16 and the InnoDB storage engine. Since both MySQL and the TPC-C benchmark client are multi-threaded, this benchmark measures the efficacy of xsyncfs's support for shared memory. TPC-C measures the New Order Transactions Per Minute (NOTPM) a database can process for a given number of simultaneous client connections. The total number of transactions performed by TPC-C is approximately twice the number of New Order Transactions. TPC-C requires that a database provide ACID semantics, and MySQL requires either disabling the drive cache or using write barriers to provide durability. Therefore, we compare xsyncfs with ext3 mounted asynchronously using write barriers. Since the client ran on the same machine as the server, we modified the benchmark to use UNIX sockets. This allows xsyncfs to propagate commit dependencies between the client and server on the same machine. In addition, I modified the benchmark to saturate the MySQL server by removing any wait times between transactions and creating a data set that fits completely in memory.

Figure 2.6 shows the NOTPM achieved as the number of clients is increased from 1 to 20. With a single client, MySQL completes 3 times as many NOTPM using xsyncfs. By propagating commit dependencies to both the MySQL server and the requesting client, xsyncfs can group commit transactions from a single client, significantly improving performance. In contrast, MySQL cannot benefit from group commit with a single client because it must conservatively commit each transaction before replying to the client.

When there are multiple clients, MySQL can group the commit of transactions from different clients. As the number of clients grows, the gap between xsyncfs and ext3 mounted asynchronously with write barriers shrinks. With 20 clients, xsyncfs improves TPC-C performance by 22%. When the number of clients reaches 32, the performance of ext3 mounted asynchronously with write barriers matches the performance of xsyncfs. From these results, I conclude that even applications such as MySQL that use a custom group commit strategy can benefit from external synchrony if the number of concurrent transactions is low to moderate.

Although ext3 mounted asynchronously without write barriers does not meet the durability requirements for TPC-C, I were still curious to see how its performance compared to xsyncfs. With only 1 or 2 clients, MySQL executes 11% more NOTPM with xsyncfs than it executes with ext3 without write barriers. With 4 or more clients, the two configurations yield equivalent performance within experimental error.



This figure shows the mean throughput achieved when running the SPECweb99 benchmark with 50 simultaneous connections. Each result is the mean of three trials, with error bars showing the highest and lowest result.

Figure 2.7: Throughput in the SPECweb99 benchmark

2.4.6 The SPECweb99 benchmark

Since my previous benchmarks measured only workloads confined to a single computer, I also ran the SPECweb99 [72] benchmark to examine the impact of external synchrony on a network-intensive application. In the SPECweb99 benchmark, multiple clients issue a mix of HTTP GET and POST requests. HTTP GET requests are issued for both static and dynamic content up to 1 MB in size. A single client, emulating 50 simultaneous connections, is connected to the server, which runs Apache 2.0.48, by a 100 Mb/s Ethernet switch. As we use the default Apache settings, 50 connections are sufficient to saturate my server.

I felt that this benchmark might be especially challenging for xsyncfs since sending a network message externalizes state. Since xsyncfs only tracks causal dependencies on a single computer, it must buffer each message until the file system data on which that message depends has been committed. In addition to the normal log data written by Apache, the SPECweb99 benchmark writes a log record to the file system as a result of each HTTP POST. Thus, small file writes are common — a typical 45 minute run has approximately 150,000 file system transactions.

Request size	ext3-async	xsyncfs
0–1 KB	0.064 (±0.025)	0.097 (±0.002)
1–10 KB	0.150 (±0.034)	0.180 (±0.001)
10–100 KB	1.084 (±0.052)	1.094 (±0.003)
100–1000 KB	10.253 (±0.098)	10.072 (±0.066)

The figure shows the mean time (in seconds) to request a file of a particular size during three trials of the SPECweb99 benchmark with 50 simultaneous connections. 90% confidence intervals are given in parentheses.

Figure 2.8: SPECweb99 latency results

As shown in Figure 2.7, SPECweb99 throughput using xsyncfs is within 8% of the throughput achieved when ext3 is mounted asynchronously. In contrast to ext3, xsyncfs guarantees that the data associated with each POST request is durable before a client receives the POST response. The third bar in Figure 2.7 shows that SPECweb99 using ext3 mounted synchronously achieves 6% higher throughput than xsyncfs. Unlike the previous benchmarks, SPECweb99 writes little data to disk, so most writes are buffered by the drive cache. The last bar shows that xsyncfs achieves 7% better throughput than ext3 mounted synchronously with write barriers.

Figure 2.8 summarizes the average latency of individual HTTP requests during benchmark execution. On average, use of xsyncfs adds no more than 33 ms of delay to each request — this value is less than the commonly cited perception threshold of 50 ms for human users [19]. Thus, a user should perceive no difference in response time between xsyncfs and ext3 for HTTP requests.

2.4.7 Benefit of output-triggered commits

To measure the benefit of output-triggered commits, I also implemented an *eager commit* strategy for xsyncfs that triggers a commit whenever the file system is modified. The eager commit strategy still allows for group commit since multiple modifications are grouped into a single file system transaction while the previous transaction is committing. The next transaction will only start to commit once the

Benchmark	Eager Commits	Output-Triggered	Speedup
PostMark (seconds)	9.879 (±0.056)	8.668 (±0.478)	14%
Apache (seconds)	111.41 (±0.32)	109.42 (±0.71)	2%
MySQL 1 client (NOTPM)	3323 (±60)	4498 (±73)	35%
MySQL 20 clients (NOTPM)	3646 (±217)	4052 (±200)	11%
SPECweb99 (Kb/s)	312 (±1)	311(±2)	0%

This figure compares the performance of output-triggered commits with an eager commit strategy. Each result shows the mean of 5 trials, except SPECweb99, which is the mean of 3 trials. 90% confidence intervals are given in parentheses.

Figure 2.9: Benefit of output-triggered commits

commit of the previous transaction completes. The eager commit strategy attempts to minimize the latency of individual file system operations.

I executed the previous benchmarks using the eager commit strategy. Figure 2.9 compares results for the two strategies. The output-triggered commit strategy performs better than the eager commit strategy in every benchmark except SPECweb99, which creates so much output that the eager commit and output-triggered commit strategies perform very similarly. Since the eager commit strategy attempts to minimize the latency of a single operation, it sacrifices the opportunity to improve throughput. In contrast, the output-triggered commit strategy only minimizes latency after output has been generated that depends on a transaction; otherwise it maximizes throughput.

2.5 Summary

In this chapter, I have used Speculator to implement an externally synchronous file system, which preserves the simplicity and reliability of synchronous I/O, and performs approximately as well as asynchronous I/O. Based on these results, I believe that externally synchronous file systems such as xsyncfs can provide a better foundation for the construction of reliable software systems.

CHAPTER 3

Speculative Execution in a Distributed File System

The last chapter showed the utility of lightweight speculations to convert the synchronous local file system abstraction into a one that is speculative. This chapter explores the use of speculations that save checkpoint state as a speculative process executes, which enables Speculator to rollback speculative state when a speculation fails. This allows a process to speculate on the correctness, rather than simply the success, of an operation. This change is needed since state is shared across many clients; thus, an operation is not known to be correct until confirmation is received from a remote server.

3.1 Introduction

Distributed file systems often perform substantially worse than local file systems because they perform synchronous I/O operations for cache coherence and data safety. File systems such as AFS [27] and NFS [6] 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 my results show, even these weaker semantics are time-consuming.

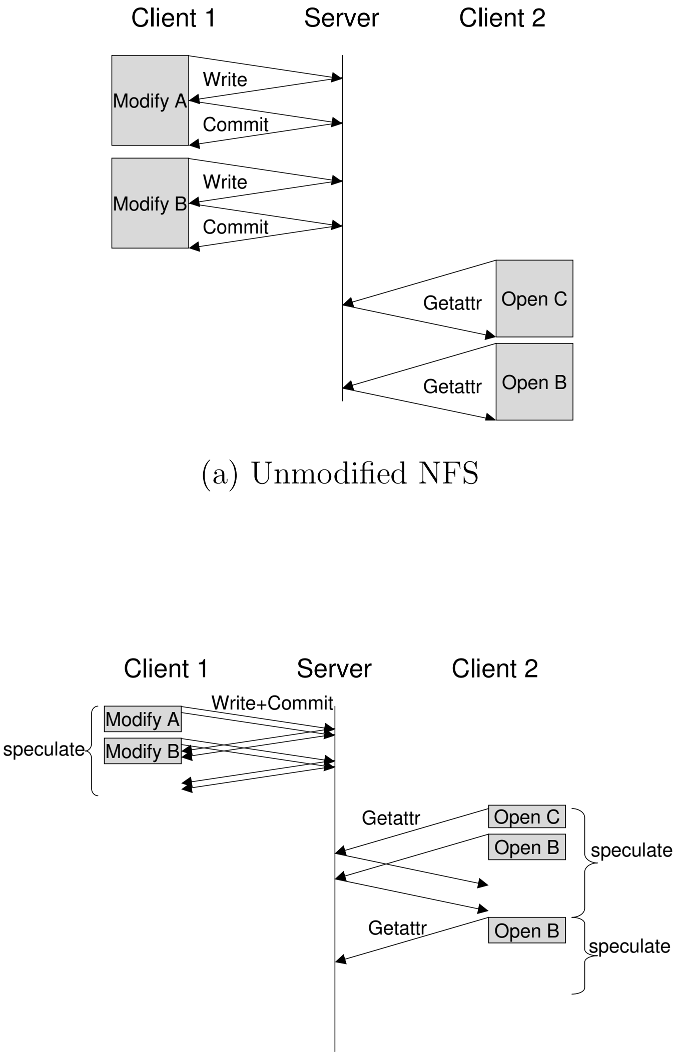
I demonstrate that, with operating system support for lightweight checkpointing, speculative execution, and tracking of causal interdependencies between processes, distributed file systems can be fast, safe, and consistent. Rather than block a process while waiting for the result of a remote communication 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.

My 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, I have added support for speculative execution, which I call Speculator, to the Linux kernel.

In my 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, e.g., 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, e.g., 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 non-speculative 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 non-speculative 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. I have also created a version of the Blue File System [52] 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, my 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 non-speculative NFS over a LAN and more than 11 times faster with a 30 ms delay.



(a) Unmodified NFS

(b) Speculative NFS

Figure 3.1: Example of speculative execution for NFS

3.2 Motivation: Speculation in NFS

Figure 3.1 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 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 receiving replies for the `write` RPCs.

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 3.1(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.

My speculative version of NFS is shown in Figure 3.1(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.

3.3 An interface for speculation

My 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 non-speculative processes.

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

Figure 3.2: Speculator interface

Speculator is implemented as part of the core Linux 2.4.21 kernel. Figure 3.2 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, e.g., a distributed file system, need not concern itself with the details of how speculative execution is performed. The flags parameter allows a process to use lightweight speculations, which do not provide the ability to rollback state should a speculation fail, which were used in the previous chapter to implement external synchrony.

The next section describes my basic implementation of speculative execution in the Linux kernel which allows processes to execute speculatively in isolation. Section 3.5 extends this implementation by allowing multiple processes to share speculative state. Section 3.6 describes how distributed file systems use Speculator.

3.4 Implementing speculation

3.4.1 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.

3.4.2 Speculation

Speculator adds two 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. I 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 non-speculative) or if any speculation fails (in which case, the object is rolled back to a prior state).

When a previously non-speculative 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. It may also prove useful in the future 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 my performance results indicate it is not needed.

3.4.3 Ensuring correct speculative execution

The previous chapter showed correctness properties for lightweight speculations within the context of external synchrony. In this chapter, I demonstrate that speculations that log state are correct within the context of distributed file systems.

I 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 non-speculative process tries to view speculative state, Speculator either blocks the process until the state becomes non-speculative, 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.

I 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. My approach to developing Speculator was to first create a correct but slow implementation that blocked whenever a speculative process performed a system call. I 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 I created to block the calling process until its speculations are resolved.

Next, I observed that system calls that do not modify state (e.g., `getpid`) are correct if performed by a speculative process. I let speculative processes perform these calls by replacing `sys_spec_deny` entries with the addresses of the functions that implement these syscalls. I 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 3.5.8). 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`.

I 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 non-speculative 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. My speculative versions of NFS and BlueFS set the read/write flag. Because file systems that I 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.

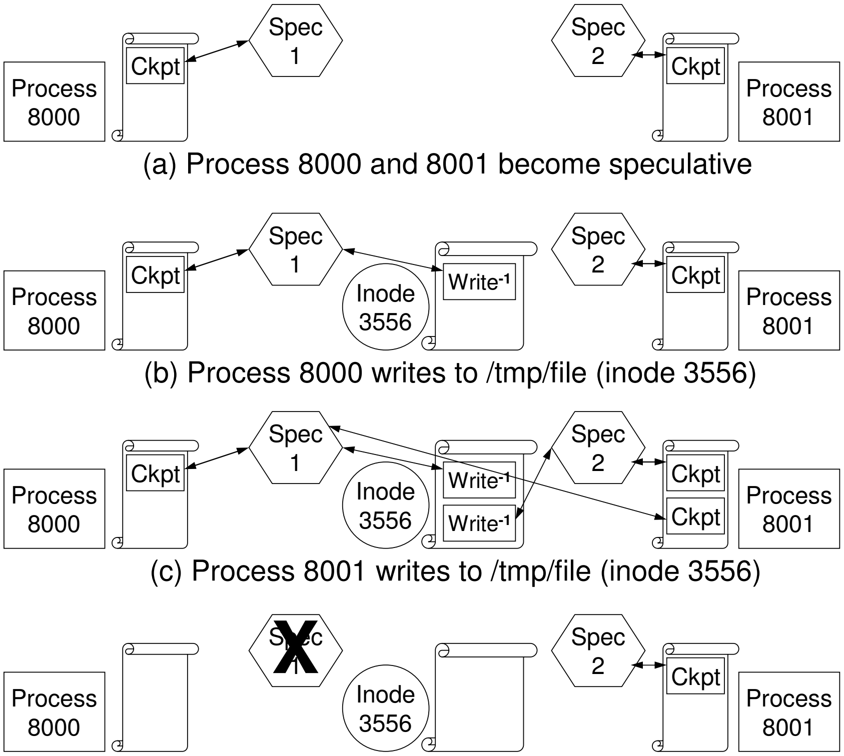
Output from a speculative process can appear before all of the speculations for that process are resolved. Consider a process that speculates on a remote operation, outputs a message, and then performs another speculative operation. Since the output depends only on the first speculation, Speculator can deliver it to the output device once the first speculation succeeds—Speculator need not wait for the second speculation to succeed or fail. In a non-speculative 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 non-speculative and speculative systems. In fact, output in the speculative system often appears *faster* than in the non-speculative 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.

3.5 Multi-process speculation

I next allowed speculative processes to participate in inter-process 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 I 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.

3.5.1 Generic strategy

Speculator's strategy for IPC allows selected data structures within the kernel to have speculative state. Figure 3.3 illustrates how speculative state is propagated. In Figure 3.3(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



(a) Process 8000 and 8001 become speculative

(b) Process 8000 writes to /tmp/file (inode 3556)

(c) Process 8001 writes to /tmp/file (inode 3556)

(d) Speculation 1 fails

Figure 3.3: Propagating causal dependencies

marked as speculative, associated with an 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 3.3(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 3.3(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 my 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 non-speculative.

Figure 3.3(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. I next describe the type-specific logic for each form of IPC that Speculator currently supports.

3.5.2 Objects in a distributed file system

In my 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.

3.5.3 Objects in a local memory file system

I 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 in the inode's undo log that describes the inverse operation needed during rollback. For instance, a `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.

3.5.4 Objects in a local disk file system

I 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, I felt a simpler solution is to never write speculative data to disk (i.e., use a *no-steal* policy [22]). 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, e.g., 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 my previously described policy, any non-speculative process performing an ext3 operation would become speculative since it would observe speculative state in the superblock.

I 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 non-speculative 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 non-speculative state. If the operation rolls back, the undo function is applied to the speculative state. Only the actual buffer containing non-speculative 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 non-speculative operations. To preserve the invariant that no speculative state is written to disk, I initially decided to block the commit of a compound transaction until all of its operations became non-speculative. However, this proved too restrictive since there are many instances in the ext3 code that wait for the current transaction to commit. I 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 non-speculative 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 non-speculative version of the buffer as part of the current transaction. This approach lets the journal immediately commit non-speculative operations without writing speculative data to disk.

3.5.5 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.

3.5.6 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.

3.5.7 Signals

Signals proved difficult to handle. My 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 non-idempotent 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. I 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 non-speculative 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, I have ensured that these system calls can be restarted correctly if a speculation on which the delivered signal depends fails.

3.5.8 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.

3.5.9 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 sub-tasks. Without this support, the parent process would block until its child's speculations are resolved.

3.5.10 Other forms of IPC

Speculator does not currently let speculative processes communicate via System V IPC, futexes, or shared memory (unless lightweight speculations are used). 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 multi-threaded applications from speculating. To support multi-threaded applications, I would need to rethink the way checkpoints are taken; using `fork` to take a checkpoint of a multi-threaded application is insufficient since a forked multi-threaded application contains only a single thread, and the state of locks within the application may be undefined. I do not believe these limitations would preclude the support of multi-threaded applications within Speculator, but it would require significant engineering work.

3.6 Using speculation

Modifying a distributed file system to use Speculator typically involves changes to the client, server, and network protocol. I 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 has not been modified after it was cached.

I 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 non-speculative, 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, I address two generic issues that occur when modifying a file system to use Speculator: correctly supporting mutating operations and implementing group commit. Sections 3.6.3 and 3.6.4 describe my speculative implementations of NFS and BlueFS in more detail. Section 3.6.5 discusses how other file system might benefit from using Speculator.

3.6.1 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 [28]. 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, I 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, I 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). I 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 I 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, I plan to block any process that attempts to write to a server while it has uncommitted speculations that depend on another server.

3.6.2 Speculative group commit

My original intent in using speculative execution was to hide the latency associated with remote communication. However, I 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. A prime example of this is group commit.

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 in-flight 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.

I 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.

3.6.3 Network File System

In order to explore how much speculative execution can improve the performance of existing distributed file systems, I modified NFS version 3 to use Speculator. My modified version, called SpecNFS, preserves existing NFS semantics, including close-to-open consistency. It issues the same RPCs that the non-speculative version issues; however, in SpecNFS, many of these RPCs are speculative and contain the additional hypothesis data described in Section 3.6.1.

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, non-speculative `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 non-deterministically, making it difficult for the client to predict which values the server will choose. The SpecNFS client chooses aliases for these 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.

3.6.4 Blue File System

I next explored whether speculative execution enables a distributed file system to provide strong consistency and safety, yet still have reasonable performance. Since I am currently developing a new distributed file system, BlueFS [52], I had the opportunity to develop a clean-sheet design that used Speculator. I 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, i.e., 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. My 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.

My 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; my 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

3.6.5 Discussion: Other file systems

While I 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 [27] and Coda [32] 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 [65] provide similar functionality. SFS [42] also extends the NFS protocol with callbacks and leases on file attributes. Echo [4] 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, I 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.

File system designers have had to choose between strong consistency guarantees, strong safety guarantees, and good performance. Speculative execution changes this equation by eliminating synchronous communication. As my BlueFS results will demonstrate: a distributed file system using Speculator can outperform current file systems while providing single-copy semantics and the safety of synchronous I/O.

3.7 Evaluation

My evaluation answers two questions:

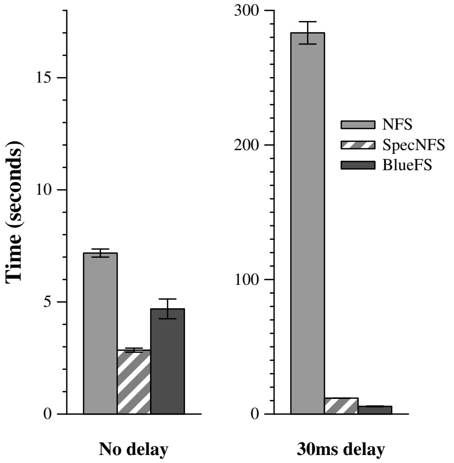
- How much does Speculator improve the performance of an existing file system (NFS)?
- With Speculator, what is the performance impact of providing single-copy file semantics and synchronous I/O (in BlueFS)?

3.7.1 Methodology

I 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. I run RedHat Enterprise Linux release 3 with kernel version 2.4.21. To insert delays, I route packets through a Dell Optiplex GX270 desktop running the NISTnet [7] 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, I run the non-speculative version of NFS with both UDP and TCP. Although results were roughly equivalent, we always report the best of the two results for non-speculative 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.

I 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.

Figure 3.4: PostMark file system benchmark

3.7.2 PostMark

I first ran the PostMark benchmark, which was designed to replicate the small-file workloads seen in electronic mail, netnews, and web-based commerce [30]. I used PostMark version 1.5, running in its default configuration that creates 500 files, performs 500 transactions consisting of file reads, writes, creates, and deletes, and then removes all files.

The left graph in Figure 3.4 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.

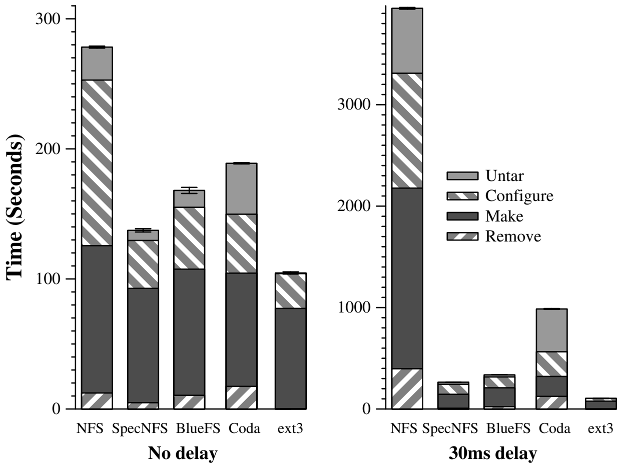
The right graph in Figure 3.4 shows results with a 30 ms delay. The adverse impact of latency on NFS is apparent by the difference in scales between the two graphs: NFS without speculation is 41 times slower with 30 ms round-trip time than in a LAN environment. In contrast, SpecNFS is much less affected 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 non-speculative 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 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, I could only have a maximum of 160 asynchronous RPCs in flight.

3.7.3 Apache build benchmark

I next ran a benchmark in which I 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 sub-tasks—these processes communicate via signals, pipes, and files in the /tmp directory. Thus, propagating speculative kernel state is important during this benchmark.

In Figure 3.5, 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



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.

Figure 3.5: Apache build benchmark

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 high without speculation since each 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 non-speculative version of NFS, SpecNFS learns the correct file handle when it issues a non-speculative `lookup` on re-execution.

BlueFS is 66% faster than non-speculative 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, I ran the benchmark using version 6 of the Coda distributed file system; I 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 I would need to implement a speculative version of Coda to determine precisely how much Speculator would improve its performance, my existing results provide some insight. Since Coda's use of callbacks results in fewer synchronous RPCs than NFS, I suspect that a speculative version of Coda would perform better than SpecNFS.

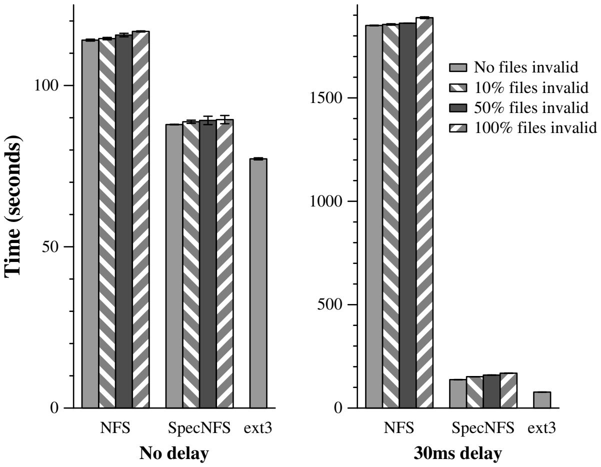
3.7.4 The cost of rollback

In the previous two benchmarks, almost all speculations made by the distributed file system are correct. I was therefore curious to measure the performance impact of having speculations fail. I 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, I 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. I 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.

Non-speculative 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. 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 3.6 shows `make` time when approximately 10%, 50% and 100% of the files



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.

Figure 3.6: Measuring the cost of rollback

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 non-speculative 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 3.7 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. I was 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 do not cause a rollback. This

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

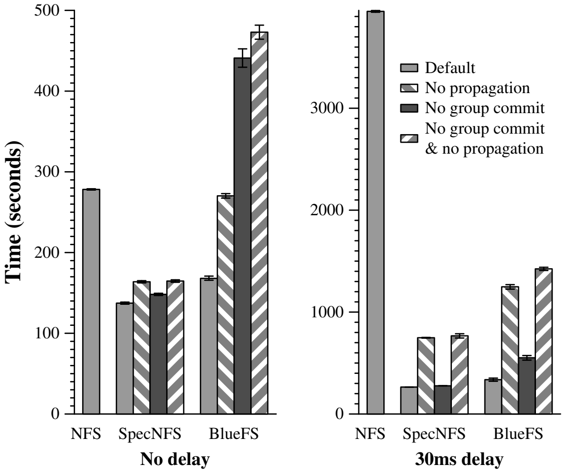
This figure shows the average number of speculations that roll back and commit during the `make` stage of the Apache benchmark when different numbers of source files are initially out-of-date in the client cache.

Figure 3.7: Rollback benchmark detail

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 [10].

3.7.5 Impact of group commit and sharing state

Figure 3.8 isolates the impact of two ideas presented in this paper. The left bar of each graph shows the time needed by the non-speculative version of NFS to perform all four stages of the Apache benchmark described in Section 3.7.3. The next two datasets show the performance of SpecNFS and BlueFS. For each file system, I show the default configuration (with full support for speculative execution), and three configurations that omit various parts of Speculator.



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.

Figure 3.8: Measuring components of speculation

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.

3.8 Summary

In this chapter I demonstrate that by converting a distributed file system's synchronous consistency and safety protocols into ones that are speculative performance is drastically improved. In addition, I built a new distributed file system that provides strong safety and consistency guarantees and is faster than existing distributed file systems.

CHAPTER 4

Parallelizing Security Checks

In this chapter I demonstrate that, by using Speculator, runtime security checks can not only be run asynchronously, but be parallelized across many cores.

4.1 Introduction

A runtime *security check* secures a system by instrumenting an application to either detect an intrusion or prevent an attack from succeeding. Security checks occur at a variety of granularities. For instance, on-access virus scanners [45] execute during each read and write to disk, call-graph modeling [79] executes during each system call, and security checks such as taint analysis [50], dynamic data-flow analysis [15], and data-flow graph enforcement [8] run before the execution of a single instruction.

To detect an attack, a security check impedes the critical path of an application by blocking it until the check completes. A security check could be executed outside the critical path, but detecting an attack after damage has been done is of little use – once an attack succeeds, it is often very difficult to undo its effects.

Unfortunately, executing powerful security checks in the critical path drastically slows down performance. For example, a security check such as taint analysis can slow down application execution by an order of magnitude or more [50].

Therefore, security researchers who wish to guard against intrusions are faced with a conundrum. Powerful security checks may cause an application to execute too slowly, but weaker checks may not detect an attack.

This paper describes a system, called Inspector, that accelerates powerful security checks on commodity hardware by executing them in parallel on multiple cores. Inspector provides an infrastructure that allows *sequential* invocations of a particular security check to run in parallel without sacrificing the safety of the system.

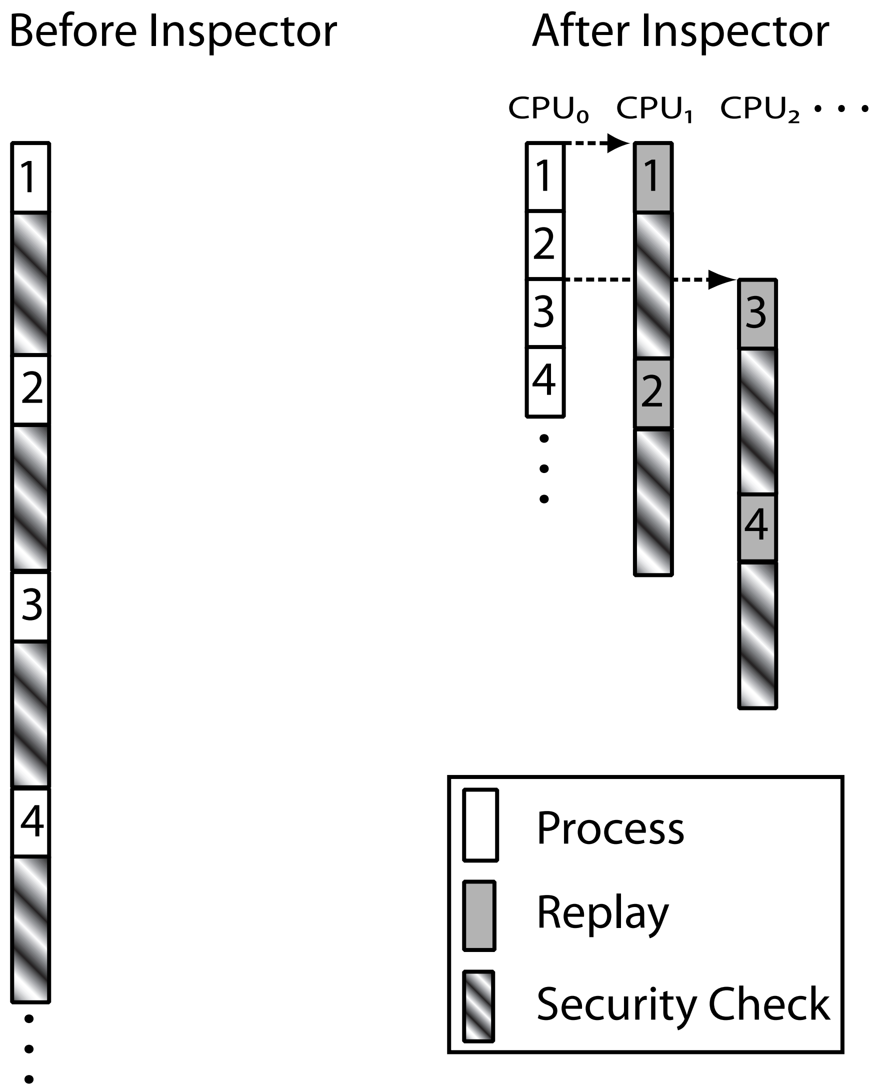
Inspector creates parallelism in two ways. First, Inspector decouples a security check from an application by continuing the application, using Speculator, while a security check executes in parallel on another core. Second, Inspector takes advantage of the application executing ahead to start and run later checks in parallel to earlier ones. Thus, Inspector creates parallelism not only between a process and a security check, but also between the sequential invocations of a security check.

Inspector uses three techniques to safely defer and parallelize security checks. First, Inspector uses Speculator to execute a process speculatively and to track and propagate its causal dependencies. Should a security check fail, Speculator rolls back each object and process within the operating system that depends upon the compromised application. Speculator supports speculative execution throughout an operating system, which allows a process to interact speculatively with the kernel, file system, and other processes.

Second, Inspector buffers external output that depends upon a speculative process, such as output to the network or the screen, until the security checks upon which the output depends have succeeded.

Finally, Inspector provides a replay system that logs sources of non-determinism as the speculative process executes. When the security check executes in parallel, Inspector replays each logged source of non-determinism to ensure that the code executing in parallel does not diverge from the speculative process.

I Inspector within the Linux 2.4 and 2.6 kernels, and I implemented parallel versions of three security checks that operate on different types of state: system calls, memory contents, and taint propagation. My results to date are promising. On an 8-core computer, Inspector improves performance by 7.6x for the system call check, 7x for the memory content check, and 2x for the taint propagation check.



The left-hand side shows a process executing with a security check interrupting execution to check for an intrusion. The right-hand side of the figure shows how Inspector parallelizes security checks. On CPU$_0$, the original process runs, using speculative execution, without a security check. Inspector forks copies of the process, and uses replay to aggregate and execute checks in parallel on multiple cores.

Figure 4.1: Parallelizing a sequential security check

4.2 Design overview

4.2.1 Inspector

Inspector's goal is to provide security equivalent to that provided by security checks executing sequentially within the critical path of an application, while achieving better performance through parallelism. The security provided by a parallel check using Inspector is equivalent to that provided by a sequential version if the output of the parallel version could have been produced by the sequential version.

The left-hand side of Figure 4.1 shows a program (denoted by the numbered boxes) and a security check (striped boxes) executing sequentially. The right-hand side of the figure shows how Inspector parallelizes security checks on multiple cores. The process running on CPU_0 executes without a security check using operating system support for speculative execution. I refer to it as the *uninstrumented process*. The uninstrumented process invokes security checks that run in parallel on other cores. Inspector replays the uninstrumented process's execution (shaded boxes) on other cores to generate the state required for each security check to run. I refer to the processes that run security checks in parallel as *instrumented clones*.

The state used by a parallel check must be identical to the state it would have had were it executing sequentially. When it is time to begin a security check, Inspector uses `fork` as a general method to copy (i.e., copy-on-write) the state of the uninstrumented process and ship it to another core. Using a software mechanism such as `fork` allows Inspector to run on commodity hardware. Unfortunately, two non-trivial costs slow the start of a security check when using `fork`: first, the page table of the application must be copied and the pages must be protected. Second, during execution, both the uninstrumented and instrumented clone will take copy-on-write page faults.

Inspector amortizes the cost of `fork` by aggregating multiple security checks into intervals of time called epochs. Increasing the epoch length reduces the frequency of `fork` and thus reduces the overhead needed to parallelize the check. However, increasing the epoch length forces the uninstrumented process to execute further ahead to begin the next epoch. For many applications, a longer epoch increases the chances that the uninstrumented process will execute a system call. Allowing the uninstrumented process to execute system calls led to a three challenges that shaped the design of Inspector.

First, some system calls (e.g., `write`) allow a process to affect other processes or kernel state. It is safe to allow a speculative process to affect other processes and kernel objects as long as the affected state is rolled back when a security check fails. I use Speculator to track and undo all changes a speculative process makes to other processes and kernel objects. If a security check fails, Speculator rolls back each object and process that depends on the failed security check. Thus, a compromised program cannot permanently damage the system.

Second, other system calls allow a process to execute an output commit (output that cannot be undone, e.g., writing to the screen). When a process attempts an action that cannot be undone, the process must block until all security checks it depends on are resolved. Some output commits can be deferred, rather than blocked, which improves performance by allowing the speculative process to continue executing further ahead. For example, Speculator buffers output to the screen and the network until the output no longer depends upon speculative state. When Speculator cannot propagate a dependency or buffer output, the process is blocked until all security checks up to that point have passed.

Finally, a set of system calls exist that affect the state of the calling process (e.g., `read`). Executing one of these system calls introduces non-determinism into the execution of the uninstrumented process. The non-determinism could cause the instrumented clone to diverge from the path taken by the uninstrumented process and thereby cause the check to miss an attack experienced by the uninstrumented process. Inspector provides a transparent kernel-level replay system, which logs sources of non-determinism while the uninstrumented process executes ahead. Inspector replays each source of non-determinism, and therefore the instrumented clone executes the same code-path as its uninstrumented counterpart.

Deterministic replay enables an instrumented clone to follow the same progression of states as the uninstrumented process. This provides the same state to a subsequent security check as having the uninstrumented process call `fork` at each check, but at much lower overhead. Thus, `fork` is the necessary mechanism to create parallelism, and replay is the necessary mechanism to amortize the cost of `fork` through the aggregation of security checks.

`Fork` and replay can efficiently provide the entire state of the uninstrumented process to an instrumented clone, which allows one to run arbitrary security checks. However, some security checks run so infrequently or check so little state that `fork` and replay is overkill. For these types of checks, Inspector can disable `fork` and replay and instead send only the subset of state required for the check (see Section 4.4.2 for an example).

4.2.2 Threat model

Two mechanisms ensure that Inspector provides security equivalent to a sequential security check. First, due to Inspector's replay system, the instrumented clones will execute the same sequential checks that would have been executed sequentially. Second, due to Speculator's causal dependency tracking and rollback, any actions a speculative process takes before the instrumented clones complete the prior checks are rolled back should a prior check fail.

Thus, Inspector's security guarantee assumes that an attacker cannot compromise the replay system provided by Inspector or the causal dependency tracking and rollback system provided by Speculator. Because Inspector and Speculator operate within the kernel, a speculative process (which may be compromised but not yet checked) must not be allowed to gain arbitrary control of kernel state. Speculator's design philosophy helps prevent speculative processes from damaging the kernel. By default, Speculator blocks speculative processes when they make system calls; only specific system calls to specific devices are allowed to proceed. Thus, while the uninstrumented process may execute speculatively beyond the point of the check, it should be unable to execute actions that compromise replay or Speculator (e.g., by writing to `/dev/mem`). Unfortunately, speculative execution creates an opportunity for a new denial-of-service attack to occur. While speculative, a compromised process could arbitrarily communicate with other processes (e.g.,by sending signals); when Inspector later detects the intrusion, these additional, otherwise independent, processes would roll back. The risk of a denial-of-service attack when using Inspector is the tradeoff that comes with the ability to run security checks that are too slow to run otherwise.

I do not exclude attacks that would compromise the kernel while running a sequential version of the security check, as such attacks would compromise the system with or without Inspector.

4.3 Inspector Implementation

I have implemented Inspector within the Linux 2.4 and 2.6 kernels. I provide a description of the implementation of epochs, the replay system, a brief overview of Speculator, and a description of the binary rewriter I use to implement 3 different security checks.

4.3.1 Creating an epoch

Inspector must choose the epoch length to balance the amount of work completed by the security check and the cost of beginning a new epoch. If the epoch length is too short, the cost of `fork` dominates the benefits derived from dividing the work of the security check among multiple cores. Alternatively, if the epoch length is too long, the parallelism in the system is limited due to the exhaustion of resources such as free memory. Resource exhaustion is a particular problem for a security check such as taint analysis that has ordering constraints in how the checks are evaluated.

I implemented Inspector to balance the cost of starting an epoch with the work accomplished during an epoch by using a time threshold. At the end of each system call, it checks whether the amount of time the uninstrumented program has executed is greater than a threshold epoch length; if so, Inspector terminates the current epoch and forks a new instrumented clone. An instrumented clone exits when it executes the last system call in its epoch. In my experiments with my implementation of the parallel taint analysis security check, I have found that an epoch length of 25–50 ms is short enough to avoid resource exhaustion, but long enough to provide a significant performance improvement.

4.3.2 Ensuring deterministic execution

Inspector uses deterministic replay to ensure that the instrumented and uninstrumented clones execute identical code paths, absent the additional security checks in the instrumented version. If the two the code paths diverged, the security checks would not be valid.

When a program executes on a single core, there are only two sources of non-determinism: system calls and signal delivery. Scheduling and hardware interrupts are transparent to application execution. After a scheduling or hardware interrupt occurs, an application resumes execution at the same point that it was interrupted. Thus, operating system virtualization, which gives the application an illusion of executing alone on a virtual processor, masks this source of non-determinism from the application, as long as all forms of inter-process communication are logged. Inspector adds support for deterministic replay to the Linux kernel by logging system call results and signal delivery [5, 16]. My current implementation of deterministic replay does not support shared memory IPC for multi-threaded programs running on multiple cores. Doing so efficiently and without hardware support is an open problem [83].

4.3.3 System call replay

Inspector associates a `save` and a `restore` function with each system call. When an uninstrumented program completes a system call, Inspector executes the associated `save` function to log the system call results. When an instrumented clone makes a system call, Inspector redirects it to a secondary *replay* system call table. Each entry in the replay system call table points to an associated `restore` function for that call, which returns the values logged by the uninstrumented version of the program. For some system calls, such as `nanosleep` and `recv`, replaying the logged value in the instrumented clone may take much less time than the original system call.

When an uninstrumented process begins a new epoch, Inspector creates a *replay queue* data structure for that epoch. Each replay queue is shared between the uninstrumented and instrumented clone for a given epoch. The two processes have a producer-consumer relationship. The uninstrumented process adds system call results to the FIFO queue, and its instrumented counterpart pulls those results from the queue as it executes.

Unfortunately, this simple save and replay strategy is not sufficient for all system calls. System calls such as `read` and `gettimeofday` require special strategies because they return data by reference rather than by value, i.e., they copy data from the kernel into the user-level program's address space. To replay such calls, Inspector saves the value of all user-level memory that was modified by the execution of the system call. For example, Inspector saves the contents of the buffer modified by `read`. When the instrumented process calls the restore function for such system calls, Inspector copies the saved values from the replay queue into the application buffer.

4.3.4 System call re-execution

Certain functions such as `mmap` and `brk` modify the layout of an application's address space. Inspector must re-execute these system calls when they are called by an instrumented process. However, Inspector ensures that the modification to the instrumented version of the address space is identical to the modification made to the uninstrumented version. For example, the `mmap` system call is passed an address as a suggested starting point to map the object into its address space. If this parameter is `NULL`, then the operating system has the freedom to choose any unused point in the program's address space to map memory. Naively replaying this call might cause the operating system to map memory to two different locations in different copies of program. To prevent this problem, Inspector saves the address returned by `mmap` instead of the `NULL` value of the parameter. When the instrumented copy later executes the same `mmap` system call, Inspector passes the saved address into the sys_mmap function. This ensures that the memory is mapped to identical locations in both copies.

4.3.5 Signal delivery

Linux delivers signals at three different points in process execution. First, if a process is sleeping in a system call such as select, Linux interrupts the sleep and delivers the signal when the process exits the kernel. Second, Linux delivers the signal when a process returns from a system call. Finally, Linux delivers a signal when a process returns from a hardware interrupt.

To ensure deterministic execution, Inspector must replay the receipt of a signal at same point in the instrumented clone as it was received by the uninstrumented process. Delivering a signal within calls such as select or on the return from a system call is handled by checking during system call replay whether a signal must be delivered. The delivered signals are saved on the replay queue during the execution of the uninstrumented process, and identical signals are delivered when the instrumented clone returns from the same system calls. Since the instrumented copy returns immediately from functions such as select rather than sleep, Inspector never needs to wake up instrumented processes in the kernel to deliver signals.

Replaying signal delivery after a hardware interrupt is more difficult. To ensure correctness, a signal must be delivered at exactly the same point of execution (e.g., after some identical number of instructions have been executed). My current implementation does not yet support this functionality, although I could use the same techniques employed by ReVirt [16] and Hypervisor [5] to deterministically deliver such signals Currently, Inspector simply restricts signal delivery to occur only on exit from a system call or after interrupting a system call such as select. In practice, this behavior has been sufficient to run my benchmarks.

4.3.6 OS support for speculative execution

Inspector uses Speculator to isolate an untrusted application until a security check determines the application has not been compromised. Inspector associates each epoch with a speculation. The success or failure of the security checks associated with an epoch determines whether the execution of code during that epoch was safe or whether an intrusion occurred. If execution is safe, Inspector commits the speculation, which allows Speculator to release all output generated by that epoch. If an intrusion occurred, Inspector fails the speculation, which causes Speculator to begin rollback. Speculator rolls the application back to the checkpoint created at the beginning of the epoch. Speculator undoes any effects of the speculative execution on other internal kernel objects such as files, processes and pipes. Any output produced by the epoch execution is discarded. My current implementation does not restart an application after an intrusion is detected; instead, the application is terminated.

Currently, there is a slight difference in the output seen by an external observer when a security check fails using Inspector and the output that would have been generated by a sequential security check. Since Inspector operates on the granularity of an epoch, any output that occurred after the last epoch began but before the instruction that led to the failed security check would not be visible. I plan to address this by simply replaying the uninstrumented process up to the point where the security check failed.

4.3.7 Pin: A dynamic binary rewriter

When Inspector begins a new instrumented clone, the structure of the clone depends upon the type of security check. Inspector can use any particular method of inserting checks. The case studies in this paper use Pin [41] to dynamically instrument the code with the necessary checks. Pin allows application programmers to insert arbitrary code into an executable at runtime. Pin uses dynamic compilation to generate new program code as it is needed, using the rules and code defined with a *pintool*.

Pintools are transparent to the execution of the application instrumented with Pin. An application instrumented with Pin observes the same addresses and registers for program code and data as it would were it running without Pin. Inspector uses this property to ensure correctness: the execution of the instrumented clone does not diverge from the execution its uninstrumented counterpart because the program cannot observe that it is being instrumented. The authors of Pin have shown that Pin performs faster than other systems such as Valgrind [49] and DynamoRIO [76].

4.4 Parallelizing security checks

In this section I discuss the properties of a security check that determine both the difficulty in parallelization and the potential for performance improvement through increased parallelism. I discuss three different classes of security checks that demonstrate trade-offs in the difficulty and overhead of parallelization, and then describe the design and implementation of a check from each class.

4.4.1 Choosing security checks

When security checks are parallelized to run using Inspector, two checks that would otherwise run in sequential order execute concurrently. Some security checks may be more suitable for use with Inspector (i.e., easier to parallelize) than others. This section describes the properties that determine the likelihood of success when parallelizing a check using Inspector.

Inspector uses the uninstrumented process to run ahead and start later security checks in parallel with earlier ones. Therefore, each security check depends upon the state received from the uninstrumented process. This dependency impacts the amount of parallelism Inspector can achieve in two ways. First, the amount of work executed by the uninstrumented process between checks determines how quickly future epochs will begin. Second, the expense of the security check determines how many checks might execute in parallel at any one time. Expensive checks create more opportunity for parallelism and therefore offer the opportunity for greater relative speedup.

The benefit of parallelization also depends upon whether a later check depends upon the result of an earlier check. If a later check depends heavily on an earlier check running in parallel, it may be difficult to parallelize the check in such a way that the later check can make forward progress. If a later check is completely independent of an earlier check, the check is very easy to parallelize as the only dependency is the state provided by Inspector from the uninstrumented process.

Incidentally, there is a third type of dependency relationship to consider when parallelizing a security check. Some checks may have very few dependencies within a single invocation of a check. Such a check may be embarrassingly parallel, and could be parallelized within a single invocation, rather than parallelized across multiple invocations as is done by Inspector. Inspector does provide one advantage, which is that the programmer does not need to reason about how to parallelize the check; parallelization is achieved automatically. Since such checks have an alternative method for parallelization, I do not consider them in this paper.

4.4.2 System call analysis

There have been many different proposed security checks within the research community that analyze program behavior using system calls. Three examples are Wagner and Dean's call graph modeling [79], Provos' systrace [57], and Ko and Redmond's non-interference check [34]. These checks can dramatically slowdown performance, which limits their usefulness in production environments. Wagner and Dean cite multiple orders-of-magnitude slowdown, and Provos up to a 4x slowdown.

System call analysis exhibits two traits that make it promising for use with Inspector. First, each check does not depend upon the result of prior checks. Therefore, later checks can be run in parallel to earlier checks without modification. Second, system call analysis requires little state to execute – often just the list of prior calls executed and/or the system call parameters. As a result, such checks do not require fork and replay; Inspector can ship the state required at each system call from the uninstrumented application to an instance of the check executing in parallel.

I have implemented a heavyweight system call analyzer for use with Inspector. I intercept system calls with `strace` and pass a log of system calls with their parameters to a policy analysis process. My example policy generates all possible merge orders between a window of system calls in the monitored program and a static list; this can be used to look for time-of-check-to-time-of-use race conditions [34].

I parallelize the check by executing a version of the application speculatively using Inspector. As system calls are intercepted, I dispatch the data to a pool of waiting policy daemons, and continue executing the application. Each group of system calls is a speculative operation, and each daemon, after completing its policy analysis, may either commit or fail the speculation.

4.4.3 Process memory analysis

While signs of many security problems appear in the address space of a process, these signs may appear in memory only for a brief period of time. For example, a virus may decrypt itself, damage the system, then erase itself [77]; sensitive, personal data may inadvertently leak into an untrusted process [14]; or a malicious insider may release classified data by encrypting it and attaching it to an e-mail. One could detect such transient problems by checking memory at each store instruction, but the overhead of such frequent checks would be prohibitive.

Inspector can reduce the overhead of these checks by running them in parallel. Parallelizing this type of check with Inspector is straightforward because the checks are independent of one another.

I implemented one such security check, which looks for inadvertent leaks of sensitive data. The security check carries the 16-byte hash of a sensitive piece of data. At each memory store, it calculates the hash at all 16-byte windows around the address being stored. I implemented the security check as a pintool, which can be run sequentially or in parallel by using Inspector. I used `fork` and replay to efficiently synchronize state between the security check and the uninstrumented process.

4.4.4 Taint analysis

Taint analysis traces the flow of data from untrusted sources (e.g., a socket) through an application's address space and detects if a critical instruction executes with tainted data. For example, if the data on the top of the stack used by `RET` is tainted, a stack smash attack has occurred.

As a process executes, the taint analysis check updates a map representing which addresses and registers are tainted within the process's address space. Each check depends upon an updated version of the map. Therefore, each check depends upon all prior checks. These inter-check dependencies make taint analysis difficult to parallelize. Running multiple checks in parallel will not work, since a later check executing in parallel to an earlier check will not yet have all the information necessary to determine whether an attack has occurred.

Because taint analysis is difficult to parallelize, I use it as a challenging case study to parallelize with Inspector. My strategy is to divide work into parallel and sequential phases of execution. The parallel phase executes within instrumented clones, and the sequential phase processes the parallel pieces to determine whether an attack has occurred. My goal is to push as much work into the parallel phase as possible, while minimizing the work done sequentially. The rest of this section gives a broad overview of instruction-level taint analysis on x86 processors. Then, Section 4.5 describes my sequential taint analysis check, and Section 4.6 describes my parallel taint analysis check.

Prior taint analysis systems [15, 50, 59] use binary rewriters to instrument a target application. A memory or register location is marked as *tainted* if it causally depends on data from an external source (e.g., the disk or a network socket). For example, if an application reads data from the network, a taint tracker would mark the memory locations into which the data is copied as tainted. Data movement instructions (e.g., `MOV`, `ADD`, and `XCHG`) propagate taint from one location to another. Such instructions may also clear an address of taint either by overwriting it with untainted data, or by executing an explicit clear instruction (e.g., `XOR` eax, eax). Taint trackers must set input, propagation, and assert policies to determine when data becomes tainted, which instructions propagate (or clear) taint, and when to check for an intrusion, respectively.

An *input policy* determines when data from outside the address space taints data within it. The input policy for my algorithm taints data from any source external to the process's address space, excepting shared libraries loaded by the dynamic loader. Setting an input policy can be reduced to a question of trust: if the loader and shared libraries are trusted, then loading a shared library does not taint the address space.

A *propagation policy* determines which instructions propagate taint from one location to another. For example, taint analysis functions typically instrument each data movement instruction to propagate taint. Taint analysis tools may also instrument control instructions such as `JMP`; e.g., a conditional jump based on a tainted value could be considered to taint all data values modified later. My propagation policy instruments all data movement (`MOV, MOVS, PUSH, POP, ADD, CMOV`, etc.) instructions within the application, including F86 and SSE instructions, to propagate taint from data sources to data sinks.

There is a tradeoff in determining a propagation policy between the *coverage* provided and the *accuracy* of the algorithm. If taint is propagated along all channels of information flow, taint analysis will generate many false positives. However, if too many channels of information flow are ignored, then attacks may go undetected. Since Inspector is independent of the specific taint analysis check, I chose policies from prior work [15, 50].

Finally, an *assert policy* determines when to check whether an attack has occurred. Assert policies share the same coverage/accuracy tradeoff that is inherent in choosing a propagation policy. We implemented two assert policies – checking for stack smashing attacks and function pointer attacks.

4.5 Sequential taint analysis implementation

I built a sequential taint analysis check as a baseline comparison by following the descriptions written by Costa et al. [15] and Newsome and Song [50].

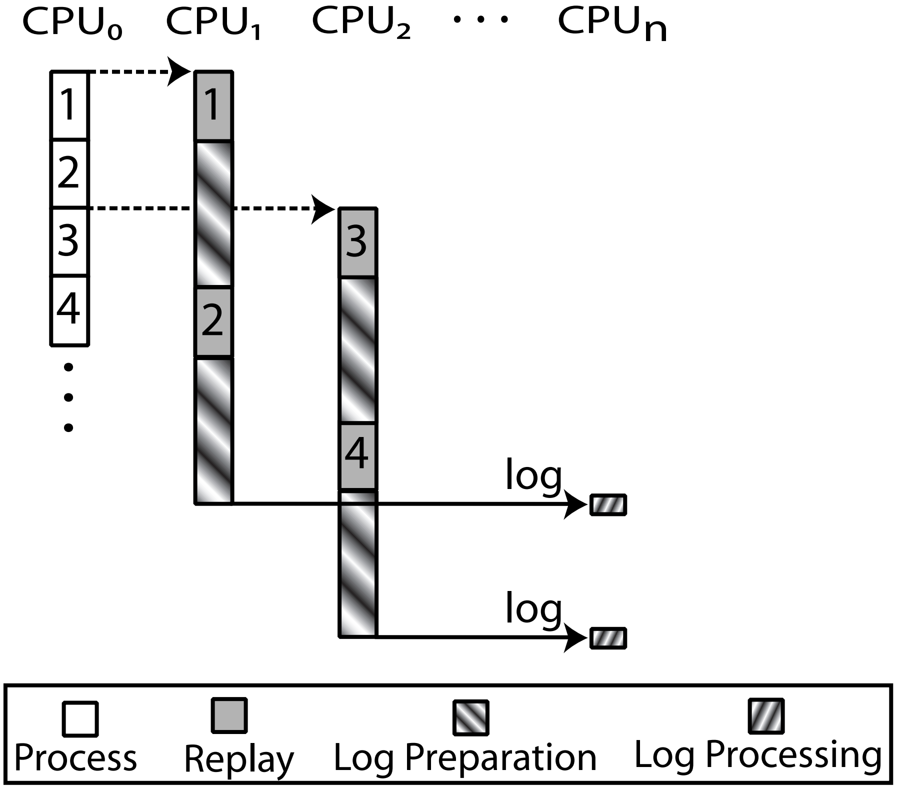
My sequential taint tracker uses a page table data structure to track the tainted memory addresses of the instrumented application. The top level of the page table is an array of pointers. Each entry points to a single page of memory in the instrumented application's address space. Each 4 KB page is a character array, such that each byte in the array represents a single byte of application memory. I use a byte rather a bit array because my initial results showed that bit operations substantially slowed down the taint tracker. To save space, initially each entry of the page table points to the same *zero page*, which is a 4 KB zero-filled array (indicating that the corresponding memory addresses are untainted). The taint values of registers are stored in another small array.

I use Pin to instrument each instruction that is covered by an input, propagation, or assert policy. The instrumentation that comprises the sequential taint tracker is carefully constructed to avoid inserting conditional instructions.

Although my focus is on the benefits of parallelization, I strove to efficiently implement Newsome and Song's sequential taint tracker using Pin. My results show that my implementation runs approximately 18 times slower than native speed; in comparison, Newsome and Song reported that their implementation ran 24-37 times slower on an identical benchmark [50]. Based on this, we believe that my sequential implementation is reasonably efficient.

4.6 Parallel taint analysis implementation

The need to sequentially update a map of tainted addresses and registers as a process executes poses a challenge for Inspector; running instrumented clones in parallel would result in incorrect results, since each epoch would have an incomplete map of tainted memory locations. I realized that the taint analysis check has two parts: first,



This figure shows the taint analysis check parallelized using Inspector. Each check depends upon the results of all prior checks. Therefore, I invented a new parallel algorithm that divides the work into parallel and sequential phases.

Figure 4.2: Parallelizing taint analysis using Inspector

the check decodes an instruction and determines whether the instruction is part of an input, propagation, or assert policy. Second, the check either updates the map of tainted memory locations, or it checks to insure a critical instruction is not acting on tainted data.

Although updating the map of tainted locations and checking for intrusions is sequential, decoding instructions and determining whether they apply to my policies can be done in parallel. Therefore, I divided the taint analysis check into parallel and sequential phases. Figure 4.2 shows an example of my parallel taint tracker. In this example, each numbered box is a single instruction, and an epoch is two instructions in length (in reality, an epoch encompasses thousands of instructions). Within the instrumented clone, the instruction is replayed, and then the first phase, called *log preparation*, is executed. Log preparation transforms the instruction into a dependency operation (described in detail in Section 4.6.1) and places the operation into a log. Once the epoch completes, the log is shipped to another core, where *log processing* takes place. Log processing processes each log in sequential order, updates a map of tainted addresses and registers, and checks for violations of assert policies.

One might imagine that the amount of data generated during log preparation could be quite large. In fact, my early results showed that the amount of data was so large that log processing took longer to complete than log generation. This problem spurred the development of *dynamic taint compression*, which is often able to reduce the number of operations in a log by a factor of 6. Dynamic taint compression is designed based on two insights: first, later operations often make earlier operations unnecessary. Second, the state of the taint map is only important before an assert check and at the end of each log. I use these observations to reduce the size of each log in parallel during log preparation. A detailed explanation appears in Section 4.6.2. Thus, log preparation is a two step process. First, log generation (described in Section 4.6.1) decodes the instructions, maps them to input, propagation, or assert policies and inserts the operations into a log. Second, dynamic taint compression reduces the size of the log, which accelerates sequential log processing.

4.6.1 Log generation

Log generation is implemented as a pintool that instruments the target application. For each instruction that could potentially trigger an input, propagation, or assert policy, the pintool writes an 8 byte record to the log that describes the operation type and size, the source location, and the destination location of the instruction.

Each entry in a log may be one of six types. A `TAINT` record is inserted when an input policy dictates that an address in the application's address space is tainted by its execution (e.g., TAINT records are inserted when data is read from a network socket). A `CLEAR` record is inserted if an instruction removes taint from a location. For example, the instruction `XOR eax, eax` generates a log entry that clears the `eax` register of taint. An ASSERT record enforces assert policies by checking to see if a location is tainted. For instance, ASSERT eax indicates that Inspector should halt program execution if the eax register is tainted.

The remaining record types implement propagation policies.

REPLACE records represent instructions such as MOV that overwrite the destination operand with the contents of the source operand. ADD records describe arithmetic and similar operations, where the destination operand is tainted if either the source or destination is initially tainted. SWAP is used for the XCHG instruction, which swaps the source and destination operands. More complex operations such as XADD are described using multiple log records (e.g., a SWAP followed by an ADD). Operations such as MOVS that move data between two addresses are encoded as two separate records. Some instructions do not generate any log records since they do not affect any taint policy.

During an epoch, multiple logs are generated. Rather than detect the end of a log by executing an expensive conditional after inserting each log record, Inspector inserts a guard page at the end of each log which is mapped inaccessible. After the log is filled, writing the next log causes the OS to deliver a signal to the instrumented application. The signal handler optimizes the filled log, as described in the next section, swaps a fresh log into the process address space, and changes the pointer to the next log entry to point the start of the log. Finally, Inspector reserves a specified amount of physical memory to store dependency logs; individual logs are created as fixed-sized shared memory segments shipped between instrumented clones executing log preparation, and the application executing log processing.

4.6.2 Dynamic taint compression

Without optimization, it takes longer to process a log than it takes to run the sequential taint tracker described in Section 4.5. This is unfortunate because log processing is fundamentally sequential. The log processing program must read each log record, decode the operation, and perform the specified operation. In contrast, the sequential taint tracker has less overhead — it simply moves or verifies taint as appropriate for the instruction it is about to execute. Further, since the sequential taint tracker executes in the address space of the application it is instrumenting, it enjoys locality of reference with the addresses it is checking.

Fortunately, I have found that there is substantial room to optimize logs after they are generated but before they are processed. Since Inspector speculatively executes instrumented code, it can potentially eliminate all taint operations that become moot due to later execution of the program. For instance, taint need not be propagated to a memory address if that address is overwritten with another value before it is checked for taint. In contrast to static analysis tools that can only eliminate a taint operation if it becomes moot along all possible code paths, the Inspector optimizer is dynamic and can eliminate an operation if it becomes moot along the code path the *application actually took*. In effect, speculative execution allows the parallel taint checker to peer into the future and eliminate work that will become unnecessary.

The goal of optimization is two-fold. First, optimization should be a parallel operation. Second, optimization should derive (and eliminate) the set of records that, after examining all records in the log, do not have an affect either on the final taint value of any address or register, or on the taint value of an address or register when it is checked for an assert policy violation. For example, if a register is tainted and later cleared, and no assert policy check on that register occurred between the two records, the earlier record can be safely discarded.

Dynamic taint compression is inspired by mark-and-sweep garbage collection. After each log is generated, it is then optimized independently of other logs. The optimizer makes two passes over the log. I use a map, similar to the one described on Section 4.5, to track the taint values of addresses and registers. During optimization, each address or register can have one of three values; either the location is known to be tainted, known to be free of taint, or its state is unknown. If the state is unknown, it depends upon the results of prior epochs, and it might depend upon one or more records within the current log. The second pass examines the map and identifies locations whose taint value depends upon a list of one or more records within the log.

The optimizer then creates a new log, which is the union of these lists. The second pass also creates a list of ranges of tainted and free locations. After the second pass, the new, smaller log (and list of tainted and free locations) is passed on to the log processing phase.

One can easily imagine more aggressive optimizations. However, we have found that the dynamic taint compression algorithm described in this section hits a sweet spot in the design space — it is very effective (achieving a 6:1 reduction in log size in my experiments), while minimally impacting performance due to its use of only two sequential passes through the log. Investigating other optimizations is an interesting direction for future work.

4.6.3 Log processing

The algorithm used in the log processing phase is very similar to that of the sequential taint tracker. It is implemented as a standalone process that reads operations from dependency logs and uses them to propagate and check taint, rather than as a pintool. However, it uses identical data structures to store taint data and implements the same input, propagation, and assert policies.

To process an optimized log, the log processing program first reads in the set of locations known to be tainted and the set known to be taint-free. It clears and sets the taint bytes for these locations accordingly. It then processes the log sequentially. If any `ASSERT` record operates on tainted data, the instrumented program is halted, the associated speculation is failed, and an attack is reported. Otherwise, the log processing program blocks until the next log becomes available.

There is an inherent tradeoff in the choice of log size. Larger log sizes create contention on the memory bus and pollute the caches on the system. Smaller log sizes lead to greater overhead, since a signal handler is executed after each log is filled. Further, since the log is the unit of optimization, smaller log sizes reduce the efficiency of optimizations that can be performed. My parallel taint check uses 512KB logs, which are small enough to fit in the L2 cache of the computers in my evaluation.

4.7 Evaluation

4.7.1 Methodology

I used two computers in my evaluation. The first computer is an 8-core (dual quad-core) Intel Xeon (5300 series, Core 2 architecture) 2.66 GHz workstation with 4 GB of memory and a 1.33 GHz bus. Each quad-core chip is made up of 2 dual-core chips stacked on top of one-another. Each dual-core on the chip shares a 4 MB L2 cache. The 8-core computer runs RedHat Enterprise Linux 4 (64 bit) 2.6 kernel. The second computer is a 4-core (dual dual-core) Intel Xeon (7000 series, NetBurst architecture) 2.8 GHz workstation with 3 GB of memory and a 800 MHz bus. Each dual-core chip shares a 2 MB L2 cache. The 4-core computer runs RedHat Enterprise 3 (32 bit) 2.4 kernel.

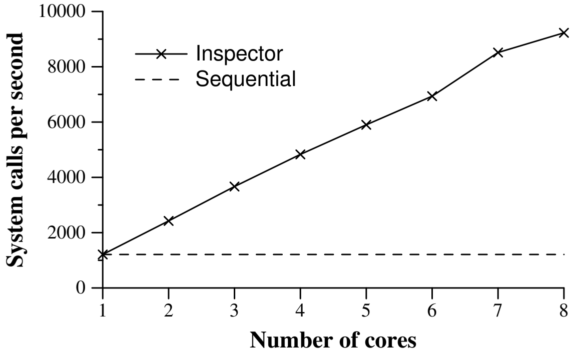
When I evaluated the parallel version of a security check, I varied the number of cores available to the operating system using the GRUB boot loader. On the 8-core computer, I could not boot any 32 bit Linux 2.4 kernel. Since Speculator currently works only with a 32 bit Linux 2.4 kernel, I could not run speculative execution for experiments on this machine. All other parts of Inspector did run for these tests, including deterministic replay as described in Section 4.3.2.

The overhead of Speculator has previously been shown to be quite small (typically adding performance overhead of a few percent or less) [51, 53]. I confirmed this by running the process memory analysis and taint analysis benchmarks on the 4-core machine with speculation. My results with the 4-core machine show that the additional overhead of speculation is negligible.

All results are the mean of 5 trials. All standard deviations are less than 1%.

4.7.2 System-call analysis

Figure 4.3 shows the rate at which `ls` can do a recursive directory listing on the 8-core machine while generating all possible merge orders between a window of system calls issued by `ls` and a static list. Inspector is able to achieve a 7.6x speedup for two



This figure shows the performance while generating all possible merge orders between a window of system calls issued by `ls` and a static list. The dashed line shows the performance of `ls` using the sequential version of the check.

Figure 4.3: System call analysis: ls performance

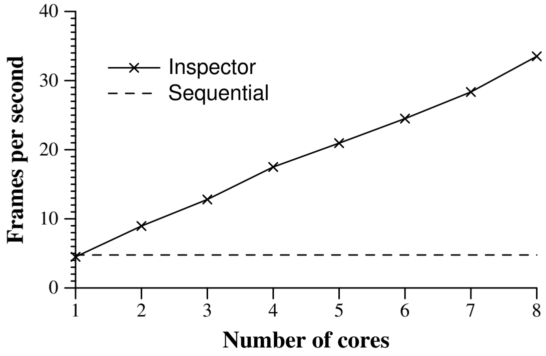
reasons. First, the parallel version of the check is identical to the sequential version, so they perform similarly on 1 core. Second, the checks are independent of each other, so performance improves almost linearly with the number of cores.

4.7.3 Process memory analysis

Figure 4.4 shows the rate at which mplayer can decode the H.264 video clip (a Harry Potter trailer) on the 8-core machine while continuously monitoring its memory for a 16-byte item of sensitive data. Inspector is able to achieve near-ideal speedups for two reasons. First, the parallel version of the check is identical to the sequential version, so they perform similarly on 1 core. Second, the checks are independent of each other, so performance improves linearly with the number of cores. By using Inspector, the movie decoded in real-time, which was not possible using the sequential version of the check.

4.7.4 Taint analysis

Figure 4.5 shows the throughput achieved when using bzip2 to compress a vim-6.2 source tarball using both the sequential and parallel taint tracker on the 8-core computer. This benchmark was first used by Newsome and Song [50] to evaluate



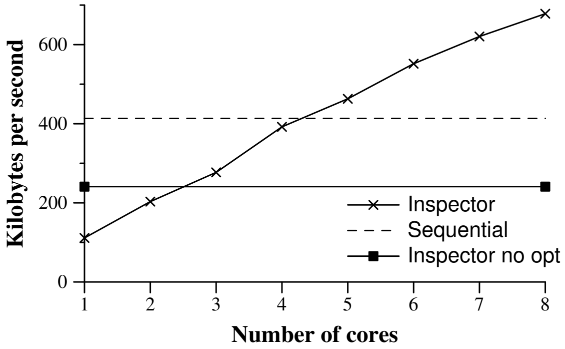
This figure shows the performance of mplayer while using Inspector to continuously check its memory contents. For comparison, the dashed line shows the performance of mplayer using the sequential version of the check. For reference, mplayer decodes 175 frames per second when running with no security checks.

Figure 4.4: Process memory analysis: mplayer performance

their sequential taint tracker. It is CPU-intensive and was regarded by them to be a particularly challenging benchmark for taint analysis. The benchmark achieves 7603.96 KBps on the eight-core machine when no security check is used. Figure 4.6 shows the throughput of the parallel and sequential taint trackers for the mplayer video decoding benchmark.

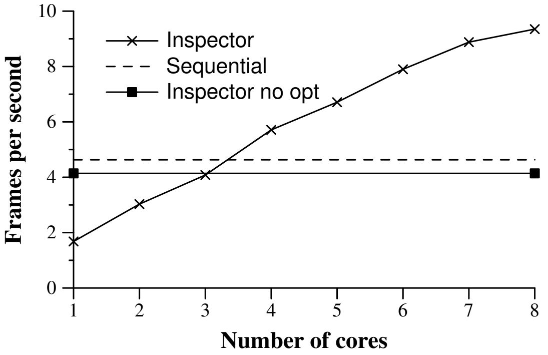
The results for 1 core show that the total computation performed by the parallel version of the taint tracker is substantially larger than that done by the sequential version, due to the overhead of generating and optimizing dependency logs. However, as the number of available cores increases, the performance of the parallel version improves. With 4-5 cores, the parallel taint tracker outperforms the sequential version. With 8 cores, the parallel version achieves a 1.6-2.0 speedup compared to its sequential counterpart.

The speedup achievable through parallelization is limited by the inherently sequential nature of processing taint propagation. Figure 4.5 and Figure 4.6 show that optimizing the log before processing it significantly reduces this bottleneck. Without first optimizing the log, the time to run the benchmark was limited by the log processing phase, which runs on a single core. In contrast, the optimized version of taint analysis shifts enough work to the parallel phase that Inspector can effectively use additional cores.



This figure shows the rate at which bzip2 compresses a 15 MB vim-6.2 tarball instrumented using the parallel taint analysis system. The dashed line shows the rate that bzip compresses the file using the sequential version of the check. For reference, bzip compresses at a rate of 7604 KB per second when running with no security checks.

Figure 4.5: Taint analysis: bzip2 compression performance



This figure shows the frames per second achieved while decoding an H.264 video clip using the parallel taint tracker. For comparison, the dashed line shows the frames per second using the sequential taint tracker. For reference, mplayer decodes 175 frames per second when running with no security checks.

Figure 4.6: Taint analysis: mplayer performance

4.8 Summary

Two trends motivate my work on Inspector. First, the difficulty of defending systems against attackers is spurring on the development of active defenses that instrument binaries with runtime security checks. Second, future processor improvements will most likely come from increased parallelism due to multiple cores, rather than from substantial improvements in clock speed.

Inspector stands at the confluence of these two trends. It reduces the cost of instrumenting security checks by decoupling them from the critical path of program execution, parallelizing them, and running them on multiple cores. My results for three security checks show substantial speedups on commodity multiprocessors available today. I hope that parallelizing security checks in this manner will enable the development and use of a new class of powerful security checks.

CHAPTER 5

Related Work

To the best of my knowledge, Speculator is the first example of support for multi-process speculative execution within an operating system. The work described in this thesis is the first to show the benefit distributed file systems, local file systems and security checks can derive from operating system support for speculative execution.

The design and implementation of this thesis has benefited greatly from prior work. First, Section 5.1 discusses prior uses of the technique of speculative execution within both hardware and software systems. Second, the remainder of the chapter discusses prior attempts at resolving the reliability/performance trade-off within the domains of local file systems, distributed file systems and security checks.

5.1 Speculative execution

The term “speculative execution” has been broadly used in many different areas of computer science. In this section, I have attempted to relate the applications of speculative execution most closely related to Speculator.

Chang and Gibson [10] and Fraser and Chang [21] 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 in distributed file systems 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 later re-done by a non-speculative thread. Speculator also allows multiple processes to participate in a speculation; Chang's speculative threads are not allowed to communicate with other processes.

Common applications of speculative execution in processor design range from branch prediction to cache coherence in multiprocessors. Steffan et al. [73] have investigated the use of speculation to extract greater parallelism from application code. Similar proposals advocating hardware speculation are [20, 24, 84]. 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 [37] and surviving software failures [58].

Speculative execution provides a limited subset of the functionality of a transaction. Thus, transactional systems such as QuickSilver [63], Locus [81], or TABS [69] 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.

Time Warp [29] 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 [28] tracks causal dependencies using Lamport's clock conditions [35]. 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. Speculator's design for buffering output was inspired by Time Warp's handling of external output.

The checkpoint and rollback-recovery techniques used within Speculator are well-studied [17]. 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. [70]. Previous work in logging virtual memory [13] might offer performance optimizations over Speculator's current fork-based checkpoint strategy. Causal dependency tracking has been applied in several other projects, including QuickSilver [63], BackTracker [31], and the Repairable File Service [86].

Fault tolerance researchers have long defined consistent recovery in terms of the output seen by the outside world [17, 39, 75]. For example, the *output commit* problem requires that, before a message is sent to the outside world, the state from which that message is sent must be preserved. In the same way, I argue that the guarantees provided by synchronous abstractions should be defined by the output seen by the outside world, rather than by the results seen by local processes.

It is interesting to speculate why the principle of outside observability is widely known and used in fault tolerance research yet new to the domain of general purpose applications and I/O. I believe this dichotomy arises from the different *scope* and *standard* of recovery in the two domains. In fault tolerance research, the scope of recovery is the entire process; hence not using the principle of outside observability would require a synchronous disk I/O at every change in process state. In general purpose applications, the scope of recovery is only the I/O issued by the application (which can be viewed as an application-specific recovery protocol). Hence it is feasible (though still slow) to issue each I/O synchronously. In addition, the standard for recovery in fault tolerance research is well defined: a recovery system should lose no visible output. In contrast, the standard for recovery in general purpose systems is looser: asynchronous I/O is common, and even synchronous I/O is usually committed synchronously only to the volatile hard drive cache.

My implementation of Speculator draws upon two other techniques from the fault tolerance literature. First, buffering output until the commit is similar to deferring message sends until commit [40]. Second, tracking causal dependencies to identify what and when to commit is similar to causal tracking in message logging protocols [18]. I use these techniques in isolation to improve performance and maintain the appearance of synchronous I/O. I also use these techniques in combination via output-triggered commits, which automatically balance throughput and latency.

5.2 Local file systems

Xsyncfs uses Speculator to mitigate the cost of providing synchronous file system guarantees to users. Prior work has focused on two different areas – changes in hardware to make synchronous I/O faster, and changes in software to improve performance by providing weaker guarantees.

While xsyncfs takes a software-only approach to providing high-performance synchronous I/O, specialized hardware can achieve the same result. The Rio file cache [11] and the Conquest file system [80] use battery-backed main memory to make writes persistent. Durability is guaranteed only as long as the computer has power or the batteries remain charged.

Hitz et al. [25] store file system journal modifications on a battery-backed RAM drive cache, while writing file system data to disk. I expect that synchronous operations on Hitz's hybrid system would perform no better than ext3 mounted synchronously without write barriers in my experiments. Thus, xsyncfs could substantially improve the performance of such hybrid systems.

eNVy [82] is a file system that stores data on flash-based NVRAM. The designers of eNVy found that although reads from NVRAM were fast, writes were prohibitively slow. They used a battery-backed RAM write cache to achieve reasonable write performance. The write performance issues seen in eNVy are similar to those I experienced writing data to commodity hard drives. Therefore, it is likely that xsyncfs could also improve performance for flash file systems.

Xsyncfs's focus on providing both strong durability and reasonable performance contrasts sharply with the trend in commodity file systems toward relaxing durability to improve performance. Early file systems such as FFS [44] and the original UNIX file system [60] introduced the use of a main memory buffer cache to hold writes until they are asynchronously written to disk. Early file systems suffered from potential corruption when a computer lost power or an operating system crashed. Recovery often required a time consuming examination of the entire state of the file system (e.g., running `fsck`). For this reason, file systems such as Cedar [23] and LFS [61] added the complexity of a write-ahead log to enable fast, consistent recovery of file system state. Yet, as was shown in my evaluation, journaling data to a write-ahead log is insufficient to prevent file system corruption if the drive cache reorders block writes. An alternative to write-ahead logging, Soft Updates [64], carefully orders disk writes to provide consistent recovery. Xsyncfs builds on this prior work since it writes data after returning control to the application and uses a write-ahead log. Thus, external synchrony could improve the performance of synchronous I/O with other journaling file systems such as JFS [3] or ReiserFS [47].

5.3 Distributed file systems

Distributed file systems have long played with the trade-off between reliability and performance by relaxing consistency and safety guarantees. For example, prior distributed file systems have provided single-copy file semantics—examples include Sprite [48] and Spritely NFS [71]. However, these semantics came at a performance cost. On the other hand, Liskov and Rodrigues [38] 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 [9], SUNDR [36], and FARSITE [1] that require additional communication to deal with faulty or untrusted servers. Finally, Satyanarayanan [62] observed that semantic callbacks could reduce the cost of cache coherence by expressing predicates over cache state. The hypotheses used by my speculative distributed file systems can be viewed as predicates expressed over speculative state.

5.4 Parallelizing security checks

Researchers in computer architecture have proposed a style of parallelization similar to mine, called thread-level speculation [24, 33, 68, 74]. Thread-level speculation is intended to take advantage of SMT (simultaneous multithreading) architectures [78] by speculatively scheduling instruction sequences of a single thread in parallel, even when those sequences may contain data dependencies. Thread-level speculation requires hardware support to quickly fork a new thread, to detect data dependencies dynamically, and to discard the results of incorrect speculative sequences. Two projects have used the proposed hardware support for thread-level speculation to parallelize checks that improved the security or reliability of software [54, 85]. Oplinger and Lam used this style parallelization to speed up basic-block profiling, memory access checks, and detection of anomalous memory loads [54]. Zhou et al. used this style of parallelization to speed up functions that monitored memory addresses (watchpoints) [85].

In contrast to prior work on thread-level speculation, my work shows how to parallelize security checks on commodity processors. The major challenge without hardware support is amortizing the high cost of starting new threads (instrumented clones). To address this challenge, I start instrumented clones at coarse time intervals, log non-determinism to synchronize the instrumented clones with the uninstrumented process, and use Speculator to enable the uninstrumented process to safely execute beyond system calls. Speculator enables speculations to span a broader set of events and data than thread-level speculation, including interactions between processes and between a process and the operating system.

In the area of security, Anagnostakis et al. use speculation to improve security [2]. They test suspicious network input in an instrumented process (a “shadow honeypot”) and roll it back if it detects an attack. Inspector differs from this work by parallelizing the work done by security checks, both with the uninstrumented process and with later security checks.

Researchers at Intel proposed hardware support to enable monitor checks to run in parallel with the original program [12]. They proposed to modify the processor to trace an uninstrumented process and ship the trace to a monitor process running on another core. As with Inspector, this enables the uninstrumented process to run in parallel with the security check. Inspector differs from this work in several ways. Most importantly, Inspector enables a second type of parallelism, which is to run in parallel a sequence of security checks from different points in a process's execution. Second, Inspector allows the uninstrumented version to safely execute system calls that externalize output (rather than blocking on every system call). Third, Inspector only ships non-deterministic events to the monitoring process (rather than a trace of every instruction), and this lowers the overhead of synchronization by an order of magnitude. Finally, Inspector requires no hardware support.

Finally, many researchers have explored ways to accelerate specific security checks. For example, Ho et. al accelerate taint analysis by enabling and disabling analysis as taint enters and leaves a system [26], and Qin et al. accelerate taint analysis through runtime optimizations [59]. These ideas are orthogonal to my work, which seeks to accelerate arbitrary security checks by running them in parallel with the original program and with each other.

CHAPTER 6

Conclusion

Often, two abstractions are provided to applications: a slow, synchronous abstraction that provides a guarantee and a faster, asynchronous abstraction, which relaxes or discards that guarantee. As long as latency disparities exist within computing systems, the tradeoff between performance and reliability will continue to affect the way applications are built.

Operating system support for speculative execution mitigates this tradeoff by creating abstractions that provide user-centric guarantees, which greatly improve performance. My thesis demonstrates this fact in three different domains. First, the implementation of xsyncfs demonstrates that a local file system can provide synchronous guarantees while providing nearly the performance of an asynchronous abstraction. Second, speculative BlueFS demonstrates that a distributed file system can provide strong safety and consistency guarantees and still be fast, and the speculative version of NFS demonstrates speculative execution dramatically reduces the time to enforce existing consistency and safety guarantees. Finally, the implementation of Inspector demonstrates that the performance of security checks can be greatly improved through parallelization across many cores.

6.1 Contributions

This thesis puts forth that application-centric guarantees are too conservative, forcing applications to wait while slow operations complete so that a guarantee might be provided. The leap to user-centric abstractions helps to relieve the tension between performance and reliability. From the viewpoint of an external observer, a user-centric speculative abstraction is indistinguishable from an application-centric synchronous abstraction, while providing greatly improved performance by freeing the application from the job of providing a guarantee to the user. User-centric guarantees are made possible with support for speculative execution within the operation system.

As a result of this thesis, several new software artifacts have been implemented. The most significant of these is Speculator. Speculator demonstrates that a reasonably efficient implementation of operating support for multi-process speculative execution is feasible. Further, the flexibility and generality of its design and implementation is demonstrated through its use in the domains of local file systems, distributed file systems and runtime security checks.

By using Speculator, four other artifacts have been implemented. First, xsyncfs is an example of an externally synchronous file system. Xsyncfs demonstrates that an existing file system can be transformed into an externally synchronous file system to provide the guarantees of synchronous file system I/O with greatly improved performance. Second, the speculative version of NFS demonstrates that by converting an existing distributed file system into one that is speculative, the performance of existing safety and consistency guarantees can be greatly improved. The implementation of the speculative version of BlueFS demonstrates the promise of designing a distributed file system around Speculator. BlueFS provides very strong safety and consistency guarantees while out-performing the non-speculative version of NFS. Finally, Inspector shows that the benefits of Speculator need not be restricted to the domain of storage. Finally, the implementation of Inspector and parallel versions of three different security checks demonstrate that the performance of sequential security checks can be greatly improved by parallelizing them across many cores.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] 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. 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, December 2002), pp. 1–14.

[2] Anagnostakis, K. G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., and Keromytis, A. D. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 2005 USENIX Security Symposium* (August 2005).

[3] Best, S. JFS overview. Tech. rep., IBM, http://www-128.ibm.com/developerworks/linux/library/l-jfs.html, 2000.

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

[5] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 1–11.

[6] Callaghan, B., Pawlowski, B., and Staubach, P. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF, June 1995.

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

[8] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006), pp. 147–160.

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

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

[11] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, October 1996), pp. 74–83.

[12] Chen, S., Falsafi, B., Gibbons, P. B., Kozuch, M., Mowry, T. C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G. R., Lin, B., and Schlosser, S. W. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. *2006 Workshop on Architectural and System Support for Improving Software Dependability* (October 2006).

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

[14] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 2005 USENIX Security Symposium* (August 2005), pp. 331–346.

[15] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 133–147.

[16] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[17] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[18] Elnozahy, E. N., and Zwaenepoel, W. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers C-41*, 5 (May 1992), 526–531.

[19] Flautner, K., and Mudge, T. Vertigo: Automatic performance-setting for Linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 105–116.

[20] Franklin, M., and Sohi, G. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers 45*, 5 (May 1996), 552–571.

[21] Fraser, K., and Chang, F. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference* (San Antonio, TX, June 2003), pp. 325–338.

[22] Haerder, T., and Reuter, A. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys 15*, 4 (December 1983), 287–317.

[23] Hagmann, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, 1987), pp. 155–162.

[24] Hammond, L., Willey, M., and Olukotun, K. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. ACM Conference on Arch. Support for Programming Languages and Operating Systems* (San Jose, CA, October 1998), pp. 58–69.

[25] Hitz, D., Lau, J., and Malcolm, M. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference* (1994).

[26] Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. Practical Taint-based Protection using Demand Emulation. In *Proceedings of EuroSys 2006*.

[27] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[28] Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404–425.

[29] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., P.Hontalas, Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Weidel, J., Younger, H., and Bellenot, S. Time Warp operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, November 1987), pp. 77–93.

[30] Katcher, J. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[31] King, S. T., and Chen, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[32] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[33] Knight, T. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), pp. 105–112.

[34] Ko, C., and Redmond, T. Noninterference and intrusion detection. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (May 2002), pp. 177–187.

[35] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[36] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 121–136.

[37] Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Technical Conference* (31–44, April 2005).

[38] Liskov, B., and Rodrigues, R. Transactional file systems can be fast. In *Proceedings of the 11th SIGOPS European Workshop* (Leuven, Belgium, September 2004).

[39] Lowell, D. E., Chandra, S., and Chen, P. M. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[40] Lowell, D. E., and Chen, P. M. Persistent messages in local transactions. In *Proceedings of the 1998 Symposium on Principles of Distributed Computing* (June 1998), pp. 219–226.

[41] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[42] Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 124–139.

[43] McKusick, M. K. Disks from the perspective of a file system. *;login: 31*, 3 (June 2006), 18–19.

[44] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (August 1984), 181–197.

[45] Miretskiy, Y., Das, A., Wright, C. P., , and Zadok, E. Avfs: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 73–88.

[46] MySQL AB. *MySQL Reference Manual.* http://dev.mysql.com/.

[47] Namesys. *ReiserFS.* http://www.namesys.com/.

[48] Nelson, M. N., Welsh, B. B., and Ousterhout, J. K. Caching in the Sprite network file system. *ACM Transactions on Computer Systems 6*, 1 (1988), 134–154.

[49] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, June 2007).

[50] Newsome, J., and Song, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium* (February 2005).

[51] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[52] Nightingale, E. B., and Flinn, J. 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, December 2004), pp. 363–378.

[53] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006), pp. 1–14.

[54] Oplinger, J., and Lam, M. S. Enhancing software reliability using speculative threads. In *Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 2002), pp. 184–196.

[55] OSDL. *OSDL Database Test 2.* http://www.osdl.org/.

[56] Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 206–220.

[57] Provos, N. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium* (Washington, D.C., August 2003).

[58] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. 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, October 2005), pp. 235–248.

[59] Qin, F., Wang, C., Li, Z., seop Kim, H., Zhou, Y., and Wu, Y. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)* (Orlando, FL, 2006).

[60] Ritchie, D. M., and Thompson, K. The UNIX time-sharing system. *Communications of the ACM 17*, 7 (1974), 365–375.

[61] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (February 1992), 26–52.

[62] Satyanarayanan, M. Fundamental challenges in mobile computing. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing* (Philadelphia, PA, May 1996), pp. 1–7.

[63] Schmuck, F., and Wylie, J. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 239–53.

[64] Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., and Stein, C. A. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 18–23.

[65] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. Network File System (NFS) version 4 Protocol. Tech. Rep. RFC 3530, IETF, April 2003.

[66] Silberschatz, A., and Galvin, P. B. *Operating System Concepts (5th Edition).* Addison Wesley, February 1998. p. 27.

[67] Slashdot. *Your Hard Drive Lies to You.* http://hardware.slashdot.org/article.pl?sid=05/05/13/0529252.

[68] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. Multiscalar processors. In *Proceedings of the 1995 International Symposium on Computer Architecture* (June 1995).

[69] Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L., and Pausch, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, December 1985), pp. 127–146.

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

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

[72] Standard Performance Evaluation Corporation. *SPECweb99.* http://www.spec.org/web99.

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

[74] Steffan, J. G., and Mowry, T. C. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 1998 Symposium on High Performance Computer Architecture* (February 1998).

[75] Strom, R. E., and Yemini, S. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems 3*, 3 (August 1985), 204–226.

[76] Sullivan, G. T., Bruening, D. L., Baron, I., Garnett, T., and Amarasinghe, S. Dynamic native optimization of interpreters. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators* (2003).

[77] Szor, P. *The Art of Computer Virus Research and Defense.* Addison Wesley, 2005.

[78] Tullsen, D. M., Eggers, S. J., and Levy, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 1995 International Symposium on Computer Architecture* (June 1995).

[79] Wagner, D., and Dean, D. Intrusion detection via static analysis. In *Proceedings of 2001 IEEE Symposium on Computer Security and Privacy2001* (2001), pp. 156–169.

[80] Wang, A.-I. A., Reiher, P., Popek, G. J., and Kuenning, G. H. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference* (Monterey, CA, June 2002).

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

[82] Wu, M., and Zwaenepoel, W. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 1994), pp. 86–97.

[83] Xu, M., Bodik, R., and Hill, M. D. A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture* (June 2003).

[84] Zhang, Y., Rauchwerger, L., and Torrellas, J. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proc. of the 5th Intl. Symposium on High Performance Computer Architecture* (Orlando, FL, January 1999), p. 135.

[85] Zhou, P., Qin, F., Liu, W., Zhou, Y., and Torrellas, J. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the 2004 International Symposium on Computer Architecture* (June 2004).

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