Speculative Execution Across Layers

by

Benjamin J. Wester

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
2012

ACKNOWLEDGEMENTS

This dissertation is not the product of my effort alone. Over these last few years of my life as a graduate student, it has taken the help of my colleagues, friends, and family to pull me through grad school and give me the opportunities I have today. I have many to thank, but a few a few people deserve a special mention.

I would like to thank my advisor Peter Chen for guiding me through the process of becoming a researcher. He has always been there to discuss new ideas and shape the old ones into something better. During my time here, he has shown me great patience and encouragement, and I cannot imagine having made it this far without him.

I want to thank my committee, Peter Chen, Jason Flinn, Satish Narayanasamy and Demosthenis Teneketzis, for helping me shape this thesis and give it direction. I especially thank Jason for being like a second advisor for me. I am fortunate to have been able to have his assistance throughout my stay at Michigan.

I am grateful to my many colleagues and friends at Michigan, those that have already moved on as well as those still studying. There has never been a lack of interesting discussion (or more often, diversion) in the office, nor of students to hear one more practice talk or gawk in wonder at how our systems ever ran in the first place. I have Jody Su, Dan Peek, Sushant Sinha, Kaushik Veeraraghavan, Arnab Nandi, Mona Attariyan, Timur Alperovich, Jessica Ouyang, Jakub Czyz, and many others to thank for keeping me sane through these years.

Lastly, I owe the greatest thanks to my family. My parents Ric and Terry and my sister Whitney have always given me their love and support. To my wife Amy: you have always been there to read through my papers, to watch me draw convoluted sketches on the whiteboard, and to force me to say what I really mean. You brought me food and blankets when I stayed late in the lab, put up with me during paper deadlines, celebrated with me when I got something published, and took care of my life when I was too wrapped up in research to know what was going on. You even still talk to me after these last few months spent finishing this degree. Most of all, you gave me your support and had confidence that I could accomplish what I wanted to, even if my methods drove you crazy. Lily, my daughter, hasn't been around very long, but she has already brought great joy into my life and helped to give me a better perspective on life. She reminded me why I chose to take my life in this direction and helped me choose which way I wanted to take it in the future.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

To achieve good performance on modern hardware, software must be designed with a high degree of parallelism. New advancements in processor design and manufacturing have produced chips that offer greater computation capacity, but they do this primarily by providing more processing cores rather than by greatly improving per-core performance. Speculative execution is a technique that can be used improve the parallelism of sequential programs by predicting the dependencies between tasks, allowing a later task to run concurrently with an earlier one.

Software systems are composed of many different cooperating layers: CPU, virtual machine, operating system, language runtime, and program code. The effects of speculative execution are traditionally confined to a single layer. Code running at higher layers is unaware that its execution is speculative, and code running at lower layers never observes any speculative behavior or output from higher layers.

This thesis explores the benefits of letting speculative executions be visible across the many layers of a software system. Components of the system can be aware of speculative computations and, with controlled access rather than isolation, can be designed to handle speculative state and effects correctly. With cooperation among layers, new opportunities for parallelization appear.

This dissertation shows how visible speculations across layers can be applied to application development, network protocols, and dynamic application analysis. I develop a new programming model for applications that separates the mechanism used to implement safety from the policy that describes how to control an individual speculation. The operating system implements the mechanism, leaving each application to describe its own custom policy. I then describe a new agreement protocol for Byzantine fault-tolerant services that is optimized for client applications capable of executing speculatively. Finally, I develop an algorithm for parallelizing data race detection by expressing speculative parallelism and handling it in the algorithm itself.

CHAPTER 1

Introduction

The ability to write parallel programs is important for many disciplines in computer science. Scientific applications have for a long time spread numerical computation and simulations across a large number of nodes to achieve reasonable performance. For systems with massive data requirements, it has often been found that spreading data and computation across a large number of slow computers can be more cost-effective than provisioning a single large, fast system [18,29], assuming the needed computations are capable of running in parallel.

Over the last decade, parallelism has become increasingly important to commodity computing systems as well. Traditionally, improvements in processor clock speed and architecture could be relied upon to improve sequential software performance at a rate of 40%–50% each year [41]. Software needing improved performance could simply wait for the underlying hardware to provide it. However, recent processor trends have diverged from this pattern. Advances in chip manufacturing and architectural design have been used to provide multiple processing cores on a single chip instead of providing improved single-core performance. Commodity chips are currently shipping with the capacity to run 12 sequential processes concurrently [3]. This number is expected to significantly increase in the next few years. Single-core performance is still improving in each successive chip design, but developers can no longer rely on the same trend to quickly improve their program's performance. To take advantage of the hardware offered by modern systems, desktop programs must be able to run in parallel across a large number of cores.

Unfortunately, writing parallel programs is a difficult task. Some tasks, like physical simulations, can be parallelized easily. Other tasks are more naturally described as a single sequence of computations where latter computations depend on the results of previous ones. This thesis focuses on one technique that systems can use to extract greater parallelism from a program in the presence of sequentially-dependent tasks: speculative execution.

1.1 Speculative Execution

A task is executed speculatively when it is started before the system knows whether the task can or should be executed. When a system uses speculation, the system will predict the effects and results of a currently-running task and then use that prediction to start future tasks in parallel with the current task. These tasks started early using predicted results are referred to as *speculative tasks*. By running the task early, its work can execute in parallel with that of its logical predecessors. Once the real inputs to the task are ready, they can be compared against the predicted inputs. When the prediction is correct, a speculative task will have performed the same computations it would have performed had it run later, i.e. when its inputs were ready. In this case, the speculation should be *committed* to preserve the work already performed. However, if the prediction turns out to be wrong, the speculative task has performed a computation that the program did not request, and the computation and any state associated with the execution must be *rolled back*. Despite this alteration of the steps of the program, we would still like the program to have the same semantics as its sequential specification. To maintain the sequential semantics of the program in either outcome, it is also necessary to delay any external output generated by the program until the output would have been generated by the sequential execution of tasks. Delayed output is released when the speculation is committed and discarded if the speculation is rolled back.

Using speculative execution has the potential to greatly improve the concurrency of a program, but there are two costs to using this technique. The first is a *direct* cost involved in creating any speculation. To support rollback, any changes made by the speculative execution must be undoable. Hence, modified objects will typically use checkpoints or undo logs to restore the original state on a rollback. Maintaining this old state may incur setup overhead (e.g., initializing a copy-on-write snapshot or making a complete one) and ongoing overhead (e.g., logging old values, writing to a copy-on-write snapshot, and buffering effects) in processing time and memory. When a speculation is committed or rolled back, these old data must be deleted or restored, consuming additional resources. For speculation to be useful, the amount of work done while speculative must exceed this overhead.

The second cost is an *opportunity* cost incurred when a speculation is aborted. Executing a failed speculative computation consumes system resources, even though the computation is ultimately ignored. Instead of being used for a wasted computation, those resources could have been used to make progress other non-speculative tasks on the system (if there are other tasks).

To decide if speculative execution should be used to predict a task, one should use a cost-benefit analysis. The expected benefit gained through increased parallelism should be greater than the cost of performing that speculation, taking into account the direct and expected opportunity cost. For some tasks, the direct cost of predicting the task and starting a speculative execution are greater than any potential savings (e.g., if the task's total duration is less than the time needed snapshot its state). The actual opportunity cost depends greatly on what other tasks are available to be run. A heavily-loaded server may allow only near-certain speculative paths to execute, while a system with many unused cores may be willing to take a greater chance by following less-certain predictions. The ideal task to predict is one that is slow, relative to the setup overhead, and predictable.

Slow and predictable tasks appear in many different components through a computer system. Consequently, speculative execution has been used to accelerate the performance of sequential tasks in a variety of areas such as processor design, virtual machines, operating systems, application runtime environments, network protocols, and user applications.

We deal with the enormous complexity of an entire computing system by building systems that are composed of individual components designed for a specific task. These tasks are often organized to provide abstractions that are layered on top of each other. Each component within a layer provides some small piece of functionality which is used by higher layers and is built on the functionality of lower layers. For illustration, processors may provide certain guarantees about program execution to operating systems, which can provide easier and safer abstractions to an application, where the runtime system can further refine the system properties to offer programmers a streamlined way to write applications.

A large number of systems (discussed in Section 2.1) apply speculative execution transparently and exclusively within a single layer. That is, one component in the larger system uses speculation internally without exposing any speculative state or behaviors to other components. A program running in higher layers (e.g., a user's program) may be executing speculatively, but it will be unable to differentiate its own speculative execution from a non-speculative one. As the speculative component interacts with components in other layers, it ensures that it never externalizes any speculative state.

Isolating all speculations within a single layer keeps the overall complexity of the system low by reducing the dependencies between layers. Doing so means that one part of the system can be modified to perform its duties using speculative execution without requiring changes to other layers.

However, a single-layer approach can miss speculative opportunities in some computations by limiting the potential benefits of the speculations. Programs that frequently use synchronous external communication may find that their speculative tasks make little progress before they block. A request external to the layer cannot be released from a speculative task (since it cannot be easily rolled back), so the reply on which the task is waiting will not come until the speculation is committed. When communication is frequent enough, the amount of work that can be parallelized becomes smaller than the overhead of using speculation.

1.2 Contributions

This thesis claims that making speculative executions visible across layers in the system creates new opportunities for parallelization and improved performance. When multiple layers, each capable of handling speculations on their own, are permitted to observe speculative state and take part in one speculative execution, interactions between those layers can be controlled and rolled back if needed. This changes the cost-benefit analysis for a communicating task. Rather than quickly blocking, the task's output can be allowed and it can continue running. By making speculative state visible to other components, more tasks become eligible to run speculatively because the expected benefit of the task may then outweigh the cost of speculative execution.

We will demonstrate this claim by designing and constructing three new systems that make speculations visible across layers. We first explore the general idea of how cooperation between two layers could be structured. We then apply speculative execution to two applications and show how speculative cooperation across layers can be used to improve their parallelism and performance. These systems are briefly described below.

1.2.1 Speculation as an OS Service

We view a speculation system as being composed of two separate parts: a *mechanism* that implements fundamental speculation primitives like state checkpointing, output buffering, and dependency tracking, and a *policy* that specifies how the mechanism should be used. Our work examines how the mechanism can be implemented in the operating system and controlled from within an application. By allowing this interaction between the OS and application layers, interesting uses of speculation can be developed in applications without significantly complicating the program design.

To speculatively execute applications transparently, the OS uses a conservative generic policy that preserves safety for any application. For some applications, this policy is too restrictive. We allow an application to specify a *custom policy* that varies from the generic policy along three main axes. A *creation policy*, specified at a level of abstraction that makes sense for the application, lets the application specify what program actions are predictable and how to predict their results. An *output policy* allows the program to specify some of its output as safe to be released non-speculatively. A *commit policy* lets an application define how to compare the results of a predicted tasks and how to recover when the results differ.

1.2.2 Fault-Tolerant Protocols

We next explore how speculative execution can be applied to Byzantine fault-tolerant (BFT) services to improve client performance. In a generic BFT system, when a server experiences a Byzantine fault, it can no longer be guaranteed to hold correct data or follow any network protocol correctly. To guard against such faults, a service must be replicated across several machines that fail independently. A client must then aggregate many replies from different replicas, since any single reply could be from a faulty server that gives incorrect results. The client only determines the true result when a quorum of replicas have all given the same reply.

We add speculative execution to BFT systems based on the observation that in the common case where no replica is faulty, the first reply a client receives is likely to be correct (though the client cannot guarantee it yet). Thus our client can predict that the first reply it receives is correct and continue executing its reply handler speculatively while it waits for more replies to be returned.

So far, only speculation within the client is required to implement this system. However, if the client is executing speculatively and wants to continue to interact with the BFT service, it must wait until its prior speculations have committed before sending another request, reducing the benefit of the speculation. To address this problem, we develop the PBFT-CS protocol that is optimized for a speculative client and permits speculative interaction between the two layers. PBFT-CS is designed to return the first reply to the client as fast as possible. Each request in our protocol is augmented with predicates that must be satisfied before a replica will execute the request. Clients can use these predicates to encode their speculative dependencies, so that if a message depends on an incorrect prediction, a non-faulty replica will discard it.

We then use our speculative client and protocol to implement a fault-tolerant NFS service. The NFS protocol itself can use speculative execution with cooperation between the distributed file server and the local OS [52]. We show how these two parts can be merged into one cohesive system that allows speculative execution across the operating system, a client proxy application, the network transport, and the distributed service layers.

1.2.3 Race Detection

Finally, we will demonstrate how a dynamic program analysis algorithm can be parallelized efficiently if it can be aware of the speculative executions of its target program. We apply speculative execution to improve the performance of a dynamic data race detector. Data races are caused when two threads access the same piece of shared memory without first synchronizing their accesses. A dynamic data race detector is a tool that instruments the memory and synchronization operations of a program to detect data races when they occur. Current detectors incur a large overhead: state-of-the-art detectors slow a program's execution by an average of 8.5× for managed code [23].

The third part of this thesis describes a new race detection algorithm, Parallel FastTrack, that is designed to parallelize the work of instrumenting and analyzing memory accesses. Our approach uses a *uniparallel* architecture as the fundamental method of extracting parallelism. In a uniparallel execution, a program's execution is time-sliced into epochs, each of which is run twice. One execution lets threads run concurrently on different cores while epochs are run in order. In this execution, we add minimal instrumentation to gather information only on the synchronization operations of each thread. The other execution runs epochs concurrently while an epoch's threads are constrained to a single core. In this execution, we include instrumentation to analyze every memory access as well as the thread synchronization. This instrumentation lets our detector find all the races that occur inside an epoch. The two executions are synchronized with each other by using deterministic replay techniques. A final sequential commit phase gathers the analysis results from each epoch to find races that span epochs. Epochs are executed speculatively, so that if a race is discovered, the program state can be rolled back to the instant where the race occurred.

As with most program analysis algorithms, race detection algorithms are designed to operate on a sequential execution of the program. By exposing the speculative execution environment to the algorithm, we construct an algorithm that parallelizes its work more efficiently while maintaining the same correctness guarantees.

1.3 Thesis Overview

Chapter 2 presents an overview of background research on which this thesis is built. We describe and categorize many existing uses of speculative execution. We also describe Speculator, an implementation of operating-system-level speculation, that underlies all the work presented in this thesis.

Chapter 3 discusses how speculative execution in an operating system can be customized by individual applications. Applications can describe their use of speculation while using a common implementation in the operating system.

Chapter 4 presents a new Byzantine fault-tolerance protocol that is optimized for speculative clients. Speculative clients on their own can hide some of the latency involved in a fault tolerant system. Protocol support allows these clients to continue to access the replicated service while executing speculatively.

Chapter 5 describes the construction of a parallel data race detection algorithm. Although race detection algorithms are formally specified for a linear sequence of program events, we build on the FastTrack algorithm to allow it to parallelize its work in a speculative environment.

Chapter 6 concludes the thesis. We summarize our contributions and discuss future work.

CHAPTER 2

Background

Speculative execution is a well-established technique that has been studied in many domains. This section presents a brief survey of many existing systems that have used speculation.

In addition, we introduce and describe the construction of the Speculator kernel, an custom Linux kernel that implements speculative execution inside the operating system.

2.1 Single-layer Speculation

A large number of systems apply speculative execution transparently and exclusively within a single layer. That is, one component in the larger system uses speculation internally without exposing any speculative state or behaviors to other components. Software running in higher layers (e.g., a user's program) may be executing speculatively but will be unable to differentiate its own speculative execution from a non-speculative one. Interactions with other components will be tightly regulated to ensure that no speculative state is externalized from the layer providing the speculation.

Although we find this model to be limiting, a significant amount of work can be parallelized without cooperation between layers. We group these works by the layer in which they are implemented.

Processor Architecture One early use of speculation was in the design of processor architectures to enable greater Instruction Level Parallelism (ILP). Programs are written as a sequence of instructions, yet the processor can choose to execute instructions concurrently by issuing instructions back-to-back through the same pipeline or by issuing multiple instructions in the same clock cycle.

Speculative execution is used in several different ways at this layer. A conditional and indirect branch instructions adds stalls to a pipeline by delaying the determination of the next instruction until an arithmetic or memory operation is evaluated. By predicting the target of each branch, the processor can begin executing instructions from the likely target before the actual outcome is known [73]. Conflicting memory accesses must necessarily be executed in program order. When the target of an access is indirect it might conflict with other operations. By predicting that two accesses do not conflict, they can be executed concurrently [57]. Even when there is a data dependence between two instructions, the data value result of the earlier instruction can be predicted to break that dependency and allow subsequent instructions to execute concurrently [45, 46].

Virtual Machines Speculative execution has also been used to provide improved performance for virtual machine replication. Primary/backup replication is a common system architecture for fault-tolerant services. The traditional approach to virtual machine replication forces VMs to agree on all non-determinism before it can affect the OS, so that output generated on the primary system is reproducible on the backup. Remus [17] uses speculative execution to instead continuously send snapshots from the primary to the backup without agreeing on non-determinism. Between snapshots, the primary executes its OS speculatively and buffers all its output until the backup has acknowledged the last snapshot. On a failure, uncommitted output is squashed before the backup takes over.

Runtime Environments We consider a runtime environment to be a software layer that runs in the same context as a program but is not specified by the program itself. Such code could be a language-based virtual machine, code inserted by a compiler, or code added through binary instrumentation. Speculative execution can be achieved by starting new speculative threads inside the process and carefully controlling what state those threads are allowed to modify and which system calls they are allowed to issue.

Speculation within this layer has been used to generate I/O hints about future file accesses [13]. When an application performs a file read that would block in the kernel, the runtime can spawn a speculative thread that runs ahead of the true thread and issues prefetching hints. In this instance, the runtime is predicting that the file access pattern does not depend on the data itself.

Operating Systems An operating system is responsible for isolating one process from another and mediating access to shared resources. A process can only directly affect state within its own address space. To do anything else, a process must use a well-defined API to issue a request to the OS, asking that it perform an action on the program's behalf. These characteristics give an OS considerable freedom to alter the execution of programs. Speculation within an OS usually involves forking a process and handling its system calls differently to preserve safety.

The I/O hint generation discussed earlier can also be performed inside the OS [13, 24]. Speculation has also been used to hide the latency of synchronous disk operations. Because the state of the disks and file systems is not directly observable, a long synchronous write can be predicted to complete without errors, allowing applications to run ahead without waiting for a full commit to disk [54]. Deadlock detection can also benefit from a speculative OS. When a potential deadlocked process is found, the OS can run the process speculatively to discover the exact resources used by each thread and process [43]. Speck [53] uses speculation to parallelize added security checks to a process through *epoch parallelism* (this execution style is discussed more completely in Chapter 5). When a check, like searching through memory for a private value or analyzing system call behavior, is needed, the system can predict that the check will succeed and allow the process to continue executing speculatively. The Rx [62] system allows software to recover from some types of software failures. As a program executes, periodic checkpoints are taken. If the program crashes, it is rolled back to a safe checkpoint and retried while the system interposes in an attempt to prevent the same bug from occurring.

User Applications To use speculation within a user application, the developer must be careful to track the speculative state and effects manually. Despite the effort, some programs do realize a significant boost from using speculative execution that justifies the added complexity.

Speculative execution has long been used in database systems as optimistic concurrency control [37]. Multiple transactions are allowed to execute concurrently under the prediction that their updated do not conflict. If they do, the database must undo the conflicting modifications and re-run the transactions serially. Remote display viewers have also benefited from speculative execution. Screen updates in the VNC protocol are highly predictable, so the user's perceived latency can be improved by showing predicted updates to the user and fixing the display afterwords to correct any bad data [40].

Network Message Delivery Replicated services rely on some form of atomic broadcast protocol to ensure that replicas execute requests in an identical order. A necessary step in such a protocol is to have each replica agree on the sequence of requests. Waiting for agreement can add significant latency to each request. By having each replica predict that its sequence is correct, it can handle a request concurrently with agreement. This general pattern has been studied for transaction processing in database systems [33] and for Byzantine fault-tolerant systems [35].

2.2 Multi-layer Speculation

Many prior system have also been designed to allow some layers to cooperate to provide speculative execution. We present these systems grouped by the layers that cooperate. Most of these systems implement speculative execution to fit the exact problem they are addressing. Our work addresses the more general issue of how two different layers can cooperate in a generic and reusable way. We also present two new systems that demonstrate cooperation between application & network layers and between OS & dynamic analysis.

Processor & Runtime A significant amount of research effort has focused on Thread-Level Speculation (TLS) [26, 74, 75], a technique where a compiler analyzes an application's code to automatically extract parallelism. The data dependencies between two successive blocks of code can be classified according to whether static analysis can guarantee that a value produced in the early block is definitely used in the subsequent block (*must* dependency), or whether the analysis cannot prove that a value is not accessed (*may* dependency). TLS analysis looks for blocks of code where the value all *must* dependencies is predictable and the *may* dependencies do not actually alias frequently. Loop bodies in scientific applications are particularly well-suited to this kind of analysis and can frequently be spawned as independent threads. Processor support is used to efficiently detect conflicts between threads and to buffer a thread's memory changes until the thread can commit.

OS & Runtime Speck [53] (mentioned previously in Section 2.1) also considered how to provide dynamic taint tracking to a program via speculative execution. Unlike the other security checks used by Speck, taint tracking is inherently an analysis over the sequential events of the program. The analysis environment must be redesigned and optimized to function correctly with an epoch-parallel architecture, although this work stops short of redesigning the taint-tracking algorithm itself.

OS & Application AutoBash [76] is a tool to help users troubleshoot configuration management problems. OS-layer speculative execution allows predicates—commands that check for configuration bugs—to perform arbitrary actions without permanently changing system state (e.g., their speculations are always aborted). Speculations are also presented directly to the user via the Bash shell. As the user attempts to fix the problem, he can explicitly roll back any command.

OS & Network Although network protocols are typically implemented by individual applications, an operating system that provides distributed services may directly interact with other systems via network protocols. By allowing speculative messages to be sent and received, local speculative processes can make progress even through remote operations.

The Time Warp operating system [31] used speculative execution to hide the latency of communication between nodes in a distributed simulation. Computation at each node occurs in fixed steps, and communication messages contain a time stamp showing exactly when they are to be read. Time Warp allows an application to run computation steps speculatively, predicting that all prior messages have been received. If a message arrives late, the node will be rolled back so that the message can be read. If the re-execution produces a different message stream, the old messages are un-sent (causing the receiver to roll back).

Speculator showed that speculative execution can hide the latency associated with remote operations in distributed file systems [52]. When a remote request is sent, the client predicts the high-level content of the reply—that the client's cache is up-to-date—and services the request from its local cache. This prediction is know to the file server, and subsequent speculative messages, which carry a list of speculative dependencies, can be ignored when they depend on an incorrect prediction.

Runtime & Application Recently, many projects have explored how to add speculative execution to a language runtime so that applications can make use of speculation without having to worry about checkpointing program objects or logging their modifications. By leaving the management tasks to a runtime, the application can avoid complicating its own design and logic.

This goal has been explored in several different contexts. Prabhu et al. develop a formal language that models speculation by explicitly defining a producer of a value, a consumer of that value, and a prediction function to generate a prediction for the speculative execution of the consumer [60]. They also explore the conditions that allow a speculative consumer to avoid needing to log memory updates. Prospect [77] uses speculative execution to implement epoch parallelism, running a fast, buggy version of a program alongside a slower more-correct program. The fast program provides hints about the slow program's future state, assuming that the slow and fast executions agree on all prior actions. Crom [50] extends the JavaScript environment to support speculative page loading and rendering. An API is presented that lets an application specify how to predict the user's interaction with the application. Fast Track [32] allows an application to specify a fast code path and an equivalent slow code path. The runtime executes the two concurrently, predicts that the two will result in identical state, and lets the fast path continue executing speculatively.

Full-system The Mojave system [72] consists of a compiler, dynamic runtime environment, and operating system that provide support for speculative execution. An API is provided so that speculative executions can be defined over application code. The runtime and OS can also start speculations when they encounter predictable events.

Mojave has been used to implement speculative execution for software distributed shared memory (DSM) [78]. DSM is implemented in the OS and performs a distributed page invalidation protocol transparently on a shared memory page access by the application. Messages exchanged between systems are logically timestamped and should be processed in a strict order. Much like the Time Warp OS, when a message is received, the OS speculates that it will not receive a logically-earlier message in the future, so the message is immediately processed. Also, when reading a shared page, the OS will speculate that its locally-cached version is up-to-date. This speculation will be aborted if it receives a write request timestamped before its read request.

2.3 Speculator

All works in this thesis use speculative execution at the operating system layer. To implement speculations at this layer, we rely heavily on the Speculator kernel [52], a modified Linux kernel with internal support for speculations.

A process running in a kernel context can call `create_speculation()` to have the process begin running speculatively. Immediately, a checkpoint is make of the current process by forking a new suspended process (referred to as the checkpoint process). The kernel's existing copy-on-write mechanism is used to preserve the program's address space in the checkpoint process. Other process state is duplicated during the fork. At any time, the kernel can call `commit_speculation()` to commit the speculation. When this happens, the checkpoint process is destroyed (without having run). Should the kernel instead call `fail_speculation()` to abort the speculation, the speculative process swaps identities with the checkpoint process and exits while the checkpoint process is awakened and starts re-executing the system call that started the initial speculation.

One powerful feature of Speculator is its ability to allow a speculative process to interact internally with the rest of the system. Speculator tracks which speculations a thread, or any other data structure, is dependent upon. In this way, aborting a speculation can roll back all state throughout the kernel that is causally dependent on the speculation. Each kernel object contains a list of dependencies and an undo log. Reading a shared kernel object causes the process to take on any extra dependencies of the object, creating a new process checkpoint if necessary. Modifying a shared object causes it to take on the process's dependencies, and an undo record is generated that can reverse the modification on an abort. Using this mechanism, speculative data can travel between processes via signals, files, sockets, pipes, and other forms of inter-process communication.

Output from a speculative process to an externally-visible device cannot be rolled back, so it is disallowed by Speculator. When possible (e.g, for sockets and terminals), the output is buffered and the program is allowed to continue. Otherwise, since the system cannot perform the action requested by the program, the speculative process must block and wait until it becomes non-speculative before it can continue.

CHAPTER 3

Custom Speculation Policies

What does it mean to make a speculation visible to other layers? This question lies at the core of our thesis. Intuitively, we call for tasks running at higher layers to somehow be aware of when they are executing speculatively. What should the task do with this knowledge? To make a speculation visible to a lower layer intuitively calls for externalizing some speculative state. How can the safety of our application be preserved if this is to happen?

To help us approach an answer to these questions, we construct a new way of thinking about speculative systems. We view an implementation of speculative execution as a system composed of two parts: (1) a *policy* that specifies what operations and values to predict, what actions to allow while speculating, and how to compare results; and (2) the *mechanisms* that support speculative execution, such as checkpointing, rollback, causality tracking, and output buffering. An application needs to make full use of both parts. The policy should be flexible enough to let an application use speculation as it requires, and the mechanism should be capable of implementing that policy while allowing the application to execute unimpeded. As we have seen in Chapter 2, speculation is used to accomplish many different goals. We would like to be able to capture these different needs within our definition of a "policy."

Existing systems typically conflate the mechanism and policy and will implement them together within a single layer, such as the processor, operating system, or application. Unfortunately, no single layer is well-suited to implement both policy and mechanism. Policy decisions are best done by higher layers in the system, such as applications, that understand the semantics of the actions that are being predicted and can more accurately predict a value and compare it with the actual result. In contrast, mechanisms that support speculation policies are best implemented at lower layers in the system (e.g., operating systems). Lower layers exercise more control over the entire system, enabling them to to propagate or coordinate speculations between applications. Implementing the mechanisms for speculative execution in the lower layer also frees application writers from re-implementing speculation for each application. By implementing speculation in a single layer, an application must choose between its need to have an effective policy and its need to interact with the larger system.

In this chapter, we show how to separate policy from mechanism in a speculation system. We implement a mechanism for speculation in the operating system, where it can easily propagate speculations between multiple applications, control the output from speculative applications, and be shared by multiple applications. We delegate policy decisions to applications, which have the semantic information needed to specify which operations to execute speculatively, what values to predict, what operations to allow during speculation, and what criteria to use when comparing predicted and actual values.

Separating mechanism from policy opens up a new design space regarding what behaviors a policy should specify and how to best describe them. An application-specific policy can address a number of issues at each phase of speculative execution:

- *Starting* the speculation: What actions are predictable? When should each speculation begin?
- *Performing* the speculation: How should output be handled? What data can be marked as speculative? How many resources should be used?
- *Ending* the speculation: Which results should be considered correct? How should the system recover from a misprediction?

Allowing applications to specify their own policies about when and how to speculate enables them to use speculative execution in ways that are difficult to implement using generic policies provided by lower layers. We demonstrate this by building a prototype speculation mechanism at the operating system layer with policies specified in user-space programs. Within this system, we modify three existing applications to demonstrate our approach:

- *Predictive application launching*: The *Bash* shell predicts the next command a user will type and executes it speculatively. An *X11 proxy* permits graphical applications to interact with the X server while being launched speculatively.
- *Firefox* performs certificate revocation checks while continuing to establish an SS-L/TLS connection with a server.
- A *Byzantine fault-tolerant (BFT) client* assumes that the first reply to a request is correct without waiting on consensus.

An OS-implemented speculation system lacks the abstractions needed to specify these features, while an application-implemented speculation system limits the scope of each speculation and complicates the development effort. To address these issues, our separated system allows custom policies to be specified in these applications by adding localized changes that reuse a common mechanism. Our changes allow predicted applications to hide 85% of their start time, reduce Firefox's SSL connection latency by 15%, and increase the BFT client's request rate by 82%. We do impose a trade-off on developers: an application using an optimized speculation implementation can improve on our results by 8%, although it must give up system support to do so.

This chapter makes the following specific contributions: first, we present a discussion of the rationale for separating speculative policy from the mechanism that implements it. Second, we use this discussion to design and implement a speculation system that places mechanism in the operating system and gives user-space processes control over policy. Finally, we demonstrate that our approach permits existing programs to use speculation for increased performance without requiring extensive modifications.

The rest of this chapter is laid out as follows. Section 3.1 describes how speculative execution works when implemented below the application. Section 3.2 explores the different behaviors a policy can customize, and Section 3.3 describes two issues that arise when applications control their own speculation. Section 3.4 describes what mechanisms we implement in the operating system to support custom speculation policies. Section 3.5

discusses the process used to locate and implement custom policies. Section 3.6 describes three case studies for using customized speculation policies and evaluates the performance improvements that custom policies enables. Section 3.7 describes related work, and Section 3.8 presents a summary of the work.

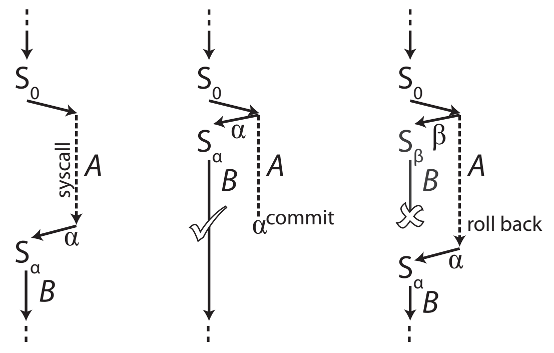
3.1 Generic Speculation

This section describes how speculative execution works when it is implemented below the application and thus does not understand the program's semantics. We refer to this as a *generic* speculation system.

Speculation can be implemented at many layers below the application, such as in hardware, a virtual-machine monitor, the operating system, or a language runtime. The layer at which speculation is implemented determines the natural unit of execution that the speculation system controls. For example, speculation implemented in a virtual machine monitor would control the execution of virtual machines, while speculation implemented inside an operating system would control the execution of processes. To make the discussion more concrete, our description assumes speculation is implemented in the operating system; the same principles apply to other layers below the application.

Implementing speculation in the operating system provides a good balance of semantic information and scope. The operating system understands the semantics of useful objects like processes, users, and files, yet is low enough in the software stack to control the execution of all applications. The natural unit of computation for OS-level speculation is a process, and the natural unit of state is a process's address space. The OS also sees objects such as files and sockets and manages related state (such as the file table) on behalf of processes. Processes communicate with the OS mainly through the system call interface.

Figure 3.1 illustrates the generic approach to operating system speculation. A speculation starts when the operating system predicts the results of an *action* (A). Actions are units of computation that have definite start and end points. An action causes a process's state to transition from one state to another; actions also may produce output. We refer to the difference in states as the action's *result*. An action is considered *predictable* if its result



(a) Sequential (b) Speculative Commit (c) Speculative Abort

Figure 3.1: Generic OS speculation. Part (a) shows a process in state S_0 execute syscall *A* with result α. When the syscall returns, the process continues to execute program action *B*. In (b) and (c), the system predicts the result of *A* and returns to user space speculatively while executing *A* in parallel. If the prediction is correct, the system commits the speculation (b). Otherwise, it aborts (c).

can be guessed at some point in time before the action completes. For OS-level speculation, actions are typically individual system calls.

Speculative execution allows predictable actions to execute in parallel with the program's future actions. When the operating system can guess the results of a process's action before it executes, the operating system marks the process as speculative, returns the predicted result to the process, and allows it to continue executing speculatively. In parallel, the operating system carries out the action and determines the actual result.

When speculating, a generic speculation system must ensure that no effects resulting from a missed speculation are visible outside the system. To hide misspeculations, the system must roll back all effects of the speculation. To support rollback, the system takes a checkpoint of the process (usually copy-on-write for efficiency) when the speculation begins.

We define the *boundary* of a speculation to be the collection of all objects whose state depends on a speculation. Initially, this boundary will include only the state of the process that initiated the speculative action. As the process interacts with the system, it may try to modify state outside its bounds by generating output. A generic speculation system may handle output in one of three ways, each of which meets the requirement of completely hiding misspeculations.

- *Expand*: the boundary of speculation is expanded to include the receiver of the output, and then the output is sent. When a new object (e.g., a process or file) becomes included in a speculation, a checkpoint of its state must be taken so it can be rolled back if the speculation fails.
- *Defer*: the write is deferred until the speculation commits.
- *Block*: the modifier's execution is halted until the speculation commits.

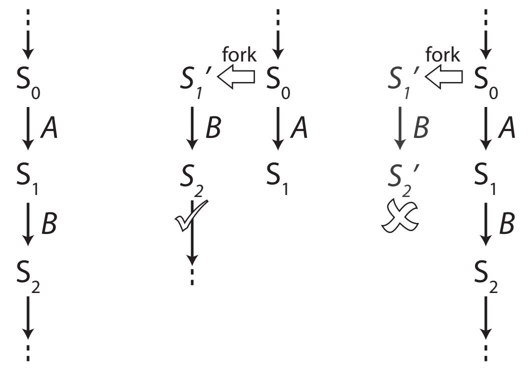
When the system finishes executing the action, it compares the predicted result with the actual result. If the actual result matches the predicted result, the system commits the speculation and releases any deferred output (Figure 3.1b). Otherwise, it aborts the speculation and rolls back all state within the speculation's boundary (Figure 3.1c).

3.2 Custom Policies

Because an application has more semantic information about its own behavior, its performance can be improved by using a speculation policy customized for that application.

A custom, application-specific policy can vary from a generic policy in several ways: creating speculations, managing output, evaluating results, and controlling the commit. Overall, custom policies benefit an application by letting it make more predictions, helping those predictions be more accurate, and allowing it to achieve more work while speculative.

Figure 3.2 shows an overview of how a sequential execution is parallelized using speculative execution with custom policies. An important distinction between OS generic speculation and custom speculation is which level controls the speculation. In OS generic speculation, the *operating system* executes the action that is being predicted and evaluates the result. In custom speculation, the *application* executes the action that is being predicted and evaluates the result. To allow the application to control the speculation, the system forks the process when the speculation begins. One copy of the process (left side of Figures 3.2b and 3.2c) incorporates the predicted result of the action and continues executing



(a) Sequential (b) Speculative Commit (c) Speculative Abort

Figure 3.2: Speculation wit custom policies. Part (a) shows a sequential process in initial state S_0 that executes actions *A* and *B*, moving its state to S_1 and then to S_2. Parts (b) and (c) show the same process predicting the result of action *A* and forking a speculative copy of the process that runs *B* in parallel with *A*. If S_1 and S'_1 are equivalent, the speculation can be committed (b). Otherwise, the speculation is aborted (c), and the process continues from state S_1. We call the left process in Figures (b) and (c) the *speculative process*, and we call the right process in Figures (b) and (c) the *control process*.

speculatively; we call this copy the *speculative process*. The other copy of the process (right side of Figures 3.2b and 3.2c) executes the action and compares the actual result with the predicted result; we call this copy the *control process*.

We explore three axes along which an application can provide a customized policy: creating speculations, handling output, and handling commits.

3.2.1 Creating Speculations

The most basic task in speculative execution is determining where to start and end the speculation, along with what value to predict for that interval. A generic speculation system is not suited to identify the best places to start and end a speculation. First, it sees only a subset of events issued by the process, e.g., system calls. Second, it has little information by which to determine which of these events are predictable: a certain system call may have a predictable result for one application but not another (e.g., reading a configuration file is more predictable than reading a user document). A generic speculation system may also fail to predict the result of the action, since the same action will often have different results for different applications.

An application sees and understands much more about its own behavior and semantics. For example, a program can start and end speculations at any line of code, rather than only at system calls. We define actions at this level to be an interval of program statements. This definition allows system calls to still count as actions, but it also lets the program speculate over many more regions, including arbitrary function calls. With so many additional actions visible, there is a greater opportunity to find predictable actions.

A custom policy on creating speculations lets the program specify which intervals of code are worthwhile to speculate on and how to predict the intervals' results. The program can pick its actions to be those at an abstraction layer that is easily predictable.

Selecting the right abstraction layer is crucial to locating predictable actions. Interfaces often exist to hide implementation details from the higher layers of a program, and we can take advantage of them to minimize the amount of state that must be predicted. Lower-level actions, themselves unpredictable, may work together to construct a high-level action whose effects are well-defined and whose outcome is predictable. Defining a high-level action can filter out the unpredictability of lower-level events and intermediate state changes that are not actually relevant to the overall task.

As an example, consider a program that calls the function `get_user_option()` to display a menu, specify a default choice, and wait for the user to interactively select an option. If we implemented our speculation in a slightly-lower layer of abstraction, the available actions concern the interaction with the menu itself. The program might find itself predicting which menu item the user would select next. At still a lower level, the program might try to speculate on the return value of the `read()` call that gets the user's next keystroke. (Note that this is all the generic system would see.)

By understanding the semantics of the high-level action, a custom policy would let the program speculate over the entire `get_user_option()` function to predict that the user will take the default option. The exact sequence of keystrokes that a user took to make a selection and the internal menu state are irrelevant details that get abstracted away to make the action predictable.

3.2.2 Output Policy

An application next needs to determine how its output should be handled while it runs speculatively. Recall that a generic speculation system must handle output by expanding the boundary of speculation, deferring the output, or blocking the speculation. Each of these handling strategies has drawbacks in certain situations:

- Expanding the speculative boundary involves more objects in the speculation. This increases complexity and increases the cost of a rollback. For example, if a heavily-shared object such as the X11 server or `/etc/passwd` became speculative, the speculation would quickly spread among other objects, and the entire system could become speculative (and thus non-responsive).
- Deferring the output prevents the receiver from getting the output and starting useful work. If the speculative sender is waiting on a reply from the recipient, it too will stop making any forward progress.
- Blocking on output is the safest, easiest option, but it performs the worst because it limits how far the application can speculate.

A generic system lacks information about the purpose of the output, the sharing patterns of objects that receive the output, how quickly that output needs to be sent, and how far the application could proceed speculatively after sending the output.

By specifying a custom output policy, an application can choose the best way to handle its output from among these options. With its knowledge of which actions are safe, what it is writing to, and whether it needs a reply, the application is in a better position to make this choice. For instance, an application can avoid deferring writes when it will spin waiting for a reply, and it can expand its speculative bounds only when it would not involve many other objects in the speculation.

In addition, the application may be able to violate the conservative restriction of completely hiding misspeculations, because the application may not care if the output produced by a misspeculation is rolled back. Hence, a custom output policy can specify a fourth strategy in addition to those available to the generic system: *allow* the output without expanding the boundary of speculation and without rolling back the receiver upon misspeculation. An application can safely follow this output strategy in the following scenarios:

- *The output does not modify external state*: Many networking applications use requests that return data without modifying important server state, such as HTTP GET or SQL SELECT requests. Since no state change needs to be undone on a rollback, these requests are safe to allow off the system.

- *The application provides its own safety guarantee*: Even if the system cannot roll back the effects of an output, the application may be able to ensure that on a rollback, the effects of its output will be undone. To guarantee this behavior, a networking application might implement a distributed speculation system by tagging its messages with its outstanding speculations and informing recipients when a rollback occurs.

- *Inconsistent output can be tolerated on a rollback*: A study by Lange shows that users are able to tolerate a limited amount of speculative and inconsistent information being displayed on their screen in exchange for faster performance [40].

By customizing its output policy, an application can ensure that its safe output does not cause it to prematurely halt forward progress. A customized output policy also directs the system to handle the unsafe output using the most efficient and appropriate strategy.

3.2.3 Committing

When the action whose result is being predicted finishes, the system must decide to either commit or abort the speculation. If the actual result is identical to the predicted result, the speculation can be committed. Without knowing what the application uses the predicted values for, this is the only condition under which a speculation can commit. If a generic system detects any differences between the actual and predicted result, it cannot determine if that difference is significant, so it must be conservative and abort the speculation.

However, some applications can tolerate differences between the predicted and actual states. Custom commit policies let the application specify what differences can be tolerated and how to to deal with those differences. Thus, custom commit policies can broaden the criterion for correct predictions from being *identical* to being *equivalent*. A custom commit policy can use this flexibility to commit more speculations, thus reducing the number of speculations that roll back and preserving more work. We consider four ways that differences can be equivalent while not being identical.

First, some differences in process state are not semantically important to a valid execution and may be ignored. For example, different patterns of `malloc()` calls may result in data structures being allocated in different locations. This is safe to ignore if there are no inconsistent pointers to these structures. Likewise, the exact contents of unused stack frames can differ if two executions take different code paths, but these are not significant.

Second, other differences in results may be unused and can also be ignored. For example, a reply to an RPC may convey the complete metadata of a shared object, but the application may only examine its time stamp. Differences in other parts of the reply can be ignored. Another example is when the application uses the time stamp only by comparing with another value. All results where the time stamp is less than the other value are in the same equivalence class.

Third, some state differences may affect execution, but the semantics of the changed state may permit updates to be lost. For instance, a cache may acquire an entry that is not predicted, but the cache's semantics allow it to drop the entry when needed.

Finally, some differences matter but can be imported into the speculative state. To do this, the control process can *forward* the difference to the speculative process to be merged into its current state. Continuing the previous example, although it may be valid for a cache to lose unpredicted entries, the program may wish to preserve them for better performance. If the speculative process has not read or written the differing state before it is forwarded by the control process, the merge can be performed easily without worrying about read/write conflicts. If the speculative process *has* read or written the differing state before it is forwarded by the control process, the updated state can be passed as a message to the speculative process.

3.3 Issues with Separation

Despite the benefits mentioned in the previous section, splitting the mechanism and policy into different layers causes two new issues that must be addressed. Both issues arise because the application participates in controlling its own speculation.

3.3.1 Committing State

Our control processes lacks effective isolation between two logically distinct portions of application state: the state used to *control* the speculation is co-mingled with the state *effected* by running the predicted action. The logic carrying out the predicted action and the logic controlling the speculation execute within the same address space (the control process). As a result, there is no easy way for the system to separate the state used by the logic controlling the speculation (which should not be preserved), the predicted results (which must be checked for equivalence), and other unpredicted state (which could be discarded, forwarded, or cause a rollback). That is, if a particular change is detected, it is not obvious how to handle that change.

This issue does not arise in a single-level system. For instance, when an OS speculates on a system call, the process switches to a separate kernel stack, isolating the speculative control logic from the application state. Furthermore, the effects of the system call on the process's state are well-defined. As a result, it is easier to check for equivalent results, and there should never be any completely unpredicted state changes.

It is left to the application's commit policy to decide how to disentangle these pieces of state. In the applications we have modified to use custom speculation, this has not been a significant burden. Still, it is an added complexity that we would avoid if possible.

3.3.2 Multi-threaded Speculation

The issue of separating state is compounded by multi-threaded processes. Our description of custom speculation so far has assumed that an application has only a single thread, which both executes the action and uses the result. With speculation, we fork this thread into a control thread and a speculative thread. The speculative copy of the thread continues using the predicted result, while the control copy of the thread executes the action and commits or aborts the speculation.

In contrast, with multi-threaded processes, a single address space is shared among many different threads. Some of these threads execute the predicted action and control the speculation; other threads use the results of the predicted action; and some threads may be independent of the predicted action. Forking a process when a speculation begins causes all threads within that process to be copied.

We designed a solution to these issues that lets us speculate using multi-threaded programs. The key issue is deciding which threads to start in each process.

We first consider the case in which the predicted action involves only a single thread (A); other threads may use the predicted result or be independent of the action. Thread A starts a speculation by predicting a result for the action and forking a speculative process. Thread A must run in the control process to execute the predicted action and control the speculation; thread A may also continue in the speculative process after skipping over the portion of its execution that is being predicted.

Threads that use the predicted result must run in the speculative process to achieve the desired parallelism; they cannot run in the control process since they would have to wait for the predicted result.

Threads that are independent of the predicted action may run in both of the control and speculative processes, but this duplicates their work and wastes computing resources. To avoid wasting work, the independent threads are allowed to run in only one of the processes; they are blocked in the other process. Most speculations are more likely to commit than abort, so we run the independent threads in the speculative process (which is more likely to survive the speculation than the control process). While we could merge the changes from the independent threads into the surviving process, this would require us to modify the independent threads to deal with the speculation.

In the general case, multiple threads may be involved in the predicted action. All threads involved in the predicted action must run in the control process, and they must skip the predicted action in the speculative process (if they run in the speculative process). Threads that cooperate on the predicted action must also cooperate on starting and controlling the speculation.

When one thread in a multi-threaded process starts a speculation, our system relies on that thread to determine which of the other currently-running threads should also be started in the control process. Unless otherwise specified, all other threads are assumed to be independent, and remain running only in the speculative process.

3.4 Mechanism Design & Implementation

Custom policies were introduced in Section 3.2 by describing what behaviors an application should be allowed to customize. In this section, we discuss how our mechanism layer is built and how applications can express those policies.

3.4.1 Overview

Our underlying mechanism for speculative execution is based on the Speculator kernel (see Section 2.3). We introduce custom policies to this system by creating a group of new system calls, which are described in Section 3.4.2. At a high level, policy decisions are executed from within the process, and the system calls are used to direct the mechanism appropriately.

When a speculation starts, the system creates two separate processes (a control process and a speculative process), each with their own address space. The isolation provided by having separate address spaces is crucial: state from the speculative process should not violate causality by influencing the execution of the control process. If a conflict is detected, the system may conservatively abort the speculation. The application should gracefully resume executing as if no speculation were created.

3.4.2 Policy API

Applications implement custom policies through a new set of system calls. An overview of each call is given in Figure 3.3.

`spec_fork(`*out* `status,` *out* `spec_id)`
Begin a speculation.

`commit(`*in* `spec_id)`
Commit a speculation.

`abort(`*in* `spec_id)`
Abort a speculation.

`set_policy(`*in* `fd,` *in* `new_pol,` *out* `old_pol)`
Set the file's output policy, returning the old one.

`get_specs(`*out* `spec_id_list)`
List the current process's uncommitted speculations.

`spec_barrier(`*in* `spec_id)`
Block until the given speculation has committed.

`start_threads(`*in* `thread_id_list)`
Start additional threads in the control process.

Figure 3.3: System call API used by an application to construct custom policies.

To create a new speculation, the application invokes `spec_fork()` and splits into control and speculative processes. After the control process executes its predicted action, it can call `commit()` or `abort()` for the speculation.

Custom output policies are specified using the function `set_policy(fd, policy)`. Each write operation can use a per-file policy or, if that is unspecified, a per-thread default policy. Policies specify one of the strategies for handling output described in Section 3.2.2 or DEFAULT. A single write operation can use its own policy by wrapping it with `set_policy()`.

We permit processes to view the status of ongoing speculations. A process can get a list of its current speculative dependencies by calling `get_specs()`. The kernel also provides a socket that broadcasts `spec_id`-s as they are created, committed, and aborted. We also found it useful to allow a speculative process to voluntarily limit its own resource usage. By calling `spec_barrier()`, a process can halt its execution until some or all of its dependencies have committed.

In a multi-threaded application, only the thread that called `spec_fork()` is initially started in the control process. All other threads in that process start blocked. If the action requires that other threads be running in the control process, they can be explicitly woken by calling `start_threads()`. If the speculation aborts, all threads will automatically be woken in the control process. (Note that all threads in the speculative process are active by default.)

3.5 Design Process

We envision the use of custom speculations as a design process consisting of three steps. First, a developer must locate interesting speculation points in a program. Second, custom speculations must be implemented safely. Finally, the system as a whole should be examined for additional optimization.

3.5.1 Determining Actions

There are three generic guidelines that should be followed when locating a suitable action to predict. First, executing the action should take longer than the overhead of creating a speculation (i.e., the cost of a fork). Blocking I/O operations (e.g., waiting for user input or network messages) often greatly exceed the overhead cost—our own case studies focus on these operations. Lengthy computations may be appropriate, as long as there are available cores to do the work in parallel. Second, it is important that the speculative process be able to make forward progress. Using a custom output policy may remove some blocking points, thus allowing more progress. Finally, the result of the action must be predictable. By using a custom commit policy, it is sufficient to predict an equivalent result rather than an identical one.

Our system imposes additional constraints on the selection of an appropriate action. We rely on the program to explicitly verify that all effects of the action were predicted. To do this correctly, the developer must be able to understand precisely the effects of the action on the local process's memory. Clean, narrow interfaces for accessing and modifying local state significantly aid the developer in performing this task. An ideal interface cleanly separates pure functions, which do not change local state, from the mutating functions. In our experience, suitable interfaces are often found at the boundary between program

```
int count;  /* Global state */

int* foo() {
  ...
  count++;
  return ptr;
}

int* spec_foo() {
  int p_cnt = count + 1;
  int p_ret = 1;
  (stat, spec_id) = spec_fork();
  if (stat == SPEC) {
    count++;
    result = new int(p_ret);
  } else if (stat == CONTROL) {
    result = foo();
    if (count == p_cnt && *result == p_ret)
      commit(spec_id);
    else
      abort(spec_id);
  }
  return result;
}

void work() {
  x = spec_foo();  /* Replaces foo() */
  p = set_policy(fd, ALLOW);
  send(*x);
  set_policy(fd, p);
}
```

Listing 3.1: Basic structure for predicting the result of simple function call.

modules. If an action seems too convoluted, it may be more reasonable to look at a different abstraction layer.

3.5.2 Implementing Custom Policies

Once a suitable action has been located, it is necessary to implement the policy in code as API calls and state modifications. We use the code in Listing 3.1 as a running example of how to use our API to predict the results of running the function `foo()`.

We found it useful to work with actions defined by a single function. When the code is structured in this way, we can write a wrapper function (`spec_foo()`) that isolates our policy implementation from the action and surrounding code.

The developer is responsible for deciding how to predict the return value and side effects of executing an action (lines 10–11). Once that prediction is made, `spec_fork` can be called to split the application into speculative and control processes. The speculative process should update local state as if the action had completed with the predicted result (ln. 14–15). The control process should execute the action (ln. 17) and then, to implement the commit policy, explicitly verify that the changed state matches the prediction (ln. 18).

It is important for correctness that all relevant side effects of the action be predicted (in the speculative process) and verified (in the control process). In the example, if `count` were ignored in the prediction (i.e., by omitting ln. 14), it would lead to odd program semantics where `count` appears to increment only when the speculation aborts. Not all differences are relevant: `foo` might have different dynamic memory allocation patterns from the speculative process's fast update (ln. 15). This difference does not affect program semantics. Hence in the example, only the value of the returned object is checked. It is a challenge to decide which state is relevant. For this reason, it is crucial that the developer be able to understand the behavior of the action.

To selectively allow speculative output on a per-message basis, a program may wrap its I/O functions with calls to `set_policy` (ln. 28–30). The developer should ensure that the receiver can handle potentially-incorrect data. Also note that after rolling back, messages sent while speculative might be retransmitted.

Although we support executing multiple threads in the control process, a single thread that makes blocking operations on local data is preferred. Acquiring a lock to access shared data may introduce a deadlock if the lock holder is not running in the control thread. If the system can detect the deadlock, the speculation can be aborted, freeing all other threads. We suggest grabbing needed locks or making local copies of data structures before starting the speculation. If multiple threads are required to run in the control process, they should synchronize with each other first before executing a `spec_fork`. The prediction must include the state changes due to *all* threads' executions.

Application	Custom Policy		
	Start/End	Output	Commit
Predictive launch:			
Bash	Y		Y
X Proxy		Y	Y
Firefox	Y	Y	Y
BFT	Y	Y	

Table 3.1: Speculative applications. A “Y” indicates that the application has a custom policy defined for that category.

3.5.3 Optimization

By examining the behavior of the system in a few key areas, it may be possible to further optimize performance. When a speculation fails, it might be the result of an overly-precise commit policy. An expanded definition of “equivalence” might allow a greater number of speculations to commit. When a speculative process blocks, it could indicate the need for a more permissive output policy. If the process is waiting for output to be released, it could be worthwhile to consider whether it is safe to allow that output. However, the system’s performance as a whole may suffer if the boundary of speculation expands too far. If a highly-shared system object becomes speculative, this may suggest that a more-restrictive output policy is needed.

3.6 Case Studies

To evaluate the effectiveness of our split-layer speculation system, we look at three case studies. We modify each application in the study to add a feature that uses custom policies to achieve greater parallelism. Table 3.1 shows the applications and which policies they use. For comparison, we discuss the difficulties involved when implementing each feature in single-layer systems at both operating system and application layers. To quantify the changes needed to implement these features, we measure the Lines of Code (LoC) added and modified in each application (excluding blank lines, comments, and braces). Finally, we quantify the improvement in performance due to each feature. Our test system uses two Xeon single-core 3 GHz processors with 8 GB of RAM.

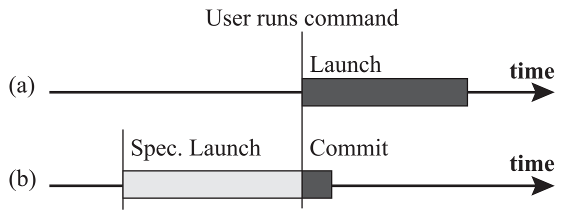


Figure 3.4: Process execution time when launched speculatively. In part (a), a process is launched normally at the time the user invokes it. In part (b), the program is launched speculatively ahead of time. We measure the execution time after the user invokes a command (dark bar).

3.6.1 Predictive Application Launching

We make use of custom speculation policies to improve perceived application startup time by predicting the launch of an application and speculatively starting it. This will not decrease the actual time needed to launch the application, but part of that time may be overlapped with the user's think time. As a result, the system will appear more responsive.

We first quantify the potential performance benefit from this technique. When it is possible to successfully predict the next program far in advance, how much work of the program launch can be hidden? Figure 3.4 illustrates our method for examining the capacity of non-interactive programs to launch speculatively. In a normal launch (Figure 3.4a), a program starts executing when it is invoked by the user's shell. We measure the run time of the process from its invocation to termination. In a speculative launch (Figure 3.4b), we begin executing the program before it is requested. Once the speculative program quits making progress, we invoke the application—committing the speculation—and measure the program's run time from that point. We examine two non-interactive applications: building a LaTeX paper, and building the Bash shell via `make`.

Interactive graphical applications do not automatically terminate, so we examine their load time from invocation instead of their run time. We end our load time measurement when the rate of X11 messages sent by the application falls below 200 messages in a 100 ms period. This threshold is arbitrary, but it effectively distinguishes drawing splash screens and main windows from handling smaller incidental actions, like redrawing buttons as the pointer moves across a window. We examine two interactive applications: GIMP 2.2 and OpenOffice 3.1.1.

Application	Normal Launch (s)		Speculative Launch (s)
	Warm \$	Cold \$	
LaTeX build[†]	2.66 ± 0.03	4.72 ± 0.07	0.092 ± 0.001
Bash `make`[†]	45.1 ± 0.02	49.0 ± 0.04	0.19 ± 0.001
GIMP[⋆]	5.1 ± 0.3	8.4 ± 0.5	0.72 ± 0.03
OpenOffice[⋆]	3.33 ± 0.05	11.8 ± 0.08	0.29 ± 0.03

Table 3.2: Application run times[†] and load times[⋆] for non-interactive and interactive programs, respectively. Normal launches are examined with both warm and cold disk caches; speculative launches were not affected by cache state. Each value is given in seconds and is the mean of 10 runs, with 95% confidence intervals.

Table 3.2 shows the run times and load times of our test applications when launched normally and speculatively. Because of the high impact on load times, we also varied the state of the disk cache. When launching speculatively, we did not find a significant difference in run/load time due to cache state. Although application load times are significantly decreased when using a warm cache, they are not eliminated. When applications are speculatively launched before invocation, almost all execution time spent running/loading the program can be performed before the program is invoked. Compared to a normal launch with a cold cache, at least 91% of the run/load time is capable of being hidden. Even with warm caches, 85% can be hidden.

Section 3.6.1 describes our modifications to the Bash shell that lets it take advantage of this potential by predicting the user's next command line and executing it speculatively. By itself, the changes to Bash are sufficient to benefit non-interactive commands. Section 3.6.1 describes how we implement an X11 proxy that lets graphical programs benefit from a speculative launch.

Bash

We modified a Bash 3.2.48 shell to predict the next full command line the user will type and begin running it speculatively. Bash predicts one command at a time, starting when the shell prompt is first displayed.

To perform the prediction, we re-implemented the EMA online machine learning algorithm [47], which predicts the next line based on the command history. One could also imagine developing an algorithm that alters its guess as the user types. Finding the best predictor is an orthogonal problem; our concern is how to effectively design a system to make use of the predictions.

Following our design process, we identified the interface between Bash and the Readline library as an ideal modification point. We used the basic pattern described in Listing 3.1 to wrap Bash's call to `readline()` (in Bash's `yy_readline_get()`), which accepts user input and returns it in a new buffer. Other program state is not modified. Our wrapper calls into EMA to generate a predicted buffer. The speculative process returns a copy of this buffer. The control process makes the call to `readline()` and compares the two strings. Note that the two executions return different memory allocations. In this program, only the buffer contents are relevant, so the commit policy makes only that comparison. Other state is assumed, without verification, to not have changed.

Later observation led us to implement two additional changes. First, we found that when a user hits Ctrl-C to interrupt Bash, the signal handler uses `longjmp()` to (incorrectly) bypass our wrapper. We modified the function `throw_to_top_level()` on the interrupt control path to abort outstanding speculations when this happens. Second, we found that tab completion could add spaces to the end of command lines. In response, we added a custom equivalence policy that normalizes commands before comparison.

Overall, only two function in Bash needed modification to permit speculative launching. Basic command prediction used 56 LoC inside Bash to invoke our EMA predictor (433 LoC). The equivalence policy added 36 LoC, mostly text manipulation functions, for a total of 525 LoC. Because Bash relies on the system's default output policy to maintain safety for arbitrary applications, no code was needed to implement the output policy. To put these numbers in perspective, the full source code for Bash is over 100K LoC.

X Proxy

Graphical applications send and receive messages over a socket to communicate with the X server. Following the generic policy used by Bash, a speculative application that attempts to use this socket will either have all of its messages buffered, preventing it from loading, or it will force the X server to become speculative, preventing further user interaction. Neither result is desirable for speculative launching.

The generic policy is unnecessarily restrictive. While loading, an X application issues many requests that read global state or modify application state without resulting in any user-visible output. These messages can be safely exposed to the X server. In particular, applications can create windows and set their properties without exposing those windows to the user (*mapping* the window, in X terminology). The X protocol is designed to operate asynchronously, so those few messages that do result in a visible change can be buffered and released only when the speculation commits.

We design and implement an X proxy that sits between the application and the X server to selectively permit messages through the boundary of speculation. By placing this functionality in a proxy, we can support arbitrary unmodified applications and avoid modifying the core X server.

For ease of development, we modify an existing proxy: xtrace 1.0.2 [44]. When a new application connects to it, the proxy forks a new server, which becomes speculative immediately after accepting the connection. The proxy takes advantage of system support for buffering output to avoid complicating its own message-handling code. Using custom output policies, requests to map, unmap, or delete widows are deferred. All other requests are allowed. The proxy rewrites sequence numbers in each message to correct for the buffered messages.

When the speculation commits, the system releases the buffered messages, and the application begins to draw its main window. The proxy is notified of the commit and performs a custom commit action: it adjusts its sequence number rewriting algorithm for the newly-released messages. If the speculation aborts, the proxy will exit, breaking its connection with the X server. The X server can recover by releasing application-held resources without rolling back.

Implementing these changes added 280 LoC to xproxy (itself 7K LoC). Most code additions are used for sequence number rewriting.

3.6.2 Firefox Certificate Checks

Verification can be a slow process whose outcome is often predictable. We use the Firefox 3.5.4 web browser as an example of how to execute verification tasks in parallel with the rest of an application. The task we speculate on is Firefox's verification of a server's public certificate.

Many Internet protocols use the SSL/TLS protocol to establish a secure link between client and server. To establish a session, Firefox sends a handshake and receives the server's public certificate. It then validates this certificate by contacting the certificate's issuer. Finally, if the certificate is valid, Firefox exchanges random data with the server to derive a session key. Encrypted data can then be sent. We modify Firefox to predict that certificates are valid and speculatively agree on a session key. The data stream should be delayed until the validation is committed.

It would be difficult for a generic speculation system to provide this feature. First, the generic speculation system would need to distinguish the requests used to verify the certificate from other network messages. Second, it would need to predict the entire reply to the client's verification request, which is especially difficult if this certificate has not been previously verified. Furthermore, once the speculation has started, the generic system must treat further output conservatively and prevent it from leaving the system.

Speculation could also be implemented entirely within Firefox. However, this would require the programmer to implement a custom checkpoint mechanism, and such a mechanism would require extensive code modifications throughout the program because Firefox is not written to isolate its state. Furthermore, the programmer would need to manually block most output while speculative.

To express this feature using custom speculations, we create a variation of the function `ocsp_GetOCSPStatusFromNetwork()` in the NSS component, which requests the status of a certificate from a remote server and caches the result. Our speculative process assumes the verification succeeds, so it places a fake success record in the cache before returning. We also use a custom output policy that allows SSL handshake data to be sent: socket output is allowed around some calls to `ssl3_GatherData()`. Certificate prediction and

Site	Spec. (ms)	Normal (ms)	Speedup
Google Accounts	297.6 ± 31.9	330.3 ± 32.7	9.9%
Windows Live ID	416 ± 46	501 ± 43	17%
Chase home page	310 ± 51	382 ± 46	19%

Table 3.3: SSL connection establishment time. Time taken to establish the first SSL connection to various sites, for speculative vs. unmodified Firefox. Error values show 95% confidence intervals. Despite the high variance, a T-Test confirms with 94% confidence that there is latency reduction when using speculation.

cache modification used 122 LoC, and the output policy was specified in 27 LoC. For comparison, the certificate validation code alone takes 8.5K LoC.

We encountered two difficulties during development. First, by default the validation request is handed off to a dedicated thread that performs simple requests. We did not expect multiple threads to be involved, and the dependency prevented our speculation from succeeding. The easiest fix was to eliminate the dependency by sending the request in the validating thread. Second, sometimes a chain of certificates must be validated. Since the speculative process only inserted a fake cache record for the first certificate, subsequent cache modifications by the control process were being lost. To preserve the data, we implemented a custom commit policy that forwards (via a message buffer in shared memory) the verification response for all certificates from the control process to the speculative process. Forwarding added 90 LoC, for a total of 239 LoC changed in Firefox.

To evaluate the impact of this feature on performance, we used a packet analyzer to measure the amount of time taken to establish an SSL connection with and without speculation. Note that certificate verification is only one step in session establishment. Our results are presented in Table 3.3. Overall, our improvement decreases the time it takes to establish an initial SSL connection by an average of 15% when certificates have not been revoked.

3.6.3 BFT Client

We next examine a client in the PBFT-CS protocol, which we fully explore in Chapter 4. We develop the PBFT-CS protocol to decrease the perceived latency of executing requests on a Byzantine fault-tolerant (BFT) cluster. We discuss the full motivation for this protocol later. For now, we are only concerned with how the client is structured.

BFT services are accessed through a shared library using an RPC interface: clients submit requests and wait for the service to return a reply. Because each reply may come from a faulty server, it is necessary to wait until a quorum of authenticated matching replies is received before the client can determine the correct reply. Servers must coordinate their execution of requests; consequently each operation typically has high latency. PBFT-CS observes that the first reply is usually correct and allows a client capable of speculation to continue executing before the reply is known to be correct. Further requests encode speculative dependencies so that the service can squash aborted requests. As a result, the client sees lower latencies for its requests and it can pipeline requests to increase its own local throughput.

This client for this case study is simple enough that we were able to construct one that implements its own lightweight checkpoint system. This is a single-purpose application-implemented speculation system. We compare this client against another client that has been implemented to use custom policies.

We can see several examples of custom policies in this client description. The BFT code decides when to make a prediction (after receiving one reply), what to predict (that the first reply will be validated), which output to allow (additional BFT requests, with modification), and when to commit (after receiving enough replies).

Our policy-based client implements its speculation logic entirely within the BFT shared library. We modified the inner message-handling routine to expose intermediate results. Then, from a layer between the internal functions and the client, we use `spec_fork()` to implement our own custom start policy. As results are returned, our layer associates the reply with the current dependency set (from `get_specs()`) to be encoded on future requests. We set the output policy on BFT sockets to allow all messages to be sent. We implement a default commit policy by requiring the actual reply to be identical to the predicted reply. These internal changes and policies were implemented in 221 LoC, out of 17K LoC for the full library.

By using custom policies, our modified BFT library can be used by any existing BFT application without further modification. Those applications can also specify their own policies for other uses without conflicting with those set by the BFT library.

In contrast, the application-implemented client is tied to the service that is using it. Instead of being written as a sequential process that uses blocking operations (the normal RPC interface), this client uses a main event loop. Making the logic event-based forces state to be isolated and saved outside of the stack, so checkpoints can be safely taken between events using memcpy(). Other applications have far more state (that may extend into the OS, if open files are considered), which will require more complex checkpointing logic.

To interpose on output, the client logic is written not to perform output directly. BFT requests are queued and handled by the mechanism so that checkpoints can be created correctly. Other application output must be queued so it can be released only when its dependencies have committed.

We see the policy-based library as an improvement in programmability. It is also necessary to consider the performance trade-offs involved when selecting between using a policy-based system or an application-implemented system. From PBFT-CS, we evaluate a simple shared counter service with a single operation that increments the counter and returns its value. The client simply executes a fixed number of requests in a tight loop.

We examine this application from two perspectives. First, we consider the improvement of our client's performance due to speculative execution. Second, we compare two different implementations of speculation: our policy-based system and an application-implemented system tuned for the application. This comparison lets us quantify the performance cost we incur by relying on heavier, generic checkpoints.

The benefit of an application-implemented speculation system is a small performance advantage over our speculation system. Figure 3.5 compares a non-speculative client against speculative clients implemented in both policy-based and application-implemented systems. We vary the amount of network latency between each server and see how it affects each client's throughput when accessing a lightly-loaded server.

Both speculative clients perform much faster than the non-speculative one. The policy-based client lets the client issue 82% more requests per second than the non-speculative client with latencies above 0.5 ms. The client using application-implemented speculation

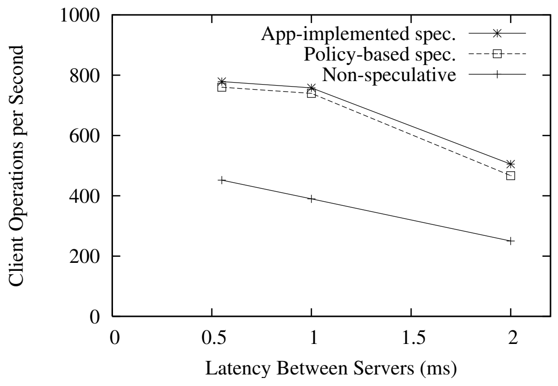


Figure 3.5: Comparison of BFT clients.

employs a checkpoint and restore mechanism that is tuned to the application. Hence, it has less overhead and is able to issue 90% more requests per second than the non-speculative client (an 8% improvement over our generic mechanism). In exchange, the development effort for the client is greatly increased, and it cannot expand its speculative boundary beyond the process itself. A developer must balance these trade-offs when deciding how to implement a feature speculatively.

3.7 Related Work

Fast Track [32] is a speculative runtime environment that allows applications to direct speculations over their own execution in a similar style to our custom policies. A programmer invokes `FastTrack()` to fork and and let one branch become speculative, like `spec_fork()`. Each side executes different version of the same action that are predicted to be *identical*: a fast but unsafe version and a slow, correct one. We go beyond the Fast Track model by giving the programmers greater control over when to commit and abort speculations in the presence of state differences that may be irrelevant. Our system also allows applications to specify a custom output policy and to speculate based on the actions of multiple coordinating threads. Fast Track, being implemented in the language compiler and runtime, cannot expand its boundary of speculation beyond its own process.

Prospect [77] is a compiler-based platform to generate programs that execute a fast program variant speculatively along with a slow variant that can include additional safety checks. Speculative system calls are allowed, although their effects are only made visible to other processes after a commit. Prospect also commits on equivalent, rather than identical, states. However, this is not verified in current implementations. In the context of our work, applications modified by Prospect could have benefited from the existence of a shared kernel mechanism to handle speculative system calls that would have allowed it to specify a default *defer* output policy. One could also view this project as an implementation of speculative mechanisms and policy at a low layer (the language and runtime) without considering the application semantics.

Crom is another framework that allows web applications to control their own speculations [50]. This mechanism is implemented as a JavaScript library that lets web application developers predict upcoming UI events. Developers flag individual events and provide lists of likely values for input controls. Equivalence functions are specified to let the system determine which speculative executions could match the user's actual event. The programming model for JavaScript is simpler than that for arbitrary binaries. Hence, custom policies must deal with a wider range of actions. Crom does not provide an analogue to custom output policies for its two I/O actions: network requests generated by the speculative code are sent and writes to the screen are kept hidden until a commit. Speculations capture the full state of the DOM tree and are isolated from each other, so causality tracking is not needed in this system.

We broadly categorize other work by considering what it is predicting, how much control it gives to applications, and what layer in the software stack implements the mechanism.

Speculative parallelism. Our work is closest to other systems that are designed to execute sequential code segments concurrently. Thread-level speculation (TLS) systems execute blocks of sequential code in parallel on separate threads, predicting that there are are *no memory conflicts* between the blocks [75]. TLS systems provide fine-grained parallelism, and the selection of the blocks is often driven by automated program analysis. The mechanism needed to support speculations at this granularity often has problems rolling back in the presence of system calls or I/O operations, so these are disallowed while speculative. Our system is built to support speculations at a much coarser granularity, and we consider system calls and I/O to be good sources for predictable actions. Because our system predicts *state* instead of *read/write sets*, the programmer can specify what value should be read by future reads.

Transactions. Speculative execution is similar in many ways to atomic transactions, and thus our system is similar to systems that provide operating system support for application transactions, such as QuickSilver [69] and TxOS [58]. Both transactions and speculation execute actions in parallel with other code, and both can commit or abort the action. The difference between transactions and speculation is the relationship between the action and other code. With transactions, other code executes in parallel with the action (with varying degrees of isolation [25]). In speculation, the outcome of an action is being predicted, and other threads are continuing based on that prediction.

Transactional memory and optimistic concurrency control are uses of transactions that also leverage a prediction [28]. As with TLS, these uses of transactions predict that there are no read/write conflicts between concurrently executing threads.

Generic speculation. There are many examples of generic low-level systems that do not take advantage of application semantics. Speculator originally predicted only system-level events such as NFS calls and disk syncing [52, 54]. Pulse speculatively resumes threads that are waiting for a resource to see if they will deadlock [43]. The Time Warp system lets processes in a distributed system run speculatively under the assumption that all their messages arrived in the correct program order [31]. Ţăpuş et al. also performed similar speculations for a distributed shared memory system [78]. These systems begin speculations only on system-visible events, and either disallow other output or handle it conservatively.

Systems offering customization. The Atomos programming language offers open transactions, which allow a thread to commit its writes back to memory while inside an uncommitted transaction [8]. Our custom output policies also allow for the same behavior, though we also consider blocking and expanding speculative boundary. The Mojave compiler also exposes an interface to start, commit, and abort speculations [72]. During a speculation, isolation is preserved, and since this is a runtime-based system, most system calls are not allowed. In Fast Track, the application customizes the actions being predicted and the predicted result, but not other policies.

Custom speculation implementations. The work by Lange et al. on speculative remote displays is an example of a program that uses an application-implemented speculation system [40]. They built a remote VNC viewer that predicts screen updates and displays the speculative view to the user. The authors also found that RDP events are also predictable, but they did not attempt to build a viewer, citing RDP's reliance on client state.

3.8 Chapter Conclusions

In this chapter, we explored the advantages of separating the mechanism to support speculative execution from the policy that describes what needs to be done. Applications that wish to use speculative execution are freed from the burden of implementing their own mechanisms such as checkpointing, rollback, causality tracking, and output buffering. Instead, they can focus on defining when to begin speculating, what results to predict, how output should be handled when speculative, and when to commit the speculation.

We demonstrate the effectiveness of our mechanism/policy split by examining three different applications that can be easily modified using our shared mechanism. First, our system reduces the startup time of programs by at least 85% when the program's launch can be predicted. Secondly, the latencies of establishing secure connections on Firefox are reduced by 15%, as our new mechanism/policy split allows it to perform certificate verification in parallel, partially removing it from a critical path.

Finally, the BFT client shows the low trade-off between performance and convenience in our system. While using an optimized application-level speculation mechanism gives an 8% performance improvement over our separated speculation system, its use prevents the application from interacting with the rest of the system while speculative.

CHAPTER 4

Fault-Tolerant Protocols

For some applications, it is not enough to be able to communicate speculatively with other components on the same machine. Applications are increasingly being developed that depend on networked services for part or all of their functionality. The utility of sharing speculative state with local processes is greatly diminished if the speculative application spends its time waiting to communicate with a remote server. For such applications, we would like to be able to make local speculations visible across the network to remote services. This chapter develops one example of how we can expose application-layer speculations to network protocols: we develop a Byzantine fault-tolerant protocol that allows clients to safely issue speculative requests.

As dependence on network services increases, so too should the services' ability to tolerate faults increase. Replicated state machines [70] provide a general methodology to tolerate a wide variety of faults, including hardware failures, software crashes, and malicious attacks. Numerous examples exist for how to build such replicated state machines, such as those based on agreement [11, 16, 35, 39] and those based on quorums [1, 16].

For replicated state machines to provide increased fault tolerance, the replicas should fail independently. Various aspects of failure independence can be achieved by using multiple computers, independently written software [4,65], and separate administrative domains. Geographic distribution is one important way to achieve failure independence when confronted with failures such as power outages, natural disasters, and physical attacks.

Unfortunately, distributing the replicas geographically increases the network latency between replicas, and many protocols for replicated state machines are highly sensitive to latency. In particular, protocols that tolerate Byzantine faults must wait for multiple replicas to reply, so the effective latency of the service is limited by the latency of the slowest replica being waited for. Agreement-based protocols further magnify the effects of high network latency because they use multiple message rounds to reach agreement. Some implementations may also choose to delay requests and batch them together to improve throughput.

In this chapter, we explore how to use speculative execution to allow clients of replicated services to be less sensitive to high communication latencies by overlapping computation with communication in agreement protocols. We observe that faults are generally rare, and, in the absence of faults, the response from even a single replica is an excellent predictor of the final, collective response from the replicated state machine. Based on this observation, clients in our system can proceed *speculatively* after receiving the first response, thereby hiding considerable latency in the common case in which the first response is correct, especially if at least one replica is located nearby. When responses are completely predictable, clients can even continue before they receive any response.

Speculative execution proves safety in the rare case in which the first response is faulty. By tracking all effects of the speculative execution and not externalizing speculative state, our system can undo the effects of the speculation if the first response is later shown to be incorrect.

Because client speculation hides much of the latency of the replicated service from the client, replicated servers in our system are freed to optimize their behavior to maximize their throughput and minimize load, such as by handling agreement in large batches.

We show how client speculation can help clients of a replicated service tolerate network and protocol latency by adding speculation to the Practical Byzantine Fault Tolerance (PBFT) protocol [11]. We demonstrate how performance improves for a counter service and an NFSv2 service on PBFT from decreased effective latency and increased concurrency in light workloads. Speculation improves the client throughput of the counter service 2–58× across two different network topologies. Speculation speeds up the run time of NFS micro-benchmarks 1.08–19× and up to 5× on a macro-benchmark when co-locating a replica with the client. When replicas are equidistant from each other, our benchmarks speed up by 1.06–6× and 2.2×, respectively. The decrease in latency that client speculation provides does have a cost: under heavy workloads, maximum throughput is decreased by 18%.

4.1 Client speculation in replicated services

4.1.1 Applicability to replicated services

As discussed in Section 1.1, speculative execution has often been used to hide the latency associated with network communication. For speculation to be applicable, communication protocols must be slow (beyond the cost of creating the speculation) and highly predictable.

Replicated services are an excellent candidate for client-based speculative execution. Clients of replicated state machine protocols that tolerate Byzantine faults must wait for multiple replicas to reply. That may mean waiting for multiple rounds of messages to be exchanged among replicas in an agreement-based protocol. If replicas are separated by geographic distances (as they should be in order to achieve failure independence), network latency introduces substantial delay between the time a client starts an operation and the time the client receives the reply that commits the operation. Thus, there is substantial time available to benefit from speculative execution, especially if one replica is located near the client.

Replicated services also provide an excellent predictor of an operation's result. Under the assumption that faults are rare, a client's request will generate identical replies from every replica, so the first reply that a client receives is an excellent predictor of the final, collective reply from the replicated state machine (which we refer to as the *consensus reply*). After receiving the first reply to any operation, a client can speculate *based on 1 reply* with high confidence. For example, when an NFS client tries to read an uncached file, it cannot predict what data will be returned, so it must wait for the first reply before it can continue with reasonable data.

The results of some remote operations can be predicted even before receiving any replies; for instance, an NFS client can predict with high likelihood of success that file system updates will succeed and that read operations will return the same (possibly stale) values in its cache [52]. For such operations, a client may speculate *based on 0 replies* since it can predict the result of a remote operation with high probability.

4.1.2 Protocol adjustments

Based on the above discussion, it becomes clear that some replicated state machine protocols will benefit more from speculative execution than others. For this reason, we propose several adjustments to protocols that increase the benefit of client-based speculation.

Generate early replies

Since the maximum latency that can be hidden by speculative execution, in the absence of 0-reply speculation, is the time between when the client receives the first reply from any replica and when the client receives enough replies to determine the consensus response, a protocol should be designed to get the first reply to the client as quickly as possible. The fastest reply is realized when the client sends its request to the closest replica, and that replica responds immediately. Thus, a protocol that supports client speculation should have one or more replicas immediately respond to a client with the replica's best guess for the final outcome of the operation, as long as that guess can accurately predict the consensus reply.

Assuming each replica stores the complete state of the service, the closest replica can always immediately perform and respond to a read-only request. However, that reply is not guaranteed to be correct in the presence of concurrent write operations. It could be wrong if the closest replica is behind in the serial order of operations and returns a stale value, or in quorum protocols where the replica state has diverged and is awaiting repair [1]. We describe optimizations in Section 4.2.2 that allow early responses from any replica in the system, along with techniques to minimize the likelihood of an incorrect speculative read response.

It is more difficult to allow any replica to immediately execute a modifying request in an agreement protocol. Backup replicas depend on the primary replica to decide a single ordering of requests. Without waiting for that ordering, a backup could guess at the order, speculatively executing requests as it receives them. However, it is unlikely that each replica will perceive the same request ordering under workloads with concurrent writers, especially with geographic distribution of replicas. Should the guessed order turn out wrong (beyond acceptable levels [36]), the replica must roll back its state and re-execute operations in the committed order, hurting throughput and likely causing its response to change.

For agreement protocols like PBFT, a more elegant solution is to have only the primary execute the request early and respond to the client. As we explain in Section 4.2.3, such predictions are correct unless the primary is faulty. This solution enables us to avoid speculation or complex state management on the replicas that would reduce throughput. Used in this way, the primary should be located near the most active clients in a system to reduce their latency.

Prioritize throughput over latency

There exist a myriad of replicated state machine protocols that offer varying tradeoffs between throughput and latency [1, 11, 16, 35, 56, 64, 82]. Given client support for speculative execution, it is usually best to choose a protocol that improves throughput over one that improves latency. The reason is that speculation can do much to hide replica latency but little to improve replica throughput.

As discussed in the previous section, speculative execution can hide the latency that occurs between the receipt of an early reply from a replica and the receipt of the reply that ends the operation. Thus, as long as a speculative protocol provides for early replies from the closest or primary replica, reducing the latency of the overall operation does not ordinarily improve user-perceived latency.

Speculation can only improve throughput in the case where replicas are occasionally idle by allowing clients to issue more operations concurrently. If the replicas are fully loaded, speculation may even decrease throughput because of the additional work caused by mispredictions or the generation of early replies. Thus, it seems prudent to choose a protocol that has higher latency but higher potential throughput, perhaps through batching, and stable performance under write contention [11,35], rather than protocols that optimize latency over throughput [1,16].

An important corollary of this observation is that client speculation allows one to choose simpler protocols. With speculation, a complex protocol that is highly optimized to reduce latency may perform approximately the same as a simpler, higher latency protocol from the viewpoint of a user. A simpler protocol has many benefits, such as allowing a simpler implementation that is quicker to develop, is less prone to bugs, and may be more secure because of a smaller trusted computing base.

Avoid speculative state on replicas

To ensure correctness, speculative execution must avoid *output commits* that externalize speculative output (e.g., by displaying it to a user) since such output can not be undone once externalized. The definition of what constitutes external output, however, can change. For instance, sending a network message to another computer would be considered an output commit if that computer did not support speculation. However, if that computer could be trusted to undo, if necessary, any changes that causally depend on the receipt of the message, then the message would not be an output commit. One can think of the latter case as enlarging the *boundary of speculation* from just a single computer to encompass both the sender and receiver.

What should be the boundary of speculation for a replicated service? At least three options are possible: allow all replicas and clients of the service to share speculative state, allow replicas to share speculative state with individual clients but not to propagate one client's speculative state to other clients, and disallow replicas from storing speculative state.

Our design uses the third option, with the smallest boundary of speculation, for several reasons. First, the complexity of the system increases as more parts participate in a speculation. The system would need to use distributed commit and rollback [21] to involve replicas and other clients in the speculation, and the interaction between such a distributed commit and the normal replicated service commit would need to be examined carefully. Second, as the boundary of speculation grows larger, the cost of a misprediction is higher; all replicas and clients that see speculative state must roll back all actions that depend on that state when a prediction is wrong. Finally, it may be difficult to precisely track dependencies as they propagate through the data structures of a replica, and any false dependencies in a replica's state may force clients to trust each other in ways not required by the data they share in the replicated service. For example, if the system takes the simple approach of tainting the entire replica state, then one client's misprediction would force the replica to roll back all later operations, causing unrelated clients to also roll back.

Use replica-resolved speculation

Even with this small boundary of speculation, we would still like to allow clients to issue new requests that depend on speculative state (which we call *speculative requests*). Speculative requests allow a client to continue submitting requests when it would otherwise be forced to block. These additional requests can be handled concurrently, increasing throughput when the replicas are not already fully saturated.

One complication here is that, to maintain correctness, if one of the prior operations on which the client is speculating fails, any dependent operations that the client issues must also abort. There is currently no mechanism for a replica to determine whether or not a client received a correct speculative response. Thus, the replica is unable to detect whether or not to execute subsequent dependent speculative requests.

To overcome this flaw, we propose *replica-resolved speculation through predicated writes*, in which replicas are given enough information to determine whether the speculations on which requests depend will commit or abort. With predicated writes, an operation that modifies state includes a list of the active speculations on which it depends, along with the predicted responses for those speculations. Replicas log each committed response they send to clients and compare each predicted response in a predicated write with the actual response sent. If all predicated responses match the saved versions, the speculative request is consistent with the replica's responses, and it can execute the new request. If the responses do not match, the replica knows that the client will abort this operation when rolling back a failed speculation, so it discards the operation. This approach assumes a protocol in which all non-faulty replicas send the same response to a request.

Note that few changes may need to be made to a protocol to handle speculative requests that modify data. An operation O that depends on a prior speculation O_s, with predicted response r, may simply be thought of as a single deterministic request to the replicated service of the predicated form: `if` $response(O_s) = r$, `then do` O. This predicate must be enforced on the replicas. However, as shown in Section 4.4, predicate checking may be performed by a shim layer between the replication protocol and the application without modifying the protocol itself.

4.2 Client speculation for PBFT

In this section, we apply our general strategy for supporting client speculative execution in replicated services to the Practical Byzantine Fault Tolerance (PBFT) protocol. We call the new protocol we develop PBFT-CS (CS denotes added support for client speculation).

4.2.1 PBFT overview

PBFT is a Byzantine fault tolerant state machine replication protocol that uses a primary replica to assign each client request a sequence number in the serial order of operations. The replicas run a three-phase agreement protocol to reach consensus on the ordering of each operation, after which they can execute the operation while ensuring consistent state at all non-faulty replicas. Optionally, the primary can choose and attach *non-deterministic data* to each request (for NFS, this contains the current time of day).

PBFT requires $3f + 1$ replicas to handle f concurrent faulty replicas, which is the theoretical minimum [7]. The protocol guarantees liveness and correctness with up to f failures, and runs a *view change* sub-protocol to move the primary to another replica in the case of a bad primary.

The communication pattern for PBFT is shown in Figure 4.1. The client normally receives a commit after five one-way message delays, although this may be shortened to

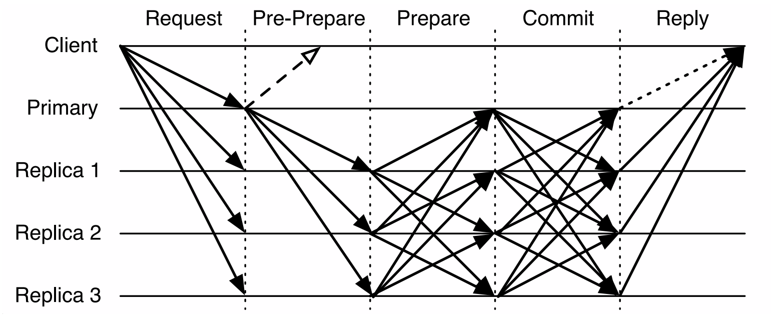


Figure 4.1: PBFT-CS Protocol Communication. The early response from the primary is shown with a dashed hollow arrow, which replaces its response from the Reply phase (dotted filled arrow) in PBFT.

four delays by overlapping the *commit* and *reply* phases using a *tentative execution* optimization [11]. To reduce the overhead of the agreement protocol, the primary may collect a number of client requests into a *batch* and run agreement once on the ordering of operations within this batch.

In our modified protocol, PBFT-CS, the primary responds immediately to client requests, as illustrated by the dashed line in Figure 4.1.

4.2.2 PBFT-CS base protocol

In both PBFT and PBFT-CS, the client sends each request to all replicas, which buffer the request for execution after agreement. Unlike the PBFT agreement protocol, the primary in PBFT-CS executes an operation immediately upon receiving a request and sends the early reply to the client as a speculative response. The primary then forms a pre-prepare message for the next batch of requests and continues execution of the agreement protocol. Other replicas are unmodified and reply to the client request once the operation has committed.

Since the primary determines the serial ordering of all requests, under normal circumstances the client will receive at least f committed responses from the replicas matching the primary's early response. This signifies that the speculation was correct because the request committed with the same value as the speculative response. If the client receives $f + 1$ matching responses that differ from the primary's response, the client rolls back the current speculation and resumes execution with the consensus response.

Predicated writes

A PBFT-CS client can issue subsequent requests immediately after predicting a response to an earlier request, rather than waiting for the earlier request to commit. To enable this without requiring replicas themselves to speculate and potentially roll back, PBFT-CS ensures that a request that modifies state does not commit if it depends on the value of any incorrect speculative responses. To meet this requirement, clients must track and propagate the dependencies between requests.

For example, consider a client that reads a value stored in a PBFT-CS database (`op1`), performs some computation on the data, then writes the result of the computation back to the database (`op2`). If the primary returns an incorrect speculative result for `op1`, the value to be written in `op2` will also be incorrect. When `op1` eventually commits with a different value, the client will fail its speculation and resume operation with the correct value. Although the client cannot undo the send of `op2`, dependency tracking prevents `op2` from writing its incorrect value to the database.

Each PBFT-CS client maintains a log of the digests d_T of each speculative response issued at logical timestamp T. When an operation commits, its corresponding digest is removed from the tail of the log. If an operation aborts, its digest is removed from the log, along with the digests of any dependent operations.

Clients append any required dependencies to each speculative request, of the form $\{c, \langle t_i, d_i \rangle, ...\}$ for client c and each digest d_i at timestamp t_i.

Replicas also store a log of digests for each client with the committed response for each operation. The replica executes a speculative request only if all digests in the request's dependency list match the entries in the replica's log. Otherwise, the replica executes a no-op in place of the operation.

It is infeasible for replicas to maintain an unbounded digest log for each client in a long-running system, so PBFT-CS truncates these logs periodically. Replicas must make a deterministic decision on when to truncate their logs to ensure that non-faulty replicas either all execute the operation or all abort it. This is achieved by truncating the logs at fixed deterministic intervals.

If a client issues a request containing a dependency that has since been discarded from the log, the replicas abort the operation, replacing it with a no-op. The client recognizes this scenario when receiving a consensus response that contains a special *retry* result. It retries execution once all its dependencies have committed. In practice an operation will not abort due to missing dependencies, provided that the log is sufficiently long to record all operations issued in the time between a replica executing an operation and a quorum of responses being received by the client.

Read-only optimization

Many state machine replication protocols provide a read-only optimization [1, 11, 16, 35] in which read requests can be handled by each replica without being run through the agreement protocol. This allows reads to complete in a single communication round, and it reduces the load on the primary.

In the standard optimization, a client issues optimized read requests directly to each replica rather than to the primary. Replicas execute and reply to these requests without taking any steps towards agreement. A client can continue after receiving $2f + 1$ matching replies. Because optimized reads are not serialized through the agreement protocol, other clients can issue conflicting, concurrent writes that prevent the client from receiving enough matching replies. When this happens, the client retransmits the request through the agreement protocol. This optimization is beneficial to workloads that contain a substantial percentage of read-only operations and exhibit few conflicting, concurrent writes. Importantly, when a backup replica is located nearer a client than the primary, that replica's reply will typically be received by the client before the primary's.

PBFT-CS cannot use this standard optimization without modification. A problem arises when a client issues a speculative request that depends on the predicted response to an optimized read request. PBFT-CS requires all non-faulty replicas to make a deterministic decision when verifying the dependencies on an operation. However, since optimized reads are *not* serialized by the agreement protocol, one non-faulty replica may see a conflicting write before responding to an optimized read, while another non-faulty replica sees the write after responding to the read. These two non-faulty replicas will thus respond to the optimized read with different values, and they will make different decisions when they verify the dependencies on a later speculative request. A non-faulty replica that sent a response that matches the first speculative response received by the client will commit the write operation, while other non-faulty replicas will not. Hence, writes may not depend on uncommitted optimized reads. This is enforced at each replica by not logging the response digest for such requests.

We address this problem by allowing a PBFT-CS client to resubmit optimized read requests through the full agreement protocol, forcing the replicas to agree on a common response. When write conflicts are low, the resubmitted read is likely to have the same reply as the initial optimized read, so a speculative prediction is likely to still be correct. After performing this procedure, we can send any dependent write requests, as they no longer depend on an optimized request.

There are three issues that must be considered for a read request to be submitted using this optimization.

- The request cannot read uncommitted state.
- The client should not follow a read with a write.
- The reply should not be completely predictable.

The first issue is required for consistency. A client cannot optimize a read request for a piece of state before all its write requests for that state are committed. Otherwise, it risks reading stale data when a sufficient number of backup replicas have not yet seen the client's previous writes. The data dependency tracking required to implement this policy is also used to propagate speculations, so no extra information needs to be maintained. Reads that do depend on uncommitted data may still be submitted through the agreement protocol as with write requests. Should a client desire a simpler policy for ensuring correctness, it can disable the read-only optimization while it has any uncommitted writes.

Second, consider a client that reads a value, performs a computation, and then writes back a new value. If the read request is initially sent optimized, issuing the write will force the read to be resubmitted. The "optimization" results in additional work. Clients that anticipate following a read by a write should decline to optimize the read.

Finally, if a client can predict the outcome of the request before receiving any replies (for instance, if it predicts that a locally-cached value has not become stale), then it should submit the request through the normal agreement protocol. Since the client does not need to wait for any replies, it is not hurt by the extra latency of waiting for agreement.

4.2.3 Handling failures

Speculation optimizes for reduced latency in the non-failure case, but it is important to ensure that correctness and liveness are maintained in the presence of faulty replicas. Failed speculations also increase the latency of a client's request, forcing it to roll back after having waited for the consensus response, and hurt throughput by forcing outstanding requests to become no-ops. It is important for our protocol to handle faults correctly in a way that still tries to preserve performance.

A speculation will fail on a client when the first reply it receives to a request does not match the consensus response. There are three cases in which this might happen:

- The most common case occurs when a write issued by another client conflicts with an optimized read. In an extreme instance, one replica's early reply could contain the stale data while all other replicas reply with current data.
- The second case occurs when there is a view change. PBFT ensures that committed requests will be ordered the same in the new view, but the client is speculating on uncommitted requests that the new replica could order differently. View changes may be the result of a bad primary, or they may be triggered by network conditions or proactive recovery [12].
- The third case occurs when the primary is faulty, and it either returns an incorrect speculative response or serializes a request differently when running the agreement protocol. We next examine this scenario further.

It is trivial for a client to detect a faulty primary: a request's early reply from the primary and the consensus reply will be in the same view and not match. If signed responses are used, the primary's bad reply can be given to other replicas as a proof of misbehavior. However, if simple message authentication codes (MACs) are used, the early reply cannot be used in this way since MACs do not provide non-repudiation.

The simplest solution to handling faults with MACs is for a client to stop speculating if the percentage of failed speculations it observes surpasses a threshold. PBFT-CS currently uses an arbitrary threshold of 1%. If a client observes that the percentage of failed speculations is greater than 1% over the past *n* early replies provided by a replica, it simply ceases to speculate on subsequent early replies from that replica. Although it will not speculate on subsequent replies, it can still track their accuracy and resume speculating on further replies if the percentage falls below a threshold. Our experimental results verify that at this threshold, PBFT-CS is still effective at reducing the average latency under light workloads.

4.2.4 Correctness

The speculative execution environment and PBFT protocol used in our system both have well-established correctness guarantees [10, 52]. We thus focus our attention on the modifications made to PBFT, to ensure that this protocol remains correct.

Our modified version of PBFT differs from the original in several key ways:

- A client may be sent a speculative response that differs from the final consensus value.
- A client may submit an operation that depends on a failed speculation.
- The primary may execute an operation before it commits.

We evaluate each modification independently.

Incorrect speculation A bad primary may send an incorrect speculative response to a client, in that it differs on the value or ordering of the final consensus value. We also consider in this class an honest primary that sends a speculative response to a client but is unable to complete agreement on this response due to a view change. In either case, the client will only see the consensus response once the operation has undergone agreement at a quorum of replicas. If the speculative response was incorrect, it is safe for the client to roll back the speculative execution and re-run using the consensus value, since PBFT ensures that all non-faulty replicas will agree on the consensus value.

Dependent operations A further complication arises when the client has issued subsequent requests that depend on the value of a speculative response. Here, the speculation protocol on the client ensures that it rolls back execution of any operations that have dependencies on the failed speculation. We must ensure that all valid replicas make an identical decision to abort each dependent operation by replacing it with a no-op.

Replicas maintain a log of the digests for each committed operation and truncate this log at deterministic intervals so that all non-faulty replicas have the same log state when processing a given operation. Predicated writes in PBFT-CS allow the client to express the speculation dependencies to the replicas. A non-faulty replica will not execute any operation that contains a dependency that does not match the corresponding digest in the log, or that does not have a matching log entry. Since the predicated write contains the same information used by the client when rolling back dependent operations, the replicas are guaranteed to abort any operation aborted by the client. If a client submits a dependency that has since been truncated from the log, it will also be aborted.

The only scenario where replicas are unable to deterministically decide whether a speculative response matches its agreed-upon value is when a speculative response was produced using the read-only optimization. Here, different replicas may have responded with different values to the read request. We explicitly avoid this case by making it an error to send a write request that depends on the reply to an optimized read request; correct clients will never issue such a request. Replicas do not store the responses to optimized reads in their log and hence always ignore any request sent by a faulty client with a dependency on an optimized read.

Speculative execution In our modified protocol, the primary executes client requests immediately upon their receipt, before the request has undergone agreement. The agreement protocol dictates that all non-faulty replicas commit operations in the order proposed by the primary, unless they execute a view change to elect a new primary. After a view change, the new primary may reorder some uncommitted operations executed by the previous primary, however, the PBFT view change protocol ensures that any committed operations persist into the new view. It is safe for the old primary to restore its state to the most recent committed operation since any incorrect speculative response will be rolled back by clients where necessary.

4.3 Discussion and future optimizations

In this section, we further explore the protocol design space for the use of client speculation with PBFT. We compare and contrast possible protocol alternatives with the PBFT-CS protocol that we have implemented.

4.3.1 Alternative failure handling strategies

We considered two alternative strategies for dealing with faulty primaries. First, we could allow clients to request a view change without providing a proof of misbehavior. This scheme would seem to significantly compromise liveness in a system containing faulty clients since they can force view changes at will. However, this is an existing problem in BFT state machine replication in the absence of signatures. A bad client in PBFT is always able to force a view change by sending a request to the primary with a bad authenticator that appears correct to the primary or by sending different requests to different replicas [10]. We could mitigate the damage a given bad client can do by having replicas make a local decision to ignore all requests from a client that 'framed' them. In this way a bad client can not initiate a view change after incriminating f primaries.

Alternatively, we could require signatures in communications between client and replicas. This is the most straight-forward solution, but entails significant CPU overhead. Compared to these two alternative designs, we chose to have PBFT-CS revert to a non-speculative protocol due to the simplicity of the design and higher performance in the absence of a faulty primary.

4.3.2 Coarse-grained dependency tracking

PBFT-CS tracks and specifies the dependencies of a speculative request at fine granularity. Thus, message size and state grow as the average number of dependencies for a given operation increases. To keep message size and state constant, we could use coarser-grained dependencies.

We could track dependencies on a per-client basis by ensuring that a replica executes a request from a client at logical timestamp T only if *all* outstanding requests from that client prior to time T have committed with the same value the client predicted.

Instead of maintaining a list of dependencies, each client would instead store a hash chained over all consensus responses and subsequent speculative responses. The client would append this hash to each operation in place of the dependency list. The client would also keep another hash chained only over consensus responses, which it would use to restore its dependency state after rolling back a failed speculation.

Each replica would maintain a hash chained over responses sent to the client and would execute an operation if the hash chain in the request matches its record of responses. Otherwise, it would execute a no-op.

We chose not to use this optimization in PBFT-CS since the use of chained hashes creates dependencies between all operations issued by a client even when no causal dependencies exist. This increases the cost of a failed speculation since the failure of one speculative request causes all subsequent in-progress speculative operations to abort. Coarse-grained dependency tracking also limits the opportunities for running speculative read operations while there are active speculative writes. Since speculative read responses are not serialized with respect to write operations, it is likely that the client will insert the read response in the wrong point in the hash chain, causing subsequent operations to abort.

4.3.3 Reads in the past

A read-only request need not circumvent the agreement protocol completely, as described in section 4.2.2. A client can instead take a hybrid approach for non-modifying requests: it can submit the request for full agreement and at the same time have the nearest replica immediately execute the request.

If the primary happens to be the nearest to the client, this is not a change from the normal protocol. When another replica is closer, the client can get a lower-latency first

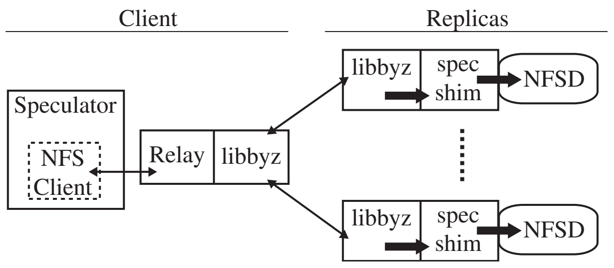


Figure 4.2: Speculative fault-tolerant NFS architecture.

reply, plus having agreement eliminates the second consideration for optimized reads (in Section 4.2.2), that a client should not follow a read with a write.

However, this new optimization presents a problem when there are concurrent writes by multiple clients. A non-primary replica will execute an optimized request, and a client will speculate on its reply, in a sequential order that is likely different from the request's actual order in the agreement protocol. In essence, the read has been executed *in the past*, at a logical time when the replicas have not yet processed all operations that are undergoing agreement but when they still share a consistent state.

We could extend the PBFT-CS read-only optimization to also allow reads in the past. Under a typical configuration, there is only one round of agreement executing at any one time, with incoming requests buffered at the primary to run in the next batch of agreement. If we were to ensure that all buffered reads are reordered, when possible, to be serialized at the start of this next batch, it would be highly likely that no write will come between a read being received by a replica and the read being serialized after agreement.

Note that the primary may assign any order to requests within a batch as long as no operation is placed before one on which it depends. Recall that a PBFT-CS client will only optimize a read if the read has no outstanding write dependencies. Hence, the primary is free to move all speculative reads to the start of the batch. The primary executes these requests on a snapshot of the state taken before the batch began.

4.4 Implementation

We modified Castro and Liskov's PBFT library, *libbyz* [11], to implement the PBFT-CS protocol described in Section 4.2. We also modified BFS [11], a Byzantine-fault-tolerant replicated file service based on NFSv2, to support client speculation. The overall system can be divided into three parts as shown in Figure 4.2: the NFS client, a protocol relay, and the fault-tolerant service.

4.4.1 NFS client operation

Our client system uses the NFSv2 client module of the Speculator kernel (see Section 2.3) to provide process-level support for speculative execution. To execute a remote NFS operation, Speculator first attaches a list of the process's dependencies to the message, then sends it to a relay process on the same machine. The relay interprets this list and attaches the correct predicates when sending the PBFT-CS request.

The relay brokers communication between the client and replicas. It appears to be a standard NFS server to the client, so the client need not deal with the PBFT-CS protocol. When the relay receives the first reply to a 1-reply speculation, the reply is logged and passed to the waiting NFS client. The NFS client recognizes speculative data, creates a new speculation, and waits for a confirmation message from the relay. Once the consensus reply is known, the relay sends either a `commit` message or a `rollback`{`reply`} message containing the correct response.

Our implementation speculates based on 0 replies for `GETATTR`, `SETATTR`, `WRITE`, `CREATE`, and `REMOVE` calls. It can speculate on 1 reply for `GETATTR`, `LOOKUP`, and `READ` calls. This list includes the most common NFS operations: we observed that at least 95% of all calls in all our benchmarks were handled speculatively. Note that we speculate on both 0 replies and 1 reply for `GETATTR` calls. The kernel can speculate as soon as it has attributes for a file. When the attributes are cached, 0 replies are needed, otherwise, the kernel waits for 1 reply before continuing.

4.4.2 PBFT-CS client operation

Speculation hides latency by allowing a single client to pipeline many requests; however, our PBFT implementation only allows for each PBFT-CS client to have a single outstanding request at any time. We work around this limitation by grouping up to 100 logical clients into a single client process.

NFS with 0-reply speculation requires its requests to be executed in the order they were issued. A PBFT-CS client process can tag each request with a sequence number so that the primary replica will only process requests from that client process's logical clients in the correct order. Of course, two different clients' requests can still be interleaved in any order by the primary.

To support this additional concurrency, we designed the client to use an event-driven API. User programs pass requests to libbyz and later receive two callbacks: one delivers the first reply and another delivers the consensus reply. The user program is responsible for monitoring libbyz's communication channels and timers.

4.4.3 Server operation

On the replicas, libbyz implements an event-based server that performs upcalls into the service when needed: to request non-deterministic data, to execute requests, and to construct error replies. The library handles all communication and state management, including checkpointing and recovery.

A shim layer is used to manage dependencies on replicas. When writes need to be quashed due to failed speculative dependencies, the shim layer issues a no-op to the service instead. Thus, the underlying service is not exposed to details of the PBFT-CS protocol.

The primary will batch together all requests it receives while it is still agreeing on earlier requests. Batching is a general optimization that reduces the number of protocol instances that must be run, decreasing the number of communications and authentication operations [11, 35, 36, 82]. This implementation imposes a maximum batch size of 64 requests, a limit our benchmarks do run up against.

Overhead Source	Slowdown
Early replies	8.2%
Larger request	4.1%
Complex client	2.8%
Predicate checking	1.8%

Table 4.1: Sources of overhead affecting throughput for PBFT-CS relative to PBFT.

4.5 Evaluation

In this section, we quantify the performance of our PBFT-CS implementation using a simple shared counter micro-benchmark we implemented and several NFS micro- and macro-benchmarks.

We compare PBFT-CS to two other modern Byzantine fault-tolerant agreement protocols: PBFT [11] and Zyzzyva [35]. PBFT is the base protocol we extend make use of client speculation. Its overall structure is illustrated in Figure 4.1. We use the tentative reply optimization, so each request must go through 4 communication phases before the client acquires a reply that it can act on. PBFT uses an adaptive batching protocol, allowing up to 64 requests to be handled in one agreement instance.

Zyzzyva is a recent agreement protocol that is heavily optimized for failure-free operation. When all replicas are non-faulty (as in our experiments), it takes only 3 phases for a client to possess a consensus reply. We run Kotla et al.'s implementation of Zyzzva, which uses a fixed batch size. We simulate an adaptive batching strategy by manually tuning the batch size as needed for best performance.

By comparison, a PBFT-CS client can continue executing speculatively after only 2 communication phases. We expect this to significantly reduce the effective latency of our clients. Note that requests still require 4 phases to *commit*, but we can handle those requests concurrently rather than sequentially. If we limit the number of in-flight requests to some number *n*, we call the protocol "PBFT-CS (*n*)."

4.5.1 Experimental setup

Each replica machine uses a single Intel Xeon 2.8 GHz processor with 512 MB RAM (sufficient for our applications). We always evaluate using four replicas without failures (unless noted). In our NFS comparisons, we use a single client that is identical in hardware to the replicas. Our counter service runs on an additional five client machines using Intel Pentium 4s or Xeons with clock speeds of 3.06–3.20 GHz and 1 GB RAM. All systems use a generic Red Hat Linux 2.4.21 kernel.

Our machines use gigabit Ethernet to communicate directly with a single switch. Experiments using the shared counter service were performed on a Cisco Catalyst 2970 gigabit switch; NFS used an Intel Express ES101TX 10/100 switch.

Our target usage scenario is a system that consists of several sites joined by moderate latency connections (but slower than LAN speeds). Each site has a high-speed LAN hosting one replica and several clients, and clients may also be located off-site from any replica. For comparison with other agreement protocols, we also consider using PBFT-CS in a LAN setting where all replicas and clients are on the same local segment.

Based on the above scenarios, we emulate a simplified test network using NISTNet [9] that inserts an equal amount of one-way latency between each site. We let this inserted *delay* be either 2.5 ms or 15 ms.

We also measure performance at clients located in different areas in our scenario. In the *primary-local topology*, the client is at the same site as the current primary replica. The *primary-remote topology* considers a client at different site hosting a backup replica. A client not present at any site is shown in the *uniform topology*, and we let the client have the same one-way latency to all replicas as between sites.

When comparing against a service with no replication in a given topology, we always assume that a client at a site can access its server using only the LAN. A client not at a site is still subject to added delay.

4.5.2 Counter throughput

We first examine the throughput of PBFT-CS using the counter service. Similar to Castro and Liskov's standard 0/0 benchmark [11], the counter's request and reply size are minimal. This service exposes only one operation: increment the counter and return its new value. Each reply contains a token that the client must present on its next request.

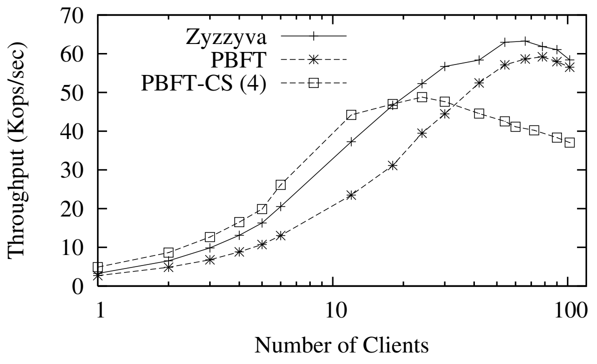


Figure 4.3: Server throughput in a LAN, measured on the shared counter service. PBFT-CS (4) is limited to four concurrent requests.

This does add a small amount of processing time to each request, but it ensures that client requests must be submitted sequentially.

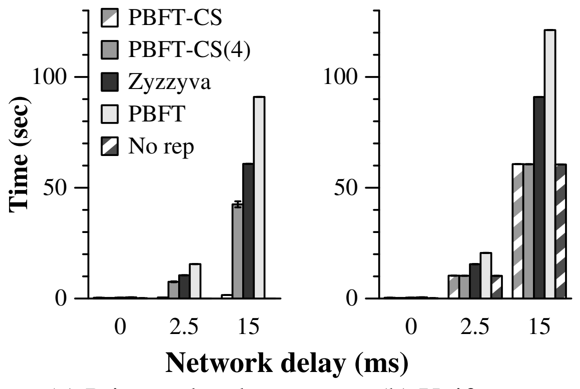
Our client is a simple loop that issues a fixed number of counter updates and records the total time spent. No state is externalized by the client, so we allow the client process to implement its own lightweight checkpoint mechanism. Checkpoint operations take negligible time, so our results focus on the characteristics of the protocol itself rather than our checkpoint mechanism.

We measure throughput by increasing the number of client processes per machine (up to 17 processes) until the server appears saturated. Graphs show the mean of at least 6 runs, and visible differences are statistically significant.

Figure 4.3 shows the measured throughput in a LAN configuration. We found that in this topology, a single PBFT-CS client gains no benefit from having more than 4 concurrent requests, and we enforce that limit on all clients. When we have 12 or fewer concurrent clients, PBFT-CS has 1.19–1.49× higher throughput than Zyzzyva and 1.79–2× higher throughput than PBFT.

In lightly loaded systems, the servers are not being fully utilized, and speculating clients can take advantage of the spare resources to decrease their own effective latency. As the server becomes more heavily loaded, those resources are no long free to use. As a result, PBFT-CS reaches its peak throughput before other protocols.

There is a trade-off of throughput for latency: PBFT-CS shows a peak throughput that is 17.6% lower than PBFT. We found four fundamental sources of overhead, summarized



(a) Primary-local (b) Uniform

Figure 4.4: Time taken to run 2000 updates using the shared counter service. The primary-local topology (a) shows a client located at the same site as the primary. The uniform topology (b) shows a remote client equidistant from all sites. 0 ms (LAN) times for both graphs are (in bar order): 0.36 s, 0.27 s, 0.41 s, 0.54 s, and 0.16 s.

in Table 4.1. First, the client implementation for PBFT-CS uses an event-driven system to handle several logical clients, needed to support concurrent requests. This design does lead to a slower client than the one in PBFT, which can get by with a simpler blocking design. Second, we found that having the primary send early replies increases its time spent blocking while transmitting. Third, each predicate added to a request makes the request packet larger, and fourth, those predicates take additional work to verify on each replica.

4.5.3 Counter latency

We next examine how latency affects client performance under a light workload when the client is located at different sites. Figure 4.4 shows the time taken for a single counter client to issue 2000 requests in different topologies. In the LAN topology where no delay is added, a PBFT-CS client is able to complete the benchmark in 33% less time than PBFT, reflecting average run times of 357 ms and 538 ms respectively. When we increase the latency between sites, run time becomes dominated by number of communication phases. With a uniform topology (Figure 4.4b), PBFT-CS takes 50% less time than PBFT and 33% less time than Zyzzyva, and its runtime is only 1% slower than the unreplicated service. This matches our intuitive understanding of the protocol behavior described at the start of this section.

For PBFT-CS, the critical path is a round-trip communication with the primary replica. Moving to a primary-remote topology (bringing one backup replica closer) does not affect this critical path, and our measurements show no significant difference between primary-remote and uniform topologies.

Figure 4.4a presents results when using a primary-local topology. As latency increases and backup replicas move further from the client, performance does not degrade significantly, since the latency to the primary is fixed. At 15 ms latency, a client using PBFT takes 58× longer than with PBFT-CS. The combination of client speculation and a co-located primary achieves much of the performance benefit of a closely located non-replicated server, while providing all the guarantees of a geographically distributed replicated service that tolerates Byzantine faults.

These significant gains are directly attributable to the increased concurrency possible in the primary-local topology. When we limit PBFT-CS to only 4 outstanding requests, the client must then wait on requests to commit, reintroducing a dependence on communication delay. In topologies where the client does not have privileged access to the primary, as in the uniform topology, limiting concurrency has little effect.

4.5.4 NFS

We next examine PBFT-CS applied to an NFS server. Considering that the NFSv2 protocol is not explicitly designed for high-latency environments, we compare against the variation of NFS that uses 0-reply speculation. All benchmarks begin with a freshly-mounted file system and an empty cache.

Unlike the counter service, this application has overhead associated with creating, committing, and rolling back to a checkpoint. Processes may have computation to perform between requests, and they may need to block before an output commit.

For comparison with non-speculative systems, we measure the performance of NFS under PBFT. Using our speculative NFS protocol, we measure PBFT using only 0-reply speculation (*PBFT + 0-spec*) and PBFT-CS. The difference between these two measure-

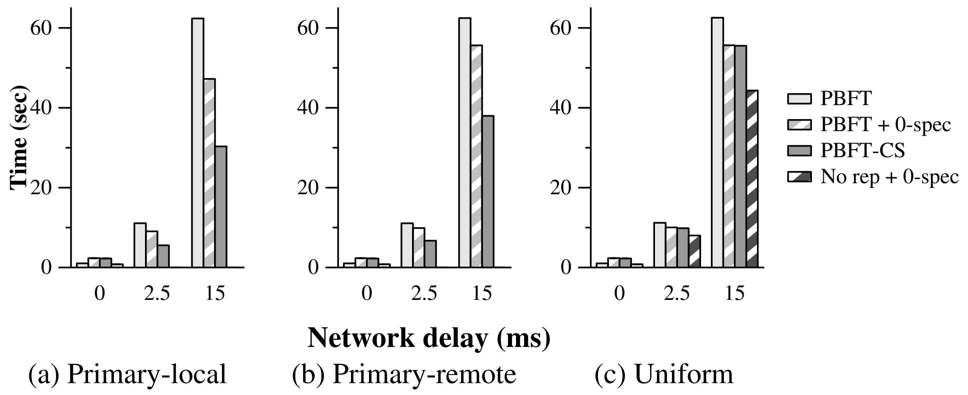


Figure 4.5: Read-only NFS micro-benchmark performance across different network topologies. The last three data sets use 0-reply speculation. At 0 ms, all three topologies are equivalent, so the same data is used for each graph. The *no rep* data show a lower bound for run time. There is only one *no rep* data set for primary-local and primary remote topologies, because the location of the server does not change with increasing latency. For these two graphs, the 0 ms bar applies to all latencies but is not repeated.

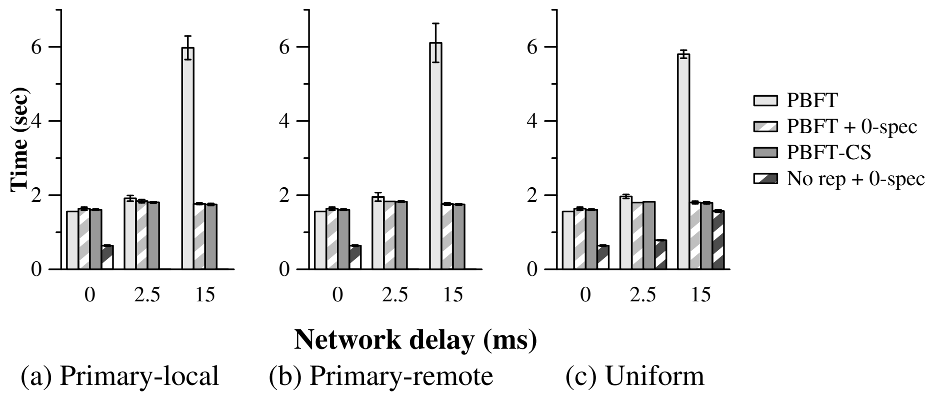


Figure 4.6: Write-only NFS micro-benchmark.

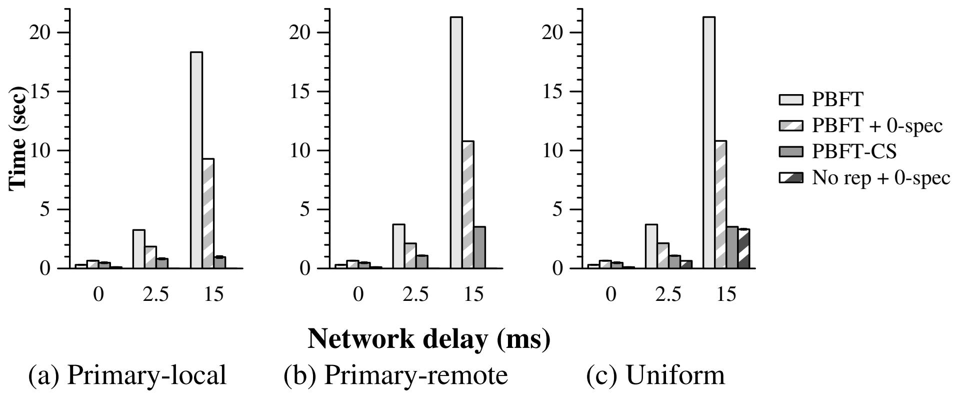


Figure 4.7: Read/Write NFS micro-benchmark.

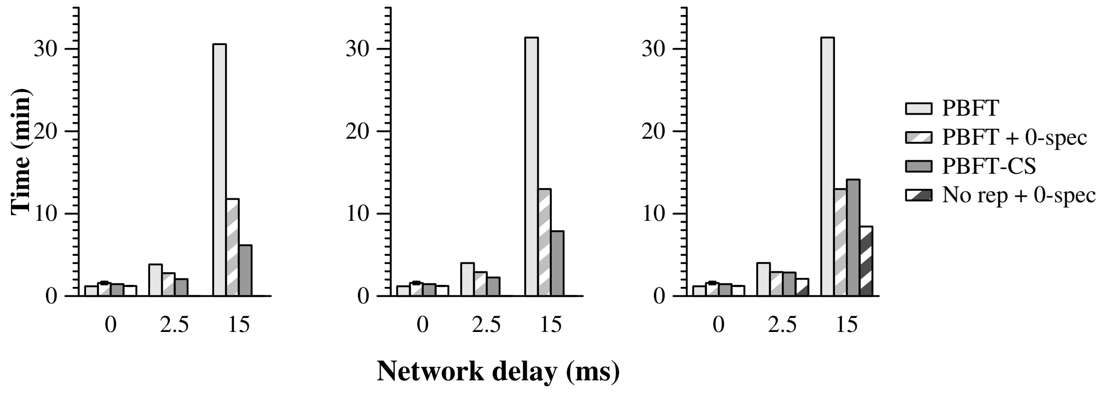


Figure 4.8: The Apache build NFS benchmark measures how long it takes to compile and link Apache 2.0.48.

ments show the benefit of 1-reply speculation. As a lower bound, we also measure the performance of a non-replicated NFS server that uses 0-reply speculation (*No rep + 0-spec*).

We use a vanilla kernel for evaluating non-speculative PBFT with a slight modification that increases the number of concurrent RPC requests allowed. Other benchmarks use the Speculator kernel.

In the *no replication* configuration, the NFS client uses a thin UDP relay on the local machine that stands in for the BFT relay.

Our modifications to the NFS client, the relay, and the replicated service have introduced additional overhead that is not present in the original PBFT. This inefficiency is particularly apparent in our 0 ms topologies, where PBFT-CS shows a 1.03–2.18× slowdown relative to PBFT across all our benchmarks. However, in all cases at higher latencies, client speculation results in a clear improvement, and we primarily address these configurations in the following sections.

At the time of publication, we had not yet ported our NFS server to use the Zyzzyva protocol, so we regretfully are unable to provide a direct comparison for these benchmarks.

All graphs show the mean of at least five measurements. Error bars are shown when the 95% confidence interval is above 1% of the mean value.

4.5.5 NFS: Read-only micro-benchmark

We first ran a read-only micro-benchmark that `greps` for a common string within the Linux headers. The total size of the searched files is about 9.1 MB. Most requests in this benchmark are read-only and are optimized to circumvent agreement.

Figure 4.5 shows that PBFT takes 2.06× longer to complete than PBFT-CS at 15 ms. 0-reply speculation lets the client avoid blocking when revalidating a file after opening it. With PBFT-CS, we can additionally read from a file without delay: a nearby replica supplies all the speculative data. Without a nearby replica (in uniform topology), 1-reply speculation is not beneficial since optimized reads complete at about the same time the client gets its first reply.

4.5.6 NFS: Write-only micro-benchmark

We next ran a write-only micro-benchmark that writes 3.9 MB into an NFS file (Figure 4.6). All writes are issued asynchronously by the file system, and the client only blocks when the file is closed. In this case, speculation is not needed to increase the parallelism of the system.

There are a very small number of read requests in this benchmark, issued when first opening a file, so there is no practical opportunity to use 1-reply speculation. Speculation at 2.5 ms reduces the benchmark run time by only 6–7%. We found that within each latency (irrespective of topology), there is no statistical difference between PBFT+0-spec and PBFT-CS.

4.5.7 NFS: Read/write micro-benchmark

We next ran a read/write micro-benchmark that creates 100 4 KB files in a directory. For each file, the client creates and writes to a file; this includes read-only operations to read the directory entries. PBFT-CS never blocks on any of these operations.

In the primary-local topology, PBFT takes up to 19× longer to complete than PBFT-CS (Figure 4.7). Furthermore, PBFT-CS shows a resilience to changes in latency as it increases from 0-15 ms: PBFT-CS execution time doubles while PBFT takes 59× longer.

On the primary-remote and uniform topologies, operations take longer to complete, but client speculation still speeds up run time by 6.03×.

4.5.8 NFS: Apache build macro-benchmark

Finally, we ran a benchmark that compiles and links Apache 2.0.48. This emulates the standard Andrew-style benchmark that has been widely used in the PBFT literature. This is intended to model a realistic and common workload, where speculation allows significant computation to be overlapped with I/O.

Within the primary-local topology, PBFT takes up to 5.0× longer to complete than PBFT-CS (Figure 4.8). In the uniform topology, PBFT takes up to 2.2× longer than PBFT-CS. Since files are often reused many times during the build process, there is less opportunity to benefit from 1-reply speculation. However, the relative difference in performance degradation as latency increases is still significant. With a co-located primary, PBFT-CS becomes 4.3× slower as delay increases to 15 ms, while PBFT slows down by a factor of 25.

4.5.9 Cost of failure / faulty primary

To measure the cost of speculation failures, we modified our PBFT-CS relay to inject faulty digests into early replies, simulating a primary that returns corrupted replies at a rate of 1%. Any speculation based on a corrupted reply will eventually be rolled back, and any dependent requests will be turned into no-ops on good replicas.

The results of this experiment are presented in Figure 4.9. We used the Apache build benchmark in the primary-local topology. The injected faults were responsible for slowdowns in PBFT-CS of 3%, 9%, and 29% at 0 ms, 2.5 ms, and 15 ms delay respectively.

These slowdowns are not identical because a client may have a greater number of requests in the pipeline for completion at a 15 ms delay than at a 0 ms delay. When one request fails, nearly all outstanding requests also fail. We observed that 1% of our speculations failed directly, and an additional 1%, 4%, and 5% of speculations (at 0 ms, 2.5 ms, and 15 ms respectively) failed due to their dependencies. These extra requests added un-

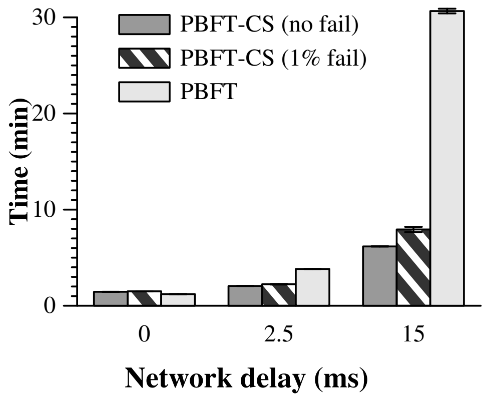


Figure 4.9: Apache build benchmark with 1% failure rate. For the Apache build benchmark in the primary-local topology, PBFT-CS is at worst 29% slower when 1% of its speculations fail.

necessary load to the replicas. By executing more requests in advance, clients must roll back a larger amount of state.

As discussed in section 4.2.3, once a client detects that 1% of requests are failing, it can stop trusting the primary to provide good first replies and disable its own speculation. If replies are signed, each primary can cause only a single failed speculation, and the resulting view change will dominate recovery time. For reference, over 100 failed speculations in this benchmark result from a 1% failure rate.

4.6 Related work

This work contributes the first detailed design for applying client speculative execution to replicated state machine protocols. It also provides the first design and implementation that uses client speculation to hide latency in PBFT [11].

Speculator [52] was originally used to hide latency in distributed file systems, and thus our work shares many of Speculator's original goals. Speculator's distributed file system application assumes the existence of a central file server that always knows ground truth. No such entity exists in a replicated state machine. For instance, non-faulty replicas may disagree about the ordering of read-only requests as discussed in Section 4.2.2. Prior to this work, Speculator was only used to speculate on zero replies. The possibility of also speculating on a single reply opens up several potential protocol optimizations that we have explored, including the possibility of generating early replies and optimizing agreement protocols for throughput.

There has also been extensive prior work in the development of replicated state machines, both in the fail-stop [38, 56, 70] and Byzantine [1, 11, 16, 34, 35, 64, 82] failure models. While Byzantine fault tolerance in particular has been an area of active research, it has seen relatively limited deployment due to its perceived complexity and performance limitations.

Our client-side speculation techniques apply equally well to reducing latency in both fail-stop and Byzantine fault tolerance protocols. However, they are particularly useful for protocols that tolerate Byzantine faults due to the higher latencies of such protocols.

PBFT [11] provides a canonical example of a Byzantine fault-tolerant replicated state machine, using multiple phases of replica-to-replica agreement to order each operation. Several systems since PBFT have aimed to reduce the latency in ordering client operations, typically by optimizing for the no-failure case [35] or for workloads with few concurrent writes [1, 16].

Byzantine quorum state machine replication protocols such as Q/U [1] build upon earlier work in Byzantine quorum agreement [5, 6, 19, 48], and provide lower latency in the optimal case. Q/U is able to respond to write requests in a single phase, provided that there are no write operations by other clients that modify the service state; inconsistent state caused by other clients requires a costly repair protocol. HQ [16] aimed to reduce the cost of repair, and reduces the number of replicas required in a Byzantine Quorum system from $5f + 1$ to $3f + 1$, but it introduces an additional phase to the optimized protocol.

Agreement protocols that use a primary replica are able to batch multiple requests into a single agreement operation, greatly reducing the overhead of the protocol and increasing throughput. While our protocol applies to both quorum and agreement protocols, the higher throughput offered by batched agreement, along with resilience during concurrent write workloads, makes them a better match for our techniques.

Our work on client speculation complements the server-side use of speculation in the Zyzzyva protocol [35]. In Zyzzyva, *replicas* execute operations speculatively based on an ordering provided by the primary, while in our system *clients* speculate based on an early response from the primary (or on 0 replies), with replicas executing only committed operations. These two approaches are complementary. Client speculation allows a client to issue a subsequent operation after only a single phase of communication with the primary, which is especially helpful for geographically dispersed deployments where some replicas are far from the client. Server speculation speeds up how fast replicas can supply a consensus response to the client, which would allow clients in our system to commit speculations faster. While we have evaluated client speculation on the PBFT protocol, it would apply equally well to Zyzzyva, where the client can receive early speculative *and* consensus responses, in the absence of failures.

4.7 Chapter Conclusions

Replicated state machines are an important and widely-studied methodology for tolerating a wide range of faults. Unfortunately, while replicas should be distributed geographically for maximum fault tolerance, current replicated state machine protocols tend to magnify the effects of the long network latencies associated with geographic distribution. In this work, we have shown how to use speculative execution at clients of a replicated service to reduce the impact of network and protocol latency. We outlined a general approach to using client speculation with replicated services, then implemented a detailed case study that applies our approach to a standard fault tolerant protocol (PBFT).

Although we studied PBFT in depth, the techniques discussed in this work should be applicable to a wider range of protocols and services. Client speculation is directly applicable to other agreement-based replication protocols [35], and it may be applicable to protocols that use more complex replication schemes, such as erasure encoding [27], although clients of such protocols may require more than one reply to predict the final response with high probability.

CHAPTER 5

Parallelizing Race Detection

The previous chapters have shown how the operating system can work with different layers of the system to improve the parallelization of applications. In this chapter, we examine how we can parallelize an algorithm for dynamic program analysis by letting it make use of speculative execution. Specifically, we develop a parallel algorithm for performing dynamic data race detection in a target program.

A prevalent model for writing parallel programs is shared-memory multiprocessing. In this model, several threads of execution share a single address space, allowing each thread to access the same memory locations concurrently. To coordinate access to shared variables, threads use *synchronization operations* (e.g., locks and condition variables) to ensure that only one thread can modify a variable at any time.

A *data race* occurs when two threads access the same memory location without proper synchronization (and at least one access is a write). Some data races (termed *benign*) may be intentionally introduced to avoid the overhead of synchronization. Otherwise, races are considered programming errors. Because there is no explicit synchronization, the outcome of a data race depends on the relative execution speed between threads. As a result, data races can be difficult to reproduce and debug with standard cyclic debugging techniques.

Data race detectors are systems designed to detect data races in existing programs. Race detectors can be implemented with the assistance of special hardware components, some of which can operate with overheads of as little as 22% [51]. However, the hardware needed to support these systems is not generally available on commodity processors. Such systems will not be discussed further.

Race detectors can also vary on when they detect the race. *Port-mortem* detectors can discover a data race at some point after the program terminates. The information generated by a post-mortem detector can be useful to point out a race to a developer, but it is too late to affect the execution of the program. An *on-the-fly* detector can discover a data race during a program execution the instant it occurs.

Some race detectors trade off precision for performance. *Imprecise* detectors group many memory locations together, typically an entire object, and treat the group as one variable. These detectors have high false-positive rates because some synchronized accesses to logically-distinct variables may be considered races. *Precise* detectors look for races at the same granularity that the underlying memory system provides (typically byte or word).

Current software-only on-the-fly race detection is slow. Production-quality race detectors for arbitrary binaries (e.g., Intel ThreadChecker [67]) slow down a target program's execution time by at least 100×. Even among research systems, the state-of-the-art precise race detector FastTrack [23] (not to be confused with the Fast Track [32] speculation environment) imposes an average of 8.5× slowdown on the execution time of Java programs.

Developers are interested in different performance characteristics for race detectors, depending on the usage scenario. Detectors can be used as part of a large automated testing system, where running code against a race detector is one step in the validation of a build. In this environment, developers are mostly concerned with having a high *throughput* (i.e., maximizing the average number of tests completed per second) and only need to be informed of a detected data race eventually.

We want to consider the scenario where a developer suspects there may be an error in his application, so he invokes a race detector as a debugging tool to help him locate a race. To maximize the productivity of this developer, the race detector should have a minimal *latency* (i.e., the time taken to run one test) and detect the race as soon as it occurs so that less of the developer's time is spent waiting for an execution to complete. Once the data race has been detected, the developer then needs access to precise information about where this race occurred.

This chapter presents a new race detector that can parallelize the work of a race detector to significantly decrease its latency, and in some benchmarks, to increase its throughput. We rely on uniparallelism [79], a kind of execution where parallelism is provided by breaking a program's execution into timeslices, which we call epochs, and running those epochs concurrently. For use in this architecture, we develop the new Parallel FastTrack race detection algorithm based on the FastTrack algorithm (which we henceforth refer to as "Sequential FastTrack") that allows each epoch to be analyzed concurrently for races. Races between two accesses that occur within one epoch can be detected as the epoch executes. As epochs finish, their accesses are checked against all prior committed epochs to detect races across epochs.

We evaluate the effectiveness of Parallel FastTrack by measuring the speedup of our detector as we increase the number of CPU cores available for race detection. We find that Parallel FastTrack can effectively use additional cores to decrease the execution time of an analyzed program. On our benchmark suite of five parallel applications, we found that Parallel FastTrack speeds up the application execution times by 2.1×, 2.8×, and 3.3× on average when by using 2, 3, and 4 times the number of worked threads used in the application.

For four out of six of our benchmarks, executing the race detection analysis using uniparallelism improves the latency of the detector even when given the same number of CPU cores as the sequential analysis. Uniparallelism allows us elide locks in the instrumentation code, significantly reducing the total amount of work and speeding up an execution by an average of about 3×.

The rest of this chapter is organized as follows. Section 5.1 discusses the background material we build upon: uniparallelism and the Sequential FastTrack algorithm. Section 5.2 presents the overall architecture of our parallel race detection system.

We then in Section 5.3 describe the Parallel FastTrack algorithm and argue its equivalence to Sequential FastTrack. Section 5.4 discusses details of our implementations of Sequential FastTrack and Parallel FastTrack. We evaluate the performance of our detector in Section 5.5.

5.1 Background

In this section we present an overview of race detection using the happens-before relation, followed by our formalism of the FastTrack algorithm. We then present an overview of uniparallelism.

5.1.1 Happens-Before Race Detection

A data race occurs when two threads access the same memory location without synchronization and at least one of the accesses is a write. There are two fundamental approaches to race detection in current literature. One approach tracks which locks are held by a thread (i.e., its "lockset") as each memory location is accessed. When two threads access the same location without holding a lock in common, a potential race is flagged. While we believe that this approach is amenable to parallelization, we leave this claim to be validated in future work.

This work focuses on a second approach, which finds accesses to the same memory location that happen concurrently according to the happens-before partial ordering of program actions. Considering only memory accesses and synchronization operations (specifically, mutex lock/unlock and thread fork/join) as program events, the *happens-before relation* is the least restrictive partial ordering of events that ensures the following conditions.

1. For two events in the same thread, one must occur before the other.

2. Releasing a lock happens before the lock's next acquisition.

3. A thread calls fork() before the new thread starts executing, and a thread exits before a join() on it returns.

These rules can be extended to cover other synchronization operations (like barriers, condition variables, and atomic accesses). A happens-before race detector looks for a pair of accesses to the same variable that are unordered and at least one access is a write.

Vector clocks [22, 49] are a mechanism that is commonly used to precisely track the happens-before relation. Many systems [20, 23, 30, 59, 83] have applied vector clocks to

Partial order	$A \sqsubseteq B \equiv \forall n\, A[n] \leq B[n]$
Merge	$(A \sqcup B)[n] = \max(A[n], B[n])$
Summary	$S_i(A)[n] = \{\text{if } i = n : A[n], \text{ else } 0\}$
Increment ith elem.	$\text{inc}_i(A)[n] = \{\text{if } i = n : A[n] + 1, \text{ else } A[n]\}$
Minimum	$\perp = \langle 0, \ldots \rangle$

Table 5.1: Vector Clocks: relations, operators, functions, and constants.

race detection in the following simplified manner. Each thread holds a vector clock that tracks the thread's local view of logical time. Every program variable has associated clocks that store the logical time at which it was last read and written. When a variable is accessed, its last-accessed timestamps are compared against the current thread's clock. If the previous accesses are unordered with the current access and at least one of those accesses is a write, then a race has been detected.

5.1.2 Sequential FastTrack

Our work is based on the version of vector clocks used by FastTrack [23]. We chose to focus on FastTrack as a representative algorithm for happens-before data race detectors that use vector clocks. Other algorithms based on vector clocks follow a similar pattern of execution and should be adaptable to our parallel architecture in a similar manner. To distinguish the original FastTrack algorithm from our modification, we refer the original version as "Sequential FastTrack."

At a high level, the Sequential FastTrack algorithm is similar to the standard vector clock algorithm previously discussed. FastTrack is notable in its use of a vector clock summary its authors call an epoch. (This is an unfortunate overlap of terminology. We will refer to this type of vector clock as a "summary" and reserve the word "epoch" for discussing uniparallelism.) This summary stores only a threads identifier and local clock value at the time of access, and the summary can be compared and updated in constant time and space. FastTrack prefers to summarize vector clocks whenever possible.

We now present our formalization of Sequential FastTrack. We begin by formalizing vector clocks and continue to discuss the details of the detection algorithm.

Event at Thread i	**Check**	**Update**
Acquire lock L		$T_i := T_i \sqcup L_C$
Release lock L		$L_C := T_i$; $T_i := \text{inc}_i(T_i)$
Fork thread j		$T_j := T_j \sqcup T_i$; $T_i := \text{inc}_i(T_i)$
Join on thread j		$T_i := T_i \sqcup T_j$; $T_j := \text{inc}_j(T_j)$
Read X	$X_W \sqsubseteq T_i$	$X_R := \begin{cases} S_i(T_i) & \text{if } X_R \text{ is a summary } \wedge X_R \sqsubseteq T_i \\ X_R \sqcup S_i(T_i) & \text{otherwise} \end{cases}$
Write X	$X_W \sqsubseteq T_i \wedge X_R \sqsubseteq T_i$	$X_W := S_i(T_i)$; $X_R := \bot$(if X_R is a full VC)

Table 5.2: The Sequential FastTrack algorithm. Operations are split between handling events for synchronization (top) and variable access (bottom). The algorithm makes different update choices depending on the representation of the last-read clock X_R. It can be a full vector clock or a summary.

A vector clock (VC) is a vector of integers of length N (e.g. $VC[n] : n < N$), where N is the number of threads in the process. A partial order is defined over VCs ($\sqsubseteq$) by comparing each index pairwise. Two VCs can be joined ($\sqcup$) by taking the maximum element at each index. For convenience, we also define functions to increment the i^{th} index and to summarize a VC by dropping all but one element. We also name the minimal element ($\bot$). These operations are described in Table 5.1.

Race detectors are often formalized as operating on program execution traces that record a sequential order of program events. For FastTrack, these events are variable accesses and synchronization operations. Java programs treat a “variable” as meaning a field of an object or the entire object. When analyzing an arbitrary binary programs, a single “variable” should be each addressable unit of memory on the architecture. To perform on-the-fly race detection, events are handled at the time they are generated.

FastTrack associates state with each program thread, lock, and variable. Each thread has a unique identifier i and a vector clock T_i. Each lock L has an associated vector clock L_C, and each variable X has two clocks: a last-read clock X_R and last-written clock X_W.

Initially, all variable and lock VCs are zero ($L_C, X_R, X_W = \bot$) and threads clocks are initialized to $T_i = \text{inc}_i(\bot)$. As synchronization events are encountered, the analysis state changes according to the rules given in Table 5.2. When a synchronization operation is handled, the corresponding update is applied to establish the proper happens-before relationship between threads. When a variable is accessed, the access is first checked to see if it races with a prior access. If the predicate given in Table 5.2 evaluates to True, then the access event does not race with a previous event, and the variable's state is updated appropriately. If the predicate is False, then the access is part of an apparent race on the variable.

5.1.3 Uniparallelism

We execute programs in a modified style called *uniparallelism* [79]. In this execution style, a program execution is divided into time slices, which we refer to as epochs. A single epoch is our fundamental unit of work. Each epoch is executed twice, once in a thread-parallel execution and again in a paired epoch-parallel execution.

A *thread-parallel* execution is similar to a normal execution of the program. Epochs are delineated by periodically interrupting all program threads and drawing a consistent boundary. As threads are spawned, the threads can be scheduled on other processing cores to provide parallelism. In this execution, each of the program's epoch are run sequentially in the program's logical order.

In an *epoch-parallel* execution, each epoch can run concurrently with other epochs. Epochs are isolated in their own address space, and the program's semantics are equivalent to a sequential execution: e.g. writes to local memory or files in one epoch are visible to epochs that logically occur later. All threads in a single epoch are constrained to execute on a single CPU core and are preempted only at well-defined points. In this execution, an application relies exclusively on the parallelization of epochs, rather than on the parallelization of threads, for its concurrency.

These two executions of each epoch are synchronized using online multiprocessor replay techniques [42]. The same program input is provided to both executions, and we ensure that both executions follow an identical happens-before for synchronization operations. In the absence of races, this guarantees that the two executions are consistent with each other [66].

In order to start future epochs in the epoch-parallel execution before previous epochs have finished, the starting state of the future epoch—or equivalently, the ending state of the previous epoch—must be predicted. The predicted state includes the architectural state of the process (the entire address space and all thread registers) as well as the relevant system state (e.g. so that file accesses are consistent). If the prediction is correct, then each epoch can be stitched together to form a single natural sequential execution of the process. However, if the prediction was incorrect—i.e. the previous epoch's ending state was different—stitching the two epochs together would result in an unnatural state transition. In this case, the epoch and its output must be discarded, and the epoch must be re-executed starting at the correct state. We execute each epoch speculatively to ensure that the epoch's output and state can be correctly discarded. As a committed epoch finishes, its final state is compared against the next epoch's starting state. If the two states match, the next epoch can be committed. Otherwise, the next epoch is rolled back and re-executed using the final committed state.

To generate these predictions for the epoch-parallel execution, we look to the thread-parallel execution. An epoch in the thread-parallel execution will typically execute faster that the corresponding epoch in the uniparallel execution. As the thread-parallel execution runs ahead, its state is used to predict the starting states for future epochs in the epoch-parallel execution execution.

5.2 System Architecture

We make a distinction between the race detection *algorithm* and the *architecture* to support it. The algorithm formally defines what events a program can generate and how to analyze those events to locate a race. The architecture defines how the program's behavior generates, collects, and presents those events to the algorithm.

In most existing race detectors, the architecture is straightforward to the point that it is not distinguished from the analysis itself. Instrumentation is added to a target program to capture relevant events, either through binary rewriting (static or dynamic) or by modifying the runtime layer (for languages that use one; e.g. Java). As events are encountered, they are sent to the race detection algorithm for immediate analysis ("on-the-fly" analysis) or logged in an execution trace file to be examined later ("post-mortem" analysis).

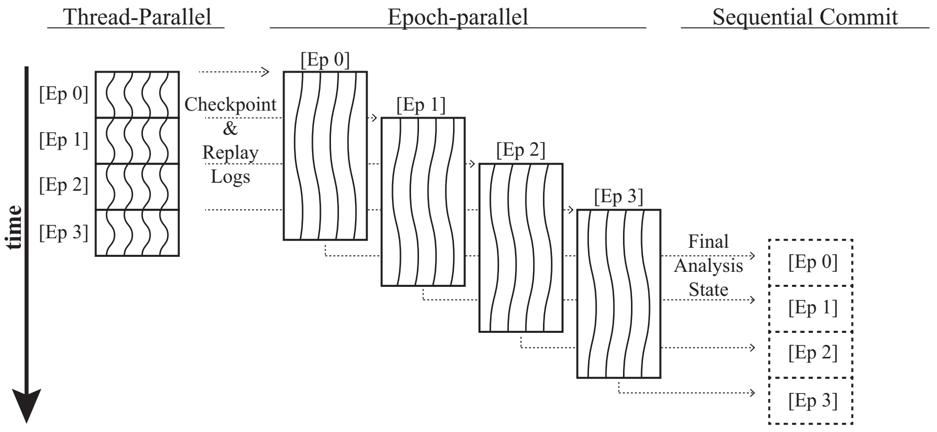


Figure 5.1: Parallel race detector architecture.

Our system uses a more complex uniparallel architecture to capture events. Figure 5.1 presents an overview of this architecture. Each program is executed twice, once as a thread-parallel execution and again as an epoch-parallel execution, and the two executions are constrained to have the same happens-before order. Full details of this execution style are given in Section 5.1.3.

Before proceeding, we must consider which parts of the race detector are slow. When we examined how much slowdown in a sequential detector was due to instrumenting and analyzing different events, we found that the amount of times spent handling synchronization operations is negligible compared to the overall execution time of program. The vast majority of work in a race detector comes from instrumenting and analyzing individual variable accesses.

Based on these results, we focused our efforts on making sure that the handlers for variable accesses would scale well. With this goal in mind, we handle synchronization events in both the thread-parallel execution and in the epoch-parallel execution, and we instrument variable accesses only in the epoch-parallel execution. Splitting the work in this way places the slowest work in the phase that we can parallelize the easiest.

Ideally, all work would be done exactly once in the epoch-parallel execution. However, race detection algorithms are inherently stateful: the analysis state gathered by analyzing prior epochs is required to correctly detect all races in the current epoch. Analyzing an epoch on its own will find all races in that epoch, but races that span epochs would not be detected. To recover the cross-epoch races, we introduce a sequential commit phase that examines epochs in program order as they finish.

To manage the cross-epoch dependencies in the analysis state, we keep a global copy of the state that reflects the final analysis state from all prior committed epochs. The commit phase is responsible for performing deferred checks against this state and then for updating it when the next epoch commits. Some of the checks that must be performed during an epoch require access to the final analysis state at the end of the previous epoch. Since that information is not available during the epoch, those checks are deferred until the epoch's commit phase (when the information is guaranteed to be available). We discuss the exact work that is deferred in Section 5.3. Once the deferred checks are made, the partial analysis state from the epoch is used to update the global committed state.

Keeping a single global committed state simplifies how epochs commit, but it does impose a strict sequential order on the tasks to be performed. If we were to keep around intermediate states after each epoch commits (i.e., committing an epoch generates a new state instead of updating a shared one), we could allow additional parallelism by processing the deferred checks for an epoch at the same time it's final state is being prepared. This however complicates our design and increases memory utilization.

5.2.1 Applicability to Analysis Algorithms

In designing this architecture, we exploit a key property of the FastTrack algorithm: the portion of the algorithm that tracks and maintains the happens-before relation can function on its own as a fast closed subset of the full algorithm. To specify this as a general property, we are interested in a subset of analysis state and events that meets the following conditions:

- The analysis of an event in the subset depends only on state in the subset.
- The state in the subset is only updated when analyzing events in the subset.

When a proper subset exists that meets these conditions, events outside the subset can be ignored while analyzing a program trace without affecting the the analysis state in the subset. Thus when we instrument only that subset of events in the thread-parallel execution, we are guaranteed that the analysis state it maintains will be identical to the state generated in the epoch-parallel execution. This property is necessary to maintain the equivalence between the thread-parallel and epoch-parallel executions when their instrumentations differ. It is also important that the analysis of events in the subset incur low overhead since this analysis will occur twice.

In general, the analysis architecture developed here can be used for any analysis algorithm that contains a suitable subset. Other race detection algorithms that are based on vector clocks perform happens-before tracking similar to FastTrack. In race detectors based on locksets, the maintenance of those locksets may also form a suitable subset.

This architecture can also be used for analyses that have no suitable subset of events and state. In this case, all instrumentation would exist only in the epoch-parallel execution. Section 5.6 discusses related work that has dealt with program analysis in this category.

5.2.2 Performance Discussion

The parallelism of our uniparallel execution is limited by two factors: the rate at which new epochs can be spawned from the thread-parallel execution, and the rate at which epochs can be committed. Spawning and committing epochs are fundamentally sequential; we extract parallelism from this system entirely through the epoch-parallel execution. Given enough cores, the rate at with epochs can finish in the epoch-parallel execution will eventually rise to some maximum supported rate, determined by the slower of the epoch creation rate or the epoch commit rate. It follows that the greatest amount of parallelism can be achieved by making these two sequential tasks as fast as possible. While this goal is hindered by handling synchronization events in the thread-parallel execution, the information gathered by this analysis is used to greatly simplify the parallel detection algorithm and reduce the amount of work that must be saved for the sequential commit phase. Our evaluation found that the slowdown to the thread-parallel execution is not significant for most benchmarks; in fact we often impose additional rate limiting on our thread-parallel execution to keep it from generating new epochs faster than our epoch-parallel execution can run them, or than our commit phase can commit them.

The amount of work done in the sequential commit phase turns out to be a key factor in performance. There can be a significant amount of work to be done in this phase, depending on the benchmark, and any processing time spent in this phase reduces the fraction of work that can be parallelized. Once there are a large number of cores available (i.e., the epoch-parallel phase has a completion rate), if it takes longer for an epoch to be committed than to be created, the commit phase will be the bottleneck in the epoch pipeline, limiting the rate at which epochs and be completed. On the other hand, if it takes less time to commit an epoch than it did to create it, then the thread-parallel execution will set the overall completion rate.

5.2.3 Handling a Race

One benefit of using an on-the-fly detection algorithm is the ability to catch the program the instant it performs its second unsynchronized access to a shared variable. Although this architecture cannot guarantee that a cross-epoch race will be detected until the end of each epoch, we can still recover the program to the exact location where the race occurred by reusing our replay logs in a method similar to RecPlay [66]. When a race is detected, the process can be rolled back to the beginning of the epoch. The replay logs can be used to advance the program's state up to the point where the race was detected. If the two racing instructions belong to the same epoch, both threads involved in the race can be advanced to the exact racing instructions.

After the correct program state is recovered, the reaction of the system should depend on the end use of the race detector. When debugging, it may be useful to generate a breakpoint and attach a debugger at the exact instant or to produce a core dump. If the goal is to generate a log for deterministic replay, two options are apparent: the racing instructions could be dynamically modified to log the outcome of the race; or the system could impose some method of ordering the racing operations, perhaps by inserting locks or momentarily executing one thread at a time.

5.3 Parallel FastTrack

This section describes the main contribution of this work: the Parallel FastTrack algorithm. It is designed to be used in a uniparallel architecture as described in Section 5.2.

The use of uniparallelism itself does not strictly require a new detection algorithm. As prior work has done [53], we might have hidden details of the uniparallel execution from the algorithm and merely used the execution as a faster way to generate a sequential log of program events. The instrumentation would be parallelized, but not the analysis. A commit phase then processes logs in order by the usual sequential algorithm. While the collection of events is a significant overhead in an execution, the analysis of those events also takes a significant amount of work.

By exposing the uniparallel architecture to the race detection algorithm, we construct a parallel race detection algorithm that distributes more of the work into the parallel phase. At a high level, Parallel FastTrack divides its work across three different phases. In the first phase (the thread-parallel execution), synchronization operations are analyzed to establish the happens-before order among program regions. The second phase (epoch-parallel execution) operates in parallel on each epoch and detects all races that occur within that epoch. The third phase (sequential commit) operates sequentially on epochs in program order to detect races that span multiple epochs.

The rest of this section describes this algorithm in detail. We first present an informal description of Parallel FastTrack. Then we build an analysis abstraction suitable for analyzing uniparallel executions. We then present the formalization of the algorithm and argue its equivalence to Sequential FastTrack. Finally, we discuss additional optimizations that can be made to the algorithm.

5.3.1 Informal Algorithm

Parallel FastTrack is an adaptation of Sequential FastTrack for use in a uniparallel architecture. The core of the algorithm is unchanged from the sequential version. Our modifications address how to divide work between parallel and sequential phases, what information to log during an epoch's execution, and how to process and merge epoch state when it commits. This section presents the algorithm informally.

We divide Sequential FastTrack into three phases. The first phase of work occurs in the thread-parallel execution, where we instrument and analyze synchronization operations to track the happens-before information for the program. This instrumentation and analysis imposes a minimal overhead on the program's execution. By performing this analysis in the thread-parallel execution, happens-before information is available for use in the second phase.

The second phase adds instrumentation to the epoch-parallel execution to analyze synchronization operations and all variable reads and writes. The uniparallel replay system will ensure that the happens-before information is the same for both the thread- and epoch-parallel executions. Within each epoch, Sequential FastTrack is used to detect accesses that race inside that epoch. However, this method cannot be used to check if the first access in an epoch races with any access in logically-earlier epochs. Without any analysis in the thread-parallel execution, the analysis state of each variable at the end of the previous epoch will not be known, so the check cannot be made. When this occurs, the check that would have been evaluated is logged for later evaluation. Once a variable's state is overwritten, which occurs at the first write to the variable, its new value can be used for the rest of the epoch to detect races in the epoch without logging.

The third sequential commit phase analyzes epoch-parallel epochs in program order as they finish executing. This phase has access to a separate *committed* version of analysis state for each variable. We maintain the invariant that the committed state is identical to the state generated by running Sequential FastTrack over all committed epochs. To commit a new epoch, its logged accesses must be verified first. The unknown values in logged checks are replaced by the true values from the committed state, and the check is reevaluated. This procedure will detect any race between the first access in the new epoch (which is logged) and the last access to a variable from all prior epochs. Then, the final state of any modified variable is used to update the committed state. When these steps are finished, the epoch has been committed, and the subsequent epoch can be processed if it has finished executing.

5.3.2 Uniparallel Analysis Abstraction

Before formally specifying Parallel FastTrack, we must consider how to reason about dynamic program analyses in a uniparallel architecture.

It is difficult to directly specify meaningful semantics for an algorithm that deals with many concurrent processes or threads (as happens in a thread-parallel execution). To make this task easier, a simplifying abstraction is used that eliminates the concurrency from the algorithm. Rather than dealing with actions performed concurrently across several cores, the program's execution is assumed to be sequentially-consistent, so it suffices to analyze *some* sequential execution of the program that is consistent with the concurrent execution. Common memory models followed by hardware and compilers preserve this property. Even programs with races can be considered sequentially-consistent (at the hardware level) up to the first race [2, 66]. Using this assumption, analyses can be specified to operate on an execution as a sequence of discrete events without considering concurrency.

We would similarly like to specify our analysis on some sequential execution of the program that is consistent with the uniparallel execution, allowing us to abstract away the concurrency. We find this execution by stitching together each epoch from the epoch-parallel execution to make a single execution. Although each of these epochs executes concurrently, the epochs are logically ordered by the program, and logically-adjacent epochs will have identical program states on their boundaries (this is guaranteed by the replay system used in uniparallelism). Hence, we can rearrange the epochs so that they occur sequentially. Because each epoch in the epoch-parallel execution runs on a single CPU core, an epoch is already sequentially-consistent internally. When arranged back-to-back, one continuous natural execution is created that is sequentially consistent. We preserve the original epoch boundaries as *epoch begin* and *epoch end* events in the execution so that the analysis algorithm can take special actions at the start and end of an epoch.

5.3.3 Formal Algorithm

We formally define Parallel FastTrack state in two components: synchronization state (T_i and L_C) and variable state, of which there are two copies. The synchronization state

Event at Thread i	**Check**	**Update**
Acquire lock L		$T_i := T_i \sqcup L_C$
Release lock L		$L_C := T_i \,;\, T_i := \mathrm{inc}_i(T_i)$
Fork thread j		$T_j := T_j \sqcup T_i \,;\, T_i := \mathrm{inc}_i(T_i)$
Join on thread j		$T_i := T_i \sqcup T_j \,;\, T_j := \mathrm{inc}_j(T_j)$
Read X	$X_W \sqsubseteq T_i \star$	$X_R := \begin{cases} S_i(T_i) & \textbf{if } \boldsymbol{X_R \sqsubseteq T_i} \\ X_R \sqcup S_i(T_i) & \textbf{otherwise} \end{cases}$
Write X	$X_W \sqsubseteq T_i \wedge X_R \sqsubseteq T_i \star$	$X_W := S_i(T_i)\ ;\ \boldsymbol{X_R := \bot}$
Epoch end	**verify log** $\star$	$\boldsymbol{\forall X :} \begin{cases} \boldsymbol{\overline{X_R} := X_R \,;\, \overline{X_W} := X_W} & \textbf{if } \boldsymbol{X_W \neq \Diamond_{X,W}} \\ \boldsymbol{\overline{X_R} := \overline{X_R} \sqcup X_R} & \textbf{otherwise} \end{cases}$
Epoch begin		$\boldsymbol{\forall X :\ X_W := \Diamond_{X,W} \,;\, X_R := \Diamond_{X,R}}$

Table 5.3: The Parallel FastTrack algorithm. Operations on synchronization events (top) are unchanged from Sequential FastTrack. In the update rules for variable accesses (middle), modified rules are given in boldface. The epoch event (bottom) is new in Parallel FastTrack. Its update rules specify how to update committed state ($\overline{X}$) with one epoch's local state (X). $\star$: Comparisons against $\Diamond$ must be logged and re-evaluated against committed state at the end of an epoch.

is the same as in Sequential FastTrack. For each variable X, it maintains read and write clocks in local state for each epoch (X_R and X_W) and in a separate committed state that is globally accessible ($\overline{X_R}$ and $\overline{X_W}$). Initially, L_C, and $\overline{X_{R/W}}$ are initialized to $\bot$. At the start of each epoch, $X_{R/W}$ are initialized to $\Diamond_{X,R/W}$, a special symbolic placeholder value. We treat $\Diamond$ as a vector clock, but it cannot combine with other clocks. That is, $\Diamond \sqcup \langle 0, 2\rangle$ must be stored as two components: $\Diamond$ and $\langle 0, 2\rangle$.

State is modified as the algorithm operates on the sequence of program events and epoch boundaries. The state update rules are given in Table 5.3. The top segment of these rules shows how program synchronization is handled. The middle segment shows how to handle read/write events within an epoch in the epoch-parallel execution. The lower segment shows how epoch boundaries affect the local and committed versions of read/write state.

Program synchronization events are the same as in Sequential FastTrack. These events are handled in both the thread-parallel and uniparallel executions. The deterministic replay system ensures that the partial order of these two executions are equivalent, so in our abstract sequential model, the state of the lock and thread clocks will be unaffected by epoch events.

Within an epoch, local variable state is updated in a similar way to Sequential FastTrack.

We alter the original algorithm so that updates will no longer depend on the representation of the vector clock (summary or full clock) but only on focus on the value of the clock. As events are handled, the access is first checked to see if it races with a prior access (according to the "check" column in Table 5.3), then the state of the variable is updated. At the time of the first write to a variable X in an epoch, X_W will be $\lozenge_{X,W}$, so this check cannot be evaluated. We log the access (X, W, T_i) so it can be checked later in the commit phase. The value of X_W is then overwritten with T_i, so subsequent comparisons to X_W need not be logged. Reads of X before its first write will update X_R but do not overwrite it, so we log (X, R, T_i) for evaluation in the commit phase.

When an *epoch end* event is handled, all logged accesses should be examined. From a log record $(X, R/W, C)$, we reconstruct and evaluate the check $\overline{X_{R/W}} \sqsubseteq C$ using the committed state for X. If one of these checks evaluates to False, then the access to X is part of a race that spans epochs. If all checks are True, then there are no races. Then, for every variable X that was accessed in the epoch, we update $\overline{X_{R/W}}$ to new state $X_{R/W}$ after substituting the old value of $\overline{X_{R/W}}$ in place of any $\lozenge_{R/W}$.

The amount of work done in this phase grows linearly with the amount of memory accessed during an epoch. Variables that are not accessed need no processing. With optimization, each variable can generate at most $N + 1$ logged checks for N threads (1 for the first write, N for the first read by N threads).

5.3.4 Analysis

We developed Parallel FastTrack to produce output identical to Sequential FastTrack. We formalize this condition as follows. For some sequential execution of a program that contains epoch boundaries (i.e., the epoch-parallel execution with epochs stitched together), consider a prefix of this execution that consists of all events from the beginning of the program to some *epoch end* event. After processing this final event, the committed state of Parallel FastTrack $(T_i, L_C, \overline{X_{R/W}})$ should be equivalent Sequential FastTrack's state $(T_i', L_C', X_{R/W}')$ *and* both algorithms should agree on whether there has been a race in the execution so far. The rest of this section argues that this equivalence condition is met by our modifications to the algorithm.

First, consider the synchronization states. Both sequential and parallel algorithms handle synchronization operations equivalently, and only synchronization operations update the synchronization state. It follows that $(T_i, L_C) = (T'_i, L'_C)$ after processing the same sequence of events.

Second, consider the variable clocks. Parallel FastTrack only has a symbolic value $\Diamond_{X,R/W}$ initially while Sequential FastTrack has a concrete value $\overline{X_{R/W}}$. However, whenever those values are needed to perform a race check, the check is deferred. In the sequential commit phase, these deferred checks will be evaluate against the concrete values for $\overline{X_{R/W}}$, resolving identically to Sequential FastTrack. If there is a data race involving the first read or write to a variable in an epoch, Sequential FastTrack will detect the race immediately while our deferred approach will not catch the race until later. However, because our sequential commit phase checks the same condition as Sequential FastTrack, the race will be discovered, and our architecture can roll back the program to make it appear as if the race were detected on-the-fly.

Once Parallel FastTrack writes a concrete value to $X_{R/W}$, future operations to X will proceed identically to Sequential FastTrack.

One potential source of differences between the two analyses may come from each analysis choosing a different way to handle each event. Sequential FastTrack may, for instance, decide not to set $X_R := \bot$ on a write, while our parallel analysis does. These small differences should not affect the outcome of any individual check, so the overall equivalence will still hold.

5.3.5 Optimizations

We use two optimizations at the algorithmic level to improve the performance of Parallel FastTrack by reducing the amount of work in the sequential commit phase.

Bounded read access logs The general algorithm as described logs every read access in an epoch up through the first write. This logs more accesses than necessary to find a race. If one read access at time C is logged (i.e., the check "$\Diamond \sqsubseteq C$" is deferred) to a variable and a second access happens at time T_i after the first access (i.e., $C \sqsubseteq T_i$), then the second access does not need its check (i.e. "$\Diamond \sqcup C \sqcup D \sqsubseteq T_i$") to be logged. Reasoning informally, if the first read did not race with the previous write, the second one, being ordered after the first, cannot either. If the first read does race, the second check is irrelevant. Formally, this is specified as $\Diamond \sqsubseteq C \wedge C \sqsubseteq T_i \Rightarrow \Diamond \sqsubseteq T_i$. With this optimization, each thread only needs to log its first read to the variable. Hence, the number of logged accesses is bounded by the number of threads.

Use natural barriers Intuitively, once a thread has synchronized with every other thread in an epoch, it is guaranteed that its subsequent accesses are ordered after every access from prior epochs. Consequently, its future race checks do not need to be logged. Formally, we use $\Diamond_{X,R/W}$ as a symbolic placeholder for the last access before the epoch. Although its exact value cannot be known until the commit phase, its range is limited. Let $E = \bigsqcup_i T_i$ for all threads i at the start of the epoch. By definition, $\Diamond_{X,R/W} \sqsubseteq E$, so $E \sqsubseteq T_i \Rightarrow \Diamond \sqsubseteq T_i$.

Even if the deferred check can be eliminated, the first access to a variable will generate some work for the sequential commit phase, since that variable's per-epoch state must be committed.

5.4 Implementation

We developed a custom implementation of our race detector that re-implements the Sequential FastTrack algorithm as well as Parallel FastTrack algorithm. This implementation consists of three parts: the replay system, the instrumentation, and the analysis library. The replay system is responsible for providing a uniparallel execution of the target program. We build our race detector on top of the infrastructure for DoublePlay [79], a system providing uniparallel executions as described in Section 5.1.3. To intercept relevant events from the program's execution, we use two techniques: first, the dynamic linker helps our race detector interpose on all program calls to pthread functions, and second, we statically instrument each load and store in the program during compilation to invoke our race detector. The events are then analyzed in library code that implements our detection algorithms.

The next two subsections describe these last two components in greater depth. We then describe the performance optimizations we use to decrease the total amount of work needed to perform race detection.

5.4.1 Instrumentation

The race detector intercepts program events through the use of two different techniques, depending on the event. Both techniques rely on altering the program at compile-time.

To intercept synchronization operations, we use dynamic library interposition. Our race detection library redefines the common pthread synchronization functions, such as `pthread_mutex_lock()`, and is statically linked with the target program, overriding the original functions. Our wrapper functions call into the race detection library before and after invoking the original functions (as needed), which are looked up by the dynamic linker.

Program loads and stores are intercepted via instrumentation added as a source program is compiled. We use the LLVM compiler with a custom pass that inserts calls to our read/write handlers (e.g., `handle_read(uint32_t addr, uint32_t size)`) into the program's intermediate representation. The exact instrumentation depends on which detector we are using. For the Sequential FastTrack algorithm, the function call is inserted unconditionally before every `load` or `store` instruction.

For Parallel FastTrack, we want the thread-parallel execution to run as fast as possible without handling loads or stores, while the epoch-parallel execution must handle each load and store. We implement this requirement by having the instrumentation first check a global variable `enable_instr` and then conditionally call the handler. This method was chosen for its simplicity, not performance. A better solution could involve switching the binary from a fast, uninstrumented version to a slow, instrumented version when a new epoch replay is forked from the thread-parallel run. Making these two binaries compatible is a challenge, though Prospect [77] presents one viable solution.

There are two consequences of our instrumentation technique. First, by instrumenting the LLVM IR, we skip handling accesses to variables that do not escape their functions. In type-safe programs, such variables are guaranteed to be inaccessible to other threads. In unsafe programs, bugs like wild pointer accesses or buffer overflows could result in a data race that our tool would miss, but these bugs could be caught and fixed by other means before employing this detector.

Second, the program's external libraries must either be compiled with instrumentation or must have their functions annotated. We implement annotations for libc through wrapper functions that explicitly invoke the read/write handlers. For all other libraries, we compile a special version that includes the appropriate instrumentation.

5.4.2 Analysis Library

The core event handler and race analysis algorithm is written in C++ and linked into a program via a static library. The exact procedure for updating analysis state is detailed in Section 5.3. We discuss in this section other interesting implementation details, particularly our use of shadow memory.

Logically, we assign a single FastTrack variable to each *byte* of program memory to detect races at byte granularity. However most program accesses are to word-sized units of memory (or multiples thereof). To save space and reduce the number of FastTrack updates, we shadow each source *word* with a single FastTrack variable. If individual bytes are accessed, we allocate an array of variables that can track each byte individually.

Parallel FastTrack requires two separate shadow memory regions. The local analysis state is kept in a private mmaped region so that each epoch has its own copy. The committed analysis state is in a shared mmap region so that all epochs can access it. Each shadow memory region is organized by a single-level page table. New regions are tracked in blocks of 4 MB. The code to look up the shadow word for address `addr` is simply `shadow_base[addr >> `*`shift1`*`][(addr & `*`mask`*`) >> `*`shift2`*`]`. The rest of the memory in each region is used for dynamic allocations of byte-level variables and for full vector clocks.

It is important that our race detector itself contains no data races and handles each program event atomically. This property does not come free when instrumenting a multi-threaded application. If the source program accesses the same location concurrently (e.g., by data races, read-shared variables, or `barrier_wait()`), instrumentation code may race with itself. Even when there is no sharing in the source program, our instrumentation adds its own sharing, since bytes are grouped together initially. To provide atomicity, our instrumentation guards shadowed memory with spinlocks. All shadow variables on a single cache line are guarded by the same lock to reduce contention. We keep a fixed number of spinlocks available; a hash of the memory address selects which lock to use.

Overall, our implementation of Parallel FastTrack has a minimum memory overhead of 4×. We use a single 32-bit integer to store a FastTrack compressed vector clocks (4 bit thread identifier with a 28 bit logical clock). Each variable needs two such clocks (last-read and -written) and has two shadowed copies (local and committed). By grouping program words, only 4 extra words of shadow memory are needed for each single word of program memory. When full vector clocks or arrays for byte-level tracking are needed, they are dynamically allocated from fixed-sized pools. For comparison, tracking bytes (not words) directly would bring the minimum overhead to 16×.

5.4.3 Optimizations

We use two optimizations to reduce the amount of work our detector is required to perform.

Lock elision In the uniparallel execution, locks are not needed to provide atomicity to our instrumentation. The uniparallel execution only allows one thread to run at a time, and context switches between threads are strictly controlled. When instrumentation is added, atomicity can be provided by ensuring that a context switch does not occur inside instrumentation code. Hence we can safely elide all spinlocks from the instrumentation. This optimization eliminates work that must be performed in Sequential FastTrack.

Omit memory comparison To ensure that the epoch-parallel execution is equivalent to a continuous thread-parallel execution, it is necessary to ensure that the final memory state of one epoch is equivalent to the beginning memory state of the next epoch. This condition allows epochs to be stitched together to form a single continuous run of the program. In prior systems that use epoch parallelism for multi-threaded applications (e.g., Respec, DoublePlay, and Frost), the presence of a data race within an epoch could cause its final memory to differ from the starting memory in the next epoch. Hence, to ensure continuity, any modified memory pages must be explicitly compared. This comparison can be omitted in our detector because the commit phase will discover the memory race on its own. This optimization reduces the overhead of uniparallelism.

5.5 Evaluation

Parallel FastTrack was designed to scale performance by increasing the number of CPU cores available. This section experimentally evaluates how well our implementation of this algorithm was able to achieve our goal. We also explore the sources of overhead in our implementation and quantify the benefit lock elision.

We first present our experimental setup and general methodology used to evaluate the system. We then present all our experimental data that measures scalability and overhead, followed by an analysis and discussion of these results.

5.5.1 Methodology

All experiments were performed on a single workstation that contains 6 GB RAM and uses dual Intel Xeon CPUs with 12 MB cache to provide a total of 8 cores running at 2.4 GHz. System software is built upon that of prior work on uniparallelism [79]. The system ran a 32-bit PAE Linux 2.6.26 kernel that has been heavily modified to provide speculative execution and uniparallel replay. We used a custom version of Glibc 2.5.1 to provide the user-space logging and replay components for uniparallel replay. Aside from Glibc, all programs and libraries were compiled in LLVM 2.9 with a custom instrumentation pass.

We primarily evaluate performance by comparing the execution times of a set of benchmark programs under different configurations. We examine a collection of five applications from the modified SPLASH-2 benchmark suite [81] — water-n^2, lu, ocean-contig, fft, and radix — and one parallel application — pbzip2. The SPLASH-2 applications are CPU and memory intensive and contain little I/O, if any. Pbzip2 is used to compress an 8 MB file, with multiple threads compressing different blocks of the file in parallel. We prefetch this file in advance to keep the application from becoming I/O-bound.

We only evaluate our system on benchmark applications that do not contain overly-frequent data races. If a program contains a frequent data race, the bug will be discovered quickly whether the detector is fast or slow. Performance is relevant mainly for programs that are mostly race-free, which may run for an extended amount of time before encountering any data race. We exclude many of the other SPLASH-2 applications on this basis. When a race is detected, our detector records the event, but does not otherwise alter the execution of the target program.

For the SPLASH-2 benchmarks, we adjust our workload sizes to produce an execution time of 100-120 second in a normal execution on a single core when possible. For some applications, we scale the input by adding a loop through program's main computation many times inside the same process. We also modify some of the more CPU-intensive applications to periodically make a system call. This change was introduced to offset a limitation of our replay system: it does not have support for deterministic interrupts which would be required to end an epoch via preemption. Inserting the system call gives our replay system opportunity to break a long-running epoch. For pbzip2, we were unable to scale the workload significantly beyond a few seconds.

Unless otherwise noted, all measurements represent the mean of at least 4 samples. We show the sample standard deviation along with means where noted.

5.5.2 Scalability Measurements

Our primary goal is to allow race detection to scale across available cores in the system, reducing the total execution time of the target application. We determine the scalability of our race detector for a given benchmark application by selecting a workload that consists of a fixed number of worker threads and measuring the execution time as we increase the number of CPU cores available. With a total of only 8 cores available on our test machine,

Application	Worker threads	System calls	Original time (s)	Sequential FT (s)	CPU cores	Parallel FT (s)	Parallel speedup
water-n^2	1	—	126 (2.6)	2983 (48)	1		
	2	2390742	72.2 (0.6)	1891 (59)	2	1419 (0.8)	1.33 (0.04)
					4	757 (2.5)	2.50 (0.08)
					6	495 (1.9)	3.82 (0.12)
					8	392 (2.9)	4.83 (0.16)
	4	4827240	44.4 (1.0)	1484 (23)	4		
					8	439 (4.5)	3.38 (0.06)
lu	1	—	103 (1.3)	3520 (68)	1		
	2	94002	53.3 (0.6)	2103 (61)	2	1623 (1.3)	1.30 (0.04)
					4	862 (4.5)	2.44 (0.07)
					6	564 (1.7)	3.73 (0.11)
					8	439 (5.0)	4.79 (0.15)
	4	144454	27.8 (0.8)	1178 (26)	4		
					8	438 (3.3)	2.69 (0.06)
ocean	1	—	118 (2.9)	4182 (98)	1		
	2	621767	64.4 (0.8)	2453 (187)	2	2026 (3.2)	1.21 (0.09)
					4	1081 (4.3)	2.27 (0.17)
					6	702 (1.5)	3.50 (0.27)
					8	534 (2.0)	4.60 (0.35)
	4	1969753	38.4 (0.6)	1731 (36)	4		
					8	551 (1.0)	3.14 (0.07)
fft	1	—	98.7 (1.7)	622 (77)	1		
	2	110764	50.5 (1.3)	2790 (123)	2	2531 (12)	1.10 (0.05)
					4	1338 (2.9)	2.09 (0.09)
					6	868 (1.8)	3.21 (0.14)
					8	662 (1.5)	4.22 (0.19)
	4	226127	27.8 (0.4)	1533 (24)	4		
					8	648 (12)	2.37 (0.06)
pbzip2	1	—	1.99 (0.04)	48.3 (1.2)	1		
	2	449	1.09 (0.03)	27.4 (0.8)	2	27.3 (0.5)	1.01 (0.03)
					4	18.2 (0.2)	1.51 (0.04)
					6	14.8 (0.9)	1.86 (0.12)
					8	14.7 (0.7)	1.87 (0.11)
	4	489	0.69 (0.04)	18.6 (0.43)	4		
					8	20.2 (2.8)	0.92 (0.13)
radix	1	—	118 (2.4)	481 (4.7)	1		
	2	9652	60.2 (0.9)	318 (68)	2	346 (1.7)	0.92 (0.20)
					4	212 (2.1)	1.50 (0.32)
					6	200 (3.0)	1.59 (0.34)
					8	198 (2.7)	1.60 (0.34)
	4	21308	31.5 (0.4)	311 (57)	4		
					8	179 (3.5)	1.74 (0.32)

Table 5.4: Scalability of Parallel FastTrack. Sample standard deviations are given in parentheses.

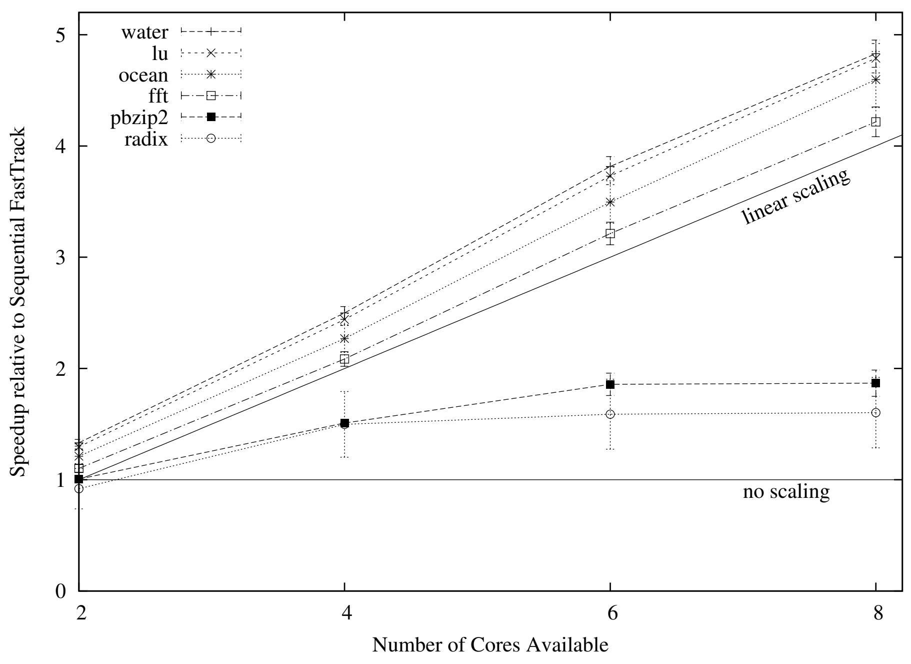


Figure 5.2: Scalability of Parallel FastTrack for two worker threads.

we chose to focus on 2- and 4-core workloads. We vary the available cores by pinning record threads to the first *p* cores for *p* worker threads and pinning replay threads to the first *N* cores, where we vary $p \leq N \leq 8$.

The results of our scalability study are presented in Table 5.4. For each application, we briefly characterize the workload as the number worker threads used and the number of system calls during one execution. We then measure the total execution time (in seconds) of the application when running without any modification or instrumentation (base) and when running using the Sequential FastTrack algorithm (Sequential FT). The next group of data present our main results: measurements of the total execution time when using Parallel FastTrack (Parallel FT) across the given number of CPU cores. Our speedup column shows the speedup of Parallel FastTrack relative to Sequential FastTrack (i.e., sequential/parallel execution time). To better illustrate the trends in this data, Figure 5.2 shows the last column of data when using two worker threads.

Overall, our implementation of Sequential FastTrack results in an average slowdown of 23.5× relative to the uninstrumented baseline (mean taken over all applications and all workloads). This slowdown is comparable to other implementations of non-sampling race detectors for unmanaged code [66–68, 71]. We note that our results are worse than the published slowdown for the original Java implementation of FastTrack. We feel that improved language safety allows a Java race detector to be better optimized than one for unsafe languages like C, but we defer this investigation for future work.

Our results show that Parallel FastTrack is indeed capable of parallelizing data race detection. We see average speedups of 2.1, 2.8, and 3.3 as we increase the number of CPU cores to 2×, 3×, and 4× the number of worker threads, respectively. There is a high variance in parallelization among the benchmark applications. Four of our SPLASH-2 benchmarks—water, lu, ocean, and fft—show super-linear speedup, with average speedups among just these benchmarks of 2.6, 3.6, and 4.6 with increasing number of cores. At the other end of the spectrum, pbzip and radix show little improvement, with a maximum speedup of 1.9 for any tested number of cores. We defer a full discussion of these results to Section 5.5.4.

5.5.3 Overhead Measurements

There are several different components that are required for Parallel FastTrack to function correctly. Figure 5.3 shows how these components affect the runtime of the detector for the workload of 2 threads. We normalize each application's runtime to its baseline measurement. This figure shows several bars for each application. The first bar on the left shows the normalized runtime of Sequential FastTrack, to serve as a reference point.

Each subsequent bar shows the runtime of Parallel FastTrack using a different number of cores. From left to right, we show 2, 4, 6, and 8 core performance. Each bar is further broken down into different components, each of which represents one part of the overhead. These components are:

- Baseline: The lowest bar on the stack shows the baseline, which is scaled for each application to be "1."
- Replay: The next higher bar shows the execution time of each process under a uniparallel replay. This result is higher than previously published results for uniparallel execution. Unlike prior work, we schedule replay threads on the same CPUs as

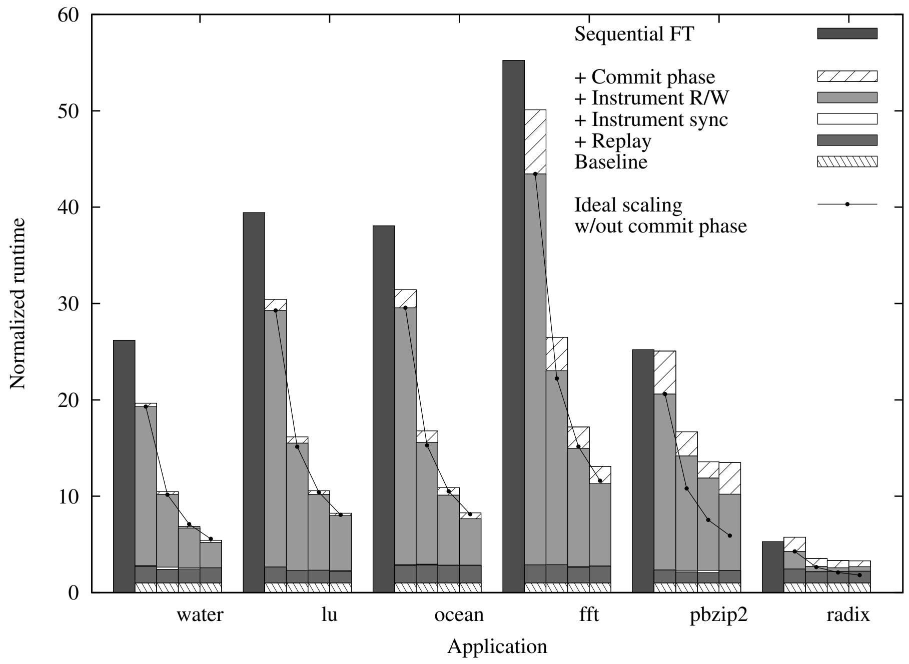


Figure 5.3: Overhead breakdown in Parallel FastTrack and the benefits of lock elision. Execution times for each application configuration are normalized to the baseline configuration. The workload shown uses 2 worker threads.

recording threads, thus slowing both down. While this scheduling strategy hurts performance when Replay is examined alone, it improves utilization and latency once instrumentation is added.

- Instrument sync: Instrumenting synchronization operations in both record and replay sides of the uniparallel execution results in a negligible increase in runtime (and is nearly indistinguishable in Figure 5.3).
- Instrument R/W: The largest slowdown in execution time overall comes from instrumenting and analyzing each memory operation within each epoch. This overhead can itself be broken into smaller components: calling into the race detector library (about 35%), locating shadow variables (20%), and analyzing accesses (40%).
- Commit phase: This component includes maintaining a log of the first access to each memory location as well as processing the log in the sequential commit phase.

The line drawn across each application shows the ideal scalability for each application based on the overhead breakdown of the 2-core trial. The scalable parts of the benchmark are Replay, Instrument Synch, and Instrument R/W. We compute the ideal line by scaling only those components of the runtime across multiple cores (i.e. $f(x)$ = sequential + scalable/x).

5.5.4 Discussion

All benchmarks showed some scalability and reduced latency, although the improvements were not consistent across applications. We discuss here three important factors that affect the performance of an application.

First, we found that the uniparallel execution of the was overall faster than the normal, sequential analysis even when using the same number of cores. This can be seen by examining the first two bars of each application in Figure 5.3, which directly compare the Sequential and Parallel FastTrack algorithms on 2 cores. It is this improvement that allows our benchmarks to exhibit super-linear scalability. Much of this improvement can be explained by our lock elision optimization. We found that on average, the use of locks in our instrumentation code was responsible for 22% of the runtime in the Instrument R/W component. These locks are unnecessary in the uniparallel execution, so we can remove them to reduce the overall execution time by about 3×. This optimization is somewhat unrelated to our parallelization of the FastTrack algorithm. This performance boost should be seen in any instrumentation that uses locks frequently and can run in a uniparallel architecture.

Second, we found that overall, the system is scaling as intended. Four out of six of our benchmarks show that the epoch-parallel phase can efficiently parallelize its work when given up to to 4 times as many cores as worker threads. This indicates that the epoch-parallel is the performance bottleneck in these benchmarks. Race detection on radix adds relatively little amount of overhead, so its bottleneck quickly becomes the thread-parallel phase (or commit phase, if enabled). Even with eight cores available, radix only has enough parallel work at any time to regularly make use of five cores. Pbzip2's trouble scaling is a result of its short length. Epoch parallelism has low core utilization at the beginning and end of replay, and pbzip2 was too short to amortize this cost. We are still working on scaling the workload for this benchmark.

Application	Total accesses	% First access	% Deferring a check (approx.)
water-n^2	43 M	0.1%	0.01%
lu	182 M	1.4%	0.2%
ocean	186 M	7%	0.07%
fft	62 M	9%	2.1%
pbzip2	209 M	5%	4.3%
radix	36 M	45%	3.9%

Table 5.5: Parallel FastTrack average memory accesses per epoch.

Third, we found that the temporal locality of memory references has a significant influence on performance and scalability. Table 5.5 looks at benchmarks using 2 worker threads and 8 cores and shows three measurements: the average number of memory accesses analyzed in each epoch, the percentage of total accesses that are the first access to a new variable, and the percentage of total accesses that defer a check (approximately). It shows a strong correlation between overall performance and the frequency of first accesses to a new memory location. Our detection algorithm can parallelize repeated accesses to the same location within one epoch, but the first access to a new location must be logged during the epoch-parallel execution and re-examined during the sequential commit phase. The sequential detector has no need for logging; this is overhead added by parallelization. Applications with high temporal locality will access many of the same memory locations repeatedly within one epoch. The resulting log will be small, and the sequential commit phase will be fast. Conversely, applications with little temporal locality will access many different memory locations with a low frequency. The optimizations mentioned in in Section 5.3.5 greatly help to reduce the number of checks that must be deferred to the commit phase. However, the optimizations cannot eliminate the need for logging altogether. Some record of each access must be made so that the shadow state for the location can be committed during the commit phase.

5.6 Related Work

Dynamic data race detection has been studied by many hardware [51, 55, 61] and software [2, 14, 15, 20, 23, 30, 59, 63, 66, 68, 83] approaches.

To our knowledge, this is the first work that attempts to parallelize data race detection, though the broader topic of parallelizing instrumentation and analysis has been previously considered. Both Speck [53] and SuperPin [80] use epoch-parallel execution to offload instrumentation and analysis of single-threaded programs to an epoch-parallel run. Their examples include a dynamic taint tracker and a data cache simulator, two analyses that depend on cross-epoch information. A similar architecture can be found in other works [77, 79, 84]. The architecture presented in this chapter is inspired by these projects, and we extend their basic architecture to allow limited instrumentation to run in the thread-parallel execution when needed.

Hardware-based systems have proposed using speculative execution to perform data race detection. ReEnact [61] uses Thread-Level Speculation hardware to detect races. Once a race is detected, recently-completed instructions are rolled back and re-executed to characterize and possibly repair the bug. SigRace [51] adds a Race Detection Module that quickly scans memory accesses for races using a Bloom filter. When a potential race is detected, program execution is rolled back and re-executed using a precise race detector. Software-based race detectors have not made use of speculative execution to our knowledge.

5.7 Chapter Conclusions

This chapter developed the Parallel FastTrack algorithm for dynamically detecting data races. By exposing the details of the uniparallel architecture at the algorithmic layer, we can parallelize the algorithm more efficiently. Each epoch in the uniparallel execution is instrumented to detect races that occur within the epoch. After epochs finish, they are handled sequentially in a commit phase that detects races across epochs.

Parallel FastTrack has attained its goal of reducing the execution time of analyzed programs. It provided average of speedups of 2.1, 2.8, and 3.3 when the number of CPU cores available are 2, 3, and 4 times the number of worker threads, respectively. Parallelization of a particular application is influenced predominantly by the application's temporal locality of memory accesses. If the application tends to access the same memory locations within an epoch, there will be less sequential work for the detector, leading to better performance scaling.

By eliding locks in the uniparallel execution, Parallel FastTrack can avoid doing about 22% of the work needed for a sequential detector, decreasing overall runtime by an average of 3× relative to a normal execution. We would expect to see similar benefits if uniparallelism were used to run other kinds of heavyweight instrumentation that makes use of frequent locks to protect small segments of code.

CHAPTER 6

Conclusion

This thesis has explored the benefits of making speculative executions visible across different layers and components of a software system. By allowing layers to cooperate to preserve safety for speculations, new opportunities are created for improved parallelization. This dissertation shows how multiple layers can cooperate, and it provides many examples of how to take advantage of the speculative opportunities present between different system layers. Our thesis is supported by the following contributions.

6.1 Contributions

We contribute to the theoretical understanding of speculative executions by introducing the idea of custom speculation policies. Speculative execution is used in different ways for different goals in a variety of systems. Even though the exact details change from system to system, we realized that there are fundamental concerns that each system addresses, and the variations fall into common patterns. These variations are captured in our idea of a speculation policy. The different dimensions of the policy let us describe how a particular use of speculation should be customized relative to a default conservative policy.

We demonstrate the utility of this idea by constructing a system for speculative execution that splits the common concerns of speculation into a shared mechanism in the operating system and a customized policy in user-level applications. By allowing these two software layers to cooperate together, speculative executions can be more easily defined by applications to take advantage of the opportunities generated by allowing speculations to be visible across system components.

Building on applications' new ability to easily control their speculative executions, we show how speculation can be used to hide the latency associated with a replicated service. We develop a new Byzantine fault-tolerance protocol which is optimized for clients that are capable of executing speculatively. The communication protocol is designed so that some speculative events on a client can be safely externalized to the replicated servers. As a result, clients are able to increase their utilization of network services and reduce the apparent latency of their requests.

As a final contribution, this thesis develops a new race detection algorithm that takes advantage of uniparallel execution to improve its parallelization. We expose the speculative epoch parallelism provided by the uniparallel execution model at the algorithmic layer. Our Parallel FastTrack algorithm can then distribute its work across the different phases of uniparallelism while maintaining the same semantics as a race detector built for normal executions.

6.2 Future Work

There are some specific limitations of our work that could be addressed in the future. The interface we use to describe speculative policies in an application permits a wide range of application behaviors and policies. In particular, it puts the burden of ensuring a safe and consistent program execution on the developer. By making speculative policies visible to the application programming language, it may be possible to shift this burden of safety from the developer to the language runtime environment. Language support for speculation policies could also alleviate some of the difficulty we found when dealing with object differences and multi-threaded applications by controlling memory allocations and providing lightweight single-thread checkpoints. One other promising direction to explore is to integrate the creation policy with the *future* concurrency construct. A future is a placeholder value for the results of an asynchronous computation running in another thread. Once the computation has finished, the future can be used like a normal variable. Until then, attempts to access it will block. We imagine that a future could serve to delineate an individual task and provide a clean definition for the "result" of that task.

Although the developer of an application is the best position to describe its speculation policy, it may be worthwhile to investigate whether an automated system could examine an application's behavior to construct a better default policy. When looking at an operating-system-layer speculation system, we can imagine that the use of machine learning algorithms may help make some system calls predictable when an application uses them in a regular pattern. It might also be capable of deciding whether an application's attempt at communication should be blocked, buffered, or used to extend the boundary of speculation, based on observed communication patterns.

We plan on extending our parallel race detection architecture to support additional algorithms. In particular, algorithms based on Locksets seem to be amenable to the kind of state partitioning we perform for Parallel FastTrack. This would permit a similar architecture to be used: synchronization operations could be handled in the thread-parallel phase, reads and writes handled in the uniparallel phase, and modified memory merged in a commit phase. Even other uses of invasive instrumentation, such as profiling or hardware simulation, may benefit from running in a uniparallel environment. In particular, lock elision provides a significant reduction in overhead for heavyweight instrumentation that shares data internally.

6.3 Final Remarks

Concurrent programming poses a significant challenge for developers, and it is one that must be mastered if we are to continue to make full use of modern hardware. It it simply not practical to build new high-performance systems that lack some measure of concurrency. Speculative execution is one tool that can help developers use concurrency in a controlled and understood way. This thesis has removed some of the technical and conceptual barriers that hindered the use of speculative executions. It is my hope that with this thesis, speculative execution becomes more practical to use as a basic concurrency primitive for use in common applications.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. Fault-scalable byzantine fault-tolerant services. In *Proc. 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 59–74.

[2] Adve, S. V., Hill, M. D., Miller, B. P., and Netzer, R. H. B. Detecting data races on weak memory systems. In *Proc. 18th Annual International Symposium on Computer Architecture* (1991), ACM, pp. 234–243.

[3] AMD. AMD Phenom II processors product brief. `http://www.amd.com/`, May 2011.

[4] Avizienis, A. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985), 1491–1501.

[5] Ben-Or, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC '83)* (New York, NY, USA, 1983), ACM, pp. 27–30.

[6] Bracha, G., and Toueg, S. Resilient consensus protocols. In *Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC '83)* (New York, NY, USA, 1983), ACM, pp. 12–26.

[7] Bracha, G., and Toueg, S. Asynchronous consensus and broadcast protocols. *Journal of the ACM 32*, 4 (1985), 824–840.

[8] Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, June 2006), pp. 1–13.

[9] Carson, M., and Santay, D. NIST Net – a Linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review 33*, 3 (June 2003), 111–126.

[10] Castro, M. Practical byzantine fault tolerance. Tech. Rep. MIT-LCS-TR-817, MIT, January 2001.

[11] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *Proc. 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, USA, February 1999), pp. 173–186.

[12] Castro, M., and Liskov, B. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. 4th Symposium on Operating Systems Design and Implementation* (October 2000), pp. 19–33.

[13] Chang, F., and Gibson, G. A. Automatic I/O hint generation through speculative execution. In *Proc. 3rd Symposium on Operating Systems Design and Implementation* (February 1999), pp. 1–14.

[14] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2002), ACM, pp. 258–269.

[15] Choi, J.-D., Miller, B. P., and Netzer, R. H. B. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems 13*, 4 (October 1991), 491–530.

[16] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. 7th Symposium on Operating Systems Design and Implementation* (November 2006), pp. 177–190.

[17] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High availability via asynchronous virtual machine replication. In *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, CA, April 2008), pp. 161–174.

[18] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation* (December 2004), USENIX Association, pp. 137–149.

[19] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *Journal of the ACM 35*, 2 (1988), 288–323.

[20] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: A race and transaction-aware java runtime. In *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2007), PLDI'07, ACM, pp. 245–255.

[21] 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.

[22] Fidge, C. J. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications 10*, 1 (February 1988), 56–66.

[23] Flanagan, C., and Freund, S. N. FastTrack: Efficient and precise dynamic race detection. In *Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2009), PLDI'09, pp. 121–133.

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

[25] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[26] Hammond, L., Willey, M., and Olukotun, K. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev. 32*, 5 (October 1998), 58–69.

[27] HENDRICKS, J., GANGER, G. R., AND REITER, M. K. Low-overhead byzantine fault-tolerant storage. In *Proc. 21st ACM Symposium on Operating Systems Principles* (October 2007), pp. 73–86.

[28] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture* (San Diego, CA, May 1993), pp. 289–300.

[29] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (March 2007), ACM, pp. 59–72.

[30] ITZKOVITZ, A., SCHUSTER, A., AND ZEEV-BEN-MORDEHAI, O. Toward integration of data race detection in DSM systems. *Journal of Parallel and Distributed Computing 59*, 2 (November 1999), 180–203.

[31] JEFFERSON, D., BECKMAN, B., WIELAND, F., BLUME, L., DILORETO, M., HONTALAS, P., LAROCHE, P., STURDEVANT, K., TUPMAN, J., WARREN, V., WEDEL, J., YOUNGER, H., AND BELLENOT, S. Distributed simulation and the Time Warp operating system. In *Proc. 11th ACM Symposium on Operating Systems Principles* (Austin, TX, November 1987), pp. 77–93.

[32] KELSEY, K., BAI, T., DING, C., AND ZHANG, C. Fast Track: A software system for speculative program optimization. In *Proc. 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Seattle, WA, March 2009), pp. 157–168.

[33] KEMME, B., PEDONE, F., ALONSO, G., AND SCHIPER, A. E. Processing transactions over optimistic atomic broadcast protocols. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 1999), IEEE Computer Society, p. 424.

[34] KIHLSTROM, K. P., MOSER, L. E., AND MELLIAR-SMITH, P. M. The SecureRing protocols for securing group communication. In *Proc. 1998 Hawaii International Conference on System Sciences* (1998), vol. 3, pp. 317–326.

[35] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *Proc. 21st ACM Symposium on Operating Systems Principles* (October 2007), pp. 45–58.

[36] KOTLA, R., AND DAHLIN, M. High throughput byzantine fault tolerance. In *Proc. 2004 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, p. 575.

[37] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.

[38] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[39] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[40] Lange, J. R., Dinda, P. A., and Rossoff, S. Experiences with client-based speculative remote displays. In *Proc. 2008 USENIX Annual Technical Conference* (Boston, MA, June 2008), USENIX Association, pp. 419–432.

[41] Larus, J. Spending more's dividend. *Communications of the ACM 52*, 5 (May 2009), 62–69.

[42] Lee, D., Wester, B., Veeraraghavan, K., Narayanasamy, S., Chen, P. M., and Flinn, J. Respec: Efficient online multiprocesor replay via speculation and external determinism. In *Proc. 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2010), ASPLOS '10, pp. 77–90.

[43] Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proc. 2005 USENIX Annual Technical Conference* (Anaheim, CA, USA, April 2005), USENIX Association, pp. 31–44.

[44] Link, B. R. XTrace - trace X protocol connections. `http://xtrace.alioth.debian.org/`, September 2010.

[45] Lipasti, M. H., and Shen, J. P. Exceeding the dataflow limit via value prediction. In *Proc. 29th Annual ACM/IEEE International Symposuim on Microarchitecture* (December 1996), IEEE Computer Society, pp. 226–237.

[46] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. Value locality and load value prediction. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), ASPLOS '96, ACM, pp. 138–147.

[47] Madani, O., Bui, H., and Yeh, E. Efficient online learning and prediction of users' desktop actions. In *Proc. 21st International Joint Conference on Artificial Intelligence* (Pasadena, CA, July 2009), pp. 1457–1462.

[48] Malkhi, D., and Reiter, M. Byzantine quorum systems. *Distributed Computing 11*, 4 (1998), 203–213.

[49] Mattern, F. Virtual time and global states of distributed systems. In *Proc. International Workshop on Parallel and Distributed Algorithms* (Chateau de Bonas, France, October 1988).

[50] Mickens, J., Elson, J., Howell, J., and Lorch, J. Crom: Faster web browsing using speculative execution. In *Proc. 7th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, CA, April 2010).

[51] Muzahid, A., Suárez, D., Qi, S., and Torrellas, J. SigRace: Signature-based data race detection. In *ISCA'09* (June 2009), pp. 337–348.

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

[53] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. Parallelizing security checks on commodity hardware. In *Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2008), ASPLOS '08, pp. 308–318.

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

[55] Nistor, A., Marinov, D., and Torrellas, J. Light64: Lightweight hardware support for data race detection using systematic testing of parallel programs. In *MICRO'09* (December 2009), pp. 541–552.

[56] Oki, B., and Liskov, B. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[57] Patt, Y. N., Melvin, S. W., mei Hwu, W., and Shebanow, M. C. Critical issues regarding HPS, a high performance microarchitecture. In *Proc. 18th Annual Workshop on Microprogramming* (1985), ACM, pp. 109–116.

[58] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating system transactions. In *Proc. 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 161–176.

[59] Pozniansky, E., and Scheuster, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 2003), pp. 179–190.

[60] Prabhu, P., Ramalingam, G., and Vaswani, K. Safe programmable speculative parallelism. In *Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2010), PLDI'10, ACM, pp. 50–61.

[61] Prvulovic, M., and Torrellas, J. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture* (June 2003), pp. 110–121.

[62] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies — a safe method to survive software failure. In *Proc. 20th ACM Symposium on Operating Systems Principles* (October 2005), pp. 235–248.

[63] Ratasaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B., Nagpal, R., and Pattabiraman, K. Detecting and tolerating asymmetric races. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (February 2009), ACM, pp. 173–184.

[64] Reiter, M. K. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, vol. 938. Springer-Verlag, Berlin Germany, 1995, pp. 99–110.

[65] Rodrigues, R., Castro, M., and Liskov, B. BASE: Using abstraction to improve fault tolerance. In *Proc. 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 15–28.

[66] Ronsse, M., and Bosschere, K. D. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (May 1999), 133–152.

[67] Sack, P., Bliss, B. E., Ma, Z., Petersen, P., and Torrellas, J. Accurate and efficient filtering for the intel thread checker race detector. In *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability* (2006), ACM, pp. 34–41.

[68] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15*, 4 (November 1997), 391–411.

[69] Schmuck, F., and Wylie, J. Experience with transactions in QuickSilver. In *Proc. 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, October 1991), pp. 239–253.

[70] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (December 1990), 299–319.

[71] Serebryany, K., and Iskhodzhanov, T. ThreadSanitizer: Data race detection in practice. In *Proc. Workshop on Binary Instrumentation and Applications* (December 2009), WBIA'09, ACM, pp. 62–71.

[72] Smith, J. D., Ţăpuş, C., and Hickey, J. The Mojave compiler: Providing language primatives for whole-process migration and speculation for distributed applications. In *Proc. International Parallel and Distributed Processing Symposium* (March 2007), pp. 1–8.

[73] Smith, J. E. A study of branch prediction strategies. In *Proc. 8th Annual International Symposium on Computer Architecture* (May 1981), pp. 135–148.

[74] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. A scalable approach to thread-level speculation. In *Proc. 2000 International Symposium on Computer Architecture* (June 2000), pp. 1–24.

[75] Steffan, J. G., and Mowry, T. C. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. 4th International Symposium on High Performance Computer Architecture* (Las Vegas, NV, February 1998), IEEE Computer Society, pp. 2–13.

[76] Su, Y.-Y., Attariyan, M., and Flinn, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proc. 21st ACM Symposium on Operating Systems Principles* (October 2007), pp. 237–250.

[77] Süsskraut, M., Knauth, T., Weigert, S., Schiffel, U., Meinhold, M., and Fetzer, C. Prospect: A compiler framework for speculative parallelization. In *Proc. 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada, April 2010), ACM, pp. 131–140.

[78] Ţăpuş, C., Smith, J. D., and Hickey, J. Kernel level speculative DSM. In *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid* (May 2003), pp. 487–494.

[79] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. DoublePlay: Parallelizing sequential logging and replay. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2011), ASPLOS '11, pp. 15–26.

[80] Wallace, S., and Hazelwood, K. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. International Symposium on Code Generation and Optimization* (2007), IEEE Computer Society, pp. 209–220.

[81] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd Annual International Symposium on Computer Architecture* (June 1995), ACM, pp. 24–36.

[82] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. Separating agreement from execution for byzantine fault tolerant services. In *Proc. 19th ACM Symposium on Operating Systems Principles* (October 2003), pp. 253–267.

[83] Yu, Y., Rodeheffer, T., and Chen, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proc. 20th ACM Symposium on Operating Systems Principles* (2005), pp. 221–234.

[84] Zilles, C., and Sohi, G. Master/slave speculative parallelization. In *Proc. 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)* (2002), pp. 85–96.