Operating System Support for Application-Specific Speculation

Benjamin Wester Peter M. Chen Jason Flinn
University of Michigan
Ann Arbor, MI, USA
{bwester,pmchen,jflinn}@umich.edu

Abstract

Speculative execution is a technique that allows serial tasks to execute in parallel. An implementation of speculative execution can be divided into two parts: (1) a *policy* that specifies what operations and values to predict, what actions to allow during speculation, and how to compare results; and (2) the *mechanisms* that support speculative execution, such as checkpointing, rollback, causality tracking, and output buffering.

In this paper, we show how to separate policy from mechanism. We implement a speculation mechanism in the operating system, where it can coordinate speculations across all applications and kernel state. Policy decisions are delegated to applications, which have the most semantic information available to direct speculation.

We demonstrate how custom policies can be used in existing applications to add new features that would otherwise be difficult to implement. Using custom policies in our separated speculation system, we can hide 85% of program load time by predicting the program's launch, decrease SSL connection latency by 15% in Firefox, and increase a BFT client's request rate by 82%. Despite the complexity of the applications, small modifications can implement these features since they only specify policy choices and rely on the system to realize those policies. We provide this increased programmability with a modest performance trade-off, executing only 8% slower than an optimized, application-implemented speculation system.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Performance

Keywords Policy, Mechanism, Speculative execution

1. Introduction

Speculative execution has been widely used as a method for increasing parallelism by allowing serial tasks to execute concurrently. It has been used to improve performance in many hardware and software systems, including processor branch predictors [Smith 1981], distributed file systems [Nightingale 2005], remote displays [Lange 2008], fault-tolerant protocols [Wester 2009], virtual-machine replication [Cully 2008], discrete event simulators [Jefferson 1987], and JavaScript interpreters [Mickens 2010].

To execute speculatively, the system predicts the outcome of a particular operation and continues its execution based on that prediction. When the operation completes, the actual result is compared with the predicted result. If the prediction was correct, the system commits the speculative state. Otherwise, the system corrects the state produced by the misprediction, usually by rolling back to a prior point in time.

Each implementation of speculative execution can be divided in 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.

Existing systems typically implement mechanism and policy 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.

In this paper, 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 SSL/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 paper 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 paper is laid out as follows. Section 2 describes how speculative execution works when implemented below the application. Section 3 explores the different behaviors a policy can customize, and Section 4 describes two issues that arise when applications control their own speculation. Section 5 describes what mechanisms we implement in the operating system to support custom speculation policies. Section 6 discusses the process used to locate and implement custom policies. Section 7 describes three case studies for using customized speculation policies and evaluates the performance improvements that custom policies enables. Section 8 describes related work, and Section 9 concludes.

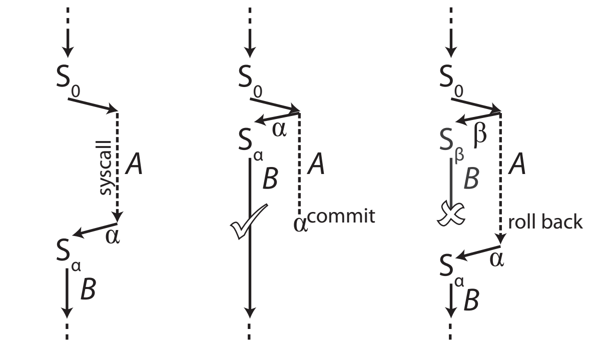
2. Generic Speculation

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 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 ac-



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

Figure 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).

tion 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 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 1b). Otherwise, it aborts the speculation and rolls back all state within the speculation's boundary (Figure 1c).

3. Custom Policies

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

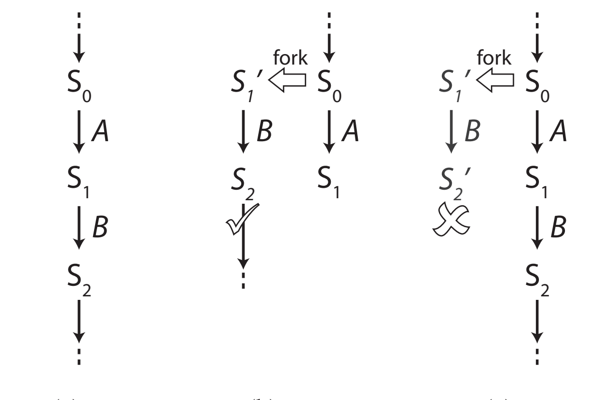
A custom, application-specific policy can vary from a generic policy in several ways: creating speculations, managing output, evaluating results, and controlling the commit. Overall, custom policies benefit an application by letting it make more predictions, helping those predictions be more accurate,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

Figure 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 2b and 2c) incorporates the predicted result of the action and continues executing speculatively; we call this copy the *speculative process*. The other copy of the process (right side of Figures 2b and 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.

3.1 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



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

Figure 2. Speculation with 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*.

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 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 [Lange 2008].

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.

3.3 Committing

When the action whose result is being predicted finishes, the system must decide whether to 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.

4. 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.

4.1 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.

4.2 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.

5. Mechanism Design & Implementation

Custom policies were introduced in Section 3 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.

5.1 Overview

Our mechanism for speculative execution is implemented at the operating system layer and is based on Speculator, a modified Linux 2.6.26 kernel that provides process-level speculative execution [Nightingale 2005]. Speculator allows processes to continue to interact with the system after becoming speculative. In particular, speculative state can be propagated through several forms of IPC: forking, waiting on children, signals, and file system operations (through files, pipes, and sockets). Each of these kernel objects can be checkpointed and rolled back as needed.

We introduce custom policies to this system by creating a group of new system calls, which are described in Section 5.2. 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 de-

`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. System call API used by an application to construct custom policies.

tected, the system may conservatively abort the speculation. The application should gracefully resume executing as if no speculation were created.

5.2 Policy API

Applications implement custom policies through a new set of system calls. An overview of each call is given in Figure 3.

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 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.)

6. 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.

6.1 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 that do not change local state from the mutating functions. In our experience, suitable interfaces are often found at the boundary between program modules. If an action seems too convoluted, it may be more reasonable to look at a different abstraction layer.

6.2 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 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

```
1  int count;  /* Global state */
2
3  int* foo() {
4    ...
5    count++;
6    return ptr;
7  }
8
9  int* spec_foo() {
10   int p_cnt = count + 1;
11   int p_ret = 1;
12   (stat, spec_id) = spec_fork();
13   if (stat == SPEC) {
14     count++;
15     result = new int(p_ret);
16   } else if (stat == CONTROL) {
17     result = foo();
18     if (count == p_cnt && *result == p_ret)
19       commit(spec_id);
20     else
21       abort(spec_id);
22   }
23   return result;
24 }
25
26 void work() {
27   x = spec_foo();  /* Replaces foo() */
28   p = set_policy(fd, ALLOW);
29   send(*x);
30   set_policy(fd, p);
31 }
```

Listing 1. Basic structure for predicting the result of simple function call.

(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.

6.3 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.

7. 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 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 dual single-core Xeon 3 GHz processors with 8 GiB of RAM.

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

Table 1. Speculative applications. A "Y" indicates that the application has a custom policy defined for that category.

7.1 Predictive Application Launching

We make use of custom speculation policies to improve perceived application startup time by predicting the launch

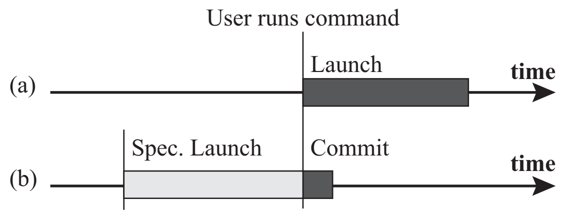


Figure 4. Process Execution Time. In part (a), a process is launched normally at the time the user invokes it. In part (b), the program is launched speculatively ahead of time. We measure the execution time after the user invokes a command (dark bar).

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 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.

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 4 illustrates our method for examining the capacity of non-interactive programs to launch speculatively. In a normal launch (Figure 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 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.

Table 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 7.1.1 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 7.1.2 describes how we implement an X11 proxy that lets graphical programs benefit from a speculative launch.

7.1.1 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 [Madani 2009], 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 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.

7.1.2 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 [Link 2010]. 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.

7.2 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, a 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 variant of the `ocsp_GetOCSPStatusFromNetwork()` function 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 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

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. 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.

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. 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.

7.3 BFT Client

We next examine a client in the PBFT-CS protocol [Wester 2009], a program already implemented to use an application-level speculation system. This allows us to compare our generic mechanism against one that has been designed for a single application.

The PBFT-CS protocol was designed to decrease the perceived latency of executing requests on a Byzantine fault-tolerant (BFT) cluster. A complete characterization of this problem is present in the cited work. Here, we summarize the system and discuss how applications used PBFT-CS.

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.

To evaluate PBFT-CS, a client was constructed that implements its own lightweight checkpoint system. This is a single-purpose application-implemented speculation system. In this work, we compare that client against our own client designed 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.

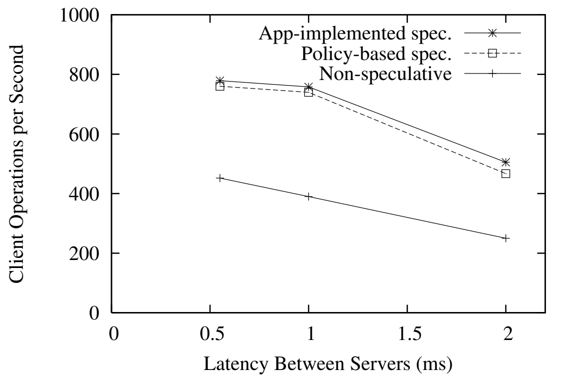


Figure 5. Comparison of BFT clients.

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 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 allows the client to issue 82% more requests per second than the non-speculative client with latencies above 0.5 ms. The client using application-implemented speculation 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.

8. Related Work

Fast Track [Kelsey 2009] 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 [Süßkraut 2010] 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 applications to control their own speculations [Mickens 2010]. 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 [Steffan 1998]. 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 [Schmuck 1991] and TxOS [Porter 2009]. 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 [Gray 1993]). 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 [Herlihy 1993]. 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 [Nightingale 2005, 2006]. Pulse speculatively resumes threads that are waiting for a resource to see if they will deadlock [Li 2005]. 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 [Jefferson 1987]. Ţăpuş et al. also performed similar speculations for a distributed shared memory system [Ţăpuş 2003]. 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 [Carlstrom 2006]. 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 [Smith 2007]. 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 [Lange 2008]. 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.

9. Conclusions

In this paper, 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.

Acknowledgments

We would like to thank our shepherd Frans Kaashoek and the many anonymous reviewers who provided valuable feedback to improve this paper. This work is supported by Intel and the National Science Foundation under awards CNS-0905149 and CNS-0614985. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

[Carlstrom 2006] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, Ottawa, Ontario, Canada, June 2006.

[Cully 2008] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, San Francisco, CA, April 2008.

[Gray 1993] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Herlihy 1993] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993.

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

[Kelsey 2009] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast Track: A software system for speculative program optimization. In *Proc. 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 157–168, Seattle, WA, March 2009.

[Lange 2008] John R. Lange, Peter A. Dinda, and Samuel Rossoff. Experiences with client-based speculative remote display. In *Proc. 2008 USENIX Annual Technical Conference*, pages 419–432, Boston, MA, June 2008.

[Li 2005] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proc. 2005 USENIX Annual Technical Conference*, pages 31–44, Anaheim, CA, April 2005.

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

[Madani 2009] Omid Madani, Hung Bui, and Eric Yeh. Efficient online learning and prediction of users' desktop actions. In *Proc. 21st International Joint Conference on Artificial Intelligence*, pages 1457–1462, Pasadena, CA, July 2009.

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

[Nightingale 2005] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles*, pages 191–205, Brighton, United Kingdom, October 2005.

[Nightingale 2006] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. 7th Symposium on Operating Systems Design and Implementation*, pages 1–14, Seattle, WA, October 2006.

[Porter 2009] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proc. 22nd ACM Symposium on Operating Systems Principles*, pages 161–176, October 2009.

[Schmuck 1991] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 239–253, Pacific Grove, CA, October 1991.

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

[Smith 2007] Justin D. Smith, Cristian Ţăpuş, and Jason Hickey. The Mojave compiler: Providing language primatives for whole-process migration and speculation for distributed applications. In *Proc. International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[Steffan 1998] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. 1998 Symposium on High Performance Computer Architecture*, pages 2–13, Las Vegas, NV, February 1998.

[Süßkraut 2010] Martin Süßkraut, Thomas Knauth, Stefan Weigert, Ute Schiffel, Martin Meinhold, and Christof Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proc. 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 131–140, Toronto, Ontario, Canada, April 2010.

[Ţăpuş 2003] Cristian Ţăpuş, Justin D. Smith, and Jason Hickey. Kernel level speculative DSM. In *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 487–494, May 2003.

[Wester 2009] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating lagency in replicated state machines through client speculation. In *Proc. 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 245–260, Boston, MA, April 2009.