SpecRPC: A General Framework for Performing Speculative Remote Procedure Calls

Xinan Yan
University of Waterloo
xinan.yan@uwaterloo.ca

Arturo Pie Joa
University of Waterloo
apiejoa@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Benjamin Cassell
University of Waterloo
becassel@uwaterloo.ca

Tyler Szepesi
University of Waterloo
stszepes@uwaterloo.ca

Malek Naouach
University of Waterloo
mm2naoua@uwaterloo.ca

Disney Lam
University of Waterloo
y7lam@uwaterloo.ca

ABSTRACT

In this paper we introduce SpecRPC, a speculative execution framework that allows applications to concurrently execute dependent operations both locally and through remote procedure calls. The framework tracks dependencies among non-speculative and speculative operations and ensures that incorrect speculations do not affect the correctness of applications that follow our suggested design pattern. By using speculation to parallelize dependent operations, SpecRPC can significantly reduce application latency even if only a fraction of the results can be correctly speculated. We evaluate SpecRPC by using it to implement Replicated Commit, a low-latency distributed transaction commit protocol for geo-replicated database systems. Our evaluation results show that, compared to RPC frameworks that sequentially execute dependent operations, SpecRPC can reduce the average transaction completion time of Replicated Commit by 58%.

CCS CONCEPTS

• **Software and its engineering** → **Middleware**; Communications management; • **Information systems** → Distributed database transactions; • **Networks** → Application layer protocols; • **Computer systems organization** → Client-server architectures;

ACM Reference Format:
Xinan Yan, Arturo Pie Joa, Bernard Wong, Benjamin Cassell, Tyler Szepesi, Malek Naouach, and Disney Lam. 2019. SpecRPC: A General Framework for Performing Speculative Remote Procedure Calls. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274808.3274829

1 INTRODUCTION

Many distributed applications perform a large number of remote procedure calls (RPCs). For example, a Facebook front-end web server performs as many as 392 RPCs to backend servers when generating a single HTTP response [12] for a page request. Generating the response can take as long as 3.1 seconds which, given the low latency of datacentre networks, suggests that many of the RPCs are performed sequentially. Sequential execution of RPCs can be due to insufficient resources leading to queuing at the involved servers. However, with the increasing abundance of low-cost computing resources in well-provisioned datacentres, sequential execution is more likely because of dependencies between RPCs, where an RPC invocation depends on the results of other RPCs. For instance, Song et al. [36] demonstrate the prevalence of dependent RPCs, providing an example containing five dependent RPCs. Likewise, Bahl et al. [2] and Natarajan et al. [29], also present complex dependencies among networked services in enterprise applications.

Unlike most performance problems, providing additional resources is minimally effective at reducing latency when sequentially executing RPCs. Although faster networks and CPUs can reduce transfer and processing time, latency improvements will likely be incremental since network latency has only seen a modest reduction in the past three decades [34], and recent CPU improvements mainly benefit concurrent operations. For cross-datacentre RPCs, data replication and caching can in some cases significantly reduce latency. Unfortunately, maintaining consistency between replicas and caches can be challenging, and unless an application can accept weak consistency guarantees, replication and caching are generally only effective at reducing latency for read-only RPCs.

Speculative execution (SE) has been used to reduce latency in operating systems [6, 30, 31, 42, 43], byzantine fault tolerance protocols [22, 44], and a number of other specialized applications [9, 23, 27, 41]. These systems take advantage of domain knowledge to determine when the result of an expensive operation can be accurately predicted. This predicted result can be used to speculatively execute the dependent operations, allowing them to be executed concurrently as long as the prediction is eventually shown to be correct.

Although SE is a powerful technique for reducing latency, it can introduce a significant amount of complexity to an application. Currently, adding SE support to a new production-quality application requires a significant investment in time and resources. Therefore, previous work has only considered using SE for the most performance-critical applications, and SE has only been implemented in research prototypes.

Middleware '18, December 10–14, 2018, Rennes, France

In this paper, we introduce SpecRPC, a general RPC framework for performing SE. A user application can be modified to predict the result of an RPC, and use the SpecRPC framework to speculatively execute the next operation, which can be a local function or another RPC, based on the predicted result. The main goal of SpecRPC is to simplify the integration of speculative techniques in distributed applications, allowing for the pervasive use of speculation to reduce application latency. SpecRPC facilitates development of applications that leverage speculation, but it is up to developers to predict RPC results as accurate prediction requires domain-specific knowledge.

To support SE in an RPC framework, each operation in SpecRPC is given a state that characterizes its current status. Speculatively executing an operation based on the predicted result of a pending RPC causes the operation to enter a *speculative* state, and creates a dependency between the operation and the RPC. An operation in a *speculative* state will transitively cause subsequent dependent operations to also be in a *speculative* state to form a dependency tree where the branching factor depends on the number of predictions made per operation. The dependency tree helps SpecRPC isolate the different branches of SE, and determine which branches are still valid as actual execution results become available.

Dependency tracking and the re-execution of computation based on incorrect speculation are done automatically by SpecRPC. To take advantage of speculation, a programmer only needs to return speculative results when partial information is available. SpecRPC provides a simple abstraction that should be familiar to any programmer who has used RPCs and callbacks; and it reduces the adoption barrier for using speculation in distributed applications.

Overall, this paper makes three contributions:

- We present SpecRPC, a general, flexible and easy-to-use framework for performing SE.
- We show that SpecRPC can fully parallelize complex communication patterns involving multiple sequential RPC calls from the same client, or an RPC call chain where an RPC function calls another RPC.
- We evaluate the effectiveness of SpecRPC by using it to implement a low-latency distributed transaction commit protocol, Replicated Commit (RC) [26]. Our results show that SpecRPC reduces the transaction completion time of RC by 58% compared to the sequential execution of dependent operations.

2 DESIGN PATTERNS

SpecRPC is an asynchronous RPC framework for Java where remote calls return immediately to the caller. A dependent operation is specified as a callback function that executes after the RPC completes. A callback function implicitly accepts the RPC's return value as a function parameter. The callback can issue additional RPCs with callbacks, which allows clients to execute a sequence of dependent RPCs by specifying a chain of callback functions.

An asynchronous RPC framework offers additional opportunities to perform operations in parallel compared to a synchronous framework. For example, clients can continue execution on non-dependent operations after issuing an RPC request. More importantly, for the purpose of speculative execution (SE), sequential operations are completely specified by a callback function chain. Therefore, by requiring that each callback function is implemented as a method in a callback object that can only modify the object's data[1], SpecRPC can encapsulate speculative results inside a collection of callback objects. This programming model allows SpecRPC to ensure that speculative results are not revealed to the rest of the application, and parallel speculations are isolated from each other.

Figure 1 illustrates an application that uses SpecRPC to parallelize client-side and server-side computation, where the client predicts the server's computation result. The *Math* class in Figure 1 (a) is provided by the RPC server and exposes the *plus* method to remote callers. The server's *main* method specifies the necessary boilerplate code to register the *plus* method, allowing remote hosts to call this method. Instead of accepting a *Math* RPC object, the *register* method accepts a factory object, which is used by the framework to create a new *Math* object for each RPC request.

Inside the *main* method in Figure 1 (b), the client binds the remote *plus* method to an RPC stub, and issues an RPC by executing the stub's *call* method. The *call* method takes as parameters a list of predicted RPC return values and a callback factory. Upon receiving a response from the server, the client uses the callback factory to generate an instance of *IncCB*, and executes the callback with the return value from *plus* as a parameter. Using factories enables the framework to speculate multiple times with different predicted values, where each SE creates a different RPC or callback object. By specifying client-predicted return values, SpecRPC allows SE to begin even before the RPC has been sent to the server.

A SpecRPC call immediately returns a future object that eventually acquires the return value of the callback method from the final, non-speculative callback object. The caller can retrieve this value by calling *getResult* on the future object, which blocks until the return value is available. The framework ensures that the method returns a non-speculative result. In the example from Figure 1, the client will block until the future receives *4* from the callback.

2.1 Single-Level Speculation

In a traditional RPC framework, operations that depend on an RPC's return value must wait until the RPC completes. This is illustrated in Figure 2 (a) where the local operation can only begin after receiving the RPC return value. However, in some applications, clients can often predict RPC results. For example, an application may perform an RPC repeatedly with the same parameters and, in most cases, receive the same return value. The client can use a client-side cache to predict the RPC result. Using SpecRPC, the client can use this prediction to speculatively execute the dependent operations, specified as callback objects, immediately after invoking the RPC. As shown in Figure 2 (b), the execution times for the RPC and its dependent operations overlap. We call this client-side speculation, where the client predicts RPC results.

In addition to client-side speculation, SpecRPC also supports server-side speculation in which the server predicts the RPC's return value before it completes its execution of the RPC function. This is useful for RPC functions that execute slowly, but their results can be accurately predicted after a small amount of preliminary

[1] This is an advisory programming model for correctness rather than a mandatory design pattern. Applications can choose to modify data outside of callback objects if they are certain that it will not affect the correctness of the application.

```java
//RPC implementation
public class Math implements SpecRpcHost{
 public Integer plus(Integer a, Integer b){
  return a + b;
 }
 ... //Defines other RPCs
}
//RPC factory
public class MathFactory implements SpecRpcHostFactory{
 public SpecRpcHost getRpcHostObject(){
  return new Math();
 }
 public String getRpcHostClassName() {
  return Math.class.getName(); //Returns "Math"
 }
}
//Server implementation
public class Server {
 public static void main(String args[]){
  SpecRpcServer rpcServer = new SpecRpcServer();
  rpcServer.initServer("./server.config");
  //Registers an RPC with its name, factory,
  //return value type, and parameter types.
  rpcServer.register("plus", new MathFactory(),
    Integer.class, Integer.class, Integer.class);
  rpcServer.execute(); //Starts the RPC server
 }
}
```

(a) Server-side code.

```java
//Callback implementation
public class IncCB implements SpecRpcCallback{
 public Object run(Object rpcResult){
  return (Integer)rpcResult + 1;
 }
}
//Callback factory
public class CBFactory implements SpecRpcCallbackFactory{
 public SpecRpcCallback createCallback(){
  return new IncCB();
 }
}
//Client implementation
public class Client{
 public static void main(String args[]){
  SpecRpcClient.initClient("./client.config");
  //Binds an RPC with its class name, method name,
  //return value type, and parameter types
  RpcSignature plus = new RpcSignature("Math", "plus",
    Integer.class, Integer.class, Integer.class);
  SpecRpcClientStub stub =
    SpecRpcClient.bind("localhost", plus);
  List preds = Arrays.asList(3); //Predicts plus(1,2)
  SpecRpcFuture future =
    stub.call(preds, new CBFactory(), 1, 2); //Issues RPC
  future.getResult(); //Callback result is 4
 }
}
```

(b) Client-side code.

Figure 1: An example illustrating the server-side and client-side code for a simple SpecRPC application.

computation. In Figure 2 (c), the client must wait until it receives the server's prediction before it can speculatively execute its dependent operations. The server can return a prediction by calling *specReturn* any time during the RPC execution. Multiple predictions can be made by both the client and the server. Each prediction creates a new callback object that executes independently.

For both client-side and server-side speculation, the framework can determine if the return value prediction is correct after the client receives the RPC's actual return value. In the case where the prediction was correct, the result from the SE that is performed via callback objects can be returned to the application. Otherwise, the framework will dispose of the speculative results and re-execute the dependent operations using the actual RPC return value. We will explain how SpecRPC manages predictions and speculative results in Section 3.

2.2 Multi-Level Speculation

In order to provide significant performance benefits for workloads with long chains of sequential operations, a speculation framework must support having multiple dependent operations execute concurrently. This allows an application to perform SEs that depend on the correctness of multiple predictions. We call an SE that depends on more than one prediction a *multi-level speculation* (MLS).

Figure 3 shows a sample sequence of operations that could benefit from using MLS. In this example, an *analysis server* (AS) provides a data analytics service, a *data server* (DS) manages user data, and a client is interested in making a purchase based on both individual user information and aggregate information from a specific userbase. The client first retrieves the purchasing interests (PIs) of a specific user from AS by invoking an RPC, *getPI*. To compute the user's PIs, AS issues an RPC, *getPH*, to DS for the user's purchase history (PH). Once *getPI* completes, the client invokes another RPC, *getAI*, to AS to retrieve aggregate information (AI) from the userbase that shares the same PIs as the user. This AI is generated in real-time by AS. Finally, once *getAI* completes, the client performs additional local computation, *comp*, before ending its execution.

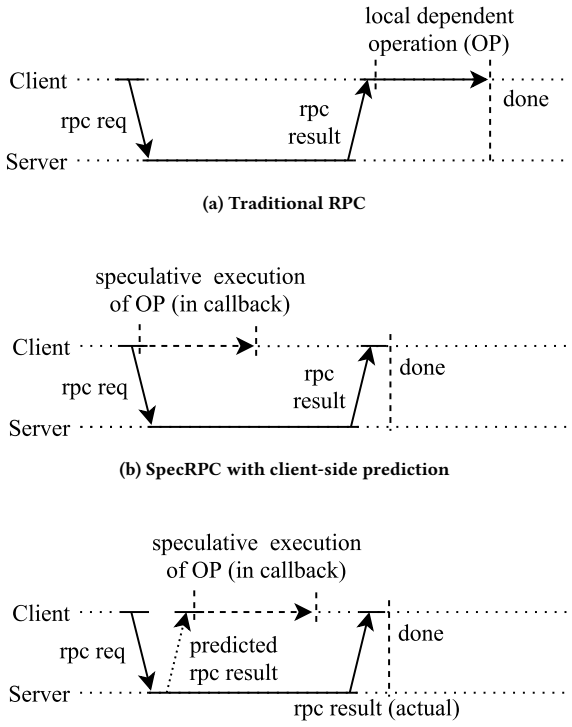
With speculation, parts of the above three RPCs and the client's local computation can execute in parallel. To service *getPH*, DS must retrieve the PH pertaining to the user specified in the request. Although the data may be available locally, additional synchronization delays may be introduced if DS is not the primary replica for the data and linearizable consistency is required. However, DS can send a speculative response using its local data to allow the caller to continue its execution without waiting for the synchronization to complete. Similarly, when servicing *getAI*, AS may be able to send a speculative response back to the client before it finishes generating the requested AI. The speculative response may be taken from the cached response of a previous request either for the same userbase, or for a related userbase with a similar PI.

Figure 3 (b) illustrates that, by predicating the result of *getPH*, AS can speculatively return the result of *getPI*. This will cause the subsequent operations to be speculatively executed in parallel. Figure 3 (b) also shows that, by speculatively executing *getAI* and predicting its result, *comp* can execute in parallel with both *getPH* and *getAI*. This is an MLS example because the SE of *comp* depends on more than one prediction.

The previous example demonstrates the need for a speculation framework to have each speculative RPC transitively depend on all of the predicted return values that its caller relies on. It also illustrates the challenge in tracking dependency information across RPCs.

3 ARCHITECTURE

The SpecRPC architecture consists of four layers, as shown in Figure 4, client and server libraries register functions for remote access,

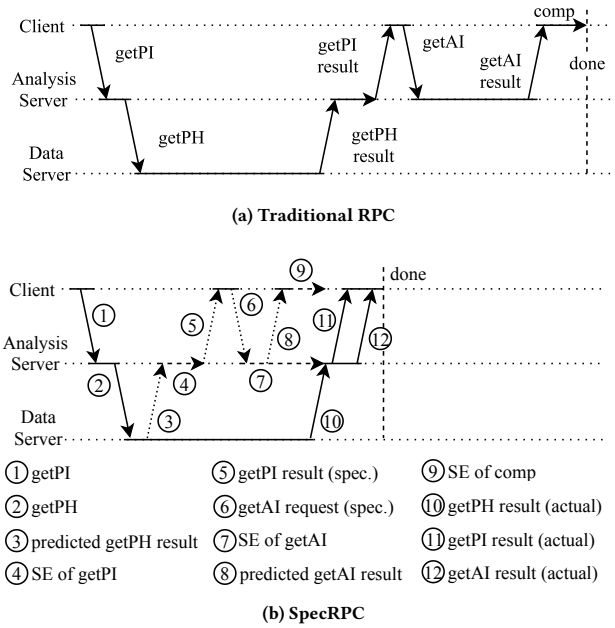


(a) Traditional RPC

(b) SpecRPC with client-side prediction

(c) SpecRPC with server-side prediction

Figure 2: An example of single-level speculation.



(a) Traditional RPC

(b) SpecRPC

Figure 3: An example of multi-level speculation.

expose the functions based on their signatures, and asynchronously issue remote RPCs. The SpecRPC controller manages speculative dependencies, uses callback and RPC factories to create new callback and RPC objects, and provides isolation between concurrent callbacks and RPCs. User-provided callback and RPC factories create new callback and RPC instances to handle RPC results and requests. Each callback or RPC instance has an object, *specObj*, which encapsulates the instance's speculative state. An RPC call inside a callback or RPC inherits the caller's speculative state through the *specObj*. The communication module manages connections between clients and server.

In the following sections, we describe how the different components work together to manage speculative dependencies across multiple nodes, handle incorrect predictions, and ensure that the final result is equivalent to what a traditional RPC framework would return.

3.1 Speculative State

In SpecRPC, computation is performed entirely within callback and RPC objects. A callback object is created when a client receives an RPC response, and an RPC object is created when a server receives an RPC request. A callback performs SE if it receives a predicted RPC response instead of an actual RPC response. If a callback issues an RPC request while it is performing SE, the RPC object created from the request will also speculatively execute its computation. When the actual RPC response arrives, the speculatively executed callback and its dependents will be discarded if the prediction was incorrect. Otherwise, they will be marked as actual execution.

To distinguish between actual and speculative execution, each callback and RPC object contains a *speculative state* that describes its speculation status and dependency information. This state is encapsulated as a *specObj*.

An RPC's speculative state can be one of the following: *caller speculative*, *speculation correct*, and *speculation incorrect*, where the last two states are terminal states. Figure 5(a) illustrates the state transitions for RPC objects. An RPC is in *speculation correct* state if the caller, which can be a client, an RPC object or a callback object, is not dependent on any predicted values. This is always the case when the caller is a client because a client's RPC request cannot be dependent on a predicted value. It is also the case when the caller is an RPC or callback object that is in *speculation correct* state. An RPC is in *caller speculative* state if its caller is dependent on a predicted value. This is equivalent to the caller being in a non-terminal state. Finally, an RPC transitions from the *caller speculative* state to the *speculation incorrect* state if its caller transitions to the *speculation incorrect* state.

Each callback is associated with an RPC and executes with the RPC's return value. Multiple callbacks can be associated with the same RPC because of multiple predictions for the return value. A callback can have one of the following speculative states: *caller speculative*, *callee speculative*, *speculation correct*, and *speculation incorrect*. Figure 5(b) illustrates the state transitions for callback objects. A callback is in *speculation correct* state if it receives a non-predicted return value from its RPC and the RPC is in *speculation correct* state. A callback is in *callee speculative* state if it executes with a predicted return value of its RPC. Upon receiving the actual return value, the callback transitions to the *speculation correct* or

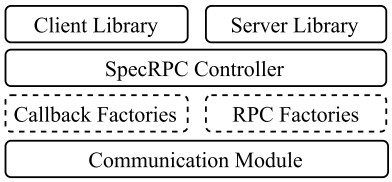


Figure 4: SpecRPC architecture.

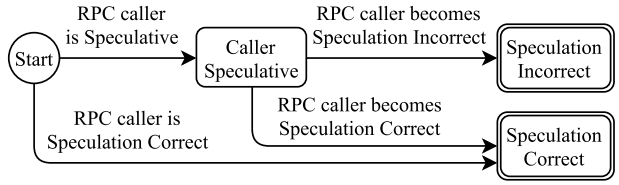
speculation incorrect state depending on the prediction, or transitions to *caller speculative* state if its RPC's caller is in a non-terminal state, i.e., either caller speculative or callee speculative. Finally, a callback in *caller speculative* state transitions to a terminal state once its RPC's caller transitions to a terminal state.

3.2 Managing Dependencies

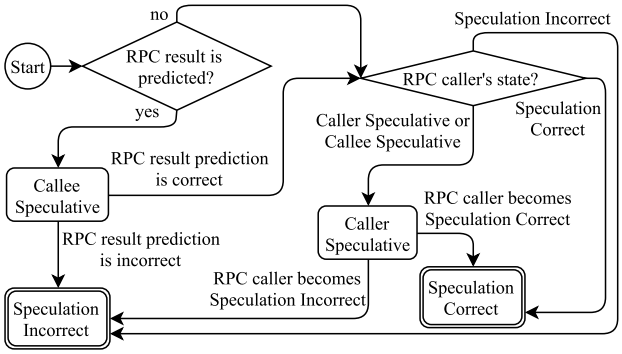
In single-level speculation, a speculative callback depends on the correctness of a single prediction. In order to keep track of dependencies, SpecRPC would only need to maintain a mapping between predictions and their corresponding speculative callbacks. Dependency management becomes more challenging in multi-level speculation as callbacks and RPCs can be dependent on multiple predictions. These dependencies are modeled as a tree with each RPC and callback object representing a node. The root in a dependency tree is the first RPC object that is issued by the user application. The root is always in the *speculation correct* state. A callback object created on a predicted return value is a child node of the RPC in the dependency tree.

A node's speculative state depends on that of its parent node (if any). Each node only needs to track its parent node's state transition and pass the state change of itself to its child nodes. SEs that are based on the same predicted return value of an RPC form a subtree under the RPC. Also, a path from the root to a leaf in the tree links dependent non-speculative and speculative execution. There is only one path standing for the actually non-speculative execution.

Figure 6 shows an example of a dependency tree in a bad scenario where predicted RPC responses are always incorrect. In this example, a client performs two dependent RPCs, rpc_1 and rpc_2, followed by a local operation that is executed in $callback_2$. For each RPC invocation, the client receives an incorrect prediction from the server. After receiving a prediction result for rpc_1, $callback_1$ is created to perform rpc_2. Therefore, rpc_2 is a child node of $callback_1$ which in turn is a child node of rpc_1. As rpc_2 executes, it also returns a predicted result which creates $callback_2$ to run the dependent local operation. In this example, rpc_2 finishes before rpc_1 even though it starts after rpc_1. When rpc_2 finishes, it returns an actual result that is different than its previous predicted result. Therefore, a new callback object $callback'_2$ is created and $callback_2$ is abandoned. Later, when rpc_1 finishes with an actual result that is different than its previous predicted result, the whole subtree of $callback_1$ is abandoned and a new $callback'_1$ is created to invoke rpc'_2. The predicted result of rpc'_2 creates $callback''_2$, which again will be abandoned when actual result of rpc'_2 is different from its predicted result. Finally, with the completion of rpc'_2, $callback'''_2$ is created to finish the remaining execution. Even with three mispredictions, the client only sees the actual execution path from rpc_1 to $callback'''_2$.



(a) State transitions of an RPC object.



(b) State transitions of a callback object.

Figure 5: State transitions.

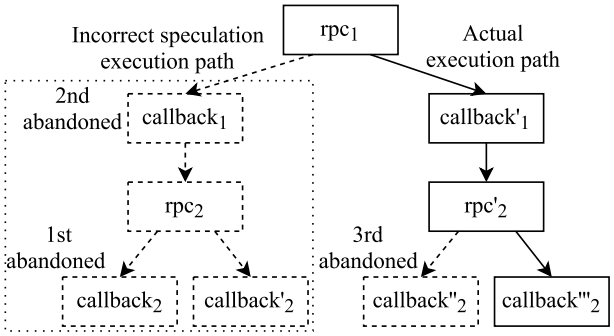


Figure 6: An example of a dependency tree.

In both single and multi-level speculations, SE includes local operations and RPCs that will be executed remotely. As a result, the dependencies between predictions and SEs span multiple nodes. SpecRPC uses dedicated state-change messages to propagate state change events between remote callback and RPC objects. This is discussed in Section 3.4.

3.3 Handling Incorrect Predictions

Upon receiving the actual result for an RPC, SpecRPC evaluates the accuracy of previous predictions of that result, and retains callbacks based on correct predictions while setting callbacks based on incorrect predictions to the *speculation incorrect* (SI) state. Any callback or RPC object that depends on an object in SI state must also be set to SI state. Objects in SI state are discarded, and their

computations are abandoned. This can be done safely without requiring data rollback since a callback or RPC in SpecRPC should only modify the fields in its associated object and should not have any side-effects. This advisory requirement is not enforced as it is up to application to decide the scope of side-effects caused by modifying data outside of a callback or RPC object.

Immediately terminating a discarded callback or RPC may require interrupting its computation, which can be difficult to perform cleanly in a language without non-local exception passing where a thread can raise an exception in a different running thread. To avoid this problem, SpecRPC allows callbacks and RPCs in SI state to finish execution before being abandoned. However, SpecRPC immediately terminates these callbacks and RPCs if they attempt to perform further speculative operations via SpecRPC, such as invoking a new RPC, returning a prediction to the client, or blocking on an operation that will generate visible output (see Section 3.5).

In the case where none of the previous predictions of an RPC result were correct, a new callback is created using the RPC's actual result. This ensures that forward progress is made even in the absence of an accurate predictor.

3.4 Propagation of Speculative State

The speculative state of a callback or RPC object depends on the state of the caller. Therefore, when a callback or RPC issues an RPC request, the caller's speculative state is sent alongside the RPC's parameters. Similarly, each RPC response contains the speculative state of the RPC and a field that specifies whether it is returning a predicted result or an actual result. The framework uses this information to create a callback object in the correct speculative state.

In multi-level speculation, when the caller of an RPC transitions to a *speculation correct* or *speculation incorrect* state from a non-terminal speculative state, both the state of the remote RPC instance and the corresponding local callback must be updated. To notify the RPC instance on the remote node, the caller sends a dedicated message indicating its new speculative state. Both the RPC and callback perform their own speculative state change based on the caller's new speculative state. This change is further propagated if another RPC is invoked by either the callback or RPC.

3.5 Implementation

The SpecRPC framework consists of approximately 3000 lines of Java code. The source code is available online at [37]. In SpecRPC, RPCs are registered by servers as signatures containing an RPC name, a return type, parameters and a server address. RPC signatures are stored in a file that is synchronized between the servers and clients using third-party tools, such as ZooKeeper [18].

3.5.1 Tracking Dependencies. SpecRPC tracks the dependencies between speculative and non-speculative executions by mapping a callback or RPC object to its parent node in a dependency tree. Instead of implementing the dependency tree as a centralized data structure, each node only tracks its child nodes. When a node's speculative state changes, SpecRPC propagates the changes only to its child nodes.

Applications using SpecRPC do not need to explicitly track speculation-related dependencies or inform the framework of what it depends on. When a prediction is incorrect, SpecRPC will discard all of the speculative callback and RPC objects that depend on the prediction.

3.5.2 Preventing Side-Effects. SE should not result in output or state changes that are irrevocable. Therefore, SpecRPC recommends that callbacks and RPCs only modify the fields in their objects, and not have any side-effects. SpecRPC's factory design pattern creates a new object when it executes a callback or RPC, and it stores speculative states inside that object. This isolates parallel SEs, which allows an application to make multiple predictions.

An application can optionally install a *rollback* function for mis-speculation in a callback or RPC. The SpecRPC framework will execute the rollback function before discarding incorrect states. This enables an application to extend its speculative states beyond the fields inside a callback or RPC object. For example, an application can store speculative states in a local database and issue a rollback for a mis-speculation.

In scenarios where it is impossible to avoid irrevocable changes or output in SE, the SpecRPC framework provides *specBlock*, a method that causes a speculative callback or RPC to block until it is in a non-speculative state. An application can call *specBlock* just before operations that will cause side-effects. Once the speculation is determined to be correct, SpecRPC will continue the application's execution. If the speculation is incorrect, the *specBlock* function will throw a mis-speculation exception.

4 APPLICATIONS

In this section, we describe how we used SpecRPC to implement a speculation-enabled version of Replicated Commit [26], a distributed transaction commit protocol for geo-replicated database systems. Also, we perform a theoretical analysis on the expected speedup from using speculative execution (SE) on another application, a multi-objective optimizer.

4.1 Replicated Commit

Replicated Commit (RC) [26] is a distributed transaction commit protocol for geo-replicated database systems. In a geo-replicated system, a transaction's completion time largely depends on the number of wide-area network roundtrips that the transaction requires to complete.

RC introduces a commit protocol that only requires one wide-area network roundtrip to complete both 2PC and consensus among replicas across datacentres. However, to achieve this, local read operations have to be replaced with quorum reads across multiple datacentres. Each quorum read introduces one wide-area network roundtrip. Writes are not affected as they are buffered until the transaction commits. RC's evaluation shows that, as the number of dependent reads increases, the transaction completion time is quickly dominated by read latency.

SE can parallelize the execution of dependent reads in RC in order to reduce the overall completion time of a transaction. This is possible because the read results from the first responding quorum member are often the same as the final quorum results. Therefore, we can use the first response to speculatively execute the next read operation. In RC, because data is fully replicated in every deployed datacentre, the first responding member will always be from the

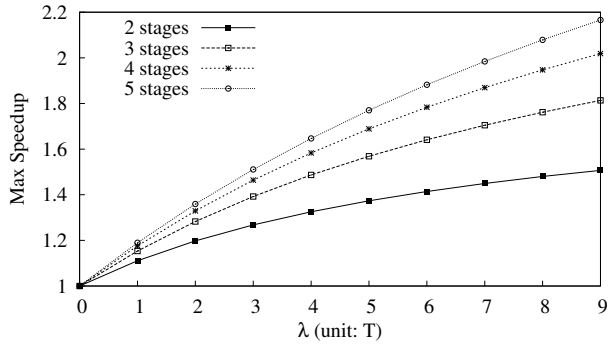


Figure 7: Maximum speedup versus λ.

local datacentre, and its response will return almost immediately. Note that we only perform SE for quorum reads before starting the commit protocol. Before calling commit on a transaction, an RC client will issue a specBlock (see Section 3.5) to wait until all quorum reads become non-speculative. We have implemented a fully working prototype of the SpecRPC version of RC, and we evaluate its performance in Section 5.

4.2 Multi-Objective Optimizer

Many scientific computing problems require solving optimization problems (OPs) with multiple objectives. A common approach for solving multi-objective problems is to construct them as a series of dependent OPs in which the output of one OP serves as an input to the next OP. Each of these intermediate OPs can be solved using an optimizer such as CPLEX [8] and Gurobi [16]. Therefore, the completion time is largely dependent on the performance of the optimizer, and the number of OPs in the series. Although most optimizers benefit from additional CPUs, their scalability is limited. Most do not achieve additional speedup beyond 16 processors [21].

SE can leverage additional CPUs to reduce computation time of dependent OPs by overlapping their computation. With SpecRPC, each OP can be registered as an RPC function, and the OPs can be deployed on a group of server nodes. To execute an OP, a client node issues an RPC to a server. Before the optimization is complete, the server can return its current best solution to the client. The client can use this result to issue an RPC to another server in order to speculatively execute the next OP. The current best solution serves as a prediction for the final result from the optimizer. If the prediction is correct, where correctness is based on the user's equivalence definition, there will be a reduction in the total completion time. The correctness probability depends on the amount of time the function was allowed to run before the current best solution was retrieved. This is because the more time the optimizer is given to run, the more likely that it has found the optimal result.

Assuming that there are n dependent OPs (stages), we define S_{lat} as the speedup of using SE to complete the n stages compared to sequentially executing them with the same total number of CPUs. We also define T_{new} and T_{old} as the expected completion time with and without SE, respectively. Therefore, we have $S_{lat} = \frac{T_{old}}{T_{new}}$.

In this analysis, we assume there are $n * N$ total CPUs. We define the amount of time it takes for stage i's optimizer to complete as: $T_i = g_i(m)$, where m is the number of CPUs, and g_i is a monotonically increasing function of m. When m is above a threshold, the increase of T_i is negligible. We also define the prediction correctness percentage at stage i as: $P_i = f_i(t_i)$, where t_i is the amount of time that stage i's optimizer executes before the best current solution was retrieved. We denote $E_{i,j}$ as the expected completion time of executing all stages from i to j. $E_{i,n}$ can be recursively calculated as follows:

$$\begin{cases} E_{n,n} = T_n \\ E_{i,n} = P_i * (t_i + E_{i+1,n}) + (1 - P_i) * (T_i + E_{i+1,n}) \end{cases} \quad (1)$$

where $1 \le i < n$. The last stage's completion time is always T_n, and no prediction occurs at this stage.

By solving the recursion in Equation (1), we have T_{new}:

$$T_{new} = E_{1,n} = \sum_{i=1}^{n-1} [P_i * (t_i - T_i) + T_i] + T_n \quad (2)$$

where $0 \le t_i \le T_i$, and $T_i = g_i(N)$ since, with SE, it is possible that all n stages run in parallel, so each stage can only use N CPUs.

Without SE, each stage can use the total $n * N$ CPUs, so we have T_{old}:

$$T_{old} = \sum_{j=1}^{n} T_j = \sum_{j=1}^{n} g_j(n * N) \quad (3)$$

With a fixed N, we can determine the set of t_is that maximize the total speedup, S_{lat}.

We illustrate our model with a two-stage example where the stages have the same completion time T. In this example, there are enough CPUs such that using N and $2N$ CPUs at each stage will achieve the same completion time, i.e., $|g(N) - g(2N)| < \epsilon$, where ϵ is negligible. We also assume that the prediction correctness percentage at the first stage can be described as a cumulative distribution, $P = 1 - \exp^{(-\lambda t)}$, where λ is a constant. This is because the convergence rates of many multi-objective optimizations have been shown to follow an exponential function over computation time [1, 3, 35]. From Equations (2) and (3), we have S_{lat}:

$$S_{lat} = \frac{2T}{(1 - \exp^{(-\lambda t)}) * (t - T) + 2T} \quad (4)$$

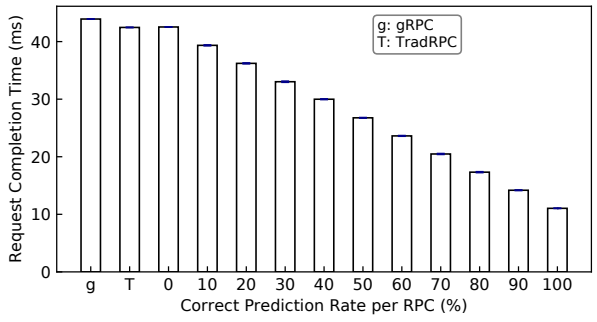
where $0 \le t \le T$. The goal is to find t_0 to maximize the speedup, S_{lat}. This is equivalent to solving:

$$1 + \exp^{(-\lambda t_0)} * (\lambda(t_0 - T) - 1) = 0, \quad 0 \le t_0 \le T \quad (5)$$

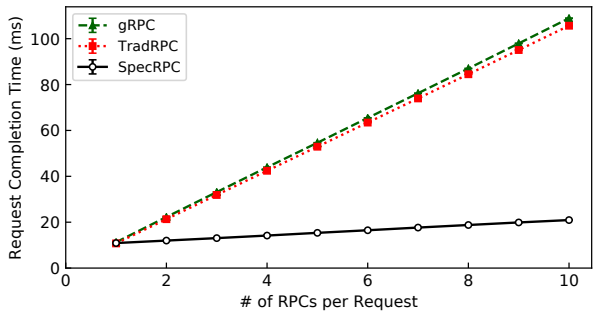
We have further generalized the previous example to support more than two stages. Figure 7 illustrates the relationship between the maximum S_{lat} and λ for different number of stages. It shows that the maximum S_{lat} increases with an increase in the prediction rate. This is not surprising since a higher prediction rate results in fewer re-executions of the stages. The figure also shows that for a given prediction rate (i.e., a fixed value of λ), the maximum speedup increases with more stages.

5 EVALUATION

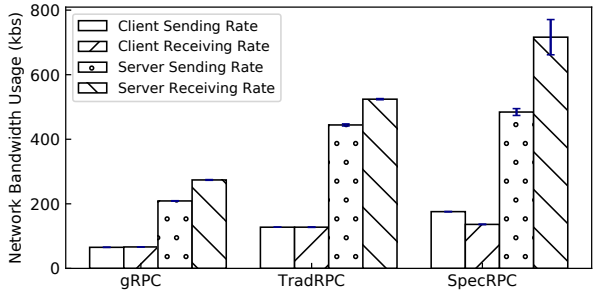
In this section, we first use a microbenchmark to evaluate the performance of SpecRPC, and then we examine the performance improvements in Replicated Commit (RC) [26] when using SpecRPC. For comparison we use Google's open-source RPC framework,



(a) Latency versus correct prediction rate



(b) Latency versus # of RPCs per request



(c) Network bandwidth usage

Figure 8: Microbenchmark results.

gRPC [14], as a baseline. Since gRPC has more features than SpecRPC, which may increase its latency, we also compare our system with *TradRPC*, an RPC framework sharing much of SpecRPC's code base without speculation.

Our experimental testbed uses standard server-class machines, each of which has two 6-core 2.10 GHz Intel Xeon E5-2620 v2 CPUs and 64 GB RAM. The machines are connected to a 1 Gbps Ethernet network. Each experiment consists of 5 runs, and each run lasts 60 seconds during which we measure the performance of the system throughout the middle 30 seconds.

5.1 Microbenchmark

Our microbenchmark consists of 16 clients with each performing a sequence of dependent RPCs, defined as a *request*, to multiple servers. Each RPC sends and receives 64 bytes of data. Unless specified, a request consists of 4 RPCs, each of which requires 10 ms to complete. The client issues one request at a time, and each client issues 10 requests per second. This system load allows us to examine the performance of SpecRPC when sufficient system resources are available.

While executing a request by using SpecRPC, the client makes a prediction for each RPC result. We define the probability of the prediction being correct as the *correct prediction rate per RPC*. Figure 8 (a) shows the mean completion time of requests under different correct prediction rates per RPC. Compared with the sequential execution of RPCs via using gRPC and TradRPC, SpecRPC achieves up to 75% reduction in request completion time. When prediction is always incorrect, SpecRPC introduces approximately 0.1 ms of overhead compared with TradRPC, which is negligible in a request that requires more than 40 ms to complete. Our experiments show that gRPC has slightly higher overhead than both TradRPC and SpecRPC. This may be because gRPC provides additional features that are not supported by TradRPC and SpecRPC. The results also show that even with only a 50% correct prediction rate per RPC, SpecRPC still provides about a 40% reduction in request completion time compared to gRPC.

We further examine the performance of SpecRPC by varying the number of dependent RPCs in a request. In the following experiments, the correct prediction rate per RPC is set to be 90%. Figure 8 (b) shows that the mean request completion time for SpecRPC increases more slowly than for gRPC and TradRPC. As expected, the request completion times for both gRPC and TradRPC increase linearly with the number of dependent RPCs per request. SpecRPC experiences a small increase in its request completion time with additional dependent RPCs because only incorrect predictions lead to sequential execution of RPCs.

Lastly, we examine the network bandwidth usage of the three different RPC frameworks. Figure 8 (c) shows that TradRPC has higher network bandwidth usage than gRPC. This is because gRPC has a more optimized implementation of message serialization than TradRPC. The results also show that SpecRPC has higher network bandwidth usage than TradRPC. This is because SpecRPC must re-execute some of its RPCs due to incorrect predictions.

5.2 Replicated Commit

In this section, we examine the performance improvements in Replicated Commit (RC) when using SpecRPC. We implement an RC prototype in an in-memory key-value store, and our implementation asynchronously persists transaction logs to SSDs. We compare three versions of RC, one using gRPC [14] as a baseline, one using SpecRPC to enable speculative execution, and one using TradRPC which shares much of SpecRPC's code base without speculation. Our RC prototype using gRPC consists of approximate 4000 lines of Java code, while SpecRPC introduces an additional 100 lines of changes on the server side and about 300 lines of changes on the client side. These changes are to modify RPC registrations and

	Ireland	Seoul
Oregon	140	122
Ireland	-	243

Table 1: RTT latencies (ms) between datacentres from [28].

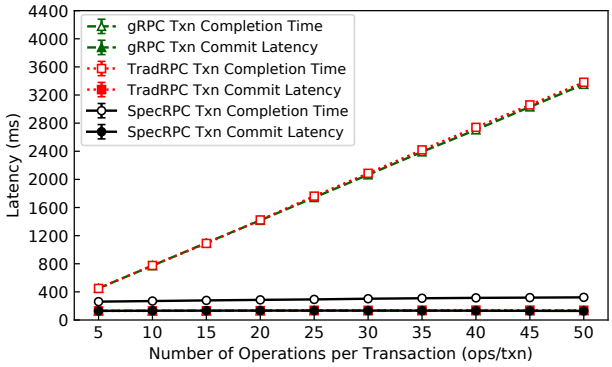
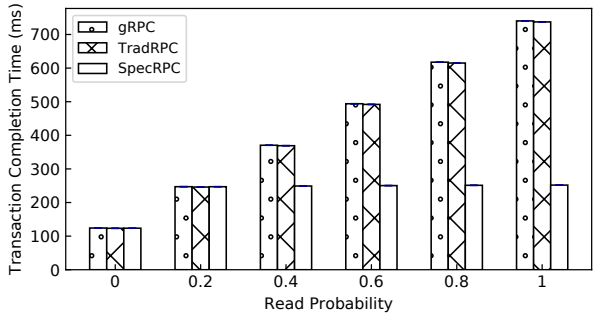
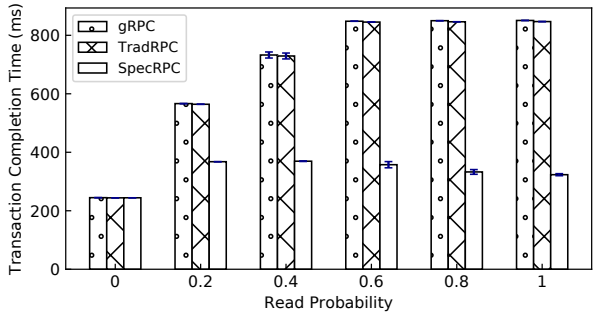


Figure 9: Mean latency versus the number of operations per transaction with YCSB+T.



(a) Median



(b) 99th percentile

Figure 10: Latency versus read probability with YCSB+T.

invocations in order to introduce speculation on read results. Our SpecRPC changes do not modify the commit protocol.

5.2.1 Experimental Setup. In our experiments, we use the Linux traffic control utility (tc) to set the network latency between machines in order to emulate the geo-distributed environment specified in [28], which consists of three datacentres. The roundtrip network latencies between datacentres are shown in Table 1.

Our transactional key-value store contains 10 million key-value pairs. The data is sharded into three partitions, with each partition having a replica at every datacentre. One RC server manages one replica. Our evaluation uses close-loop experiments, in which a client sends transactions back-to-back, and there are 16 clients in each datacentre.

Our evaluation uses two workloads, YCSB+T [11] (an extension of the YCSB workload [7] with transactional support) and Retwis [24] (a Twitter-like workload). By default, the data access pattern in both workloads follows a Zipfian distribution with *alpha = 0.75*.

5.2.2 YCSB+T Workload. We first use YCSB+T to repeat the RC experiments in [26]. Without using speculation, read latency dominates the transaction completion time as the number of reads increases. In this experiment, the number of operations (reads and writes) varies from 5 to 50, and the ratio of reads and writes is 1:1 (mirroring the values used in the original experiments in [26]). Figure 9 shows that the average transaction completion time of our RC prototype with gRPC and TradRPC increases linearly with the number of reads, which matches the results in [26]. In contrast, the average transaction completion time of the SpecRPC version of RC is nearly independent of the number of reads in a transaction. Going from 5 to 50 operations per transaction, the transaction completion time only increases by 23% for SpecRPC, compared to more than 600% for the non-speculative systems. This low increase in completion time is a result of correct speculation, which allows SpecRPC to parallelize dependent read operations. This experiment also shows that the read result from the first responding replica is a good predictor of the final result of a quorum read. This approach correctly predicted the final quorum read result with more than 95% accuracy.

We further examine the impact of the probability that a request in a transaction is a read (instead of a write) on transaction completion time. In this experiment we use 5 operations per transaction. Figure 10 shows both the median and 99th percentile of the transaction completion time. As expected, with gRPC and TradRPC, the median transaction completion time grows linearly with the read probability. The tail transaction completion time grows even more quickly, as even with a 0.2 probability, the transactions in the tail consist mostly of read operations. At 0.6 probability and higher, nearly all of the transactions in the tail consist of 5 read operations. With SpecRPC, the median and tail completion times are largely unaffected by read probability. This is because the correct prediction rate for this workload is above 99%, with the rate growing with increasing read probability.

5.2.3 Retwis Workload. In this section, we use a Twitter-like workload, Retwis, to evaluate the performance of RC using SpecRPC. We use the same transaction profile as the Retwis workload in [46], which is shown in Table 2. Figure 11 shows the CDF of RC's transaction completion time when using gRPC, TradRPC, and SpecRPC.

Transaction Type	# gets	# puts	workload%
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1, 10)	0	50%

Table 2: Retwis transaction profile from [46]

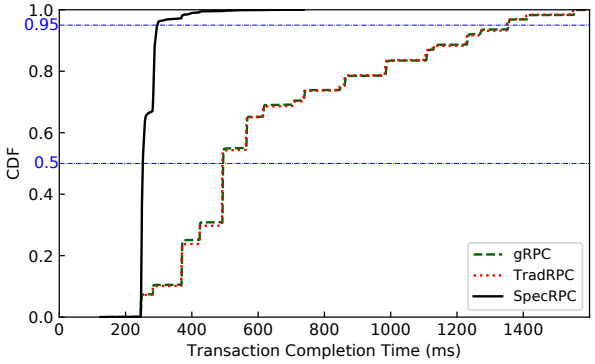


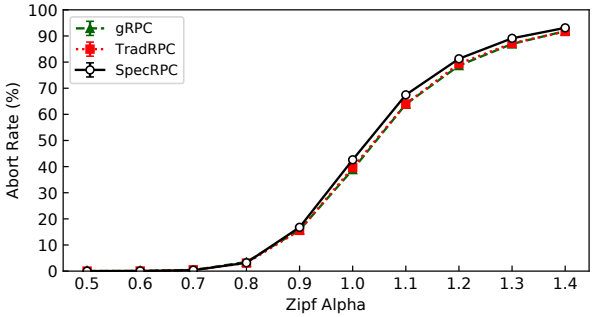
Figure 11: Transaction completion time with Retwis.

Compared with gRPC and TradRPC, SpecRPC reduces the average transaction completion time by 58%.

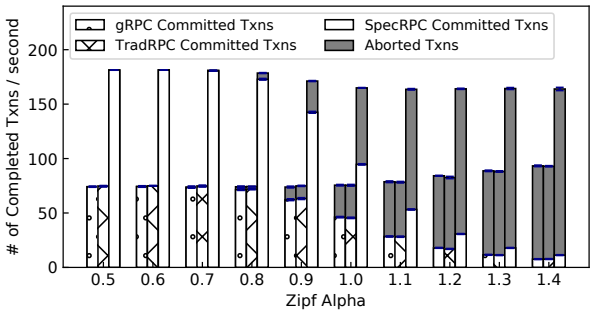
We then adjust the Zipfian alpha value to examine the impact of transaction contention on the performance of these systems. Figure 12 (a) and (b) show that SpecRPC's abort rate is only marginally higher than gRPC and TradRPC despite processing twice the number of transactions per second in this closed loop experiment with a fixed number of clients. For example, when the Zipfian alpha value is 0.9, using SpecRPC introduces about a 1% higher abort rate than using gRPC and TradRPC. However, SpecRPC is able to commit 142 transactions per second, while gRPC and TradRPC can only commit 62 and 63 transactions per second, respectively. The higher transaction processing rate of SpecRPC is due to its transaction completion time being half of that of the other systems.

In order to measure the maximum throughput of these systems, we must saturate them with client requests. To accomplish this in our cluster, we have to reduce the computing resources of the RC servers. For these experiments, we set the roundtrip network latency between datacentres to be 5 ms and limit the number of CPU cores per RC server. Since the number of CPU cores is artificially limited, these experimental results do not represent the maximum throughput of the systems in practice. Instead, these experiments aim to compare the throughput of the three systems under the same resource limit and to examine the impact of increasing computing resources on the performance of the three different systems.

As shown in Figure 13, all three systems have near perfect speedup in throughput when increasing the number of cores from 2 to 3, where the throughput is indicated by the vertical lines in the graphs. As expected, SpecRPC's throughput is lower than TradRPC's due to speculation overhead. Surprisingly, gRPC has a lower throughput than both other systems, which may be due to additional features that it provides which are not supported by SpecRPC and



(a) Abort rate



(b) Number of completed transactions

Figure 12: Retwis workload with varying alpha values.

TradRPC. Although SpecRPC introduces processing overhead, we believe that this is a reasonable tradeoff since throughput can be increased by increasing the number of cores or data shards, whereas transaction completion cannot be reduced with more resources without speculation. In this experiment, it is not possible for gRPC or TradRPC to achieve SpecRPC's 14 ms transaction completion time.

6 RELATED WORK

Speculation is a latency-hiding technique that enables parallel execution of dependent operations. This is especially useful in cases of otherwise unavoidable or lengthy wait times. This section compares SpecRPC to previous systems that implement speculation or other latency-hiding techniques.

6.1 OS-Level Speculation

Speculator [30] modifies the Linux kernel to provide OS-level SE support. It can reduce the latency of a distributed file system by allowing applications that would normally block during a remote I/O operation to continue execution using a predicted result. Speculator can also reduce the latency of synchronous local file I/O operations [31]. Because speculation is provided as part of the OS, Speculator must checkpoint the state of the entire process in order to undo changes when SE occurs on an incorrect prediction.

OS-level SE has also been proposed to reduce the I/O latency for microsecond devices [42]. However, its effectiveness is limited

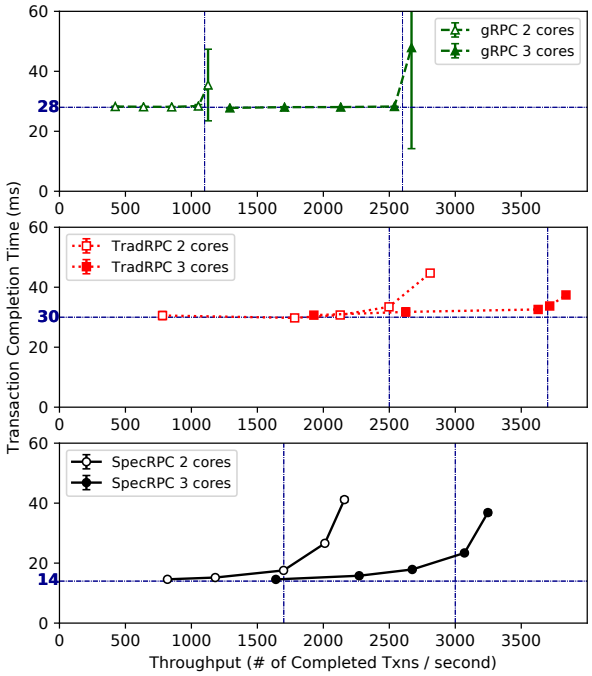


Figure 13: Average transaction completion time versus throughput with Retwis.

by the narrow system call interface and the lack of application semantics [43]. Comparatively, SpecRPC supports SE across multiple systems, and it provides a high-level framework that reduces the barrier to using speculation in large-scale distributed applications.

6.2 Application-Level Speculation

Supporting SE at the application level can leverage application semantics to provide additional latency reduction compared to OS-level approaches. Fast Track [20] and Prospect [38] allow users to implement fast but unsafe code for SE. Wester et al. [43] extend Speculator to enable developers to define customized speculation policies. Thread-level speculation techniques [4, 10, 19, 32, 33] can increase the parallelism of a sequential program to reduce the total execution time. Most of these speculation frameworks only support SE on a single machine. In contrast, SpecRPC provides application-level SE support for distributed applications.

SE has been used to improve the performance of Byzantine fault tolerance systems. The scope of the speculation techniques used in these systems is fairly limited, as each system aims to address a specific performance problem without introducing significant complexity to an already complex system. For example, Zyzzyva [22] only performs speculative execution on a single machine, and PBFT-CS [44] does not support propagating speculative results between servers.

Many other applications also have their own SE support for workload-tailored benefits. SpecHint [6, 13] uses SE to hide hard-disk latency in local file systems. Crom [27] speculatively prefetches web page data and computes web browser layouts. Lange [23] uses speculation to improve remote screen display protocols. Moreover, SE is used to improve the performance of logging and replay systems [25, 41], to protect software from hardware failures [9], and to detect run-time race conditions [40] and inconsistency in distributed systems [45].

These application-level approaches differ from SpecRPC by focusing on improving a specific application or protocol. SpecRPC is a general framework for providing SE support for distributed applications. Furthermore, most of these speculation systems were designed from scratch and required significant effort to implement and verify correctness. One of SpecRPC's goals is to reduce the barrier for using speculation in distributed applications.

Correctables [15] provides clients with server-side predictions for data access in replicated-object storage systems. However, a Correctables client has to implement its own dependency tracking between speculative and non-speculative executions. In contrast, SpecRPC is a more general framework that allows applications to perform arbitrary SE on both clients and servers.

6.3 RPC Batching

RPC batching can reduce the latency of a client performing dependent RPCs to a server. For example, BRMI [39] allows a client to explicitly send multiple RPCs in a batch to a server. Batched futures [5] implicitly defer a client's RPCs until the results are requested. However, RPC batching techniques cannot batch RPCs to different servers or use speculation to execute dependent operations in parallel.

7 DISCUSSION

A common pitfall to SE is that stateful execution is not correctly undone after an incorrect speculation. We address this pitfall by recommending or requiring SpecRPC users to follow design patterns that avoid this problem. For example, instead of passing a speculative object directly to a speculative callback, SpecRPC requires users to follow a factory design pattern in which a new speculative object is created for each callback that encapsulates all intermediate results. This allows programmers to not worry about cross-contamination between results from different speculative and non-speculative executions.

Another potential issue with speculation is in supporting operations with side-effects that is outside of the control of the speculation framework. We provide a *specBlock* function that prevents the computation from proceeding until it is in a non-speculative state. However, because SpecRPC is a user library, a decision that we made to simplify and promote adoption, it cannot enforce correctness in much the same way that a threading library cannot enforce thread safety.

SpecRPC provides the necessary tools to enable a developer to implement a speculative application without manually managing dependencies, discarding incorrect speculations, and hiding speculative state from non-speculative execution, which allows developers to focus on leveraging their domain knowledge to improve prediction rather than spending their time implementing a speculative execution infrastructure. However, the developer must still make reasonable decisions with respect to predictions, and must follow provided guidelines to avoid potential problems.

In addition to the applications described in Section 4, many other latency-sensitive applications can benefit from SpecRPC. For example, web applications often execute a chain of services to generate a response for a client request. These applications can use caches to predict service results, enabling services in the chain to execute in parallel. Social network applications can also benefit from speculation as they often perform multiple dependent graph computations. Many of these graph computations, such as triangle counting, are expensive but their results can be estimated quickly using approximation algorithms [17]. These estimates can be used as predictions. A social network application can perform many of its dependent graph computations in parallel if the predictions are correct or within some error bound.

8 CONCLUSION

In this paper, we introduce SpecRPC, a general RPC framework for performing speculative execution. By managing dependencies between callbacks and RPCs, SpecRPC simplifies the process of using speculation and reduces application latency. We evaluate SpecRPC by implementing a distributed transaction protocol using the framework. Our experimental results show that SpecRPC can significantly reduce transaction completion time by using speculation to perform dependent reads in parallel.

We designed SpecRPC with the hope that it would be used by many applications. As a result, some of the core designs of the framework were chosen to simplify adoption for developers and allow deployment in various environments. For example, instead of requiring operating system support for state rollback after an incorrect speculation, we perform state rollback completely within the framework. These design choices necessitate an advisory programming model where correctness relies on applications following our suggested design pattern. Although we believe this is acceptable for most applications, we are exploring other designs, such as introducing language-level changes, that can provide stronger guarantees. We plan to work closely with developers interested in our recently open-sourced implementation, and use their feedback to improve future versions of our framework.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Patrick Eugster, and the anonymous reviewers for their valuable feedback. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). We also wish to thank the Canada Foundation for Innovation and the Ontario Research Fund for funding the purchase of equipment used for this research.

REFERENCES

[1] Aristeidis Antonakis, Theoklis Nikolaidis, and Pericles Pilidis. 2017. Multi-objective climb path optimization for aircraft/engine integration using Particle Swarm Optimization. *Applied Sciences* 7, 5 (2017).

[2] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. 2007. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *Proceedings of the SIGCOMM Conference (SIGCOMM'07)*.

[3] Kalyan Shankar Bhattacharjee, Hemant Kumar Singh, and Tapabrata Ray. 2016. Multi-Objective Optimization With Multiple Spatially Distributed Surrogates. *ASME Journal of Mechanical Design* 138, 9 (2016).

[4] Anasua Bhowmik and Manoj Franklin. 2002. A General Compiler Framework for Speculative Multithreading. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*.

[5] Phillip Bogle and Barbara Liskov. 1994. Reducing Cross Domain Call Overhead Using Batched Futures. In *Proceedings of the Annual Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA'94)*.

[6] Fay Chang and Garth A. Gibson. 1999. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'10)*.

[8] CPLEX. 2017. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[9] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*.

[10] Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2001. The R-LRPD test: speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*.

[11] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proceedings of the VLDB Endowment* 4, 8 (2011).

[12] Nathan Farrington and Alexey Andreyev. 2013. Facebook's Data Center Network Architecture. In *Proceedings of the IEEE Optical Interconnects Conference*.

[13] Keir Fraser and Fay Chang. 2003. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proceedings of the USENIX Annual Technical Conference (ATC'03)*.

[14] Google. 2017. gRPC-go. https://github.com/grpc/grpc-go.

[15] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[16] Gurobi Optimization. 2017. http://www.gurobi.com/.

[17] Mohammad Al Hasan and Vachik S. Dave. 2018. Triangle counting in large networks: a review. *WIREs Data Mining and Knowledge Discovery* 8 (2018).

[18] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'10)*.

[19] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative Separation for Privatization and Reductions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*.

[20] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. 2009. Fast Track: A Software System for Speculative Program Optimization. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'09)*.

[21] Thorsten Koch, Ted Ralphs, and Yuji Shinano. 2012. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research* 76, 1 (2012).

[22] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*.

[23] John R. Lange, Peter A. Dinda, and Samuel Rossoff. 2008. Experiences with Client-based Speculative Remote Display. In *Proceedings of the USENIX Annual Technical Conference (ATC'08)*.

[24] Costin Leau. 2013. Spring Data Redis - Retwis-J. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[25] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*.

[26] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proceedings of the VLDB Endowment* 6, 9 (2013).

[27] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. 2010. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*.

[28] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[29] A. Natarajan, Peng Ning, Yao Liu, S. Jajodia, and S. E. Hutchinson. 2012. NSDMiner: Automated discovery of Network Service Dependencies. In *Proceedings of the IEEE INFOCOM Conference (INFOCOM'12)*.

[30] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2005. Speculative Execution in a Distributed File System. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'05)*.

[31] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the USENIX Symposium on Operating*

Systems Design and Implementation (OSDI'06).

[32] Manohar K. Prabhu and Kunle Olukotun. 2005. Exposing Speculative Thread Parallelism in SPEC2000. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05).*

[33] Lawrence Rauchwerger and David A. Padua. 1999. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999).

[34] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. 2011. It's Time for Low Latency. In *Proceedings of the USENIX Conference on Hot Topics in Operating Systems (HotOS'11).*

[35] Bryan Van Scoy Scoy, Randy A. Freeman, and Kevin M. Lynch. 2018. The Fastest Known Globally Convergent First-Order Method for Minimizing Strongly Convex Functions. *IEEE Control Systems Letters* 2, 1 (2018).

[36] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. 2009. RPC Chains: Efficient Client-server Communication in Geodistributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'09).*

[37] SpecRPC. 2018. *https://github.com/xnyan/specrpc.*

[38] Martin Susskraut, Thomas Knauth, Stefan Weigert, Ute Schiffel, Martin Meinhold, and Christof Fetzer. 2010. Prospect: A Compiler Framework for Speculative Parallelization. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10).*

[39] Eli Tilevich, William R. Cook, and Yang Jiao. 2009. Explicit Batching for Distributed Objects. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'09).*

[40] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and Surviving Data Races Using Complementary Schedules. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'11).*

[41] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).*

[42] Michael Wei, Matias Bjørling, Philippe Bonnet, and Steven Swanson. 2014. I/O Speculation for the Microsecond Era. In *Proceedings of the USENIX Annual Technical Conference (ATC'14).*

[43] Benjamin Wester, Peter M. Chen, and Jason Flinn. 2011. Operating System Support for Application-specific Speculation. In *Proceedings of the European Conference on Computer Systems (EuroSys'11).*

[44] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. 2009. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'09).*

[45] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'09).*

[46] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'15).*