A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications

Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, Irene Zhang

ABSTRACT

When writing today's distributed programs, which frequently span both devices and cloud services, programmers are faced with complex decisions and coding tasks around coping with failure, especially when these distributed components are stateful. If their application can be cast as pure data processing, they benefit from the past 40-50 years of work from the database community, which has shown how declarative database systems can completely isolate the developer from the possibility of failure in a performant manner. Unfortunately, while there have been some attempts at bringing similar functionality into the more general distributed programming space, frequently called "exactly once execution", a compelling general-purpose system must be performant, support a variety of machine types with varying resiliency goals, and be language agnostic, allowing distributed components written in different languages to communicate. This paper introduces the first system, Ambrosia, to satisfy all these requirements. We coin the term "virtual resiliency", analogous to virtual memory, for the fundamental mechanisms (already present in data processing systems for decades) that allow programmers to write their applications in a *failure oblivious* way. Of interest to our community is the effective reapplication of much database performance optimization technology to make Ambrosia more performant than many of today's non-resilient cloud solutions.

1 Introduction

When writing today's distributed programs, which frequently span both devices and cloud services, programmers are faced with complex decisions and coding tasks around coping with failure, especially when these distributed components are stateful. For instance, consider the simple case of two objects, one called Client, and the other called Server, where Server keeps a counter, initially 0, and exposes a method called Inc() to increment the counter and return the new value. Furthermore, assume Client calls Inc() twice and prints the value of the counter after each call. If both objects are in a single process, the outcome is clear: the values 1, and 2 are displayed in the Client output. In contrast, consider the possibilities when Client and Server run on different machines, where method calls are performed through an RPC (remote procedure call) mechanism.

First let's consider possible outcomes when the Client fails and is naively restarted from scratch: If the Client fails after the first call and after the return value is received, the output will instead be 2,3, which is incorrect. If the Client fails after successfully issuing the RPC request, but before receiving the return value, the Server will initially try to provide to the Client an unexpected return value, which is problematic. Even worse, consider that Client may be restarted on a different machine, with a different IP address.

Outcomes when the Server fails are further complicated by the loss, and subsequent reinitialization, of the counter. If the Server fails after the Client has completed the first RPC, the output will be 1,1, which is incorrect. Furthermore, if the Server fails after receiving the first RPC request, but before communicating the return value, the Client is left waiting for a return value which never arrives.

In order to get the answer we all expect after taking our first programming class, which is consistent with *no* failures occurring, developers face very different challenges, depending on the type of application they are writing.

If their task can be cast as pure data processing, they benefit from the past 40-50 years of work from the database community, which has shown how declarative database systems, along with technology to make database sessions robust ([1]) can completely isolate the developer from the possibility of failure in a performant manner. Most recently, map-reduce and its progeny ([2, 3]), by pursuing similar strategies, have achieved similar results.

Unfortunately, while there have been some attempts at bringing similar functionality into the more general distributed programming space, frequently called "exactly once execution" ([1,4]), the failure to address a number of important issues has prevented widespread use of these technologies. As a result, developers either give up entirely on fully reliable applications, or implement solutions that involve complex, error-prone, and difficult to administer strategies to make applications reliable with today's cloud environments (Section 3). A compelling general-purpose solution to this problem must address the following:

- **Performance/Cost** – In order to offer failure-obliviousness as a general capability, performance must be comparable to failure-sensitive code with a good application specific strategy for achieving exactly once execution (e.g. within a factor of 2). Only data processing systems have achieved this.
- **Machine Heterogeneity** – While machines inside a datacenter are homogenous, complete distributed apps typically span devices and datacenters. While some devices may be heavy and able to persist information necessary to hide failure, others may be best effort. The end-to-end semantics must be easy to understand, reason about, and code against. Today, [4] is the closest to achieving this goal.
- **Language Heterogeneity** – Because distributed applications span across a wide variety of machines and settings, distributed components written in different languages must be able to work together. Protobuf, Avro, HTTP, and JSON effectively solve this problem.
- **Determinism** – In order to build a general purpose exactly once system, all computation needs to be deterministically replayable. Achieving this in the face of internal race conditions and non-deterministic calls (e.g. `GetTimeOfDay`) is challenging. Again, databases, with their notion of serial ordering captured in the log, are the only widely used successes.

In this paper, we coin the term "virtual resiliency", which provides developers the illusion that machines never fail, by automatically healing the system after physical failure, analogous to how virtual memory provides developers the illusion that physical memory never runs out by automatically paging memory to disk. While most data processing platforms already provide efficient programming and execution environments with virtual resiliency, there are no commonly used analogous systems for general purpose distributed programming.

We address this problem with Ambrosia (Actor Model Based Reliable Object System for Internet Applications), the first general purpose distributed programming platform with virtual resiliency, high performance, and machine and language heterogeneity. Ambrosia is a real system. For instance, Ambrosia is used in a cloud service which manages the machine images of hundreds of thousands of machines running a cloud application (Section 5).

Ambrosia's high performance was achieved by incorporating the decades' old wisdom used to build performant, reliable, and available database systems. For instance, we make extensive use of batching, high-performance log writing, high-performance serialization concepts, and group commit strategies.

Our performance with respect to resiliency is comparable to distributed data-processing systems in wide use today. For instance, the resiliency behavior of both Hadoop and Spark can be replicated with Ambrosia, making similar performance for distributed plans possible.

Through employing the technology mentioned above, we achieve throughput results which, in some cases, exceed gRPC by up to a factor of 12.7X, despite gRPC lacking failure protection. Compared to gRPC, using Ambrosia to add geo-replicated persistence increases ping latency only by 5.5ms. We vastly outperform today's cloud-based serverless, stateless compute, exactly-once strategies, in some cases achieving over a 100x improvement in cost per unit of work served.

Because Ambrosia's virtual resiliency implementation is based on database logging technology, we also offer familiar related features, like transparent high availability through active standbys. With these mechanisms, we are also able to provide application-centric features less familiar to databases, like time-travel debugging, retroactive code testing, and in-flight application upgrades.

The next section describes, abstractly, the resiliency strategy implemented both by today's cloud applications, as well as Ambrosia itself, in common terms. Section 3 describes how to implement this by hand using typical cloud application building blocks. Section 4 describes the Ambrosia design and implementation. Section 5 contains an in-depth case study of how Ambrosia was used to build a real cloud service which manages the images of hundreds of thousands of machines. Section 6 contains an experimental evaluation which compares Ambrosia against the strategy described in Section 3, as well as a comparison to gRPC. Section 7, 8, and 9 are further related work, future work, and conclusions, respectively.

2 Achieving Distributed Resiliency

While Ambrosia's architecture and implementation details are quite different from the standard resiliency implementa-

tions in today's cloud services, the overall approach in both is fundamentally the same, and has been proposed in the systems and database communities decades ago [5,6].

Both approaches involve encapsulating the application to be made resilient, logging its input, and relying on deterministic replay to reconstruct state after failure.

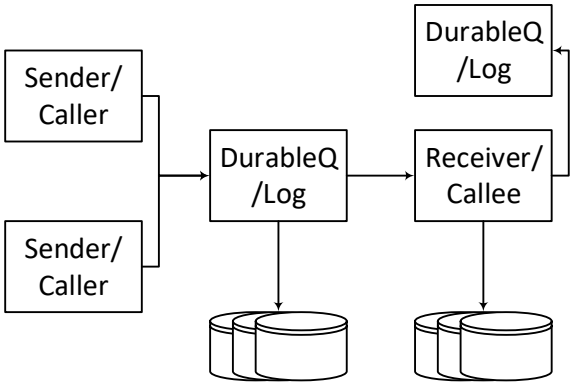

Figure 1: Distributed resiliency, abstractly

To illustrate, consider Figure 1. In this figure, we have three communicating distributed components, labeled Sender/Caller, and Receiver/Callee. Logically, a durable, distributed replicated queue (i.e. log) records ordering decisions made regarding requests from the two callers. Once this ordering has been determined, the callee deterministically processes the requests, resulting in outgoing side effects, which are first sent to another durable queue and then handled. In this fashion, multiple components may be stitched together in a peer-to-peer arrangement.

If the callee fails, the queue is used to replay input until the state is reached *which produced the last successfully enqueued output*. Processing then continues normally.

Checkpoints of the callee to distributed replicated storage are periodically taken, which limits the number of queue entries needed to replay during recovery, and consequently allows reclamation of unneeded queue space.

Database logs optimize further, eliminating the need to log read-only transactions, which can reduce network bandwidth by as much as 2x, but with additional logging potentially needed on the caller to complete an end-to-end failure-oblivious developer story [1].

Once these logs exist, they can be used to maintain any number of active standbys, which continuously read the log in recovery mode, until the primary which writes the log, fails, at which point control can fail-over to one of the active secondaries. This is a tried and true technique for implementing highly available databases [7].

3 Distributed Resiliency in the Cloud Today

While it is possible to build fully resilient distributed applications with today's cloud development tools, application writers don't usually fully implement this guarantee, due to the impression that such an implementation would be difficult to code and perform badly. To better understand this point of view, we consider a typical arrangement of today's commonly used service components. Then we explore what an implementation of distributed resiliency would look like.

For instance, Kafka, Event Hub, or Kinesis provides the durable queue. The receiver is typically microservice code executed with a serverless, stateless service, like Azure or Lambda Functions, or with manually deployed Docker containers running on a Kubernetes cluster.

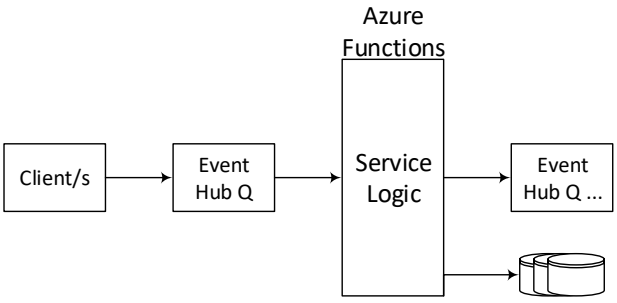

Figure 2: Resiliency using stateless compute

The above Figure shows a typical configuration for a cloud application today. In this particular example, a client, which may or may not be in the datacenter, first durably records its service request in event hub, ensuring that the request, along with its ordering relative to other requests, is preserved in replicated storage. The application logic is then expressed as an Azure function and is called on batches of event hub input. Any output (e.g. to other microservices), is then sent to other event hub queues, and the pattern is potentially repeated with other services.

Since Azure functions are stateless, to make the application resilient, every call must begin by retrieving all application state necessary to process the requests. Before the Azure function completes, it must write modified state back atomically, along with the sequence numbers associated with all input and output queues at the time of completion. These sequence numbers are used by the application developer to recover from *partially* executed Azure functions; it becomes part of the application logic.

While only one Azure function can be run at a time and still guarantee correct behavior, most applications naturally partition into independent identical pipelines, which may be run in parallel to achieve higher application throughput. For this reason, they typically store application state in key/value stores keyed on the partition id.

Example. Consider a counter service with increment and reset operations, which reports its state to a monitor service after every 1000 increments. The application state is the current count, and there are three types of messages:

```
class State { int count; }
class IncrementMessage: Message {}
class ResetMessage: Message {}
class ReportMessage: Message { int count; }
```

To achieve virtual resiliency, we label participants with a unique id, add sequence numbers to messages, and track sent and received sequence numbers per participant:

```
class PMessage {
  Id origin;
  int seqno;
  Message payload;
}
class PState {
  Id id;
  State state;
  map<Id, int> received;
  map<Id, int> sent;
}
```

To process a batch of messages, the state is first loaded from storage. Then, the messages (excluding duplicates) are processed, updating the state and sending messages. Finally, the state is written back to storage.

```
void process(Id id, List<Message> batch)
{
  State state = LoadState(id);
  foreach(var m in batch)  {
    if (m.seqno <= state.received[m.origin])
      continue; // ignore duplicate
    Process(state.state, m.payload,
      (dest,payload) =>  send(dest,
        new Message(id,++sent[dest],payload);
    state.received[m.origin] = msg.seqno;
  }
  SaveState(state);
}
void Process(State state,
             Message message,
             Action<Message, PId> send) {
   if (message is ResetMessage)
      state.count = 0;
   else if (message is IncrementMessage) {
      state.count++;
      if (state.count % 1000 == 0)
         send(monitorId,
              new ReportMessage(count));
   }
}
```

This technique of combining at-least-once delivery with idempotent processing to achieve exactly-once semantics is sometimes called “effectively-once”.

4 Ambrosia and Virtual Resiliency

In this section, we first describe the Ambrosia architecture, contrasting important design decisions with the state of the art in cloud programming, and hypothesize the performance consequences, which will be explored in Section 6.

Next, we define virtual resiliency abstractly, and discuss the difficulties which arise from trying to support virtual resiliency in a general-purpose programming environment. We then explain how we implement virtual resiliency in the context of the Ambrosia binding for C# (which is the language Ambrosia is implemented in), and briefly discuss how different choices could be made for different programming languages without compromising component interoperability.

4.1 Ambrosia Architecture

We begin with the following diagram, which illustrates the architectural components of two communicating Ambrosia services/objects/actors, called “immortals”.

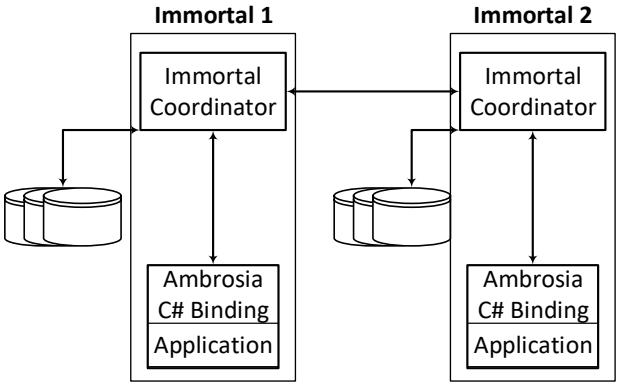

Figure 3: Resiliency using Ambrosia

First, note that each immortal is composed of two running processes, which are expected to run on the same VM/container/machine, meaning that they fail and recover together. The choice to run each immortal as two separate processes is an implementation convenience which allows us to more easily add support for multiple languages, at additional cost, but is not fundamental to Ambrosia’s design. The two pieces communicate through local TCP, and on Windows, make use of TCP loopback, minimizing latency and maximizing throughput.

The first process is the immortal coordinator, which handles all log interactions, and communication with other immortals. As a result, this process is responsible for orchestrating recovery, including handling the correct reconnection of a recovered immortal to other immortals, or even just reestablishing connection after a TCP connection breaks. It is also responsible for handling failover when active secondaries are present.

Note that the immortal coordinator relies on an open-source application hosting layer, called CRA [20], which virtualizes the communication links between a graph of immortals. For instance, when an immortal fails and is restarted on another machine, all previously connected immortals are automatically reconnected by CRA to the restarted instance, going through a connection protocol in the immortal coordinator guaranteeing fully resilient behavior.

The second process is divided into 2 parts: the first part is a language-specific Ambrosia binding, responsible for interacting with the immortal coordinator, and executing application logic in response to requests. As such, the Ambrosia language-specific binding creates a language specific development framework in which it is straightforward to author serializable immortals with deterministic replayability. The second part of this process is the application logic itself, which is authored in the context of the language specific framework.

Note that each immortal coordinator has a connection to storage, which may be backed by either local storage, or cloud replicated storage for highly reliable applications. Like a database, the immortal coordinator creates deterministic replayability through logging. In Ambrosia's case, we log all incoming requests (which can come from multiple sources). The position of these requests in the log determines the order in which they are submitted to the application process for (re)execution.

As a result, the application process must uphold the following contract: from some initial state, any execution of the same requests in the same order results in both *the same final state*, as well as *the same outgoing requests in the same order*.

In addition, the language-specific binding must also provide a state serializer. To avoid replaying from the start of the service during recovery, the immortal coordinator must occasionally checkpoint the state of the immortal, which includes the application state. Again, the way this serialization is provided can vary from language to language, or even amongst bindings for the same language.

At this point, it is interesting to point to a few important differences between Ambrosia's architecture, and the architecture described in Section 3:

- Because the durable queue is hidden in the immortal coordinator, and immortal coordinators are direct peers, the immortal coordinator is now free to store the queue wherever, and however it likes. For instance, it could store the queue in a local file, or to some form of cloud storage. Furthermore, this decision can even be delayed until deployment time, with different decisions made for different deployments.
- Application developers no longer write logic to recover from partial executions, since this is all handled by the immortal coordinator. For the same reason, they also no longer write code to retrieve and store state, since all state is implicitly durable through the combination of logging and deterministic execution. As we'll show in Section 6, this can have profound performance implications.
- Since the log implicitly contains all state changes for the application, debugging is greatly facilitated. To perform "time travel debugging" [23], we simply execute from a checkpoint before a bug occurred, and roll forward with the debugger attached, without involving any distributed components outside one immortal itself. This kind of debugging convenience is very difficult to replicate when applications explicitly write recovery code and durable state.

4.2 Virtual Resiliency and C#

Virtual resiliency, a term coined in this paper, is defined as follows:

Virtual Resiliency *– A mechanism in a (possibly distributed) programming and execution environment, typically employing a log, which exploits the replayably deterministic nature and serializability of an application to automatically mask failure.*

Like virtual memory, virtual resiliency is a mechanism. Like virtual memory, the effect of working with an environment which has virtual resiliency can be easily described: the presence of virtual resiliency removes the need for application writers to write logic for recovery or state protection. This paper describes one reference implementation, although there are many others.

Note that data processing systems, which typically express their queries in SQL variants, have provided their query writers virtual resiliency for decades. Map-reduce

systems, which don't necessarily use SQL, also provide this capability. Note that in all these cases, this feature leverages the ability to deterministically replay, like Ambrosia.

Also, note the use of the phrase "replayably deterministic". Transactional databases provide replayable determinism, even though they have many sources of non-determinism, like thread scheduling. They are, however, replayably deterministic, which means that with the aid of the recovery log, they can recover the state which affects external visibility

In the case of C#, we interpret messages between immortals as RPC calls. The vocabulary of calls handled by a given immortal, including the arguments and their types, is expressed in C# as an interface. For example, the following example illustrates how the counter example in the previous section is implemented in Ambrosia-C#.

Example. In the C# binding for Ambrosia, the counter service defines interfaces for two immortals:

```
public interface ICounter {
   void Increment();
   void Reset();
}
public interface IMonitor {
   void Report(int count);
}
```

The implementation of the counter contains the application logic, some attributes to allow state serialization, and initialization to set up a proxy for sending messages to the monitor:

```
[DataContract] class Counter:
  Immortal<ICounterProxy>, ICounter
{
  [DataMember] int count;
  [DataMember] IMonitorProxy monitorProxy;

  public Counter() {
    monitorProxy = GetProxy<IMonitor>("mon1");
  }

  public void Reset() { count = 0; }

  public void Increment() {
    count++;
    if (count % 1000 == 0)
      monitorProxy.ReportFork(count);
  }
}
```

From the two interfaces, we automatically generate C# libraries which contain abstract base classes with associated abstract method calls, which are implemented by the application writer. For instance, class `Counter` in the above example implements `ICounter`, which is in the associated generated C# library.

These generated libraries also contain proxies for making method calls on immortal instances of this type from other Ambrosia applications. For instance, in the above example, `monitorProxy` is of type `IMonitorProxy`, which is a generated type for interacting with immortals which implement `IMonitor`. `GetProxy` is used to get a handle to an immortal registered in a catalog of immortals stored in a table. (Ambrosia uses Azure tables.)

In C#, Ambrosia calls to other immortals can be executed in either an *awaitable* (called async), or *non-awaitable* (called fork) fashion. For instance, the Report call on the `monitorProxy` is a forked call, which means it is not awaitable. Both RPC versions are automatically generated in the proxy for using an Ambrosia immortal. If an RPC is executed in a non-awaitable fashion, no return value is expected or sent, similar to sending an event. If an Ambrosia call is awaited, the executing call is suspended until the return value arrives through the message queue from the coordinator. Handling a return value simply involves waking up the suspended RPC and continuing execution.

Within the C# language binding, we execute all arriving RPC requests in the order in which they arrive (i.e. were logged), in a single threaded manner. Therefore, as long as the application code is deterministic, we have met the determinism requirement for Ambrosia. Since the handling of return values relative to new RPC calls is deterministically ordered and single threaded, determinism w.r.t. the handling of return values is preserved.

Note that for expensive partitionable workloads, Ambrosia applications can be sharded, where each shard runs on its own set of cores, like many other systems. While Ambrosia does not yet support elasticity, this is a subject of future work, and we expect elastic sharding designs similar to other stateful systems, like databases [9], and Orleans [16] to be effective.

4.2.1 Replayable Determinism and Impulses

Of course, there are unavoidable sources of non-determinism, like user input, or calls to `GetTimeOfDay`. In the first case, deterministic replay would require that a user reenter their input. In the second case the clock would somehow have to be rewound to the exact point in time it was called in previous runs. Fortunately, we have a log! Like a database, we can log all non-deterministic data before acting on it, resulting in deterministic replayability.

When non-state changing method calls are deterministically made, but the return value differs on reexecution, as

in `GetTimeOfDay`, it suffices to wrap the `GetTimeOfDay` call in an Ambrosia *self-RPC*, returning the measured value. This will ensure that all reexecutions will use the first successfully logged execution of the call, whose return value will also be logged, solving our determinism problem.

When data arrives from an asynchronous unreplayable source like user input, the data is passed to an "Impulse Handler", which is a specially marked Ambrosia RPC, which logs the data (since it's an argument), and continues processing as a typical RPC call after logging. Such impulse handlers differ from regular RPC calls in that they are faithfully replayed during recovery but cannot be called unless the node is handling incoming requests (not recovering or a secondary).

Example. After declaring an impulse handler

```
public interface IMyImmortal {
   [ImpulseHandler]
   void UserInput(string line);
}
```

we can start a background thread that reads nondeterministic console input and sends it to the impulse handler:

```
class MyImmortal:
Immortal<IMyImmortalProxy>, IMyImmortal
{
  public MyImmortal() {
    new Thread(() => {
      string line;
      while ((line=Console.ReadLine())!=null)
         thisProxy.UserInputFork(line);
    }).Start();
  }
  public void UserInput(string line) {
    // process input deterministically here
  }
}
```

Common scenarios using impulse handlers include user input, data from lightweight non-resilient sources as in IOT applications, and periodic self-calls like health checking. Note that it is perfectly fine for background threads to be used to collect this data, if the only side effects or state changes happen in the impulse handler itself, *after* the data has been logged. In part for this reason, Ambrosia Immortals have an overloadable method called `BecomingPrimary`, which is executed after recovery, and just prior to dispatching the first RPC. All impulse gathering threads may be started here, ensuring that new impulses are not arriving during recovery.

Great care must be taken by the immortal coordinator in correctly handling impulses during recovery. For instance, the outgoing messages after replay will not contain the impulses generated by threads that don't run during replay (e.g. collecting input from the user). As a result, sequence numbers on the reproduced output will no longer match the sequence numbers of other services which received impulses missing from the reproduced RPC stream. The solution is to correctly bookkeep sequence numbers with and without impulses, and for a recovering node to send the first non-impulse RPC after the last non-impulse received by the listener.

State serialization is a more straightforward affair. The immortal coordinator has, built into it, the ability to serialize its state, along with all necessary buffers. The C# language binding must also provide a way to serialize the state of the user code in the immortal. Fortunately, C# has a built-in ability to serialize and deserialize classes labeled as `[DataContract]`. This simply requires that members be marked which are required to correctly serialize and deserialize objects of that type. We therefore make use of this standard feature, mostly solving our state serialization problem.

The only tricky part of serialization/deserialization in C# is the state of suspended tasks waiting for return values. These suspended calls, along with their call stacks, must be serialized upon checkpointing, and deserialized upon replay. Fortunately, we were able to find a library which does exactly this for C# [24].

4.2.2 Nondeterministic Task Wrapping

By using the impulse handler feature, applications can perform arbitrary nondeterministic operations, yet remain deterministically replayable. For convenience, Ambrosia provides a `NondetTask` function that lets users wrap arbitrary nondeterministic tasks:

```
time = await NondetTask(() => GetTimeOfDay());
x = await NondetTask(() => new Random().Next());
```

The result of the task is automatically persisted in the log by an impulse handler, and therefore deterministically replayable.

Asynchronous tasks containing I/O can also be wrapped, allowing applications to easily consume external nondeterministic non-Ambrosia services. For example, we can load or store a blob in cloud storage:

```
byte[] content = await NondetTask(
  async () => await LoadBlob("name"));

await NondetTask(
  async () => await StoreBlob("name", content));
```

Wrapped tasks execute with at-least-once semantics: if the primary that starts the task the first time around fails before recording a response via an impulse handler, the next primary restarts the task.

Conceptually, exceptions are not failures, but special return values. Thus, the task wrapper implementation logs and rethrows exceptions, after which they may (or may not) be caught and/or retried by the application.

Because the wrapped tasks execute on the thread pool, task wrapping also provides an appropriate solution for running CPU-intensive computations without locking up the scheduler:

```
x = await NondetTask(() => HeavyComputation());
```

This also has the benefit of caching the heavy computation so it is not rerun on replay. Since wrapped tasks execute outside of the immortal scheduler, they must not update the state of the immortal directly.

4.3 Getting high performance

Because the resiliency approach employed by Ambrosia is similar to a transactional database, many of the optimizations developed by the database community can be gainfully employed to give Ambrosia dramatically improved performance over existing methods for resilient cloud programming. This section details these optimizations.

4.3.1 Adaptive Batching in Memory

Similar to the approach used in [8], whenever there is an important latency/throughput tradeoff in Ambrosia's implementation, adaptive batching is used to simultaneously guarantee low latency in a lightly loaded system, and high throughput in a heavily loaded system.

Since a single TCP connection can only transmit one message at a time, when part of the system becomes throughput challenged, messages, if buffered, accumulate while a message is transmitted. The next message then contains all buffered messages, wrapped up into a single message which begins with a count of the number of messages in the batch. This results in very efficient message handling code on the other side, which is a tight loop over the messages in the batch.

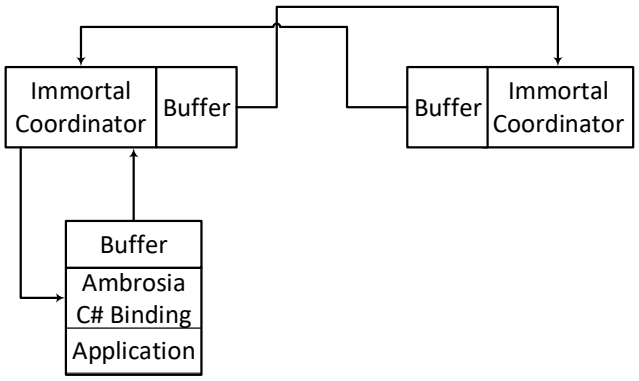

Figure 4: Adaptive Batching in Ambrosia

The Figure above shows 2 sensitive locations in Ambrosia, where messages can accumulate, represented by buffers used to form batches prior to sending. First, in the language binding, if the application generates large amounts of traffic, as in streaming workloads, systems such as Trill [8] have shown that adaptive batching is greatly advantageous. Second, note the shuffle in the immortal coordinator, which sends the individual messages received from the language binding to their corresponding outputs. Quill [17] showed a similar advantage to batching the in-memory output of a shuffle. The adaptive batching used for the immortal coordinator is almost identical to what is described in Quill. Note that, as a result, we associate with each output in the immortal coordinator, a queue of buffer pages which grabs and releases pages from and to a shared pool. This ensures that outputs will respond flexibly to changes in relative load, without allocating unreasonable amounts of memory.

4.3.2 Writing the Log to Storage Efficiently

Like a database, Ambrosia uses a log to durably record system choices which could differ on reexecution, making the computation deterministic. For instance, the order in which messages appear in the log is the order in which those messages are presented to a language binding, regardless of whether it is the first execution, or replay. Log writing is handled inside the Coordinator processes (Figure 3).

Like a database, which can have multiple threads associated with different database sessions simultaneously writing to the log, Ambrosia has multiple threads, associated with different callers, simultaneously writing to the log.

We therefore take the standard approach described in [18], where each thread grabs the position in the current log page in which it will write its bytes. Threads can then concurrently write their bytes to the log record, where the last writer, which closes the page to further writes, waits for the concurrent writers to finish before writing the page to storage. After the page is closed to writing, new writers write

to the next log page etc. Our implementation uses compare and swap to execute this strategy in a highly efficient manner, as is described in [18].

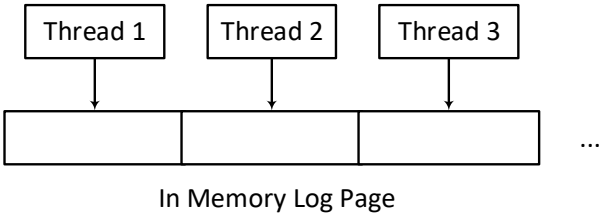

Figure 5: In Memory Log Page

The above figure shows the in-memory log page for an immortal, with 3 threads, corresponding to 3 immortals making RPC calls to this immortal.

They quickly grab locations in the log to write their messages, which allows them to copy the bytes into the log in parallel. The last writer to close out the page waits for other writers to finish, at which time it submits the page for asynchronous writing. As soon as the last writer determines that it is closing out the page, a bit is set which redirects all future writers to the next in memory log page, unblocking future writers. The number of in memory log pages is fixed, which ultimately forces delays on the senders if the system becomes I/O bottlenecked.

4.3.3 Batch Commit

Group commit [19] is the mechanism employed by transactional databases to get the benefits of delaying a transaction commit until the log record which contains the commit, which also contains many other commits, is flushed. This results in the delay of commit notification until a flush is performed, at which time, many transaction commits become externally visible.

We achieve a similar effect by writing many requests to the log concurrently and waiting to submit these requests to the language binding until after the requests have been flushed to storage. This ensures that no outgoing messages (which are the consequence of the input messages) are produced until the input messages, and their relative order, are made durable in the log.

It is worth noting that with some extra bookkeeping, we could further reduce latency by submitting the requests to the application after their relative ordering is determined, but before the log record is written, if we withheld the resulting output messages until after the log write completed. We have found, though, that in practice, the latency savings are not large or necessary for most real applications.

4.4 Other Log Based Ambrosia Features

There are four additional useful features enabled by virtual resiliency based on logging and state serialization.

4.4.1 High Availability

The first of these features is high availability through active standbys. In Ambrosia, the log, and associated checkpoints, are written to a directory specified by the immortal deployer. In both Windows and Linux, that directory can be backed by either local storage, or cloud-replicated storage. For instance, Azure Files may be mounted on all internet connected Linux and Windows machines. Alternatively, Azure Managed Disks offer a performant and very cost-effective alternative for immortals running inside Azure.

At any given moment, there is one primary, which produces the log and is connected to other Ambrosia immortals, and secondaries, which consume the log in recovery mode, until they become primary. Leader election is simply the result of all instances continuously (e.g. every half second) trying to acquire the exclusive write lock on the log. When an instance acquires the lock, is becomes primary, and CRA establishes all connections to other immortals. If a primary ever loses the file lock, it commits suicide.

The log is broken into deployer-specified chunks, such that whenever a threshold is reached, a new log file is created with an incremented chunk number as part of the filename. When a secondary becomes primary it immediately starts a new log file.

In Ambrosia's implementation of high availability, checkpoints are generated by a secondary, such that each time a new log file is started, there is an associated generated checkpoint which contains the state of the immortal instance at the *start* of the log file. The secondary-based checkpointing prevents loss of primary availability while checkpointing and turns out to be the optimal strategy in a resource-reservation based environment like the cloud [21]. A new secondary then starts from the latest checkpoint and rolls forward until it is caught up.

4.4.2 Time Travel Debugging

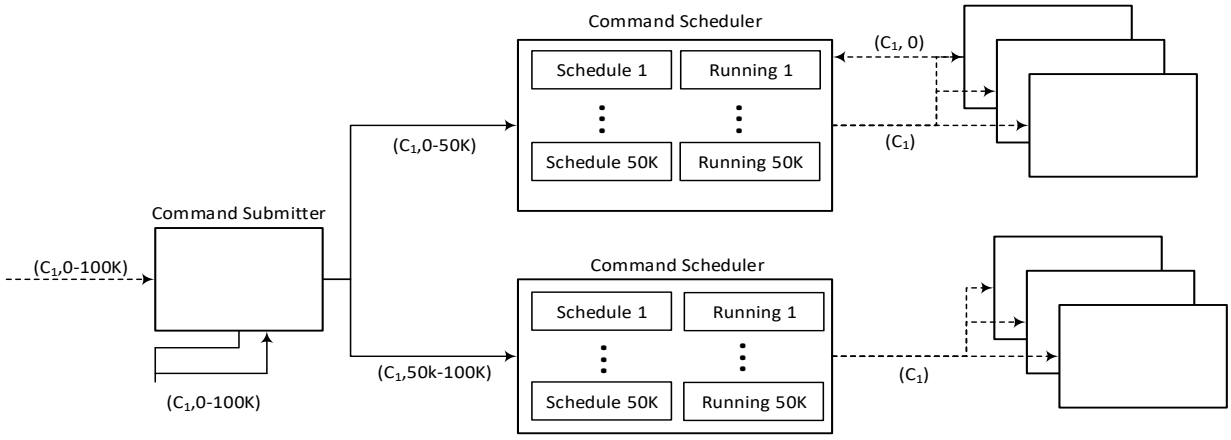

Figure 6: Anatomy of a real ambrosia application

Using the checkpoints and log files, Ambrosia exploits application replayable determinism to implement time travel debugging. With time travel debugging, the developer starts the application process and attaches the debugger. The developer then starts the immortal coordinator in time travel mode. In this mode, the developer points the coordinator to the log and checkpoint files (which may still be live) and specifies the checkpoint number to begin recovering from. The immortal coordinator then runs recovery, never becoming primary.

Since the debugger is attached to the application process, all the usual debugger features may be used, like setting breakpoints, and stepping through code. Because replaying the log is deterministic, the same application behavior may be replayed and debugged as many times as desired, even against a live log.

4.4.3 Retroactive Code Testing

Related to time travel debugging, if the application writer wants to test an alternate version of the service which has the same interface and state (as is frequently the case when fixing bugs), they can perform time travel debugging with the updated version of the application, using the debugger, to find a bug or test a fix. A developer may even use this feature to create new user-level logs against the replay.

Observe that new versions of services may even be rolled out this way, where the new version starts as an active secondary and becomes primary when all instances associated with old versions are killed.

4.4.4 Live Service Upgrades

Occasionally, services go through significant upgrades, where the API to the service broadens, and/or where the type of the application state changes (e.g. the addition of new counters). For such situations, Ambrosia allows developers to define an "upgraded Immortal", where both old and new versions of the application code are present in the process.

When such an immortal is deployed, it recovers using the old version of the service. When it becomes primary, it calls a constructor for the new version of the service, which takes as an argument the state of the old version at the time it becomes primary. A new checkpoint is then taken of the new version of the service, and the upgrade is complete.

To deploy such an upgrade, it is initially added as an active secondary. While killing all the instances of the old service, the new version becomes primary and the service continues. Note that any old versions of the service still running simply die once the new version becomes primary.

5 Anatomy of a Real Ambrosia Application

Ambrosia is a real system, used in production today. This section describes a real Ambrosia service used to manage the submission of commands to a collection of hundreds of thousands of machines running another service.

Figure 6 shows the overall arrangement of the two services. Working backwards from the machines whose images are being maintained, there is a command scheduling and monitoring service, composed of multiple machines, such that each machine is responsible for scheduling, submitting commands for, and monitoring a subset of the machines being maintained. Since there are between three and four hundred thousand machines being maintained, we need about 10 machines to satisfy the overall load, and simply hash partition the overall collection of machines. Each of these 10 machines accept user-submitted commands from a command submission service. Note that while the command submission service, and the command scheduling/monitoring service are Ambrosia immortals, Ambrosia is not running on either the machines which

submit commands to the command submitter, or the machines being managed.

Note that the command submitter and all command schedulers are running in active/active configurations, resulting in high availability of the service, overall.

Starting from the command submission service and working forwards, commands arrive through a web service interface, which are serviced using a collection of threads in the Ambrosia immortal. Non-Ambrosia calls are represented in our diagram by dashed lines, such as (C_1, 0-100K) in the example above. When a command arrives at the command submitter, a self-call is made using an impulse handler, called `AcceptRequest`, such as in the example above, represented by the solid line. This makes the request durable. The request itself contains the command to submit, in the form of a `Command` class, and a list of machines on which the command needs to be run. In this example, C_1 must be executed on machines 0 through 100,000.

`AcceptRequest`, upon execution, partitions the set of machines on which the command must be run into ten sets, corresponding to the 10 immortal instances in the command scheduling service. Each of these machines, then, receives the command, and the associated set of machines it must be run on, through the `ScheduleRequest` method. The `ScheduleRequest` method then adds the new command to the schedule of each machine it must be run on. Note that some commands can be run in parallel, with other commands, and others not. It is up to the scheduler to maintain a legal schedule for each of the machines it is responsible for. Note that the schedule is part of the serializable state of the immortal, and therefore is automatically protected from failure. In the example above, the first command scheduler is responsible for managing the first 50,000 machines, so the `ScheduleRequest` call sent to that Ambrosia instance only applies to those machines. `ScheduleRequest` now adds the new command C_1 to each relevant schedule, serializing conflicts while maximizing parallelism.

Associated with the schedule of each machine is a list of currently running commands. which is also part of the serializable state of the immortal. Periodically, the scheduling immortal scans through the list of currently running commands, and (re)issues all open requests to individual machines through a web service API. This results in *at least once* submission of commands to individual machines maintained. The submission of requests is made idempotent on the maintained machine, resulting in *exactly-once* execution of commands. In the above example, we see C_1 sent to all managed machines, since it must be run on them all.

It is worth pointing out that the thread which periodically scans the running commands and reissues them only runs on the primary. For instance, we don't want secondaries to reissue these commands. For such situations, we overload a method in the immortal called `OnBecomingPrimary`, which runs just prior to establishing connections. The scanning thread may be started there.

Once the command has completed, a finished message is sent by the managed machine, to the correct scheduler through a web service interface, which immediately calls the impulse handler `CommandFinished`, which updates the state of the machine's schedule and running commands, and schedules the next command/s if appropriate. Note that in the case of a lost `Finished` message, the running command is periodically reissued automatically, resulting in a resend of the `Finished` message without re-execution. This guarantees exactly-once execution of the command on the managed machines, regardless of failure.

It is interesting to observe that the interaction of the scheduler and managed machines is a special case of the `NonDetTask` logic shown in Section 4.2.1. In this case, since all retries in the scheduler may be performed at once on the same schedule, the logic is implemented directly with impulse handlers, rather than using `NonDetTask`, which has more flexible behavior.

6 Experimental Evaluation

In this section, we explore the performance of several different implementations for our mirror service, described in Section 4. We focus on 4 different implementations:

- A non-resilient implementation using gRPC
- Ambrosia (C# client)
- Ambrosia (native client in C++)
- Azure Serverless - Based on the design described in Section 3.

We consider 2 types of experiments. The first is a throughput experiment, which tests the possible throughput (or dollar cost of throughput in the case of Azure Serverless) for payload delivery of various sizes. This is a streaming workload, where acknowledgements for each delivered payload need not be delivered back to the sender.

The second experiment is designed to test the latency of these various approaches under light load, which reflects the best latency achievable by these systems. For this we perform pings, where only one outstanding ping is allowed.

6.1 gRPC

This implementation is the ringer, as it represents a performance-oriented cloud RPC framework that does not do any logging or recovery. It is just straight-up RPC. As such, we would expect it to soundly beat Ambrosia, and it *should* represent an upper bound of what's possible.

Of course, theory and practice are not always the same in practice. One of the standard data streaming tricks we employed, adaptive batching, allows us to navigate the throughput/latency tradeoff very effectively. Only experimentation will determine if gRPC has effectively incorporated this technology.

Note that for the throughput benchmark, we used the gRPC streaming implementation in C++, which according to [22], is the most performant option for this benchmark. In this mode, there is a client and a server, where the server has a streaming RPC, called Receive, which takes a byte array of the appropriate size and keeps a running total of all bytes received. The choice of a byte array is designed to minimize serialization and deserialization overhead, which is orthogonal to the issues tested here.

For the latency test, we use a single RPC call, which performs the fastest round trip available in gRPC, performing one at a time to ensure minimum interference.

6.2 Ambrosia – C#

This implementation is the main focus of this paper. While we provide implementations for both .NET framework, and .NET core, and run on both Windows and Linux, we focus in these experiments on the .NET framework implementation on Windows. We wrote two immortals: a client and a server. Both are fully recoverable and generate their own logs and checkpoints. Each write their logs to Azure storage. In particular, we wrote our logs to 8x P10 Azure Premium Managed Disks which were pooled together in a software RAID configuration with aggregate bandwidth of 800 MB/s. Note that this RAID configuration represents the cheapest way to allocate the bandwidth we anticipated we'd need for our tests.

Like the gRPC implementation, the server keeps track of the total bytes sent, which is part of the serializable state of the server and as a result is marked as a data member. Note that checkpointing (but not logging) was turned off for these experiments.

Like gRPC, we use our streaming RPC calls (Fork). Conceptually, there is very little difference between the code written to implement this microbenchmark in gRPC and Ambrosia, although differences in C# and C++ make the Ambrosia-C# version more readable.

6.3 Ambrosia – C++

In this version, we wanted to test the performance of our C# language binding compared to a very lean binding written in C++. It uses a simpler ring-buffer for communication between a networking thread and an application thread. Note that we can arbitrarily mix and match C# and C++ clients and servers, since we implemented a C++ binding that understands our serialization format for byte arrays.

For these experiments, we are using the same immortal coordinator, written in C#, regardless of the language binding being used for the immortal code. We would expect that the C++ implementation would be at least as performant as the C# implementation, although this may not be relevant if we are bottlenecked by the coordinator.

6.4 Azure Serverless and Stateless Compute

A popular design (see Figure 2) for microservices is to ingress data into a messaging layer such as Azure Event Hubs, a fully-managed, real-time data ingestion service. Event Hubs feeds data to a serverless execution fabric such as Azure Functions, which pulls data batches from EventHub and executes the user code. The user code is stateless; it loads state from a persistent back-end such as Azure Tables, runs application logic, and writes back the state at the end of execution. We can compute the total cost to run a microservice using this architecture, in terms of dollar amount per month, per MB/sec of ingress. We assume that both the messaging layer and the serverless functions layer can parallelize as much as needed. The cost components of a deployment are:

(1) Cost to ingress data into EventHub: It currently costs $0.028 per million messages, plus $0.015 per hour, per throughput unit (1 MB/sec ingress, 2 MB/sec egress). Event Hubs also offers a dedicated option that costs $4999.77 per month; we choose the lower cost between these options for our computations.

(2) Azure Function costs have two components. First, there is a cost of $0.20 per million executions, we assume that a function is invoked with batches containing up to 256KB of data from Event Hubs. Second, there is an *execution time* cost of $0.000016 per GB-sec, a unit of resource consumption. Resource consumption is calculated by multiplying average memory size in gigabytes by the time in milliseconds it takes to execute the function. We assume 128MB of average memory (the lowest allowed) and that it takes 0.1ms per event in the batch fed to Azure functions.

(3) We perform one read and write to storage per function invocation. Azure Tables cost $0.00036 per 10,000 transactions, with a $0.07 per GB cost for the actual first terabyte of storage. We assume that 1GB is enough to hold the state for our example.

We vary the per-message size from 16 bytes and up, and compute costs over a month if the service ingests 100MB/sec over the entire month. We then scale the result down to report the cost per month, per MB/sec of ingested data.

6.5 Results

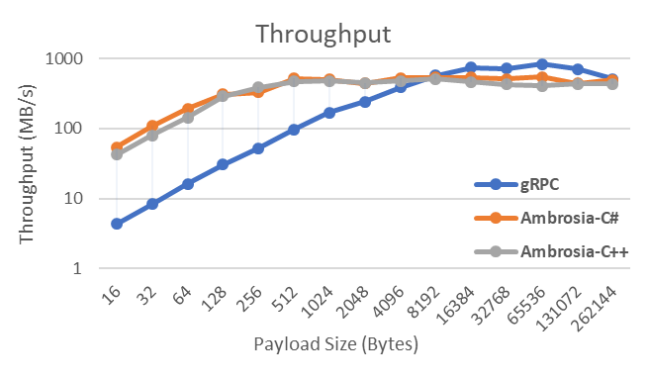

Figure 7: Throughput as a function of message size

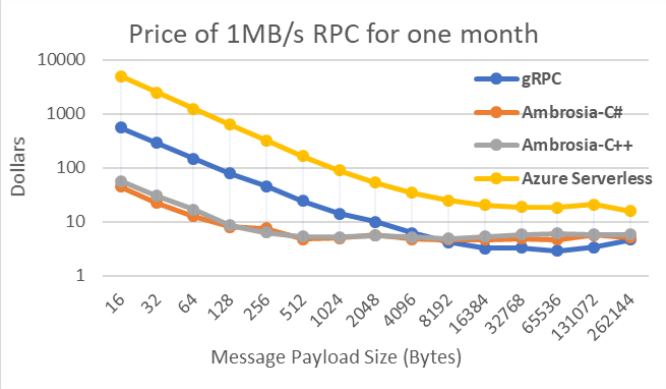

Figure 8: The price of RPCs that update state. In the case of gRPC, the state is in memory (not resilient).

Note that in all cases except Azure serverless, we perform a throughput experiment and a latency experiment on 2x D14v2 Azure instances, where one acts as the client and the other acts as the server. The same actual instances were used in all experiments to eliminate hardware variation.

For the resilient implementation based on the design in Section 3, which uses only serverless components, we simply calculate costs for comparable work done, and measure the latency of individual components, assuming latency is additive.

The results of the throughput experiments are shown in Figure 7. First, observe that our ringer isn't a ringer after all! Even though gRPC is a bare RPC framework, without any notion of failure resiliency, it is nevertheless significantly *less performant* for small message sizes than either of the Ambrosia variations. For 16-byte arguments, it is actually more than 10x slower than Ambrosia-C#! We attribute this to our careful implementation of adaptive batching, which is particularly helpful for small message sizes. Note that gRPC briefly outperforms Ambrosia-C# a bit near the throughput limit, but pulls back, for some reason, to efficiency levels indistinguishable from Ambrosia-C#. We saw this trend continue for even larger message sizes.

As expected, there is very little difference between the Ambrosia-C# and Ambrosia-C++ versions. The Ambrosia-C++ slightly underperforms, which we believe to reflect the relative maturity levels of the bindings.

The results of the ping experiment are shown in Table 1:

	0.5	0.9	0.99	0.999	Mean
Ambrosia-C#	6.57	7.1	8.71	11.34	6.63
Ambrosia-C++	6	6.47	8.79	12.48	6.1
gRPC	0.5	0.59	0.8	61.85	0.58

Table 1: Latency in milliseconds

The first four columns show the latencies for various percentiles. For instance, 0.5 is the median, 0.9 is the value for which 90% of the latencies are lower, etc. Unsurprisingly, gRPC, which simply sends a message across the wire from one machine to the other, is the clear latency champ. Ambrosia, on the other hand, must make two sequential round trips to our P10 disks. What we see here is that adaptive batching and asynchrony completely closes the gap (and then some) on throughput, but not latency. Oddly, gRPC has the higher tail latency around 60ms. These are not one-time outliers; they occur regularly throughout the workload. It likely reflects global locking associated with gRPC periodically cleaning up resources. We left out numbers for stateless compute due to measuring difficulties, but expect latencies for the workflow to exceed 100ms [25].

For the final experiment, we compare the costs of performing our ping experiment in terms of cost per month per MB if we ran the experiment continuously for a month on the hardware described. Performing any comparison of this sort is fraught with difficult decisions which can make one strategy fare better or worse.

For instance, EventHub can be run in either basic or standard mode. There are several differences, one of which is the ability to write queue history to cold storage for later processing. The difference in price is a factor of 2. Of course, Ambrosia provides this capability, making the log directly available. Nevertheless, in our calculations, we chose the basic level of support, as some users may not care about this feature.

Another example of this is that we chose relatively expensive instances to run Ambrosia and gRPC on, wanting

to test the limits of performance. As such, the numbers for both Ambrosia and gRPC can likely be significantly improved by optimizing for price/performance, rather than performance. Also, we are assuming the on demand price of these VMs, which can be reduced by about 30% with long term reservations.

With these caveats in mind, the results are shown in Figure 8. First, note that both gRPC and Ambrosia are significantly cheaper than stateless compute. For gRPC, this isn't a surprise, as stateless compute has resilience to failure, and gRPC doesn't. The bigger surprise is that Ambrosia is even more dramatically cheaper. This is due to the combination of effective adaptive batching, which dramatically helps for small message sizes, and the very low cost of storage bandwidth relative to compute. For instance, the monthly cost of a DS14 instance is $1541.03, while the monthly cost of 100MB of continuous storage bandwidth is only 19.71.

At larger message sizes, all the options become closer, as differences in adaptive batching and CPU efficiency become irrelevant. In these experiments, the state is a single counter which can be read once for each batch, then updated with a single write per batch. More typical is that the application is partitioned, resulting in close to one read and write for each message in a batch. This would increase the cost of stateless compute for small messages significantly.

7 Further Related Work

Throughout the paper, we have mentioned much related work in the DB and systems communities, which we will not revisit here. We have, however, not talked about related work in deterministic computation, which is relevant for language bindings, or maybe even the construction of languages and runtimes specifically for use with Ambrosia.

Typically, determinism is accomplished via a combination of programmer obligations and language-specific mechanisms. Ensuring deterministic execution has been the subject of a substantial body of research in operating systems [10,11], threading libraries [12,13], and programming languages [14,15]. When developing a service to run on top of Ambrosia, any *combination* of these approaches may be used, as determinism is a *local* property of each communication endpoint.

8 Future Work

While Ambrosia in its current form is immediately useful, as is shown in Section 5, there are many more interesting research problems to think about and solve, as well as associated exciting possible capabilities.

The most obvious next step is elastic scaleout. While databases have certainly solved the problem for transactional systems, they rely on the ability to abort in flight transactions. In an exactly once system, this is not an option, and performant solutions to this problem must be found as part of a desirable implementation.

While this work has made little of the ability of immortals to be relocated on other machines, this is potentially very exciting in the world of devices, where Ambrosia facilitates the construction of easily migratable apps from one device to another, without loss of state.

Figuring out how to support other languages, including Javascript, and Java, is both useful and interesting. For instance, the language binding choices made for Javascript, a single-threaded language, may be quite different from a language like C#, where thread non-determinism can complicate achieving replayably deterministic behavior.

Finally, as the number of CPUs and distributed state proliferates with IOT, the problem of distributed state management in distributed applications will become excruciating. Ambrosia provides a crucial building block to tame this complexity. Understanding Ambrosia's role, and potential gaps, for these scenarios is very important.

9 Conclusions

This paper introduces Ambrosia, the first general purpose distributed platform that provides its developers virtual resiliency with high performance, and the flexibility of working across a variety of machines, operating systems, and languages. Furthermore, Ambrosia supports high availability, time-travel debugging, retroactive debugging, and live service upgrades. Ambrosia is a real system, used to build a service which manages hundreds of thousands of machines.

Ambrosia's performance depends upon technology, developed by the database community, used to develop performant data processing systems. For instance, we make extensive use of adaptive batching from the streaming community, efficient log writing, and batch commit.

Therefore, Ambrosia achieves dramatically higher throughput than gRPC, a widely used non-resilient RPC framework, outperforming gRPC by as much as 10x for smaller message sizes, typical of cloud services, but at higher latency costs due to cloud storage latency. Furthermore, Ambrosia is both simpler to program, and cheaper to run, than a typical stateless compute cloud configuration designed to be resilient to failure, outperforming this configuration by over 100x for small message sizes. These results also indicate that the stateless compute approach embraced by most cloud developers is likely a temporary workaround until systems like Ambrosia mature.

REFERENCES

[1] Roger S. Barga, David B. Lomet.. Phoenix: Making Applications Robust. SIGMOD Conference 1999: 562-564

[2] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI Conference 2004: 137-150

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica: Spark: Cluster Computing with Working Sets. HotCloud 2010

[4] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, Henry M. Levy: Customizable and Extensible Deployment for Mobile/Cloud Applications. OSDI 2014: 97-112

[5] Laura M. Haas, Patricia G. Selinger, Elisa Bertino, Dean Daniels, Bruce G. Lindsay, Guy M. Lohman, Yoshifumi Masunaga, C. Mohan, Pui Ng, Paul F. Wilms, Robert A. Yost: R*: A Research Project on Distributed Relational DBMS. IEEE Database Eng. Bull. 5(4): 28-32 (1982)

[6] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3): 375-408 (2002)

[7] Wilschut, A., and Apers, P.: Dataflow Query Execution in a Parallel Main-memory Environment. In Distributed and Parallel Databases 1(1), 1993.

[8] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James F. Terwilliger: Trill: Engineering a Library for Diverse Analytics. IEEE Data Eng. Bull. 38(4): 51-60 (2015)

[9] Manish Mehta, David J. DeWitt: Data Placement in Shared-Nothing Parallel Database Systems. VLDB J. 6(1): 53-72 (1997)

[10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, 2010.

[11] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven Gribble. Deterministic process groups in dOS. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, 2010.

[12] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pages 97–108, New York, NY, USA, 2009. ACM.

[13] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In Symposium on Operating Systems Principles. ACM, 2011.

[14] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In POPL, pages 257–270, 2014.

[15] Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A type and effect system for deterministic parallel Java. In Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09, page 97, Orlando, Florida, USA, 2009.

[16] Philip A. Bernstein, Sergey Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Computing 20(5): 71-75 (2016)

[17] Badrish Chandramouli, Raul Castro Fernandez, Jonathan Goldstein, Ahmed Eldawy, Abdul Quamar: Quill: Efficient, Transferable, and Rich Analytics at Scale. PVLDB 9(14): 1623-1634 (2016)

[18] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta: LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888 (2013)

[19] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84). ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/602259.602261

[20] Common Runtime for Applications (CRA) – a runtime for distributed dataflow applications. https://github.com/Microsoft/CRA.

[21] Badrish Chandramouli, Jonathan Goldstein: Shrink - Prescribing Resiliency Solutions for Streaming. PVLDB 10(5): 505-516 (2017).

[22] gRPC Benchmarking. http://grpc.io/docs/guides/benchmarking.html

[23] ET Barr, M Marron: Tardis: Affordable time-travel debugging in managed runtimes. ACM SIGPLAN Notices 49 (10), 67-82

[24] AsyncWorkflow. http://github.com/ljw1004/blog/tree/master/Async/AsyncWorkflow

[25] Blog Post Concerning Event Hub Latency. http://blogs.msdn.microsoft.com/opensourcemsft/2015/08/08/choosing-between-azure-event-hub-and-kafka-what-you-need-to-know/