Arming Cloud Services with Task Aspects

Zhenyu Guo Cheng Chen Haoxiang Lin Sean McDirmid
Fan Yang Xueying Guo Mao Yang Lidong Zhou
Microsoft Research

ABSTRACT

Our cloud services are losing too many battles to faults like software bugs, resource interference, and hardware failures. Many tools can help us win these battles: model checkers to verify, fault injection to find bugs, replay to debug, and many more. Unfortunately, tools are currently afterthoughts in cloud service designs that must either be tediously tangled into service implementations or integrated transparently in ways that fail to effectively capture the service's problematic non-deterministic (concurrent, asynchronous, and resource access) behavior.

This paper makes tooling a first-class concern by having services encoded with **tasks** whose interactions reliably capture all non-deterministic behavior needed by tools. Task interactions are then exposed in **aspects** that are useful in encoding cross-cutting behavior; combined, tools encoded as *task aspects* can integrate with services effectively and transparently. We show how task aspects can be used to ease the development of an online production data service that runs on a hundred machines.

1 INTRODUCTION

Scalable fault-tolerant cloud services are notoriously difficult to develop and maintain. Service developers are in continuous battle against faults like software bugs, hardware failures, and mis-configurations, losses of which can lead to significant financial loss [24]. Tools play a very important role in improving the cloud service robustness. Failure injection tools test how services handle various failures [34]; model checkers systematically explore state spaces to uncover misbehaviors [51, 35, 37]; and assertion tools check global invariants [32, 6]. When errors occur, record-replay tools are used to reproduce bugs [20, 30, 44]; and trace analysis tools diagnose performance anomalies [21, 19, 42].

Unfortunately, tooling is currently an afterthought in cloud service design. Invasive approaches to tool integration is expensive and tedious, often resulting in tools not being used at all. On the other hand, transparent integration approaches sacrifice tool effectiveness through the use of inappropriate mechanisms; e.g. system call interception [23] can be unreliable, miss important service interactions as well as capture interactions that are too low level for tools to efficiently reason about.

So that tools are both effective and can be integrated with services transparently, tooling must be considered as a first class concern in cloud service design. In a study of cloud service tooling, we found that non-deterministic behavior such as asynchronous, concurrent, and IO operations are the primary focus of these tools. Accordingly, we propose a new programming model where cloud services are encoded using **tasks** whose interactions are both useful and available to tools. Tasks are contiguous chunks of deterministic execution that, to service developers, resemble event handlers; all non-deterministic behavior is then exposed to tools as **task interactions**.

We leverage *aspects* [25] to enhance services with tooling features that essentially "cross cut" their implementations. Tools can reliably impose on service behavior through *join points* that correspond to task interactions. Through *state extensions*, tools can also inject and propagate state along join point occurrences, allowing for aggregate reasoning about service behavior. Finally, tools can customize *task scheduling* for precise control over when and how tasks will execute, which is useful for tools like replay that manipulate task flows more invasively. Aspects that manipulate services through their tasks, i.e. *task aspects*, then provide a very reliable and complete abstraction for expressing tools.

This paper presents Zion where services are implemented against an enriched task model and all non-deterministic behavior is gated through a concise set of Zion APIs. Tools are then encoded as task aspects that can transparently and effectively manipulate service behavior. Zion coherently integrates a variety of tools into one platform so that they are applicable across multiple stages of service development and deployment.

We used Zion to successfully develop a production data service that runs on a hundred machines, tooling it with global assertions, fault injection, model checking, configuration-guided unit testing, record/replay, and so on. We also ported to Zion services like memcached [17] and LevelDB [13] to show that the platform is general. Based on our experiences in real production development and in porting popular online services, we conclude that tooling in Zion is both effective and transparent.

The paper continues in Section 2 with observations on current tooling practices for cloud services. Section 3 presents our cloud service framework, Zion, showing the value of tasks and the effectiveness of task aspects. Section 4 elaborates on Zion's task-centric platform while Section 5 evaluates Zion on a real production service as well as other services to show breadth. Related work is surveyed in Section 6 and Section 7 concludes.

2 Tales From the Trenches

Services are often initially written without much concern for tools, which are then either *invasively* adapted into each service as needed, or integrated with multiple services *transparently*. The invasive integration approach is very expensive in terms of development effort, and often results in useful tools not being used at all. For example, imposing an implementation-level model checker on a service requires knowing the service's semantics to identify its states and actions [26, 36]. Likewise, fault injection often requires that developers tediously identify where faults should be added to service code.

On the other hand, transparent approaches often sacrifice tool effectiveness through interposition granularities that are poorly suited to tooling needs. Integrating a model checker into a service transparently via system call interception causes it to capture uninteresting system behaviors that increase exploration time for no benefit. For example, loading a library involves manipulating runtime locks, while RPC involves low-level network operations, both of which pointlessly increase the model checking state space enormously. Likewise, doing fault injection via system call instrumentation alone can only introduce low-level failures, making the injection ineffective. A fault injection tool can introduce network message loss in a replication service to simulate network failure; but if the service automatically resends lost protocol-level messages, a network-level failure will not reliably trigger higher-level failure recovery logic.

Transparent approaches also suffer from service platforms that are uncooperative and not supportive of these approaches. Consider a record-replay tool that must capture enough interactions to faithfully reproduce program executions. Using system call interception, the tool developer has to manually inspect *all* system calls to mark those with side effects; e.g. they must go through more than 800 such API calls for Windows [20]. Similarly, Linux 3.14 has 326 system call entries [31]. Additionally, hidden system invariants can be broken by tools executing along side system calls; e.g. both Linux and Windows API use thread local storage to store auxiliary data such as error codes that can be polluted by tool code.

There are also serious challenges in ensuring that tools are integrated correctly at all. We applied a distributed trace analysis tool similar to Dapper [43] to diagnose correctness and performance issues in a map-reduce like system and its underlying replicated storage. So our tool could capture the system's complete execution flow across multiple machines, we manually inspected its source code, modifying places where asynchronous flow transitions are made.Although most of the system's asynchronous transitions were implemented in utility libraries, we found 18 places that needed to be modified in non-utility modules still without any assurance that other places were yet to be found.

What can be learned from these experiences? First and foremost, the current situation where tools are basically **afterthoughts** in service design is very problematic. Tooling instead must be considered as a first-class concern in the design of a service if tools are to be effective. However, transparency between service and tools is strongly desired so that tool integration is economical. Both of these seemingly contradictory requirements can be satisfied by constraining the service programming model to ensure that all service behavior relevant to tools is somehow exposed. For tools that focus on robustness, of especial relevance are non-deterministic behaviors such as concurrent, asynchronous operations (such as asynchronous I/O), as these are the areas where most robustness problems occur. Additionally, to ensure that tools themselves are correctly integrated, the programming model should also guarantee that all such behaviors are gated through principled programming model points that tool developers can reasonably enumerate.

The grain at which service behavior is exposed heavily influences tool effectiveness. Exposing low-level behaviors can kill tool efficiency as well as miss significant higher-level interactions. The grain at which a service's non-deterministic behaviors must be exposed also does not correspond cleanly to procedure call boundaries, and so a tool's functionality "cross cuts" the grain of a service implementation. The rest of this paper acts on these lessons in the design of a new service platform.

3 Tasks and Aspects

According to Section 2's lessons, services should be developed with tooling in mind from the start. We observe that constraining the service programming model can guarantee that relevant behavior is always exposed to tools. Services in *Zion*, our cloud service platform, are encoded using explicit *task* constructs whose interactions are useful and available to tools. A task is a contiguous chunk of deterministic execution that, to service developers, resembles an event handler. Tasks execute concurrently on thread pools, where, as discussed later, their scheduling can be customized to encode tools like replay.

As an example of a service built with Zion's tasks, Figure 1 shows the simplified pseudocode of a real-world replicated storage service[1]. To achieve high availability, the storage service replicates data on multiple machines using four tasks that are defined using the task keyword. Figure 2 illustrates one execution flow through these tasks: a ClientWrite task on the primary replica initiates an RPC call that is handled by a Prepare task on a secondary replica by writing the mutation to disk; after the write is completed, a LogCompleted task on the secondary initiates an RPC reply that is handled by a PrepareAck task back on the primary.

Looking at Figure 1, Zion's task model would be quite familiar to service developers who often handle events and deal with asynchronous method calls; the novelty of the model lies solely in how it supports tooling. Tasks in Zion are designed to expose service behaviors that our experience in Section 2 indicates are necessary and relevant to robustness tools. In particular, we found that non-deterministic behavior such as asynchronous, concurrent, and IO interactions challenge many of these tools. All such behavior then occurs through a concise set of service APIs so that they can be exposed to tools as *interactions* between tasks. Consider this call in Figure 1's ClientWrite task definition:

```
Rpc.Call(RPC_PREPARE, mutation, PrepareAck)
```

This call uses Zion's Rpc.Call API to initiate RPC on a primary replica, asynchronously causing a Prepare task on a secondary replica to handle the RPC call. The call also specifies that a PrepareAck task on the primary will handle an RPC reply to the call made by the secondary. APIs like Rpc.Call that involve non-deterministic behavior must be Zion service APIs, while APIs that are deterministic, like those that do string manipulation, do not as they are not of interest to tools.

Tools as Task Aspects

By gating all non-deterministic behavior through Zion's service APIs, they can be exposed to tools without invasive service modifications. But how can these behaviors be effectively presented to tools? As discussed in Section 2, an approach like system call interception does not correspond very well to the service-level non-deterministic behavior that tools need to manipulate. Instead, we leverage *aspects* [25] to transparently add tooling features to a service that essentially "cross cut" its implementation. With aspects, principled points in a program's execution are manipulated by aspect constructs.

[1]For presentation clarity we use Python-like pseudocode, although services and tools are actually written in C++.

```
service Replication:
  Rpc.Register(RPC_CLIENT_WRITE, ClientWrite)
  Rpc.Register(RPC_PREPARE, Prepare)

  task ClientWrite(mutation):
    if Role != Primary: return
    ... # log mutation
    for secondary in replicaList: # ↓ join with Rpc::OnCall (a)
      Rpc.Call(RPC_PREPARE, mutation, PrepareAck)

  task Prepare(mutation):        # ← join with Rpc::OnRequest (b)
    if Role != Secondary: return # ↓ join with File::OnCall (c)
    File.Write(LPC_LOG, mutation, LogCompleted)

                                 # ↓ join with File::OnComplete (d)
  task LogCompleted(mutation, err, length)
    if err: ... # handle local failure
    else if Role == Secondary:   # ↓ join with Rpc::OnReply (e)
      Rpc.Reply(RPC_PREPARE_ACK, err, ...)
    else: ...

  task PrepareAck(err, req, ack): # ← join with Rpc::OnResponse (f)
    if err or ack.err: ... # handle timeout or remote failure
    else: ...              # ack secondary; commit
```

Figure 1: Zion pseudocode for a replication service; join point occurrences are annotated in comments (#) with arrows pointing to the triggering statement.

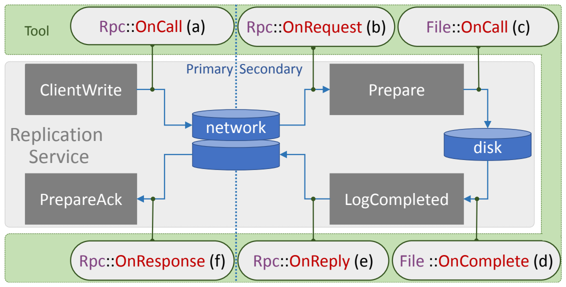

Figure 2: An illustration of how tasks (rectangles) and task aspect join points (capsules) execute in a flow starting from Figure 1's ClientWrite task; join point occurrences are labeled with letters that match those in Figure 1's comments.

In Zion, these principled points correspond to task interactions, where tools are encoded as *task aspects*.

Task aspects manipulate task interactions in three ways. First, kinds of task interactions are exposed as event-like *join points* that can be acted on by join point actions (aka "advice" [45]). Second, task aspects can use *state extension* to inject and propagate tool-specific state across multiple join point occurrences, allowing for comprehensive reasoning about service behavior. Third, task aspects can customize *task scheduling* for precise control over when and how tasks will execute on a thread pool. The rest of this section discusses these mechanisms.

```
tool FaultInjector:
  do "Drop" on Rpc::OnCall(inTask, msg, replyTask):
    if Env.Random32(...):
      replyTask.Timeout() # timeout reply because it will never run
      return false        # do not send RPC message via msg
    else: return true     # no fault, all is normal
  do "Unused" on Rpc::OnRequest(msg, requestTask):
    # no fault injection for Rpc::Request
  do "FailWrite" on File::OnCall(inTask, fileTask):
    if Env.Random32(...):
      file.SetError(...)  # IO will never complete
      return false        # return false to not do IO
    else: return true
  do "SlowWrite" on File::OnComplete(fileTask):
    if Env.Random32(...):
      file.Delay(...) # make IO slower
  do "FailReply" on Rpc::OnReply(inTask, msg):
    if Env.Random32(...):
      return false # do not send RPC reply
    else: return true
  do "SlowRpl" on Rpc::OnResponse(msg, replyTask):
    if Env.Random32(...):
      replyTask.Delay(...) # make RPC reply slower
```

Figure 3: Leveraging join points to inject faults.

Join Points

A task aspect encoding of a fault injector tool is shown in Figure 3. This tool can inject faults into any service built on top of Zion, including the replication service of Figure 1. An action on a join point is expressed using the do keyword in the body of a tool, where the join point being acted on proceeds the on keyword; e.g.

```
tool FaultInjector:
  do "Drop" on Rpc::OnCall(inTask, msg, replyTask):
    ... # do manipulation here
```

The fault injector is defined as a task aspect (keyword tool) with a join point action labeled "Drop" that does work at Rpc::OnCall join point occurrences. Each join point has parameters that depend on the semantics of the join point; e.g. the Rpc::OnCall join point is parametrized by inTask that initiates the call; msg that is the message being sent; and replyTask that will eventually handle the reply to the RPC call. For a join point occurrence that handles the call Rpc.Call(RPC_PREPARE, mutation, PrepareAck) in Figure 1, inTask is the ClientWrite task that initiates the RPC call; msg is formed from a mutation message, and replyTask is bound to a PrepareAck task created to handle the reply. Figure 2 (a) – (f) illustrates how the six join points acted on in Figure 3 occur in the replication service's execution, which are also annotated in Figure 1.

```
tool DependencyTracker:
  do "TraceA" on Rpc::OnCall(inTask, msg, replyTask):
    causalEvent = # label: node Id, thread id, logic clock
      { nodeId, threadId, ++clock }
    MsgStateExt::Set(msg, causalEvent) # attaching
    return true
  do "TraceB" on Rpc::OnRequest(msg, requestTask):
    ++clock # advance the logic clock and retrieve the label
    causalEvent = MsgStateExt::Get(msg)
    TaskStateExt::Set(requestTask, causalEvent)
    ...
    return true
```

Figure 4: State extension for dependency tracking.

As task aspects, tools can insert behavior into a service's execution via join points without requiring service modifications. Tools also impose on service behavior at a more appropriate granularity, abstracting away uninteresting lower level details. In our example, fault injection can manipulate RPC operations directly rather as opposed to lower-level network operations that could then be counteracted by RPC's error recovery code.

State Extension

Join points alone cannot expose aggregate service behavior to tools. Consider that in Figure 2, imposing just on Rpc::OnCall (a) and Rpc::OnRequest (b) cannot reveal to a tool that the Prepare task that handles the RPC call on a secondary replica is causally related to the ClientWrite task that initiated the RPC call on the primary replica. Task aspects can provide for this awareness by extending tasks, and task-oriented objects like RPC messages, with state that can be propagated along task execution flows.

The code in Figure 4 extends tasks and RPC messages with state to perform dependency tracking using "state labels" MsgStateExt and TaskStateExt. Later join point occurrences can read the labels of earlier join point occurrences when they share arguments. For the execution flow of Figure 2, the Rpc::OnCall (a) join point occurrence shares its msg parameter with the next Rpc::OnRequest (b) join point occurrence, even though msg travels across the network between them. The "TraceA" join point action in Figure 4 attaches a causal event to msg, and so the "TraceB" join point action in Figure 4 is able to "propagate" the causal event attached from msg in "TraceA" to the requestTask that will handle the RPC call on the secondary. The requestTask will then become input to a later File::OnCall (c) joint point occurrence, where it can be propagated further (not shown).

With state extensions, tools can inject customized state into task interactions at join points. Zion applies

```
tool Replay:
  enqueue(inTask):
    causalEvent = TaskStateExt::Get(inTask)
    readyTasks[causalEvent] = inTask
    if blockingThreads.ContainKey(causalEvent):
      # task for causalEvent is ready, signal blocked thread
      blockingThreads[causalEvent].Signal()
      blockingThreads.RemoveAt(causalEvent)
  dequeue():
    causalEvent = LocalThreadLog.GetNext()
    while(true)
      if readyTaks.ContainKey(causalEvent):
        outTask = readyTasks[causalEvent]
        readyTasks.RemoveAt(causalEvent)
        return outTask
      else:
        blockingThreads[causalEvent] = CurrentThread()
        # wait for the task corresponding to causalEvent to be ready
        CurrentThread().Wait()
```

Figure 5: Zion task aspect pseudocode for a replay tool.

state extension to tasks, RPC messages, and synchronization constructs (locks and semaphore) that connect tasks indirectly (Section 4). Tools can then aggregate multiple task interactions to uncover service behavior that cannot be observed in single task interactions alone.

Custom Task Scheduling

Many tools that focus on robustness must impose on how tasks are scheduled in order to be effective. For example, during the replay phase of a record-replay tool, replay must impose the same task scheduling order that occurred in a recorded execution. Figure 5 shows how a replay tool can be implemented through custom task scheduling. In this pseudocode, dequeue and enqueue statements replace original task scheduling policy with one that ensures tasks are replayed in their recorded order. When a task is scheduled to run using enqueue, it is put into a readyTasks dictionary according to the causal event associated and propagated with the task in Figure 4. LocalThreadLog is then used in dequeue to ensure tasks execute in a previously recorded order. If tasks are enqueued out of the recorded order, the thread executing the task blocks until the task associated with the next expected causal event is enqueued.

Combined with join points and state extension, custom task scheduling is very powerful. Besides replay, custom task scheduling can prioritize the disk IO requests of interactive jobs over long-running jobs by using state extension to differentiate between both kinds of requests. Also, to ease debugging, a tool can also ensure that only one task runs at a time; in this case, a custom task scheduler coordinates what task to execute while Task::OnBegin and Task::OnEnd join points are used to acquire and release a scheduling token that ensures only one thread is executing a task at a time.

Together, join points, state extension, and custom task scheduling allow tools to impose on service behavior in powerful and reliable ways. By focusing on task interactions, tools are exposed to service behavior at a grain appropriate to their needs, abstracting over lower-level details that would otherwise make them less effective. Tools are also guaranteed access to all of a service's non-deterministic behavior, and so can be implemented reliably. We provide more details about Zion's task-centric architecture in Section 4 while we validate how Zion enhances tooling for services in Section 5.

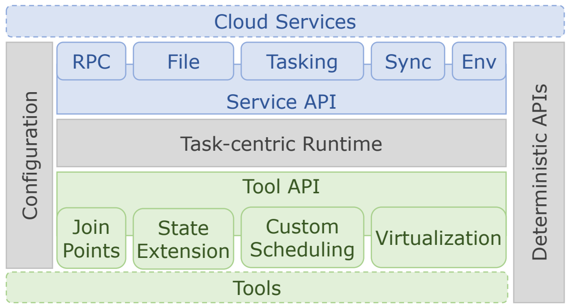

Figure 6: Zion's architecture.

4 A Task-Centric Architecture

Zion's architecture is illustrated in Figure 6. Cloud services are built on top of Zion's RPC, file, synchronization, tasking, and environment APIs. Cloud services can also use other APIs that do not involve non-deterministic behavior, such as hash functions like MD5 or SHA1; these do not need to be gated through Zion as their behavior is uninteresting to robustness tools. Tools are written according to join points, state extension, and custom task scheduling that we introduced in Section 3.

Because many tools like model checkers also rely on virtualizing system resources (e.g. disk or network), Zion also provides built-in virtualization support. Tools mainly use virtualization to make the underlying system deterministic, and so it is not configured by tools in diverse ways. Between the tool and service APIs is a task-centric runtime that owns task-executing thread pools, the RPC engine, and component wiring. Zion also supports the configuration of which tools are applied to services, and how services are deployed; e.g. within one process or on many machines. Configuration also allows control over how tasks are dispatched to thread pools.

A discussion of virtualization and configuration are

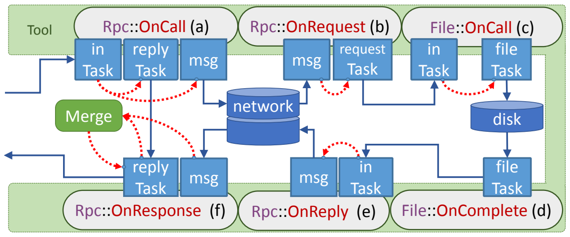

Figure 7: An illustration of how state is propagated for the task flow of Figure 2; solid edges connect shared parameters while curvy dashed (red) edges is tool-induced propagation.

beyond the scope of this paper, and we only mention them here for completeness reasons. Zion provides APIs to interact with an environment; e.g. to procure a random number or get the current time. Most kinds of tools do not need to deal with these APIs unless they require them to be deterministic, in which case virtualization is usually sufficient. The rest of this section elaborates on the RPC, File, Tasking, and Synchronization subsystems.

RPC and File

The RPC and File APIs provide services with respective IO capabilities. All APIs in these two sub-systems are captured by join points: Rpc.Call by Rpc::OnCall; Rpc.Reply by Rpc::OnReply; File.Write and File.Read by File::OnCall. API join points are paired with join points that impose on the scheduling of API call completing tasks. The Rpc::OnCall join point is paired with Rpc::OnRequest, which imposes on the scheduling of a task to handle a RPC call; likewise Rpc::OnReply is paired with Rpc::OnResponse while File::OnCall is paired with File::OnComplete.

Understanding Rpc and File join points relationships is crucial in writing effective tools. Relationships between join point occurrence parameters are illustrated in Figure 7 for the flow of Figure 2 in Section 3. It can be seen in this figure that the msg parameter of Rpc::OnCall (a) is shared by Rpc::OnRequest (b), so any state propagated through an RPC call must be attached to msg. Zion will transmit state attached to msg through the network to Rpc::OnRequest, although developers must provide serialization logic as needed. Other relationships are set by call order; e.g. the requestTask parameter of (b) becomes the inTask parameter of (c) in Figure 7 because that is what the task does next.

Except for state that is copied through the network, state can be propagated by reference; this state must be protected because it can be accessed concurrently. RPC presents another challenge: the Rpc::OnResponse join point receives the message from Rpc::OnReply, but also uses replyTask from Rpc::OnCall! There is then two

```
tool LockDependencyTracker:
  ... # from Figure 4
  do "LogAcquire" on Lock::OnAcquire(inTask, lock):
    ++clock
  do "LogAcquired" on Lock::OnAcquired(lock,contTask):
    LockStateExt::Get(lock).ToolLock.Acquire()
    Log(lock, ACQUIRED)
    LockStateExt::Get(lock).ToolLock.Release()
  do "LogRelease" on Lock::OnRelease(inTask, lock):
    ++clock
    LockStateExt::Get(lock).ToolLock.Acquire()
  do "LogReleased" on Lock::OnReleased(lock, contTask):
    Log(lock, RELEASED):
    LockStateExt::Get(lock).ToolLock.Release()
  do "LogTryLock" on Lock::OnTryAcquire(inTask, lock):
    ++clock
    LockStateExt::Get(lock).ToolLock.Acquire()
  do "LogTried" on Lock::OnTried(lock, success, contTask):
    if (!success):
      while(LockStateExt::Get(lock).LastKind == RELEASED)
        LockStateExt::Get(lock).ToolLock.Release()
        LockStateExt::Get(lock).ToolLock.Acquire()
    Log(lock, TRIED)
    LockStateExt::Get(lock).ToolLock.Release()
  def Log(lock, kind):
    LockStateExt::Get(lock).LastKind = kind
    RecordCausalEvent({ nodeId, threadId, clock } )
```

Figure 8: Pseudocode for recording lock behavior.

versions of any state propagated completely through an RPC call and reply that must be "merged" by the tool.

Tasking and Synchronization

Zion provides service APIs to explicitly fork ("enqueue") new tasks that run concurrently, allowing work to be divided up as appropriate. Tasks can also explicitly wait on other tasks; since this is a task interaction, Zion splits the current task and spawns a new **continuation** [22] task that executes the current task's remaining logic after waiting completes. Waiting is exposed to tools by a pair of join points that mirror those for RPC and File APIs.

Generic join points are provided to tools to capture when a task begins (Task::OnBegin) and ends (Task::OnEnd) executing, which is useful for imposing on all task executions. As mentioned in Section 3, tools can also customize task scheduling to implement tooling features. Each thread pool that executes tasks is associated with its own scheduler that can be customized separately, the details of which are beyond this paper's scope.

Because tasks can run concurrently, Zion provides familiar synchronization APIs, e.g., locks or semaphores,

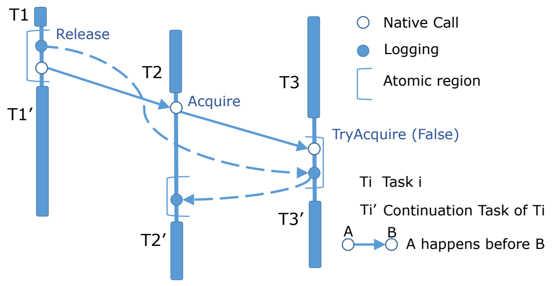

Figure 9: Capture causal dependencies between tasks via locks. The dotted lines are the captured (false) dependencies while the solid lines are the ground-truth.

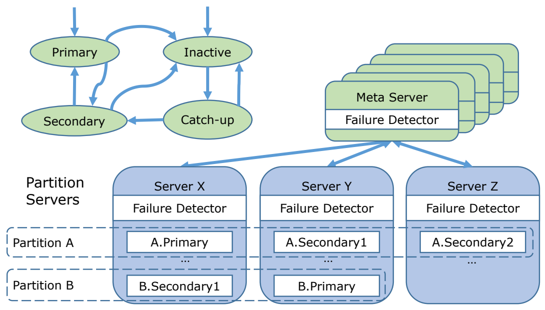

Figure 10: Architecture of Moab. Upper-left corner shows the status transition graph of a replica.

to protect the state that is shared between tasks. Similar to task waiting, tasks fork continuation tasks when they invoke synchronization APIs; Zion exposes these behavior to the tools with joint point pairs that captures before the API is called and when the continuation task is scheduled. Consider the pseudocode of a recording tool shown in Figure 8 that records the causal order of a service's lock usage. Unlike the direct causal order recorded in Figure 4, lock causal order is non-deterministic since task worker threads can be interleaved in different ways. Consider the execution illustrated in Figure 9 where three tasks interact with each other; solid lines are the ground-truth of their causal dependencies. Merely recording each invocation of lock methods in join points can capture false dependencies as shown by dotted lines within the figure: recording after a Lock.Acquire call can be executed after a Lock.TryAcquire call has begun.

To solve this problem, state extension is used in Figure 8 to attach a ToolLock to the service-visible lock object (lock) that is then acquired by join point actions; recording is now atomic. However one corner case remains: because Lock.Acquire is a blocking call, we cannot really use ToolLock to lock its invocation. When Lock.TryAcquire fails to acquire the lock, it must wait until recording for Lock.Acquire is finished. The "LogTried" join point action in Figure 8 must therefore wait until the last locking operation is from the "LogAcquired" join point action, which is accomplished using state propagation via LastKind.

Dealing with locks in Figure 8 can be tricky, where this trickiness is unavoidable in general. However, task aspects provides to tools all the mechanisms needed to deal with them: join points to capture their execution, state extension to propagate state through task and lock objects, and custom scheduling to manipulate locking orders as needed. Through these join points, Zion can also convert service executions into task execution even in the presence of blocking calls, greatly simplifying the building experience of tools as shown in Section 5.

5 Experience and Evaluation

We have implemented Zion's task-centric runtime, service API, and tool API in around 7,700 lines of C++ code. The number of the service and tool APIs are 18 and 40 respectively, including API calls and interface classes. Over the past one year, we have used Zion in the development of *Moab*, a production distributed and replicated table service; we have implemented and are using a set of useful tools in the development, testing, debugging, diagnosis, and pre-deployment of this service. We have also ported five popular legacy cloud service components to Zion to understand how close our programming model is to those used in common practice.

Applying Zion to Moab

With the lessons of Section 2 in mind, we initiated the Zion project at the same time that Moab began development to solve service-tool integration problems by considering tooling as a first-class concern. For the needs of this paper, Moab can be regarded as implementing a distributed reliable table service that supports both query and update on table entries. Figure 5 shows Moab's high-level architecture. A (large) table is split into a large number of partitions that form a *replication group* and are replicated on a set of *partition servers*. Each replication group runs a primary/backup protocol, relying on a Paxos-based *meta service* to manage membership. A partition server participates in the replication groups of multiple partitions, taking different roles as needed.

Roles and their transitions are shown in Figure 5 (upper left part): normally a single *primary* orders update requests, committing them on all replicas; a replica can be removed from the group to become *inactive*; and a

Tool	Phase	LOC	Mechanism
Failure injection	D, O	200	J
Model checking	D	3.9K	J, S, T, V
Targeted flow checking	D, O	450	J, S
Config-based unit testing	D	550	J
Task tracing & profiling	D, O	270	J
Single-box debugging	D	430	J, T, V
Global assertion checking	D	90	J
Record and replay	D, O	1.4K	J, S, T, V

Figure 11: Tools we built on top of Zion, including their names, usage phase, line of code (LOC), and used mechanisms; D and O stand for whether the tool is used during Development and/or when the service is Online; and J, S, T, and V are short for Join points, State extension, Task scheduler, and Virtualization, respectively.

new partition must through the *catch-up* role to get its state up-to-date before joining as a *secondary*. The meta service monitors each replication group, making changes for load balancing and failure handling; e.g. on primary failure, a secondary is upgraded to be the new primary. Monotonically increasing *decrees* are attached to membership configurations to indicate which is newer.

Moab is implemented in around 234K lines of C++ code on Zion; 50K lines of code is related to replication, implemented using 52 different kinds of Zion tasks. Correctness of replication is crucial to data consistency in Moab, which is challenged by performance optimizations to achieve low-latency query and update requests. Tools are then very important in understanding, testing, debugging, and diagnosing replication related issues.

Figure 11 lists the tools that we have built on Zion, some used during development/testing and others used during deployment—Moab has been pre-deployed on a cluster of 100 machines, each running 50 instances of partition managers that simulates a testbed of 5000 machines. Moab will be deployed to over ten thousands of machines within months. Also listed for each tool are lines of code (LOC) as well as the Zion mechanisms (i.e. join points, state extension, task scheduler, and virtualization) used. Because Moab is designed to maintain correctness in the face of a variety of failures, we focus here on failure-injection type tools. Where we showed how basic failure injection capabilities can be built using join points in Section 3, we now highlight the advanced failure injection capabilities that we have built on Zion for Moab like model checking, targeted flow checking, and config-based unit testing. Some of those capabilities leverage other tools, such as a global assertion checker; we will describe them briefly when used.

Advanced failure injection

Failure injection is a powerful mechanism [34] that is particularly appropriate for validating Moab's replication implementation. With Zion, failure injection is done through joint point actions (e.g. on a File::OnCall to mimic disk failures and on File::OnCompleted to mimic slow disk operations) or a special task (e.g. Replica::Close to mimic crashes). In our test deployment, failures are injected randomly and task tracing capability built on Zion is leveraged to capture traces related to any misbehavior.

We use implementation-level model checking to explore different service behavior systematically by controlling non-determinism such as scheduling and deciding when to inject failures. Better yet, Zion further enables us to easily build other advanced failure injection capabilities that leverage application semantics to thoroughly explore interesting state spaces. We describe these advanced failure injection capabilities here.

Model checking. We built an implementation-level model checker on Zion that is also capable of injecting failures systematically. We previously implemented a model checker [51] using system-call interception where the contrast between the two approaches is particularly revealing. The new model checker required only one person-month of work with 3,700 lines of code in total, while the previous one took more than a year of work, handling nearly 200 common system calls that alone requires more than 22,000 lines of code.

Figure 12 uses file read as an example to compare model checking using system call interception with that using Zion. Figure 12(a) shows the procedures for intercepting the ReadFile and SleepEx system calls. For ReadFile, callback must be exposed to the model checking scheduler so the model checker can gain control. OS data structure used in callback must also be translated to the model checker's *action* data structure IoCallbackAction. In reality, there are five different kinds of asynchronous I/O methods that must be handled; we only show IoCallbackAction here. Similarly, SleepEx must expose a ContinuationAction that further needs to get instruction from the model checker scheduler to resume (as done in WaitSchedule). One additional complication is that the model checker scheduler must track the constraints associated with certain actions; e.g. an IoCallbackAction requires that the caller thread be waken up from an *alertable* state.

In contrast with Zion, the callback fileTask in File::Read is already a Zion task; nothing extra needs to be done. The TP_TaskScheduler, which manages the thread pool, simply exposes tasks to the model checker and schedules task as instructed. Model checking with Zion uses tasks as actions, while the system call interception is forced to

```
bool MC_ReadFile(hFile, buffer, count, offset, callback) {
  ... r = ...
  auto action = new IoCallbackAction(callback, ...);
  action->Submit();
  return r;
}
void MC_SleepEx(...) {
  auto action = new ContinuationAction();
  action->Submit();
  while ((act = WaitSchedule(FLAG_ALERTABLE)) != action)
    act->Execute();
}
```
(a) system call interception approach.

```
void File::Read(hFile, buffer, count, offset, fileTask);
class TP_TaskSchduler {
  void Enqueue(task) {
    (new TaskAction(task))->Submit();
  }
  Task Dequeue() {
    return WaitSchedule().Task;
  }
};
```
(b) Zion approach.

Figure 12: Comparison of model checking with the system call interception approach and the Zion approach.

define actions at the system call level. Our statistics for a 10-minutes stress test of Moab show that an action with Zion (i.e. a task) requires 5.39 actions, on average, at the system-call level; e.g. a RPC caller task involves 10 lock operations and 2 socket related system calls. This allows the Zion model checker to search a much smaller state space at the task level, making it more effective.

Targeted flow checking. Although model checkers are effective in exploring corner cases, the state space is generally too large for them to be exhaustive. We therefore build other advanced tools to inject failures systematically like target flow checking that aims to exhaustively explore a space along one task flow.

Figure 13 shows a task flow of how a replica transitions from a catch-up role into a secondary by learning all committed requests from the primary. The flow starts with the meta server sending a proposal to the primary node to ADD_SECONDARY (①). The primary then instructs a new replica to enter the catch-up role to start learning from the primary (②,③). The primary sends some meta information about what should be learned, e.g. the collections of on-disk files (④). The catch-up replica starts an asynchronous generic task in a separate thread pool to pull these files from remote machines (⑤). The task notifies the replica once all remote files are copied to the local machine (⑥). After that, the replica tries another round of learning just in case the primary

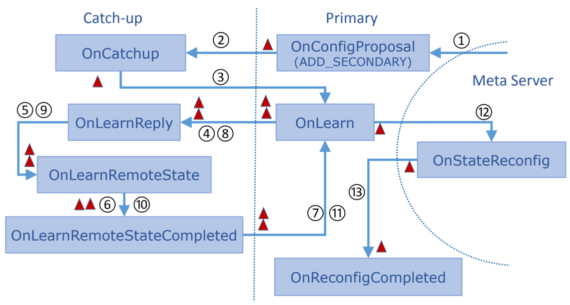

Figure 13: Learning task flow in Moab. The numbers represent the flow steps and the triangles are the join point occurrences where fault injection is applied.

has generated new files during the learning process (⑦–⑪). When the primary decides that the catch-up replica's state is ready to be served as a secondary, it updates the replication group membership on the meta server (⑫,⑬). At this point, learning is completed and the status of the catch-up replica will be lazily updated once it gets the new membership configuration. The task flow graph is constructed from the output of our tracing and profiling tool using join points.

This task flow captures an important part of the replication protocol and so it is meaningful to inject failures at various flow points to check robustness. For the task flow in Figure 13, there are 15 join point occurrences (some join points are executed twice), and so, to inject a single failure, we only need to explore 15 failure cases. When injecting multiple failures, the task flow graph can branch after the first failure injection. Our tool keeps track of branching and can choose to inject failures further in that branch based on the specified policy.

Targeted flow checking uses state extension to track the task flow graph, where the graph is constructed along the execution by attaching and propagating state. It further uses join points to inject failures.

Config-guided unit test. Replication in Moab can work in different replication-group configurations, handle different kinds of failures, and at the same time process update requests. Unit tests must be constructed to test the code for these different cases. In particular, the unit test must look at different configurations: with up to three replicas, we can have 6 configurations: (P), (P,S), (P, S, S), (S), (S, S), (S, S, S), where P is a primary and S is a secondary. In any of those configurations, it will inject a set of pre-defined failures, such as disk failure, network disconnection, and node crash, and at the same time possibly process an update request. Writing such unit tests would usually require adding lots of code into the service

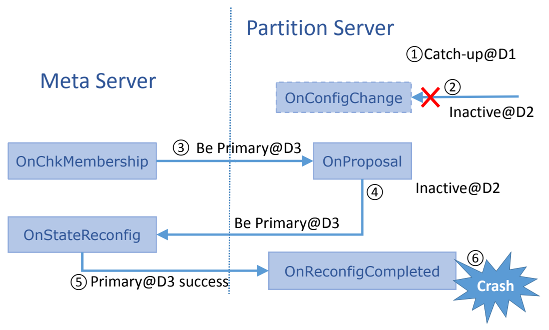

Figure 14: A simplified task flow for a bug triggered by using model checking.

to control execution, check global state, and inject failures systematically; but this becomes feasible with Zion.

Our unit test tool leverages the single-box debugging tool that we developed on Zion, which makes all service nodes run in a single process with serialized task execution and logical time to avoid false RPC timeouts and concurrent state updates during service debugging. It does this with a custom task scheduler coordinated with join point actions in Task::Begin and Task::End, as well as a virtualized network. The single-box debugging tool also serves as a platform where other tools can be built on top and integrated; e.g. our unit test tool further uses the global assertion checking tool, built on single-box debugging, that can inspect the global state at consistent points by using Task::End join point actions.

The unit test is constructed by getting execution following a specified sequence of steps into a targeted configuration and then injecting failures systematically. The unit test also monitors if the system can be fully repaired, reporting an error after a configured time interval.

Tools in action

No single tool is a panacea, and multiple tools are usually combined to expose bugs more effectively and find root causes more quickly. Zion makes it easy to use tools together as they share the same underlying platform. Here we describe how we used a set of tools together to discover a bug, analyze a trace, and find its root cause.

We first ran the model checking tool with fault injection enabled. After making 4,224 scheduling decisions, the tool flagged a bug identified by a local assertion on a partition server. Figure 14 shows a simplified task flow of how this bug occurs. A catch-up replica at configuration decree 1 (marked as D1) on a partition server is learning state from the primary (marked ① in the figure). The primary removes itself due to a disk failure injected by our tool, causing all catch-ups to abort and transition into inactive in the new configuration at decree 2 (D3). The meta server sends a OnConfigChange RPC message to all catch-ups to stop their learning process, which is further dropped due to failure injection (②). This creates an inconsistency between the meta server and the replica, which still believes that it is in the configuration.

The meta server runs periodic routines to checks memberships of each replication group, and decides to suggest that the catch-up replica be the new primary at decree 3 (D3) via an RPC message, which also piggybacks the current configuration for this replica (i.e, inactive at decree 2) (③). The OnProposal task handles the proposal (④). There is a map data structure on each partition server that tracks all replication groups that it participates in. The task handles a proposal in the group only after ensuring that the group is in that map data structure, performing necessary initialization if the group is not yet in the data structure. However, in this case, when processing the proposal, the replication group is in the map. The task then starts processing two operations in a batch, with the first one removing the replication group in the map data structure. The processing of the second operation assumes that the replication group is already in the map (while it is no longer true), leading to inconsistencies. But the bug is not exposed yet. After the replica updates the configuration with the meta server (⑤), the violation is detected (⑥) in the task OnReconfigCompleted with an assertion checking that any active replicas (e.g. primary) must be included in the map.

It was hard to figure out what caused inconsistencies when assertion failures occur. We first used the tracing tool to build a task flow graph, verifying flow's validity itself. The task flow graph exposes only structure at task granularity, which is insufficient to discover the root cause. We then replayed execution using our replay tool, eventually finding the root cause through breakpoints.

Beyond debugging and testing tools. During Moab's development, we realized that task aspects can be applied beyond debugging and testing, like performance and model-based reasoning. Moab has a strict query latency SLA with a somewhat higher tolerance for write latency. Because replication can interfere with query processing, we wrote a custom task scheduler so that query requests are processed with higher priority, and write requests are throttled when the query latency is close to the upper limit of its SLA. We also built a simulator for Moab because adopting the task model in Zion leads to a naturally born queuing network [14], making it possible to do some performance analysis and estimation using the simulator.

Tooling experience summary. In general, we found that tasks and task aspects enable many tools. The task abstraction and limited pre-defined task interaction points greatly reduce the cost of building tools; the completeness of task interactions through a small tool API eases the correctness and/or reliability of the tools; and the tools are successfully applied to a service with almost no additional integration cost.

We found a few limitations to our approach. First, Zion assumes that tasks will acquire locks before accessing shared data, leading to the possibility of data races, which tools like replay are especially vulnerable to. Second, although Zion can easily capture task-level execution flow, it does not understand higher level semantics like batching. This prevents us from implementing some tools like tracking privacy and security information for a given request using state propagation. When task flow hits a batch operation, the flow ends and propagated information is lost, making it impossible to check later when the request is recovered from a batch.

Porting Services

We ported five popular services to Zion, memcached [17], LevelDB [13], Xlock (similar to Chubby [7]), a thumbnail server, and Kyoto Cabinet [16], with changed or added lines of code ranging from 200 to 500. The changed code mostly replaces original invocations of low-level system services to that provided by Zion, including locks, time, random, and file operations. For those services used as local components (e.g. LevelDB), we added code for wrapping their interfaces (e.g. read or write a key/value pair) as RPC request and response tasks. We also removed original thread code because all tasks are now redirected to Zion's thread pool for execution.

While porting was mostly straightforward, a couple of features did not fit into our task model very cleanly. The first was background threads; e.g. LevelDB uses a background thread that waits for jobs from write request handlers to compact disk files as necessary. To fit into Zion's task model, background threads are broken down by considering each job as a task that are executed in a separate thread pool.

The second problematic feature is synchronous IO where Zion only supports more batch-performance friendly asynchronous IO. However, many services use synchronous IO to optimize for latency while the use of flash disks already improves performance substantially. Task.Wait calls are then combined Zion's asynchronous interface to mimic synchronous IO.

Performance

We further measure the overhead of Zion and tools by applying them to the five ported services above. Overhead of some tools is not measured because it is applied only when services are not online (e.g. model checking) or overhead is not applicable (e.g. fault injection); we use a tracing and profiling tool as well as a replay tool as two representatives. The experiments are done on two machines, acting as client and server respectively, which have 4 cores at 2.0GHz with 8 hyper threads, 32 GB memory, 1Gbps network, and 3 TB hard disk. For each native version of the service (without Zion), the client sends RPC requests to the server with a large throughput so that the server always has 100% CPU utilization. The request type and size vary for each service, and the CPU, not network, is always the bottleneck. We then change the configurations on the server side, and measure the request throughput on the server. We tried five kinds of different configurations: (i) native; (ii) (Zion w/o ext) with Zion but without tooling primitives; (iii) (Zion w/ ext) with Zion and all tooling primitives enabled; (iv) (Zion T&P) with the tracing and profiling tool enabled for all tasks in the service; and (v) (Zion record) with the record phase of the replay tool. Because MemCached, KyotoCabinet, and LevelDB are local components, we instead use the second configuration as their base. For all cases, we use 8 threads to run all configurations for 5 minutes with 10 rounds, and observed that the CPU utilization is constantly 100% except MemCached where the CPU utilization is only around 50%, indicating bad scalability.

Figure 15 depicts the overhead of Zion under these five configurations. Our ported versions of Xlock and Thumbnail server actually have slightly better performance than the native version, mostly due to replacement of RPC related code. (We adapted the original network code to work with Zion's RPC engine to minimize the change of the underlying runtime.) As shown in figure, the overhead of Zion itself is negligible, both with or without tooling extensions. The overhead of the tools is acceptable ($< 8\%$) for all cases except MemCached recording, which is because memcached is not scalable and the tool's overhead accumulates across threads.

6 Related Work

This paper does not focus on traditional systems tools that detect memory leaks, detect data races, or handle misconfigurations, but rather it focuses on supporting tools that promote robustness in cloud services. Consider distributed trace analysis tools like Magpie [5], Pinpoint [9], X-Trace [19], Pip [42], and Dapper [43]; configuration and performance troubleshooting tools like

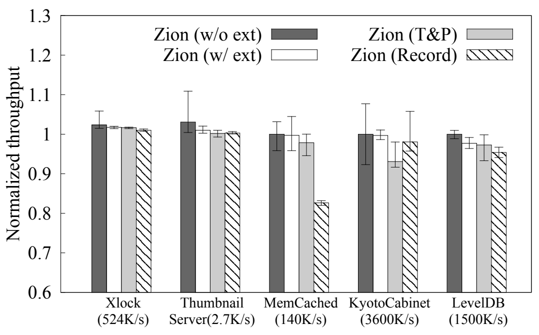

Figure 15: Overhead of Zion and the tools atop. The number below the service name is the request throughput we used.

ConfAid [3] and X-Ray [2]; failure injection tools like LFI [34] and AFEX [4]; record-replay tools like ODR [1], R2 [20], PRES [40], Respec [30], and DoublePlay [46]; and model checking tools like MaceMC [26], Chess [38], and CrystalBall [49].

Tools traditionally impose on program behavior in a variety of ways. Pip [42] relies heavily on application-level instrumentation, while Pinpoint [9] and Magpie [5] leverage instrumented libraries and middleware; Zion also leverages an instrumented runtime for transparent integration, but exposes task interactions to retain the effectiveness of application-level instrumentation. Many tools also leverage system call interception and binary instrumentation frameworks like [23, 41] to achieve transparent integration without runtime modifications, but as observed in Section 2, these techniques are often too unreliable and low level for cloud service tooling needs.

Failure injection has been recognized as an effective tool for testing system recovery code [34, 4]. LFI [34] proposed to inject failures at the library level to test recovery code effectively. AFEX [4] further proposes a metric-driven approach for fast black-box testing using fault injection. We also rely on failure injection heavily in our replication service while Zion allows us to efficiently explore interesting state spaces.

Some tools also propagate information along the communication paths. For example, ConfAid [3] does dynamic information flow analysis to automate configuration troubleshooting, which requires interposition on communication primitives to propagate taints. X-Ray [2] uses both deterministic replay and ConfAid for performance troubleshooting using performance summarization. Zion's task model, together with join points and state extensions, would be convenient for building such tools. Tools are often built with hardware support [39, 48] or virtual machines [15, 29] to record and replay all aspects of an application's environment, including scheduling decisions made by the underlying system; Zion tools can achieve this instead with just join points, state extension, and custom task scheduling.

Zion's task model descends from event-driven systems that can be more performant and robust than purely threaded systems; e.g. see libasync [12] and SEDA [47]. Libeel [11] observes that by making event callbacks explicit, the writing, reading, and verification of event handler code can be significantly simplified. Work in [28] shows how scheduling of an explicit event handling "task" construct can be customized for better performance. Zion goes beyond these works by showing how tasks can further be leveraged to tool cloud services.

There is a lot of work on platforms for building distributed systems like cloud services. Analogous to Zion, Mace [27] has unified APIs for networking and event handling, imposes a restricted programming model, and leverages aspects to enable model checking [26] and causal-path debugging. In contrast to Mace, Zion focuses on supporting cloud services with a wider range of tools. Similar to state extension in Zion, meta-applications in Causeway [8] can inject and access meta-data (state) along distributed system execution paths, which can then be used for scheduling or debugging. Causeway services, however, are implemented according to an actor model that is much less conventional than Zion's task model. Work in [50] propose a model where services expose environment-related "choices" to be made that are then maximized by a model checker [49]; a Zion tool could do this by imposing on task interactions.

Our work is inspired by aspect programming as pioneered by Kiczales et al [25]. Aspect techniques have been previously applied to systems; e.g. [10, 33] demonstrate that concerns that crosscut traditional OS layers can be cleanly defined as aspects, while [18] applies aspects to the chore of patching operating system behavior. Zion additionally shows how aspects can be combined successfully with tasks to tool cloud services.

7 CONCLUSION

We have shown how tools that enhance cloud service robustness can be effectively encoded as task aspects in Zion. By restricting the cloud service programming to an enriched task model where all non-deterministic behavior is gated through service APIs, non-deterministic behavior is completely and reliably exposed to tools as task interactions. Task aspects that impose on task interactions then provide a convenient way to tool services. We have successfully applied Zion to the development of a large-scale production replicated data service, and have found that it does indeed make tools easier to apply. Future work includes focusing on enhancing what is meant by a "task interaction" in Zion to provide better feedback to tools; e.g. by adding data flow tracking between tasks to better support information flow tools.

ACKNOWLEDGMENT

We would like to thank the PDI Data Service team in Microsoft Bing for the collaboration and valuable feedbacks of applying Zion.

REFERENCES

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proc. of SOSP*, pages 193–206, 2009.

[2] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proc. of OSDI*, 2012.

[3] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. of OSDI*, 2010.

[4] R. Banabic and G. Candea. Fast black-box testing of system recovery code. In *Proc. of EuroSys*, pages 281–294, 2012.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of OSDI*, volume 4, pages 18–18, 2004.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. of FMCO*, pages 115–137, 2006.

[7] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, pages 335–350, 2006.

[8] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proc. of HotOS*, pages 18–18, 2005.

[9] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. *Path-based failure and evolution management*. PhD thesis, University of California, Berkeley, 2004.

[10] Y. Coady, G. Kiczales, M. J. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proc. of FSE*, pages 88–98, 2001.

[11] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Proc. of HotOS*, pages 3–3, 2005.

[12] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proc. of ACM SIGOPS European Workshop*, pages 186–189, 2002.

[13] J. Dean and S. Ghemawat. LevelDB: A fast persistent key-value store. https://code.google.com/p/leveldb.

[14] R. L. Disney and D. KÃűnig. Queueing networks: A survey of their random processes. In *SIAM Rev.*, pages 335–403, 1985.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS*, 36(SI):211–224, 2002.

[16] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet.

[17] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, Aug. 2004.

[18] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Patch (1) considered harmful. In *Proc. of HotOS*, 2005.

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proc. of NSDI*, pages 20–20, 2007.

[20] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proc. of OSDI*, pages 193–208, 2008.

[21] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proc. of USENIX ATC*, pages 27–27, 2011.

[22] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Proc. of LISP and Functional Programming*, pages 293–298, 1984.

[23] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proc. of WINSYM*, pages 14–14, 1999.

[24] P. Institute. 2013 study on data center outages. http://www.emersonnetworkpower.com.

[25] G. Kiczales, J. Lamping, M. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP*, 1997.

[26] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. of NSDI*, pages 18–18, 2007.

[27] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *Proc. of PLDI*, pages 179–188, 2007.

[28] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *Proc. of VEE*, pages 101–110, 2009.

[29] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of USENIX ATC*, pages 1–1, 2005.

[30] D. Lee, B. Wester, K. Veeraraghavan, P. M. C. Satish Narayanasamy, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proc. of ASPLOS*, 2010.

[31] Linux Community. syscall_table.s. http://lxr.free-electrons.com/source/arch/m32r/kernel/syscall_table.S.

[32] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging deployed distributed systems. In *Proc. of NSDI*, pages 423–437, 2008.

[33] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program instrumentation for debugging and monitoring with AspectC++. In *Proc. of ISORC*, pages 249–256, 2002.

[34] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, pages 11:1–11:38, 2011.

[35] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[36] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS*, 36(SI):75–88, Dec. 2002.

[37] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proc. of PLDI*, pages 362–371, June 2008.

[38] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of OSDI*, pages 267–280, 2008.

[39] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. *ACM SIGOPS*, 40(5):229–240, 2006.

[40] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proc. of SOSP*, pages 177–192, 2009.

[41] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *Proc. of WCAE*, 2004.

[42] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. of NSDI*, pages 115–128, 2006.

[43] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[44] S. M. Srinivasan, S. K, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. of USENIX ATC*, pages 29–44, 2004.

[45] W. Teitelman. *PILOT: A STEP TOWARDS MAN-COMPUTER SYMBIOSIS*. PhD thesis, Massachusetts Institute of Technology, 1966.

[46] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *Transactions on Computer Systems*, 30(1):451–490, February 2012.

[47] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of SOSP*, pages 230–243, 2001.

[48] M. Xu, R. Bodik, and M. D. Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *Computer Architecture*, pages 122–133, 2003.

[49] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing incon-

sistencies in deployed distributed systems. In *Proc. of NSDI*, volume 9, pages 229–244, 2009.

[50] M. Yabandeh, N. Vasić, D. Kostić, and V. Kuncak. Simplifying distributed system development. In *Proc. of HotOS*, pages 18–18, 2009.

[51] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proc. of NSDI*, 2009.