DDOS: Taming Nondeterminism in Distributed Systems

Nicholas Hunt, Tom Bergan, Luis Ceze, Steven D. Gribble
Computer Science and Engineering, University of Washington
{nhunt,tbergan,luisceze,gribble}@cs.washington.edu

Abstract

Nondeterminism complicates the development and management of distributed systems, and arises from two main sources: the local behavior of each individual node as well as the behavior of the network connecting them. Taming nondeterminism effectively requires dealing with both sources.

This paper proposes DDOS, a system that leverages prior work on deterministic multithreading to offer: 1) space-efficient record/replay of distributed systems; and 2) fully deterministic distributed behavior. Leveraging deterministic behavior at each node makes outgoing messages strictly a function of explicit inputs. This allows us to record the system by logging just message's arrival time, not the contents. Going further, we propose and implement an algorithm that makes all communication between nodes deterministic by scheduling communication onto a global logical timeline.

We implement both algorithms in a system called DDOS and evaluate our system with parallel scientific applications, an HTTP/memcached system and a distributed microbenchmark with a high volume of peer-to-peer communication. Our results show up to two orders of magnitude reduction in log size of record/replay, and that distributed systems can be made deterministic with an order of magnitude of overhead.

Categories and Subject Descriptors C.2.4 [*Computer Communication Networks*]: Network Protocols and Distributed Systems; D.4.1 [*Operating Systems*]: Process Management; D.4.5 [*Operating Systems*]: Reliability

Keywords Distributed Systems, Reliability, Determinism, Record-and-Replay

1. Introduction

Nondeterminism makes the development of distributed systems difficult. Without the ability to precisely reproduce a buggy execution, it is challenging for developers of distributed applications to track down the root cause of a bug. Further, replicating distributed systems for fault tolerance is hard since nondeterminism can cause the replicas' executions to diverge. Ideally, the behavior of distributed systems would be solely a function of well-defined inputs.

However, nondeterminism is pervasive in distributed systems. Not only does each individual node encounter nondeterminism locally due to timing variations in the operating system and hardware, but the network connecting the nodes is also a major contributor: arbitrary messages can be lost or reordered. Without mechanisms to control both sources, it is exceptionally challenging to replicate distributed systems and to reproduce buggy distributed executions.

There are two main approaches to handling nondeterminism. The first is to *record* and later *replay* the nondeterminism, simplifying debugging by making executions repeatable. The second approach eliminates nondeterminism altogether by executing the program *deterministically* with respect to its explicitly specified inputs. This approach simplifies both debugging and fault-tolerant replication.

This paper describes DDOS, a system we built to explore the trade-offs between record/replay and determinism in distributed systems. In record/replay mode, DDOS generates space-efficient replay logs that can be used to replay an entire distributed system. In deterministic execution mode, the *entire distributed system is executed deterministically* with respect to inputs from outside the distributed system, such as external network packets and user input. To our knowledge, this is the first proposal for fully deterministic execution of arbitrary distributed systems.

Both implementations exploit local-node determinism as provided by prior work in dOS [4]. For record/replay, we observe that given local-node determinism, it is sufficient to record just the arrival time of internal node-to-node messages, and not the message's contents; local determinism guarantees that message contents will be regenerated deterministically during replay. For distributed determinism, we propose an algorithm for delivering internal node-to-node messages deterministically. When combined with local-node determinism, this algorithm guarantees deterministic distributed execution.

DDOS advances prior work in two ways. First, existing record/replay systems [10, 14, 18] often produce prohibitively large logs (multiple gigabytes per hour) because they work at the boundary of each local-node and thus need to record the contents and delivery times of internal network messages. DDOS generates space-efficient logs by exploiting local-node determinism to eliminate the need to log the contents of internal messages. DDOS leverages observations similar to those made by DejaVu [18], but because DejaVu is a full record/replay system, it is not able to achieve the considerable log size reductions that DDOS achieves by exploiting local-node determinism.

Second, existing deterministic execution systems have focused on multithreaded programs. These systems either support single-node determinism only [4, 5, 8] or support distributed systems with a limited communication model [1]. As part of DDOS, we propose and implement an algorithm to make inter-node communication fully deterministic and for arbitrary applications and communication patterns.

Our DDOS implementation supports *arbitrary* distributed POSIX-compatible applications—including applications with data races—without requiring changes to application binaries. Our evaluation shows that our record/replay mechanism can reduce the required

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03…$15.00

log sizes by up to 70%, and that our deterministic execution mode imposes up to an order of magnitude overhead relative to nondeterministic execution—these overheads vary wildly with frequency and pattern of communication.

Finally, our evaluation demonstrates the following trade-off between record/replay and deterministic execution: the record/replay mechanisms in DDOS impose a lower performance overhead at the expense of larger logs, whereas the deterministic execution mechanisms lessens the space requirements for logs at the expense of performance.

1.1 Use Cases for Distributed Replay

Debugging. Debugging distributed systems is a daunting task. In addition to tracking down bugs that occur locally within a single node of the system, bugs in distributed systems can be dependent on deep communication chains involving a large number of nodes across the network. Local-node record/replay can help with the first class of bugs, but requires logging all communication between nodes, which can be prohibitive for some applications. Further, local-node record/replay does little to help find bugs of the second class because the root causes of these bugs are visible only when the entire distributed system is replayed as a unit. The space-efficient, full-system record/replay mode provided by DDOS makes this debugging technique more practical.

1.2 Use Cases for Distributed Determinism

Debugging. Determinism helps with debugging by making execution repeatable. For debugging, main difference between determinism and record/replay is that a deterministic system enforces a particular communication schedule between nodes, obviating the need to actually log this communication. The user can observe buggy behavior again by simply re-running the application with the same inputs.

Replication. A second application for distributed determinism is to enable replication of an *entire distributed system* for fault-tolerance. By instantiating multiple copies of a distributed system and replicating the inputs to all copies, distributed determinism guarantees that each replica evolves identically. Prior work on replicating distributed databases has shown that deterministic execution can remove the need for expensive cross-replica two-phase commit protocols, simplifying code and potentially improving performance [30].

1.3 Outline

The rest of this paper is organized as follows. Section 2 overviews the DDOS architecture and provides some background on dOS, which is used by the DDOS implementation. Sections 3 and 4 describe our high-level algorithms for distributed record/replay and deterministic execution. Section 5 describes implementation details of these algorithms. Section 6 offers a evaluation of our system, Section 7 summarizes related work, and Section 8 concludes.

2. DDOS Architecture

DDOS builds on dOS [4], which is an operating system that provides deterministic execution of single-node, multiprocess, multithreaded programs. The basic abstraction provided by dOS is a deterministic process group (DPG), which is a group of local threads and processes that execute as a single deterministic unit. In DDOS, each node of the distributed system executes within its own DPG, and the set of DPGs is collectively referred to as a *distributed DPG* (DDPG).

Sections 3 and 4 detail how we leverage the single-node determinism provided by DPGs. In the rest of this section, we summarize the semantics of DPGs, and then describe how DPGs are combined to form a DDPG.

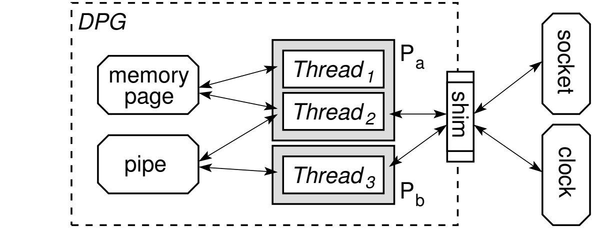

Figure 1. A Deterministic Process Group (from [4])

2.1 Deterministic Process Groups

A DPG is an abstraction that allows a group of multithreaded processes to execute deterministically on a single node. Specifically, DPGs provide a mechanism to partition the sources of nondeterminism that can affect a process into *internal* sources and *external* sources: internal sources of nondeterminism are eliminated from DPGs, but external sources of nondeterminism are fundamentally nondeterministic and cannot be eliminated. Examples of internal sources of nondeterminism include thread scheduling, hardware timing variations, and communication between threads involving pipes, shared-memory, local files, and so on. External sources of nondeterminism include reading physical clocks and communicating with processes outside of a DPG.

Additionally, dOS defines a *shim layer* API that allowed programmers to interpose on all external nondeterminism. Any time a DPG receives input from a nondeterministic external source, dOS funnels that event through the shim layer, allowing the shim programmer to control how that nondeterminism is introduced. For example, a shim may implement the record half of a record/replay mechanism by creating a log entry on every nondeterministic event.

Figure 1 shows an example of a DPG containing two processes and three threads communicating with each other using shared-memory and a pipe. DPGs guarantee this communication happens deterministically. Reading a physical clock or communicating over the network occurs outside of the DPG and is therefore nondeterministic, so this communication must first pass through the shim layer. To enable deterministic execution, dOS schedules a DPG's execution onto a deterministic logical timeline. The scheduling algorithm is described in [4]. dOS performs this scheduling in the operating system kernel, transparently to applications, allowing it to execute arbitrary, unmodified POSIX-compatible applications inside of a DPG.

2.2 Distributed DPGs

DDOS defines a distributed system as a set of deterministic process groups which may be distributed across many different physical machines. Hence, we call this set of DPGs a distributed DPG (DDPG). Each DPG can itself contain one or more multithreaded processes participating in the distributed system. Many DPGs can execute on the same physical machine, and processes not in any DPG can execute alongside DPGs on the same machine. In a DDPG, the network is partitioned into an *internal* network that connects all deterministic process groups in the DDPG, and an *external* network that connects the DDPG to external processes and machines.

Figure 2 shows an example of a DDPG. The DDPG shown is composed of three deterministic process groups executing on two different machines. All communication between DPGs 1, 2, and 3 will occur on the internal network. DDOS controls the internal network to implement either space-efficient record/replay (Section 3) or distributed determinism (Section 4). Communication with Process F, which is outside of the DDPG, occurs on the nondetermin-

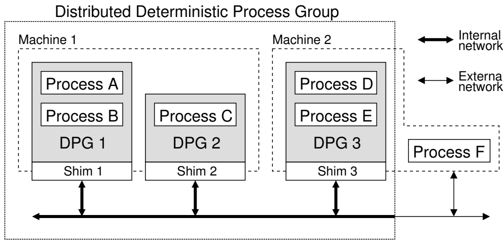

Figure 2. A Distributed DPG

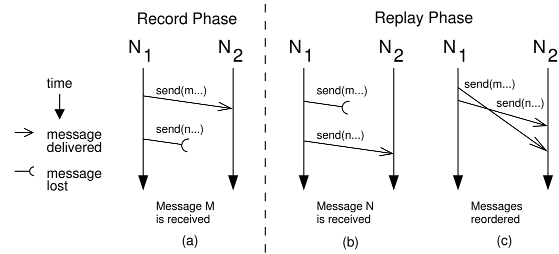

Figure 3. When messages are regenerated during replay, different UDP messages can be lost or reordered

istic external network. All messages received from the external network are considered nondeterministic input.

DDOS exposes the same POSIX-compatible sockets API that Linux provides. Distributed applications can run inside DDPGs without modification: record/replay and distributed determinism are both provided transparently by DDOS.

3. Record/Replay with DDPGs

This section describes our algorithm for space-efficient record/replay of DDPGs. To help understand the trade-offs, we first discuss how a traditional record/replay system would be implemented, and then describe our algorithm in comparison to the traditional approach.

3.1 Single-Node Record/Replay

A traditional approach to record/replay for distributed systems would record each node in the distributed system independently. To reproduce a buggy execution, the node on which the bug manifested would be replayed *in isolation* using the generated log to reproduce the nondeterminism encountered in the original run.

While replaying the failed nodes independently may be convenient, this approach suffers from two drawbacks. First, to ensure accurate replay, *all* network communication between nodes in the distributed system must be recorded. For some applications, the amount of internal communication is so large that this approach becomes infeasible. Second, some distributed system bugs can be understood only after looking at chains of communication between multiple nodes. This information is lost by single-node replay, making these multi-node bugs significantly more difficult to debug.

3.2 Distributed Record/Replay

DDOS records each node independently but replays the entire distributed system as a unit. In this approach, messages between nodes in the distributed system need not be recorded since they will be regenerated during replay. Instead, we record just two sources of nondeterminism at each node: 1) the logical arrival time of internal messages; and 2) all external inputs that any traditional record/replay system would record, including external network packets, clocks, and user input. During replay, we replace these sources of nondeterminism exactly as specified in the log.

To understand why replay is deterministic, recall that all nodes execute in DPGs. Thus, each node's execution evolves deterministically and therefore the network messages generated by each application will be deterministic: as long as internal messages and external input are delivered at the same logical time during replay, messages sent by a DPG will occur at a deterministic logical time and contain deterministic contents.

Reliable Channels. For reliable transport protocols such as TCP, messages sent by one endpoint will be received completely and in-order on the remote side; packet loss is handled transparently by the underlying protocol. During the record phase, a process that consumes data from the network can simply record the number of bytes read and the logical time the data was consumed. Replay looks at the recorded timestamp and waits for that logical time to arrive before receiving exactly the recorded number of bytes from the network.

Unreliable Channels. For unreliable datagram protocols such as UDP, message loss and reordering are exposed to the application. As a result, the network may drop or reorder a different set of datagrams during the record and replay phases, as shown in Figure 3. We handle this by appending a unique sequence number to all UDP messages during the record phase. This sequence number is stripped by the remote node on reception and written to the replay log. We then perform replay with a reliable connection and simulate dropped (or reordered) packets during replay by dropping (or buffering) unexpected messages on the receiving end.

Connection Management. Similar issues arise when establishing TCP connections during replay. Connections to internal addresses need to be reestablished during replay if they were successfully completed in the record phase. However, connection requests that succeeded in the original execution may be dropped during replay. We handle this by retrying connections during replay until they succeed.

Further, connection requests can be reordered by the network, causing them to be accepted in a different order in the record and replay phases. We avoid this by buffering new connections at the accepting node during replay to ensure that new connections are delivered to the application in the same order they were accepted during the record phase.

Dealing With Failures. Node failures due to program bugs are replayed deterministically. Node failures due to hardware crashes are also easily replayed: the node hits the end-of-log during replay, and stalls.

Hardware failures during replay are also possible. Although our approach drastically reduces the required log size for many applications, it requires replaying all nodes in a distributed system at once, increasing the possibility that some node will fail unexpectedly during replay (e.g., due to a hardware failure). Our system assumes no failures during replay: if a node fails during replay, the system must be restarted. We believe this is a reasonable assumptions because the typical use for record/replay is to observe a short segment of execution in a debugging setting. Another feasible solution would detect node failure during replay and then restart that node from a checkpoint, as discussed later in Section 4.3.

Reconfigurable Systems. Because all recording operations are local to a node (there is no global coordination), we naturally support *reconfigurable* systems in which nodes may dynamically enter or leave. This requires no changes to the above protocols: a node enters a system by establishing connections, as described above, and leaves by closing all open connections.

4. Deterministic Execution with DDPGs

Beyond record/replay, DDOS also implements a deterministic distributed execution algorithm. Our algorithm operates at the socket layer, and its job is to ensure that messages on the internal network of a DDPG are delivered deterministically. We first explain the guarantees we desire from this algorithm, and then we describe the algorithm itself.

4.1 Deterministic Guarantee of DDPGs

Our deterministic guarantee is that the entire DDPG executes as a single deterministic unit, relative to input received from clocks, users, the external network, and other sources of external nondeterminism. Our guarantee can be specified in two parts. The first is *local* determinism, provided by DPGs, which ensures that each node of the distributed system executes deterministically relative to its own local inputs. The second part is *global* determinism, provided by DDPGs, which ensures that all messages sent on the internal network are received at a deterministic logical time and have deterministic contents.

Our algorithm guarantees that messages sent on the internal network of a DDPG are delivered deterministically. Specifically, we ensure that network receive operations complete at a deterministic logical time and return a deterministic number of bytes, with deterministic message contents. Further, we ensure that internal listening sockets accept connections in a deterministic order. These guarantees apply to both TCP and UDP channels.

Processes in a DDPG can also communicate via network messages with processes outside of the DDPG. However, messages exchanged with these processes are transferred on the external network, so they are delivered nondeterministically. Record/replay techniques are needed to later reproduce these inputs.

4.2 Deterministic Network Protocol

The basic idea is depicted in Figure 4. We begin by observing that communication occurs when a node *receives* a message from a remote node, not when the message is sent. To ensure deterministic communication, we need to ensure only that nodes consume messages from remote nodes at deterministic points in their logical time; DPGs will guarantee the sends will occur at a deterministic time by construction.

We ensure deterministic message delivery by dividing execution of the DDPG into global *quanta*. Conceptually, messages sent along internal channels are buffered until the end of the global quantum in which they were sent, and are delivered to the receiving node before the next quantum begins. Global quanta end once every node has signaled that it is ready to start the next global quantum; this happens once the local node's DPG has executed some deterministic number of logical time steps. The number of time steps need not be fixed and need not be the same for each node, as long as it is chosen deterministically. The mapping of local logical time to global quanta is illustrated in Figure 4.

A naïve implementation of this algorithm might use a global barrier to stall nodes at the end of every global quantum until all buffered messages have been delivered. However, this is not necessary. Global quanta can be propagated *lazily*. We do this by having each node push an end-of-quantum marker down its internal network channels every time it reaches the end of a global quantum. Remote nodes buffer data as it arrives and are allowed to consume data only up to the end-of-quantum marker of the *previous* quantum. This ensures that remote nodes consume data only if that data was sent in a prior global quantum.

Figure 5 illustrates this lazy protocol. In this figure, DPG_A is currently executing in global quantum 3, and is communicating with three other DPGs over network channels which are shown as queues. DPG_1 and DPG_2 are currently executing in global quantum 4, so they have sent three end-of-quantum markers down their network channels, while DPG_3 is executing in global quantum 2, so it has sent just one marker. The darker shaded region in this figure shows the portion of data that is currently available for DPG_A to read: since DPG_A is executing in the third global quantum, only data sent from the first two global quanta are available. Further, note that DPG_3 is currently executing behind DPG_A and has not yet sent any data, other than the end-of-quantum marker. If DPG_A attempts a `recv` operation on the connection with DPG_3, that `recv` cannot complete until DPG_3 has either sent another end-of-quantum marker or pushed enough data onto the channel to satisfy the request.

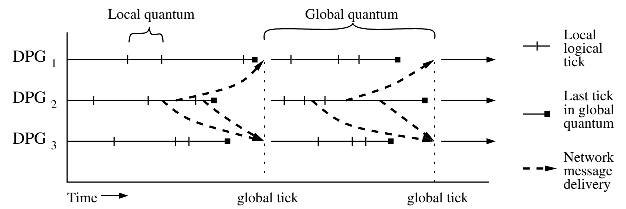

Figure 4. Conceptual view of deterministic communication. Messages are buffered until the end of the current global quantum before being delivered. Global quanta are formed from deterministic numbers of logical time steps in the local DPGs.

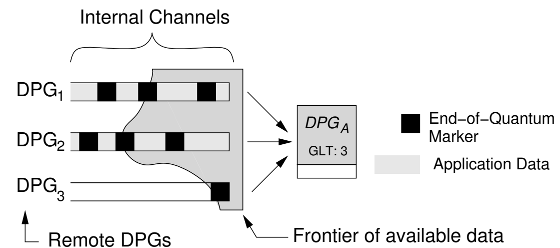

Figure 5. Deterministic communication with lazy barriers. The tokens in each network channel define the *frontier* of data that a DPG can safely consume while preserving determinism.

Figure 5 also illustrates how we allow each node's notion of current global time to drift from the other nodes, as long as the basic communication constraints are enforced: threads can only consume data from the internal network that was sent in an earlier global quantum. This relaxation can be a significant optimization when compared to a naïve protocol using global barriers. In the limit, when the nodes do not communicate, each node can advance its own view of global time arbitrarily, without waiting for the other nodes to catch up. Only when a node receives data from a remote node must it potentially synchronize with that node.

Promiscuous Operations. Some network operations—such as accepting connections, polling for data with `poll`, or promiscuous receives on UDP sockets—can receive data from multiple remote nodes in a single call. These operations require that the caller waits for *all* nodes it *could* communicate with to catch up in global time before the operation can complete. Once all such nodes have caught up, we deliver incoming data from one sending node at a time, after ordering sending nodes in some deterministic order. For example, a `accept` operation first waits until it has received an end-of-quantum marker for the prior quantum from all other nodes. It then gathers all connection requests, orders them, and finally delivers the connection requests in that order to ensure they are processed deterministically.

In the worst case, where all nodes call globally promiscuous operations such as `accept` in every global quantum, every node will be forced to wait for every other node at the end of every quantum, degrading the relaxed protocol to the naïve protocol that uses global barriers.

Unreliable Channels. Messages sent over UDP channels can be arbitrarily dropped or reordered by the network. We must guarantee in-order lossless message delivery to ensure determinism. Thus, in deterministic execution mode, DDPGs tunnel all UDP connections over a reliable TCP connection.

Reconfigurable Systems. Nodes may leave a DDPG via exiting: this is already deterministic because of the local determinism provided by dOS. Joining a DDPG, however, represents a fundamental source of external nondeterminism in the time at which a node joins. For this reason, we do not consider joins to be useful operations in deterministic execution mode, and we do not support them. (We do support joins in record/replay mode; see Section 3.2.) If desired, `join(N,DDPG)` could be implemented by adding a two-phase commit protocol, to allow all nodes to agree on the quantum at which `N` joins the DDPG, and by logging that quantum for use during further replays.

4.3 Dealing With Failures

Failures in large distributed systems occur frequently, so it is important to understand how these failures affect DDOS. When one DPG attempts to receive data from another DPG along an internal channel, that node needs to wait until the correct end-of-quantum marker is received before data can be returned. If the sender of the data (and thus the marker) fails, the receiver can not continue. This sender/receiver synchronization constrains communication onto a deterministic *logically synchronous network*.

An initial strategy for coping with failure might be as follows: detect a failed node, restore the node from a checkpoint, replay its inputs up to the current state of the system, and finally resume distributed execution. However, distinguishing a failed node from a slow node is difficult, and further, when a node fails, all other nodes in the system must agree on the exact logical time at which that node failed. Detecting failures with this degree of accuracy is a consensus problem which is impossible to solve on asynchronous physical networks given an arbitrary failure model [13].

Thus, a side-effect of the logically synchronous network model is that DDOS cannot guarantee forward progress while preserving determinism without making assumptions about the failure rate of the distributed system. If these assumptions are broken, DDOS must give up either determinism or forward progress. As a further consequence of consensus impossibility, DDOS cannot detect with complete certainty when these assumptions are broken. Therefore, our deterministic guarantees are at best probabilistic, even though these probabilities may be quite high in practice.

One simple way to deal with this problem is to implement a physical timeout mechanism that triggers when a node has not received an end-of-quantum marker from a remote node in some predefined amount of physical time. If the node is blocked on an operation waiting for an end-of-quantum marker to arrive and the timeout triggers, the blocked node can use this timeout to detect a *possible* failure, and alert the system. A user could then decide to restart the node, to continue to wait, or to continue without waiting and sacrifice determinism from that point forward. This might even be incorporated into a larger consensus protocol, such as Paxos [22]. We have not yet implemented this mechanism or investigated it further.

Finally, we note that the problem of failures is common to all systems for deterministic execution, not just DDOS. The fact that DDOS deals with distributed systems simply exacerbates the problem. For example, even in a hardware implementation of determinism, such as DMP [8], determinism could be lost as the result of an alpha particle causing a random bit flip in memory.

```
tom   5000 t:80,u:512 web-serve
jerry 6000 t:10000 memcached
```

Figure 6. An example of a manifest file specifying a two-node DDPG

5. Implementation

DDOS is implemented on top of dOS [4] as a distributed shim program. It uses dOS's shim API to interpose on all socket-related system calls made by a DPG. Depending on the mode of the DDPG (either record/replay or deterministic execution), the shim will either record the results of these system calls to a log, replay the results of the system call from a log, or execute the system call in a deterministic way.

Implementing DDOS as a dOS shim means the physical network is outside of DDOS's control. In other words, the DDOS shims replay or make communication deterministic at the system call layer, but an eavesdropper watching the packets in transit between physical machines could see either imprecise replay or nondeterminism. The shims at either side of the connection, however, ensure that communication observed by the applications is properly controlled.

5.1 Creating a DDPG

A DDPG consists of a set DPGs executing on physical machines, along with a specification that partitions the network into *internal* and *external* channels. To construct a DDPG, the developer writes a *manifest file*. The manifest contains an entry for each of the DPGs, specifying the command to execute, the node to execute the DPG on, as well as the set of TCP and UDP ports on that node that are considered internal to the DDPG. When used for deterministic execution, the manifest also indicates which port the DPG should use for *control channel* connections (Section 5.3). Note that internal channels can only be used to communicate with DPGs: if a port will ever be used by both DPGs and processes outside of a DPG, the port cannot be marked as internal.

Figure 6 shows an example of a manifest file specifying a two-node DDPG. `web-serve` will be run within a DPG on the host `tom`, and connections to TCP port 80 and UDP port 512 will be treated as *internal*; all other connections will be treated as *external*. Similarly, `memcached` will run within a DPG on host `jerry` with a single internal TCP port.

5.2 Record/Replay

The DDOS record/replay shim is implemented as an extension to the single-node record/replay shim originally implemented for dOS. Most of the implementation effort was tedious but straightforward, following prior work on record/replay. Our record shim emits an execution log that can be read by the replay shim. The log files are not compressed, and thus further reduction in log size is likely possible if compression was included.

UDP Sequence Numbers. As described in Section 3.2, UDP messages are tagged with a unique sequence number before being sent. Each thread in a DPG maintains a count of the number of UDP messages it has sent. The sequence number assigned to each outgoing UDP message is a tuple of the sending DPG's id, the sending thread's id, and the current UDP message count of that thread. Because the DPG ids, thread ids, and the UDP message count are all deterministic, these sequence numbers are deterministic as well and can be used to replay UDP communication.

Establishing Connections. If a connection on the internal network was successfully completed during recording, the connection must be successfully completed during replay. However, due to timing variations (Section 3.2), a connection that succeeded during record may fail during replay (for example, the connector may call `connect` before the remote node has had a chance to invoke `listen`). To ensure the same behavior during replay, the replay shim may pause the DDPG execution until the connection can be reestablished and will retry the connection attempt until it succeeds.

Replaying Connection Order. Similarly, DDPGs must ensure that incoming internal network connections are accepted during replay in the same order observed during the record phase. To replay the correct order, the DDOS shims buffer all incoming connection requests during replay; when a process calls `accept`, the buffered connections are checked against the log see if the expected connection has already arrived. If so, the connection is returned to the caller and the DPG that initiated the connection is notified. If the expected connection is not already buffered, the call to `accept` blocks until the anticipated connection is received.

Log Durability. Our current implementation writes log entries to disk asynchronously, as synchronous writes would be prohibitively expensive. This may introduce a window of record loss during a hardware failure. More advanced storage systems [27] or upcoming new hardware [24] may make synchronous writes more viable.

5.3 Deterministic Execution

DDOS implements our deterministic message delivery algorithm with a distributed shim program. This shim program interposes on all socket-layer operations performed by applications to ensure that internal network communication happens deterministically.

Upon startup, the DPGs enter a brief initialization phase where the manifest file is read, pairwise control channels are established between DPGs, and DDPG configuration values—such as the size of a global quantum—are exchanged. Each DPG is assigned a unique id derived from its position in the manifest file. After the initialization phase, the DPGs begin executing their respective programs.

Control channels are used by the DDOS shims to coordinate their execution. Messages sent on the control channel have one of four types, described in Table 1. CMT_ENDQ messages broadcast by a DPG at the end of every global quantum. CMT_CONNREQ and CMT_CONNRESP messages are used to establish virtual network connections between the applications. CMT_DATA messages are used to transfer application-level data along a virtual connection.

Control channel are reliable, in-order network connections. All internal network connections between two nodes are multiplexed on to the single control channel connecting the nodes, and both application-level data and the DDOS protocol are transferred along this connection. Because the control channels are in-order, when a DPG sees a CMT_ENDQ message from a particular node for quantum n, it is guaranteed to have already seen all other messages sent by that same node in quantum n.

Connection Management. To establish an internal network connection, the local DPG sends a CMT_CONNREQ message to the remote DPG containing the target port number, as well as the local port number it's connecting from. Because Linux randomly assigns port numbers to sockets, the port numbers used in DDOS are *virtual port numbers*, assigned deterministically by the DDOS shims.

The remote DPG buffers all connection requests in a sorted queue for the target listening socket. Each request is tagged with a tuple consisting of the global quantum in which the request was sent, the unique id of the connecting node, and the port number the request originated from. The queue is sorted based on this tuple. Requests from prior global quanta are satisfied from the queue in sorted order, while requests from future quanta are kept buffered. Once a connection is accepted (or if the connection request is denied because a listening socket does not exist on the target port), the remote DPG responds with a CMT_CONNRESP message.

Type	Description
CMT_ENDQ	Broadcast by a node at end of global quantum
CMT_CONNREQ	Connection request sent from one DPG to another
CMT_CONNRESP	Response for a connection request, indicating connection success or failure
CMT_DATA	Contains application-level data being sent between process in a DDPG

Table 1. Control channel message types

Communicating on Internal Channels. Sending data on an internal channel creates a new CMT_DATA message containing the length of the message and the data being sent. The message is sent along the control channel to the receiving node. Upon reception of a CMT_DATA message, a receiving node reads the entirety of the message and buffers the message in a queue representing the virtual port of the receiving socket. Each message is tagged with the global quantum in which it was sent. Messages from prior global quanta are delivered from the queue in-order, while requests from future quanta are kept buffered. For promiscuous UDP channels, messages are additionally tagged and sorted in the same way that connection requests are tagged and sorted.

Poll/Select Optimization. As mentioned previously in Section 4.2, promiscuous network operations such as `poll` and `select` can potentially add significant overheads to deterministic execution due to the high cost of synchronizing logical timelines across nodes, but this synchronization can sometimes be avoided. For example, when performing a `poll` operation on a set of file descriptors, synchronization with remote nodes can be avoided if either: 1) the remote node's view of global logical time is later than the node's performing the `poll`; or 2) the remote node's view of global logical time is earlier than the caller's, but *all* file descriptors being examined by `poll` from that node have *some* amount of data already available.

In the first condition, the remote node is operating in the future relative to the caller's notion of global logical time. Thus, any data sent by the remote node has already been received by the caller, and no additional synchronization is needed to satisfy the `poll` request.

The second condition exploits the fact that the `poll` family of functions return only a Boolean result indicating whether or not data is available to read—not how much data is available. As long as all file descriptors associated with a remote node have some data that is currently available from a logical time earlier than the caller's notion of global logical time, `poll` can deterministically return a true result without first synchronizing; synchronization can be delayed until the two nodes communicate with a `recv` operation.

Resource Exhaustion. System limits on resource usage, such as limits on open file descriptors, can pose difficulties for correct deterministic behavior. To understand why, consider running a DDPG twice, where no other processes are run on the system during the first execution, but a nondeterministic background process runs during the second execution. If that background process in the second execution opens a large number of file descriptors, the DDPG might be unable to open all of the descriptors it needs to complete, possibly causing the two executions to behave differently. This scenario could be dealt with in two ways: 1) pause the execution of the DDPG until the request can be completed (for example, by waiting for the other process to close some of its descriptors), or 2) report the error to the user and abort execution of

the DDPG. For simplicity, our system implements the second approach, leaving implementation of the first method to future work.

6. Evaluation

This section evaluates DDOS. The goal of our evaluation is to answer the following three questions: 1) What are the performance overheads of using DDOS for record/replay and for deterministic execution, relative to nondeterministic execution outside of DDOS? 2) How effective is our space-efficient record/replay mechanism at reducing the size of replay logs? 3) Where do the runtime overheads come from? A more qualitative goal of this evaluation is to demonstrate that we can support real-world applications.

6.1 Applications

We evaluated DDOS with three different types of distributed applications: a microbenchmark with frequent peer-to-peer traffic, a reactive web application with a remote file server, and a distributed scientific application based on the OpenMPI libraries. The distributed scientific application comes from the NAS Parallel Benchmark suite [25].

Peer-to-peer Microbenchmark. The peer-to-peer microbenchmark is a custom program adapted from the `racey` deterministic stress test [15] that we call `racey-dist`. `racey-dist` is a distributed application with single-threaded nodes. The nodes in `racey-dist` form a completely connected graph and send and receive data to the other nodes in the system with high frequency; the data sent and received is combined by each node to compute a signature that is highly sensitive to the pattern of communication, including both the order and content of messages.

Reactive Webserver. The reactive webserver is composed of two applications: a multithreaded server that handles web requests from clients, and a multithreaded instance of `memcached` that serves the files to the webserver from an in-memory database. For our evaluation, each application was run on a separate evaluation node, and a third computer external to the DDPG was used to generate web requests. We used the `ab` benchmarking utility from Apache to generate the workload. Both the webserver and the `memcached` server were configured to run with 8 threads.

OpenMPI Scientific Application. From the NAS OpenMPI parallel benchmarks, we ran `is`, a large-scale integer sorting kernel. We vary the number of MPI processes from 2 to 16, with the threads evenly divided amongst the nodes. Because OpenMPI uses the ssh protocol to remotely spawn tasks, we also ran a separate `dropbear` ssh daemon inside each of the DPGs to accept these connections.

6.2 Correctness

We ran `racey-dist` in a deterministic DDPG 20 times and verified that all executions produced the same signature. For comparison, 20 nondeterministic runs of `racey-dist` produced 20 different signatures.

`racey-dist` stresses the connection management and end-to-end communication of our implementation, so we believe these components are correct. This benchmark also stresses the lazy barrier implementation by buffering connection requests and message contents until global barriers. Our selection of other applications provide an indirect demonstration that the remaining operations—such as promiscuous communication, UDP transmissions, `poll` and `select`, etc.—likely behave properly. The NAS benchmark in particular exercises complex connection management and relies heavily on promiscuous socket operations. We have released the source code [1] and encourage readers to independently test our implementation.

[1] Source code is available at `http://sampa.cs.washington.edu`.

Benchmark	Execution Time			Log Sizes (MB)	
	Nondet	DPGs	DDOS	DPGs	DDOS
`racey-dist`					
2 threads	3 ms	236 ms	135 ms	0.27	0.15
3 threads	22 ms	349 ms	176 ms	0.41	0.23
4 threads	7 ms	307 ms	168 ms	0.54	0.30
`www`					
8 threads	2.5k r/s	188 r/s	217 r/s	599	146
`is`					
2 threads	1.53 s	2.09 s	1.59 s	180	4
4 threads	1.11 s	1.73 s	1.19 s	272	7
8 threads	1.00 s	29.48 s	26.63 s	374	107
16 threads	1.23 s	49.05 s	40.79 s	443	168

Table 2. Execution time of the recorded execution and the resulting log sizes. Nondet is nondeterministic execution; DPGs is each node performing a local record; and DDOS is a DDPG record, where nodes only record communication external to the DDPG.

6.3 Methodology

The evaluation was performed on a five node cluster, each running a dOS-enabled 2.6.24.7 Linux kernel. Four of the machines have 2 Intel Xeon X5550 quad-core processors and 24GB of RAM; the fifth node has 2 Intel Xeon E5520 quad-core processors and 10GB of RAM. Each of the experiments below was executed three times, and the average measure of performance is reported.

We use an application-specific measure of performance for each application. For the `racey-dist` and MPI benchmark, we measure the wall-clock time required to complete the workload. For the webserver benchmark, we use the number of requests handled per second.

6.4 Record/Replay

Table 2 shows the runtime of the recorded executions and the resulting log sizes in MB. For `www`, requests per second is used instead of execution time to report performance in a more meaningful way. Each row contains the results for a single application run with the specified number of threads.

Column 2 shows performance when the distributed system is run nondeterministically, without the use of DPGs or DDPGs. Column 3 shows the results when each node is its own DPG, performing local node recording with the record/replay shim developed in [4]. This column demonstrates the overhead of using a traditional record/replay mechanism as described in Section 3.1, which does not exploit the deterministic properties of DDPGs when logging network communication. Column 5 provides the log sizes produced with this local recording strategy. Finally, Column 4 provides the execution times when DDPGs are used to reduce the logging of network traffic to only that which is external to the DDPG, as described in Section 3, with Column 6 giving the resulting log sizes.

A comparison of Columns 2 and 3 shows that the overhead of dOS's deterministic scheduler plus the overhead of logging local-node inputs results in a 1-2 order of magnitude slowdown in performance. The logs in these executions vary between 200 MB and 600 MB in size.

A comparison of Columns 5 and 6 shows that not logging the internal traffic of a DDPG results in a significant reduction in log size (about 70% in some cases). We also observe that when executing within a DDPG with internal sockets, the application always recovered some performance compared to separate DPGs with record/replay shims. Some of this performance likely comes from not needing to write as much data to disk.

For the `is` benchmark with 8–16 threads, both the log size and performance overheads grew significantly from the execution with 4 threads. In these cases, we observed a large increase in the invocations of `poll` and the `rdtsc` instruction (an x86 instruction

that reads a hardware timer) by the application. We suspect this behavior is due to a mechanism in OpenMPI that is enabled when more than one MPI task is executed on a single node. In these cases, over 95% of the entries in record log were for these two operations, and likely contributed to the higher performance overhead as well. This is an example of a case where OS-level record/replay isn't the most efficient solution: a record/replay mechanism at the MPI layer would not be exposed to the `poll` system call and `rdtsc` instruction, and so would probably not see the same performance and logging penalty at the higher thread counts.

Although the record shim does not currently compress the logs during execution, manually compressing the logs after the execution shows that a 50-60% reduction in log size for these applications is possible.

6.5 Deterministic Execution

To evaluate the overheads of deterministic distributed execution, we ran each benchmark in four different modes. Table 3 reports these results. Column 2 shows the application-specific performance measurement when the benchmark is run nondeterministically, without the use of DPGs or DDPGs. Column 3 shows the performance when each node is run as a separate DPG, *without* an encompassing DDPG. In other words, each node runs dOS, but network communication between the nodes is still nondeterministic. Column 4 shows the performance metric when the distributed system is executed within a *fake* DDPG; the fake DDPG does not actually enforce deterministic network communication, but allows us to measure the overhead of just constructing the DDPG and interposing on all socket-layer operations. Finally, Column 5 shows the performance metric when the application is run within a deterministic DDPG, enforcing deterministic network communication between the nodes.

The difference in performance between Columns 2 and 3 is the overhead directly attributable to dOS. In theory, this represents the best performance that DDOS could achieve. In general, we see that the performance tends to be worse for higher thread counts. This is due to serialization of threads by the dOS scheduling algorithm due to resource sharing between the threads in the DPGs (for instance, shared-memory locations or the kernel's file descriptor table). A more complete evaluation and characterization of dOS's performance is available in [4]. dOS does not use a state-of-the-art deterministic scheduler; the addition of more advanced algorithms, such as those in [9], could contribute a substantial performance improvement. Most importantly, in this paper, we are primarily concerned with the overhead added by our algorithm for deterministic network communication, not with the overhead of local-node determinism.

Column 4 adds the cost of creating the DDPG, multiplexing network communication over a control channel, and sending end-of-quantum tokens between the nodes. In this mode, communication remains nondeterministic. We see that this additional communication imposes almost no additional overhead for `racey-dist`, and between 30% to 50% for `www` and `is`. Some results in this column, such as the 3 and 4 thread runs of `racey-dist`, show better performance than the corresponding value in Column 3. We would expect that the results for Fake DDPG would always be worse than DPG executions and believe these non-intuitive results are within measurement error for these experiments.

Finally, column 5 shows the performance when all communication between nodes occurs deterministically. The sources of the additional overheads include nodes waiting for remote nodes to catch up in logical time, and a small amount of additional coordination between the nodes. We see that for `racey-dist`, where communication occurs very frequently and with many nodes, the overheads of deterministic communication are the highest, with a roughly two

Benchmark	Nondet	DPGs	Fake DDOS	DDOS
`racey-dist`				
2 threads	$<$0.01 s	0.46 s	0.48 s	5.33 s
3 threads	0.02 s	0.47 s	0.44 s	1.77 s
4 threads	0.01 s	0.49 s	0.48 s	3.82 s
`www`				
8 threads	2.5k r/s	331 r/s	225 r/s	155 r/s
`is`				
2 threads	1.53 s	1.91 s	1.52 s	10.85 s
4 threads	1.11 s	2.70 s	1.13 s	11.34 s
8 threads	1.00 s	2.64 s	2.41 s	40.17 s
16 threads	1.23 s	4.77 s	6.15 s	55.35 s

Table 3. Execution times for the deterministic execution evaluation. Nondet is nondeterministic execution; DPGs is local determinism only; Fake DDOS sends end-of-quantum markers but doesn't enforce deterministic communication; and DDOS is fully deterministic distributed execution.

order of magnitude slowdown. This is attributable to the frequent synchronization of logical clocks, where one node must pause its execution waiting for the other node to catch up. For `www`, we observe a 16x slowdown compared to nondeterministic execution. We characterize the sources of overhead in more detail in the next section.

6.5.1 Record/Replay and Deterministic Execution Trade-offs

We briefly mentioned the trade-offs between the record/replay mechanism and the deterministic execution mechanisms in Section 1. The above evaluation demonstrates this trade-off empirically.

Consider, for example, the `racey-dist` benchmark. Executing in the record/replay mode of DDOS, `racey-dist` was able to finish its execution in less than a second for all of the configurations evaluated. However, it needed to record about 150 KB - 300 KB to do so. The deterministic executions all required over a second to finish, but did so with minimal logs. This same trade-off can be observed in the results for the other two applications as well.

6.5.2 Overhead Characterization

Reasons For Waiting. Our algorithm might introduce slowdown whenever it forces a node to wait for an end-of-quantum marker from a remote node. Table 4 partitions the total time DDPGs spend waiting into three categories: *promisc* shows the percentage of wait time due to the promiscuous `select` or `poll` operations; *connect* shows the percentage of wait time spent establishing connections with `connect` and `accept`; and *data* shows time spent waiting for data to become available on a deterministic socket (a blocking `recv`, for example).

Consider the `www` benchmark. `memcached` and `www` communicate constantly with one another to satisfy incoming web requests. As a result, a majority of wait time is due to nodes synchronizing their logical clocks while communicating: 91% of wait time is due to this synchronization. A small percentage of the time is attributable to connection setup and tear-down between the webserver and `memcached` at the beginning and end of the execution.

The `is` benchmark does more computation than communication, demonstrated by the fact that only 10% of its wait time occurs during data transmission. Surprisingly, the remaining 90% of the wait time is due to synchronization in the `poll` family of functions. These functions, due to their promiscuous nature, can be extremely expensive, in the worst case requiring the caller to synchronize logical time with all nodes in the system. In this application it was the `dropbear` ssh daemon running in each of the DPGs that was responsible for these promiscuous operations, rather than the actual `is` MPI tasks. This suggests that removing `dropbear` from DPG

Benchmark	Reason for Waiting		
	Promisc.	Connect	Data
racey-dist	18%	21%	61%
www	<1%	9%	91%
is	90%	0%	10%

Table 4. Percentage of time spent blocked waiting for promiscuous operations, connection requests/responses, or data

after the initial MPI tasks have been spawned could lead to a significant performance improvement.

Finally, for the `racey-dist` microbenchmark, the time spent waiting for connection setup is higher that other benchmarks, likely due to its relatively shorter running time.

Quantum Drift. We also measured quantum *drift* to evaluate to what degree distributed execution was synchronous; high values of drift suggest that lazy quantum barriers had a significantly positive effect on performance. We define quantum drift between node A and remote node B to be the difference between node A's current global quantum and the most recent end-of-quantum marker that node A has received from node B.

We instrumented DDOS to compute the maximum quantum drift at every quantum boundary and capture the summary as a histogram. All three benchmarks exhibited a similar trend: the quantum drift was mostly either 0 or 1 quantum, meaning the nodes executed mostly synchronously, but all benchmarks had a long tail of larger drifts, in some cases extending to hundreds of quanta. For `racey-dist`, 86% of drifts were either 0 or 1 quantum, with a tail spread evenly out to 16 quanta. For `www`, these numbers were 99% with a tail to 220 quanta, and for `is`, 98% with a tail to 580 quanta.

7. Related Work

Record/Replay. There have been many proposals for software-based implementations of record/replay. These include systems that provide record/replay of local nondeterminism only, as well as systems that provide record/replay for an entire distributed application.

Of the single-node systems, some record nondeterminism at the system call layer only [29], making replay of multithreaded applications impossible. Others record nondeterministic thread schedules to support multithreaded programs. These include systems that assume uniprocessor execution and record only scheduling decisions [7], as well as systems that support multiprocessors and either record all shared-memory accesses [11, 19, 23] or just synchronization operations [28, 32]. Nondeterminism arising from local sources is eliminated by DPGs, so DDOS never needs to record it.

Other systems provide record/replay for entire distributed systems. `liblog` [14] provides consistent distributed system replay, but is implemented as a userspace library and thus does not support multithreaded execution at the nodes (multithreaded applications are supported, but execution of the threads is serialized). It uses techniques similar to Jockey [29] to record local nondeterminism introduced through `libc` library calls, and additionally tags network messages sent between nodes in the system with Lamport clocks to coordinate replay. All messages received over the network by `liblog` are logged to disk, along with the time they were received.

DejaVu [18] provides record/replay for distributed Java applications and is implemented in a custom JVM. Cooperative ReVirt [2] provides record/replay for arbitrary distributed applications. DejaVu, Cooperative ReVirt and DDOS all make the same observation: internal network messages can be regenerated during replay to save log space. Unlike DDOS, however, both DejaVu and Cooperative ReVirt must record all local-node nondeterminism–including shared-memory access interleavings–so they are not able to achieve the same log size reduction that DDOS achieves by exploiting local-node determinism.

Deterministic Execution. Prior systems for deterministic execution have focused on single-node multithreaded programs. These systems eliminate the nondeterminism due to thread scheduling and shared-memory access interleavings. Some systems require custom hardware support [8, 9, 16] while others work purely in software [1, 3–5, 26]. A common approach divides execution into *quanta*, where communication is prevented for the majority of a quantum and then allowed to happen in a deterministic way when the quantum completes. The deterministic algorithm we proposed for DDOS is most similar to the buffering algorithm proposed by CoreDet [3]: both buffer communication during the quantum, then release communication at the quantum boundary. However, CoreDet's algorithm does not support message queues, and further does not include the lazy-barrier optimization, which is vital for performance on networks that have much higher latency than shared-memory.

Kahn networks [17] are a deterministic model of network communication where processes communicate through unbounded point-to-point queues. It is difficult to schedule arbitrary programs onto Kahn networks without introducing deadlock [12]. Determinator [1] supports multi-node deterministic distributed execution through a limited tree-based communication model that is a restricted form of Kahn network. This limited communication model is mostly used to implement a deterministic distributed shared-memory system. Like all Kahn networks, Determinator does not support many-to-one operations such as a promiscuous UDP `recv`.

Distributed Systems. Our algorithm for deterministic execution is a vector clock [20] algorithm: a node's current vector clock is defined by its current global quantum combined with the end-of-quantum markers it has received from remote nodes. Consequently, our algorithm bears resemblance to other distributed systems algorithms, most notably those for taking distributed snapshots [21], which broadcast local time coordinates in similar ways to our algorithm.

Virtual synchrony [6, 31] provides an atomic multicast primitive. Processes can join one or more *process groups*, and messages can be sent by processes to these groups. A process does not need to be a member of a group to send that group a message. Virtual synchrony ensures that all nodes in the receiving process group receive the same set of messages and in the same order, even if those messages were sent from multiple processes. Virtual synchrony does nothing to address the nondeterminism at the nodes in the system. DDOS's implementation of a logically synchronous network, however, has one important difference: it guarantees *deterministic* communication, rather than simply guaranteeing a consistent order of message delivery.

8. Conclusions

The goal of our work is to understand and explore the role of determinism in distributed systems. We leverage prior work on deterministic multithreading to: 1) significantly reduce log sizes when recording execution of distributed applications; and 2) deterministically execute entire distributed applications from the ground up. We proposed the first algorithm and system for deterministic execution of arbitrary distributed applications and prototyped it in a system we call DDOS.

We have shown that our record/replay mechanism significantly reduces log-sizes, resulting in log savings of 70% in some cases, although at a cost of an order-of-magnitude slowdown in application performance. We also evaluated our deterministic execution algorithm and showed that we can completely eliminate the log of communication between the nodes of a distributed system, although the cost of doing so is usually higher.

Acknowledgements

We thank the members of the UW Sampa and systems research groups for their feedback and fruitful discussions. We also thank our anonymous reviewers and our shepherd, Bhuvan Urgaonkar, for their guidance. This work was supported in part by the National Science Foundation CAREER award 0846004, NSF grants CNS-1016477 and CNS-1217597, a Google PhD fellowship awarded to Tom Bergan, and a Microsoft Faculty Fellowship.

References

[1] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.

[2] M. Basrai and P. M. Chen. Cooperative revirt: Adapting message logging for intrusion analysis. Technical Report CSE-TR-504-04, University of Michigan, 2004.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[5] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe and Efficient Concurrent Programming. In *OOPSLA*, 2009.

[6] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[7] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SIGMETRICS SPDT*, 1998.

[8] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[9] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, 2011.

[10] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[11] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[12] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *EMSOFT*, 2005.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[14] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Abstract replay debugging for distributed applications. In *USENIX Annual Technical Conference*, 2009.

[15] M. Hill and M. Xu. Racey: A Stress Test for Deterministic Execution. `http://www.cs.wisc.edu/~markhill/racey.html`.

[16] D. Hower, P. Dudnik, D. Wood, and M. Hill. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.

[17] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Information Processing*, pages 471–475, 1974.

[18] R. Konuru. Deterministic replay of distributed java applications. In *In Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium*, pages 219–228, 2000.

[19] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*, 2010.

[20] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[21] L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM TOCS*, 3(1), 1985.

[22] L. Lamport. The Part-Time Parliament. *ACM TOCS*, 16(2), 1998.

[23] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE TC*, 36(4), 1987.

[24] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.

[25] NASA Advanced Supercomputing Division. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[26] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[27] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[28] M. Ronsse and K. D. Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM TOCS*, 17(2), 1999.

[29] Y. Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *International Symposium on Automated Analysis-driven Debugging*, 2005.

[30] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[31] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4), April 1996.

[32] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, 2011.