Deterministic Process Groups in dOS

Tom Bergan Nicholas Hunt Luis Ceze Steven D. Gribble

Department of Computer Science & Engineering, University of Washington

Abstract

Current multiprocessor systems execute parallel and concurrent software nondeterministically: even when given precisely the same input, two executions of the same program may produce different output. This severely complicates debugging, testing, and automatic replication for fault-tolerance. Previous efforts to address this issue have focused primarily on record and replay, but making execution actually deterministic would address the problem at the root.

Our goals in this work are twofold: (1) to provide fully deterministic execution of arbitrary, unmodified, multithreaded programs as an OS service; and (2) to make all sources of intentional nondeterminism, such as network I/O, be explicit and controllable. To this end we propose a new OS abstraction, the *Deterministic Process Group* (DPG). All communication between threads and processes *internal* to a DPG happens deterministically, including implicit communication via shared-memory accesses, as well as communication via OS channels such as pipes, signals, and the filesystem. To deal with fundamentally nondeterministic *external* events, our abstraction includes the *shim layer*, a programmable interface that interposes on all interaction between a DPG and the external world, making determinism useful even for reactive applications.

We implemented the DPG abstraction as an extension to Linux and demonstrate its benefits with three use cases: plain deterministic execution; replicated execution; and record and replay by logging just external input. We evaluated our implementation on both parallel and reactive workloads, including Apache, Chromium, and PARSEC.

1. Introduction

Nondeterminism makes the development of parallel and concurrent software substantially more difficult. Software testers face daunting incompleteness challenges because nondeterminism leads to an exponential explosion in possible executions [27]. Developers must reason about large sets of possible behaviors and attempt to debug without precise repeatability [31, 36]. Moreover, standard techniques for fault-tolerant replication do not work when the software being replicated executes nondeterministically [38]. At the same time, the growing popularity of multicore architectures is making parallel and concurrent software more and more important.

Unfortunately, nondeterminism is pervasive; thread scheduling, memory reordering, and timing variations at the hardware level can all affect the interleaving of threads and cause a multithreaded program to produce different outputs when given the same input. We define this as *internal nondeterminism*. Internal nondeterminism is entirely hidden from the programmer and thus is undesirable. However, as we demonstrate in this paper, it is not fundamental and can be completely removed. On the other hand, events such as user input and the arrival of network packets are triggered nondeterministically by the external world. We define this as *external nondeterminism*; this kind of nondeterminism, if present, is fundamental and cannot be removed.

What we want is a software environment where internal nondeterminism is completely eliminated. What we want is more than just deterministic record and replay: multithreaded programs should always execute deterministically relative to their explicitly specified inputs. Moreover, where external nondeterminism exists, it should be made explicit and controllable.

Recent research has begun to explore ways of reducing internal nondeterminism in multithreaded programs. However, current proposals fall short in several aspects: they do not deal with nondeterministic channels other than shared-memory; they do not offer ways of making external nondeterminism explicit and controllable; they either require new hardware [14], apply to only a subset of programs [7, 29], or require recompilation [6]; and they do not support multiprocess applications.

Our goals are to completely eliminate nondeterminism where possible, including channels beyond shared-memory like pipes, signals, and the filesystem, and to make all intentional, external nondeterminism explicit and controllable. To this end, we propose a new OS abstraction, the Deterministic Process Group (DPG). A programmer uses this abstraction to define a *deterministic box* inside which all communication happens deterministically. All of the nondeterministic input received by a DPG is interposed upon by the *shim layer*, an interface that can be used by programmers to observe and control external nondeterminism in a flexible way.

A DPG is effectively a high-level deterministic virtual machine. The deterministic guarantees are provided transparently by the OS without intervention from the programmer; thus, DPGs can host arbitrary, unmodified application binaries. At any given time there may be many DPGs running alongside many conventional nondeterministic processes. An alternative design is full-system determinism, in which a hypervisor executes an entire OS deterministically relative to inputs triggered by the hardware. The DPG approach is more flexible because the programmer can select the desired granularity of determinism for each individual application.

1.1 DPG Use Cases

Debugging and Testing Many applications do not continuously interact with the external world, but instead read inputs at deterministic points in their execution. Since DPGs provide internal determinism by default, these applications will execute completely deterministically when run within a DPG. This has obvious benefits for debugging, since execution is directly repeatable. Moreover, removing internal nondeterminism has the potential to reduce the problem of testing multithreaded programs to the problem of testing sequential programs by making execution a function of only the explicit inputs, including external nondeterminism.

Record/Replay Controlling external nondeterminism with the shim layer makes determinism useful even for applications that interact continuously with the external world. As an example, one can run an application inside a DPG and extend the shim layer to log all external nondeterminism. This log can be used later to faithfully replay an application's execution for debugging and other analyses. Most prior work on record and replay of multithreaded applications focuses on how to record internal nondeterminism caused by shared-memory accesses. This leads to either unwieldy logs and high overheads [16, 22] or imprecise replay [1, 31, 36]. The internal determinism offered by DPGs completely subsumes this problem; only external inputs need to be recorded.

Replication for Fault Tolerance DPGs naturally enable replication of multithreaded applications. By running multiple copies of an application inside DPGs on several machines and replicating the inputs, all replicas will behave the same way because there is no internal nondeterminism. This can be implemented by extending the shim layer to ensure that all replicas receive the same input at the same point in their execution. Because DPGs eliminate all forms of internal nondeterminism, there is less to log and replicate. This is a major issue in prior work [5, 38–40] on replication mostly because shared-memory is a very large source of such nondeterminism.

1.2 Outline and Contributions

This paper makes several conceptual and architectural contributions. First, we identify the fundamental distinction between internal and external nondeterminism, and we demonstrate that internal nondeterminism can be eliminated from programs. To do this, we expand on earlier work that removed shared-memory nondeterminism by also removing internal nondeterminism from signals, pipes, the filesystem, and other OS channels.

Second, we propose the Deterministic Process Group abstraction (Section 2), which lets programmers define the boundary between internal and external nondeterminism. As part of this abstraction we introduce the shim layer, whose interface lets programmers observe and control all external nondeterminism.

This paper also presents and evaluates our implementation of these ideas. In Sections 3–4, we describe dOS, a Linux-based implementation of DPGs and the shim layer that enables the deterministic execution of arbitrary, unmodified binaries. Section 5 demonstrates the usefulness of the shim layer by using it to implement deterministic filesystem services, replicated execution of a multithreaded server, and record/replay. Section 6 provides a detailed evaluation of dOS and our shim applications on a variety of workloads. Finally, we end with related work and closing remarks.

2. The Abstraction

Figure 1 illustrates the abstract model of a Deterministic Process Group and Figure 2 illustrates the major components of our system. A DPG consists of a group of threads and processes along with the kernel objects they share. Kernel objects include shared-memory pages, pipes, and sockets. Threads communicate by performing operations on shared kernel objects, for example by reading from a shared page or writing to a shared pipe. A kernel object is *internal* if it can be modified only by threads inside the DPG, and is *external* if it can be modified by threads or devices outside the DPG. We refer to a thread executing inside a DPG as a *deterministic thread*, and we refer to a DPG's set of threads and internal objects collectively as a *deterministic box*.

Figure 1 shows three deterministic threads, $Thread_1$, $Thread_2$, and $Thread_3$, two internal objects, the `memory page` and the `pipe`, and two external objects, the `socket` and the `file`. $Thread_1$ and $Thread_2$ are members of the same process, P_a. The deterministic box is illustrated with a dotted outline. Note that internal objects need not be shared by the entire DPG; in this example, the `memory page` is shared by just two threads.

The final component of a DPG is a user-space service called a *shim program*. A shim program sits on the boundary of a deterministic box, and its job is to interpose on communication that crosses the deterministic boundary. Shim programs are written using a system call interface called the *shim layer*. This interface provides new opportunities for systems programmers that we explore in detail throughout this paper.

2.1 DPGs and Their Guarantees

A new DPG is created with the `sys_makedet` system call, and initially hosts just the calling thread. Each new thread spawned by the initial thread is added to the DPG, and in this way the DPG expands to include all descendant threads and processes. A thread leaves a DPG when it exits. We have not found DPG `join` and `leave` primitives necessary and so have not defined them. Threads

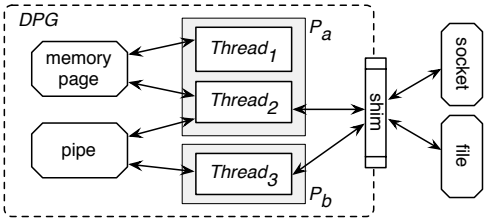

Figure 1. A Deterministic Process Group

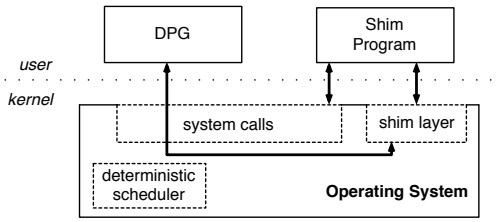

Figure 2. System Overview

```
void shim_attach(tid, SYS|SIG)
void shim_trace(*event)
void shim_resume(tid, result)
void shim_queue_sig(tid, siginfo)
void shim_ctl(tid, ...)
```

(a) Interposing on Nondeterminism

```
void shim_sleep(tid)
void shim_add_barrier(tid, logical_time)
int  shim_gettime(tid)
```

(b) Controlling Logical Time

Figure 3. Shim layer system calls

hosted in a DPG need not share an address space, which means a DPG can host many multithreaded processes.

Deterministic threads invoke system calls and read and write shared-memory just like ordinary threads. However, DPGs distinguish between operations on internal objects, which happen deterministically, and operations on external objects, which happen nondeterministically. Interactions with external objects represent a DPG's *only* source of nondeterminism; essentially, these external interactions represent the inputs a DPG receives from the external world.

Given the same initial state and the same stream of external inputs, a DPG is *guaranteed* to execute the same steps of inter-thread communication and produce the same output. More precisely, as a DPG executes it performs shared-memory `loads` and `stores`, invokes system calls, and handles asynchronous signals; each of these operations introduces nondeterminism *only* when it involves an entity outside the DPG. This is a stronger guarantee than output determinism [1, 23], which guarantees that replaying a program will produce the same output, but not that it will reproduce all inter-thread communication steps that lead to that output.

For example, when operating on a network socket, the `read` system call returns nondeterministic data. Additionally, `read` is a *blocking* call; it does not return until data is available, which means `read` will block for a nondeterministic amount of time. However, when `read` operates on a device that is internal to a DPG, such as an internal pipe, `read` behaves deterministically.

In summary, a DPG experiences nondeterminism only when it: (1) reads data from an external source; (2) blocks to wait for external data; or (3) handles a signal sent from an external source. Our guarantee is that DPGs execute deterministically relative to a stream of such nondeterministic input, and also relative to the initial state of the DPG at the call to `sys_makedet`. Note that this guarantee holds even across different machines.

Logical time Conceptually, a DPG executes as if it was serialized onto a *logical timeline*, where *logical time* is represented by a single global counter. Blocking system calls occupy two points on the logical timeline, one to initiate the call and the other to complete the call. dOS ensures that internal communication is mapped onto the logical timeline in a deterministic way. (Section 3 describes how our implementation groups instructions into atomic epochs in order to extract parallelism.) Note that logical time and physical time are distinct: DPGs guarantee deterministic output, but not deterministic performance. Input from the external world is mapped onto the logical timeline in a way controlled by the shim layer, which is the subject of the next section.

2.2 The Shim Layer

Every DPG is monitored by a user-space service called a shim program, also referred to as a *shim*. Shim programs use the shim layer interface (Figure 3) to observe and control nondeterministic input.

At a high level, there are two kinds of nondeterministic input: the *what* and the *when*. The *what* includes the values of external input, such as data read from the network. Then *when* includes the blocking times of nondeterministic system calls, as well as the delivery times of external signals. Shims can observe and control both kinds of nondeterministic input.

As a motivating example, consider record and replay implemented with a pair of shim programs. The record shim observes execution: for every nondeterministic system call, the shim logs the number of logical time steps the call spent blocked, along with the return value of the call. The replay shim controls execution: it ensures that every nondeterministic system call is scheduled to return at the specific logical time and with the specific value specified in the log.

The following sections first describe how shims observe and control the *what* (Figure 3a), and then how shims observe and control the *when* (Figure 3b), using record and replay as running examples.

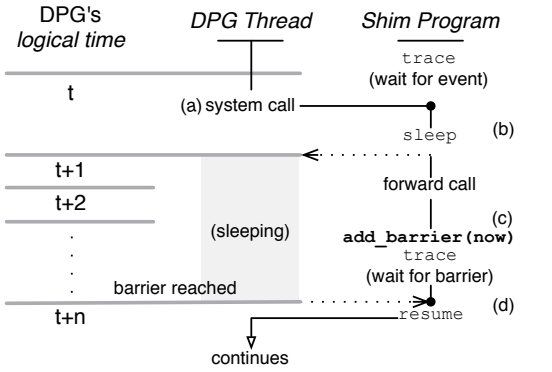

Figure 4. Observing a blocking system call

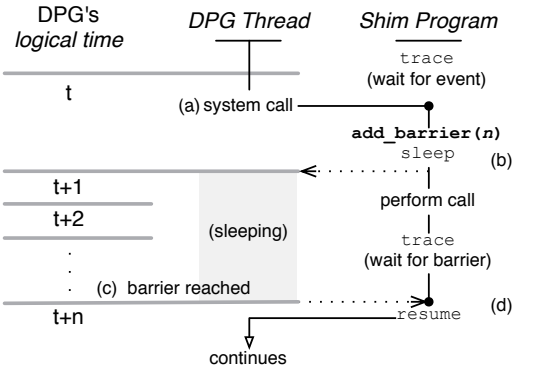

Figure 5. Controlling a blocking system call

2.2.1 Interposing on Nondeterminism

Shims use `shim_trace` to wait for a DPG to encounter nondeterminism. `shim_trace` blocks until either (a) a deterministic thread is about to perform a nondeterministic system call, or (b) an external signal is about to be delivered to a deterministic thread. In both cases, the deterministic thread stalls, execution transfers to the shim program, and `shim_trace` returns. The shim can interpose on this nondeterministic event and then return control back to the deterministic thread by calling `shim_resume`. In this way, execution of a deterministic thread alternates between itself and a shim program, much like execution of an ordinary thread alternates between user-space and kernel-space.

For system calls, `shim_trace` populates the given $event$ structure with the system call number and arguments. The shim should perform the system call on behalf of the deterministic thread and then transfer control back to deterministic thread by calling `shim_resume`, using the $result$ parameter of `shim_resume` to specify the system call's return value. The shim might perform the call by forwarding the call to the OS (*e.g.*, for record) or by ignoring the OS entirely (*e.g.*, for replay).

For external signals, the $event$ structure includes the `siginfo_t` of the pending signal. The shim can queue the signal for delivery by calling `shim_queue_sig`, save the signal internally for later delivery, or discard the signal entirely. In each case, the shim returns control to the deterministic thread by calling `shim_resume` with an empty $result$.

2.2.2 Controlling Logical Time

A shim program monitors the passage of logical time in a DPG by registering logical time barriers using `shim_add_barrier`. A logical time barrier is a timer tied to a specific deterministic thread (through the tid parameter); when the timer goes off, the deterministic thread stalls and the shim is notified through `shim_trace`. The barrier time is specified as an offset relative to the current logical time of the DPG, which can be obtained with `shim_gettime`. Time barriers can be used to control the nondeterministic *when*, as described below.

System Call Blocking Time Figures 4 and 5 illustrate how to observe and control the number of logical time steps that a system call blocks. Both examples follow a similar pattern; the only difference is the way in which `shim_add_barrier` is called.

Figures 4 illustrates observing a blocking system call (*e.g.*, for record). When deterministic thread T performs a system call (a), the call is trapped by the shim, which returns from `shim_trace`. At this point, thread T stalls and the DPG's logical time does *not* advance. The shim can now forward the call to the OS, but before doing so it puts T to sleep by calling `shim_sleep` (b). While T is asleep it is detached from the logical timeline and does not execute; this allows a nondeterministic amount of logical time to pass in the DPG while the system call is being performed. When the system call finally completes, the shim synchronizes with the DPG by registering a time barrier for T to happen at the very next logical time step in the DPG (c). Once that barrier triggers, the shim returns control to T via `shim_resume` (d).

Figure 5 illustrates controlling a blocking system call (*e.g.*, for replay). Again, deterministic thread T performs a system call which is trapped by the shim via `shim_trace` (a). The key difference in this example is that the shim decides, *a priori*, that the system call should complete in exactly n logical time steps. For example, a replay shim would read n from a log. To enforce this, the shim registers a barrier for T that will trigger n steps in the future and then puts T to sleep (b). While T is asleep it does not execute; the rest of the DPG executes normally for exactly n logical time steps, but no further. At this point the barrier triggers: T wakes and notifies the shim (c). Finally, the shim returns from the system call and returns control to thread T (d).

Signal Delivery Time Now suppose a shim wants to deliver a signal to thread T at logical time n. To do this, the shim should simply register a barrier for time n. When that barrier is reached, the shim can queue the

signal for immediate delivery using `shim_queue_sig` and then resume the thread using `shim_resume`.

2.2.3 Shim Use Cases

Shim programs can implement the record/replay and replicated execution services discussed earlier, but we envision many other kinds of shim programs as well. Some shims will be generic, application-independent services written by systems programmers, while others will be written by application programmers and tailored to enhance a specific application. Additionally, a shim program can be used to adjust the boundary of a deterministic box in two ways described below.

Expanding the Set of Deterministic Services An OS that supports DPGs may decide to implement some system calls nondeterministically to reduce kernel complexity, even when deterministic implementations are possible under the right assumptions. For example, in dOS, interaction with local files remains nondeterministic due to variations in disk latency, even though this nondeterminism can be considered internal and thus eliminated under the right assumptions. Section 5.1 explores how a shim can make local file access deterministic.

Further, a shim can virtualize global resources such as process identifiers in a deterministic way, as in [30]. A shim can even convert physical times (*e.g.*, used by `sleep` and `alarm`) into virtual, logical times. This would eliminate nondeterminism introduced by real time, but of course is only meaningful for applications that do not require a precise correspondence with real time.

Customizing the Nondeterministic Interface System calls are a DPG's basic interface to the nondeterministic world. However, it is often beneficial to let applications define the nondeterministic interface at a more abstract level. For example, a server application might want to hide many low-level `read` and `write` system calls behind a single high-level, nondeterministic `getmsg` call. Previous work has argued that this flexibility is valuable for record/replay systems [19], but we consider this flexibility to be even more general; for example, Section 5.3 shows how it is useful for replicated execution.

We enable this flexibility in dOS by defining a new system call, `dpg_callshim`, which makes a direct call from a DPG into its shim. Effectively, `dpg_callshim` allows developers to divide an application into two parts: the deterministic part that runs in a DPG and the nondeterministic part that runs in a shim.

3. Deterministic Execution Algorithm

The first implementation choice we make is which algorithm dOS uses to enforce determinism. Prior work on shared-memory determinism has proposed a family of deterministic execution algorithms, including DMP-O, DMP-B, and DMP-TM [6, 14]. dOS implements the DMP-O ownership-tracking algorithm; we selected it for its relative simplicity of implementation, but any deterministic execution algorithm can support DPGs as long as the shim layer can be implemented on top of it.

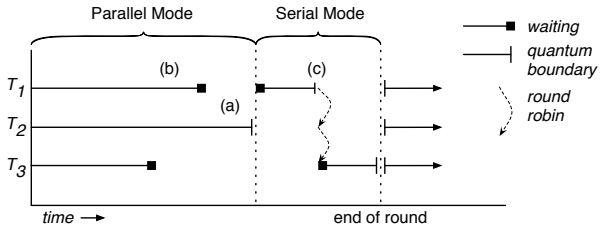

Figure 6. Timeline of a quantum round in DMP-O. T_2 finishes its quantum in parallel mode (a), while T_1 and T_3 have work left for serial mode (b,c).

One constraint imposed by the shim layer is that logical time should be representable with a single global counter. DMP-O, DMP-B, and DMP-TM all satisfy this constraint, but other (as yet uninvented) algorithms may require a more complex notion of logical time, such as a vector clock. We believe the shim layer could be extended to support such algorithms, but the details are left for future work.

In the rest of this section, we first summarize our earlier work on using DMP-O to enforce deterministic execution of multithreaded programs that communicate via shared-memory. Next, we describe how to generalize DMP-O to include communication via channels other than shared-memory, such as pipes and signals.

3.1 Shared-Memory Determinism

Two key observations underlie DMP-O. First, if threads do not touch shared data, *i.e.*, if they do not communicate, their execution will be deterministic no matter how they are scheduled. Second, when threads do communicate, a trivial deterministic schedule is to divide each thread's execution into chunks and then execute all chunks in a deterministic serial order.

Following these observations, execution in DMP-O is divided into chunks called *quanta*. A *round* consists of all threads executing one quantum each. Each round is divided into a *parallel mode* and a *serial mode*. In parallel mode, threads run in parallel but are isolated; they do not communicate. In serial mode, threads run serially but can communicate arbitrarily. A thread ends its parallel mode once it has reached an instruction that might communicate with other threads. Serial mode begins once all threads have completed parallel mode, and ends once all threads have had a chance to run. The parallel and serial modes are thus isolated by global barriers into two-stage rounds, as illustrated in Figure 6.

Notice that the parallel and serial modes are directly inspired from the two key observations stated above. DMP-O is deterministic as long as threads are (1) bro-

ken into quanta at deterministic boundaries, (2) ordered deterministically in serial mode, and (3) correctly isolated in parallel mode. The first two constraints are easily satisfied: we define a quantum to be some deterministic number of *dynamic* instructions, and we order threads in serial mode by sorting them by creation order.

DMP-O achieves isolation in parallel mode by partitioning ownership of shared-memory across threads. Each memory location is in one of two ownership states: *owned-by-T* for some thread T, or *shared*. A location that is *owned-by-T* is private to T; no other thread can access the location during parallel mode. A location that is *shared* is globally read-only; all threads can read the location during parallel mode, but none can modify it. A thread waits for serial mode before performing an operation that does not meet these conditions.

Ownership states evolve during serial mode by following two rules: (1) before thread T writes to a location, it sets ownership of that location to *owned-by-T*; and (2) before T reads a location that is not *owned-by-T*, it sets ownership of that location to *shared*.

Logical time Finally, we say that logical time increments on every *mode transition*, *i.e.*, on every transition from parallel mode to serial mode and back. Note that within a single mode every thread appears to execute atomically. From this property it follows that mode transitions are meaningful increments of logical time.

3.2 Beyond Shared-Memory Communication

Our model of a DPG from Figure 1 is that threads communicate by performing operations on shared kernel objects, which includes more than just shared-memory. To generalize DMP-O to this model we first observe that we can track ownership of shared kernel objects just as for shared-memory locations: if an operation mutates a kernel object it acts as a "write," while if an operation only observes a kernel object it acts as a "read." In fact, our implementation (Section 4.1.1) tracks ownership of shared-memory at the page granularity, effectively treating a memory page as just another kernel object.

To fully generalize DMP-O we need two additional changes: the first deals with blocking operations, and the second deals with asynchronously delivered signals.

Blocking Operations When a system call blocks, the calling thread ends its current mode (either parallel mode or serial mode) and is not scheduled to run again until it unblocks. While a thread is blocked, the rest of the DPG continues to execute. A thread can only unblock during a mode transition; this ensures that threads unblock at discrete points on the logical timeline.

Signal Delivery Incoming signals are queued during the current mode then delivered immediately on the next mode transition. Queued signals are partitioned into *internal* and *external* signals, depending on whether they were sent from a thread inside or outside the DPG, respectively. If there are N threads in a DPG then each deterministic thread has N logical queues: one queue for external signals and one queue for signals sent from each of the $N-1$ other deterministic threads.

On a mode transition, internal signals are delivered first and external signals are delivered last. The internal signal queues are emptied in a deterministic order, *e.g.*, by using the ID of the sending thread as a sort key.

This strategy ensures, first, that internal signals are delivered deterministically, and second, that external signals are delivered at meaningful logical times. Note that when a thread sends a signal to itself (as with `SIGSEGV`) the signal is synchronous; such signals are always delivered instantly. dOS implements the N-queue model described here using a single sorted list. Additionally, dOS always delivers external `SIGKILL` signals immediately (rather than forwarding them to the shim) so that a DPG can be killed even when its shim program misbehaves.

4. Linux-Based Implementation

We now describe how we implemented dOS, which is a variant of Linux that implements the DPG abstraction. dOS makes two major changes to Linux: first, it implements the shim layer; and second, it implements DMP-O, which includes an object ownership-tracking mechanism and a deterministic scheduler that constrains the execution of each DPG to a deterministic logical timeline. dOS exports a traditional system call interface to DPGs along with the `sys_makedet` and `dpg_callshim` system calls.

Our implementation of DMP-O was the most challenging and invasive change. Overall, we added roughly 5800 lines of new code to the Linux 2.6.24-7/x86-64 kernel and changed roughly 2500 lines of existing code in 53 files. Below we summarize the low-level implementation details of dOS and discuss engineering challenges (Sections 4.1–4.4). We end with a summary of the strengths and limitations of our implementation (Section 4.5).

4.1 Ownership Tracking

4.1.1 Shared-Memory Pages

dOS tracks ownership of shared-memory at the page granularity by using hardware page-protection to verify that a deterministic thread does not access a page without appropriate ownership.

Conceptually, dOS maintains a *shadow page table* for each thread. A thread's shadow table mirrors its real page table exactly, except that shadow permission bits are modified to reflect the current distribution of page ownership. dOS exposes only the shadow page tables to hardware: on a context switch to thread T, dOS installs T's shadow table onto the CPU even if the previously scheduled thread shared an address space with T.

Page ownership is encoded into shadow page table permissions so that ownership violations such as a `store` to a *shared* page will trigger a page fault. dOS intercepts this page fault, notices it is due to an ownership violation, and stalls the faulting thread until it is scheduled to run in serial mode. dOS then assigns ownership of the page to the faulting thread and continues its execution.

Every conventional process has one real page table representing its *address space*. All address space modifications are expressed in terms of the real page table and then transparently applied to the shadows. To limit memory overheads, dOS maintains just N shadow tables per address space, where N is the number of CPUs, and then assigns threads to shadow tables, effectively bucketing the threads in a given process into N ownership groups. This requires a slight tweak to the DMP-O scheduler: during parallel mode, all threads that share a shadow table must be serialized in a deterministic order (*e.g.*, scheduled serially in thread creation order). We bucket threads using a simple greedy algorithm.

This strategy is not limited to shared-memory within a single process. dOS supports shared-memory across processes by tracking ownership of physical pages; we use Linux's `rmap` facility to enumerate all user-space addresses that map a given physical page.

Finally, there are two corner cases worth mentioning. First, dOS disables address space randomization for DPGs so that every DPG has a deterministic address space layout. Note that we can enable address space randomization in DPGs if we expose the seed as external nondeterminism. Second, page swapping can introduce nondeterministic changes to page tables. To preserve determinism, when a page is swapped out, dOS preserves the page's ownership state using extra bits in the shadow page tables. When a page fault triggers a swap-in, dOS stalls the thread until the page is read from disk, and then restores the saved ownership state of the page.

4.1.2 Other Kernel Objects

Other kernel objects, such as pipes and sockets, are operated on by system calls. dOS instruments the kernel so that a system call never operates on a kernel object unless the calling thread has the appropriate level of ownership.

Adding this instrumentation presents two engineering challenges. First, where should the instrumentation be placed? It is tempting to lazily acquire ownership of an object just before a system call actually uses the object, but doing this requires reengineering kernel locking protocols. To see why, note that acquiring ownership may require sleeping the calling thread to wait for serial mode. However, a system call may not decide to use an object until inside an atomic region, *e.g.*, while holding a spin lock, and it is not safe to sleep in such regions.

dOS avoids this difficulty by conservatively acquiring ownership of all objects a system call may use before executing the call. This requires adding instrumentation in just two places: at the system call entry point, and in the code that wakes up a thread. dOS instruments thread wakeup to reacquire any privileges lost while the system call was asleep, *e.g.*, while waiting for input.

Behavior	(Total Syscalls) Examples
use *pages*	(14) mprotect, read
use *address space*	(6) mprotect, mmap, brk
use *inode*	(32) read, write, lseek, close
use *fd table*	(9) open, dup, close
use *fs path*	(22) open, chdir, chroot, access
read *untracked*	(54) getpid, gettimeofday
modify *untracked*	(172) kill,setrlimit,sigaction

Table 1. System call behaviors

The second challenge is that Linux is a large, complex system with over 250 system calls and many unique types of kernel objects. To simplify our implementation, we track a few kinds of kernel objects precisely and then conservatively merge all other kinds of objects into an *untracked objects* group. For all but the untracked objects, dOS tracks ownership using a hash table that maps an object to its current owner. Freshly allocated objects are initially *owned-by* the allocating thread. Ownership of the untracked objects is implicit: during parallel mode they are *shared*; and during serial mode they are *owned-by* the thread currently running. Thus, read-only operations on untracked objects can execute in parallel mode, while all other operations on untracked objects must wait for serial mode. This strategy is summarized in Table 1.

An `inode` is Linux's internal name for files, sockets, pipes, and anything else that can be referenced by a file descriptor. System calls like `read` that operate on file descriptors can modify the contents of memory pages, map new pages into the address space, or even modify the `inode` itself. These system calls must acquire ownership of all of these objects before proceeding.

4.2 Scheduling

The dOS scheduler is implemented as a filter in front of the default Linux scheduler—it does not push a deterministic thread into the Linux scheduler until the thread has been scheduled to run by its DPG. This filter implements the DMP-O scheduling algorithm.

Thread Creation The `fork` and `clone` system calls always execute in serial mode. This ensures that deterministic threads are spawned in a global serial order. The newly spawned thread will be scheduled to run during the next parallel mode.

Logical Time Barriers The dOS scheduler checks for pending time barriers on each mode transition. To prevent deadlock, dOS instantly fast-forwards logical time to the next pending time barrier whenever all threads in a DPG are simultaneously asleep.

Quantum Formation Recall that parallel mode ends when all threads have either reached a quantum boundary or stalled to acquire ownership, and serial mode ends once all threads have reached a quantum boundary, where quantum boundaries must occur at deterministic points in a thread's execution. A possible implementation is to mark quantum boundaries with system calls, but this does not guarantee forward progress because a thread may loop forever without making any system calls. Additionally it does not guarantee *balance*; imbalance leads to excessive waiting at the end of parallel mode, which leads to poor performance [6].

To guarantee forward progress, dOS defines a *quantum budget*, which is the maximum amount of work a thread can perform in a quantum. dOS estimates work by counting instructions. The quantum budget is simply a deterministic number of instructions, typically in the range of tens to hundreds of thousands of instructions.

dOS counts instructions using the hardware "instructions retired" counter that is available on all modern x86 CPUs. dOS configures this counter to trigger an overflow interrupt after the quantum budget expires. There are well-documented caveats about using this counter [15, 43]. Specifically, the counter suffers from nondeterminism that can be engineered around. We follow the solution outlined by [15]: to overcome imprecise interrupt delivery, dOS must single-step the DPG (via the x86 trap flag) for up to about 200 instructions per quantum, which can introduce large overheads. To avoid those overheads, as an optimization, dOS deterministically ends a quantum when returning from a system call if the remaining quantum budget is low, but not yet exhausted.

4.3 Additional Optimizations

As demonstrated in [6], DMP-O performs best when parallel mode is balanced and when serial mode is empty. dOS implements a few optimizations to bias execution towards these conditions. dOS automatically adjusts a DPG's quantum budget: when dOS detects significant parallel mode imbalance, the budget is decreased to reduce imbalance, and when dOS detects well-balanced parallel modes, the budget is increased to reduce quantum barrier overheads. To limit the time spent executing in serial mode, dOS ends a quantum after a few (heuristically determined) ownership transfers. All of these optimizations preserve determinism, since the parameters used evolve deterministically.

4.4 Shim Programs

In concrete terms, a shim program is composed of a collection of threads called *shim threads*. Shim threads begin life as ordinary user-space threads, *e.g.*, after being spawned by `fork` or `clone`. An ordinary thread becomes a shim thread by calling `shim_attach` to attach to some deterministic thread T. Once attached to T, the shim thread is the distinguished thread that will intercept all of T's nondeterminism through `shim_trace`. If the shim thread crashes, T will stall on external operations until attached to by another shim thread.

A thread can act as a shim thread for more than one deterministic thread. Additionally, to simplify the implementation of shims, a shim thread can elect to receive only the nondeterministic system calls or only the external signals for a given deterministic thread (by setting the second parameter of `shim_attach`). Our usual strategy is to spawn one shim process for every DPG. Within this process we spawn one shim thread to intercept signals for the entire DPG, and for every deterministic thread in the DPG we spawn one shim thread to interpose on the system calls performed by the corresponding DPG thread.

Intercepting System Calls When a shim program intercepts a system call it has two options: (1) it can emulate the system call completely; or (2) it can simply instrument the system call's entry and exit, allowing the deterministic thread to actually execute the body of the system call. These options resemble those allowed by `ptrace`.

The option to simply instrument a system call is selected by passing a special result to `shim_resume`. This option gives a shim limited control over how the system call executes in logical time. For example, if a shim simply instruments `read` instead of emulating it, the shim cannot observe or control when the kernel writes to the given user-space buffer (the writes will happen nondeterministically, in an unrecordable way). We provide instrumentation as a convenience for cases where full emulation is not necessary. During system call emulation, a shim can use `shim_ctl` to perform side effects in a DPG, such as writing to or reading from a user-space buffer.

RDTSC dOS allows shim programs to interpose on the nondeterministic `RDTSC` instruction. Our implementation uses the time stamp disable flag of the x86 `cr4` register to fault on user-mode accesses to `RDTSC`; these events are exposed to the shim via `shim_trace`.

4.5 Discussion

Guarantees Provided by dOS dOS guarantees that communication via the following kernel objects is deterministic as long as the objects are completely internal to a given DPG: shared-memory pages, including across multiple processes; pipes allocated with `pipe`; and `futexes` (used to implement pthreads synchronization). Additionally, dOS guarantees that file descriptors and memory pages are allocated in a deterministic order; that the address space evolves deterministically (as via `mmap`); that internal signals are delivered deterministically; and that `wait` is deterministically notified when threads in the same DPG exit.

Note that some system calls are deterministic except in error cases. For example, `mmap` allocates pages deter-

ministically within an address space, but will fail nondeterministically if there is not enough physical memory available to service the request.

Guarantees Not Provided by dOS Our deterministic guarantees may not translate across different versions of program binaries no matter how slightly different (*e.g.*, after a patch). Also, although our guarantees hold across different host machines, an application can read host configuration as part of its inputs, for example to dynamically adjust its resource usage; these inputs must be duplicated exactly to guarantee determinism.

Additionally, dOS does not guarantee deterministic access to shared-memory pages that can be modified by threads or devices outside the DPG. Ideally we might interpose on this external communication using the shim, but this would require adding excessive restrictions to non-DPG processes. For example, page ownership might transition between "exclusive to a DPG" and "exclusive to the external world," but this would require stalling external threads as they wait to reacquire page ownership. Relatedly, DPGs may encounter nondeterminism when memory is modified through backdoors in `/proc`.

Retrospective Implementing DPGs in a monolithic kernel such as Linux raises many thorny issues. The example of `mmap` is instructive: reasoning about the cases in which `mmap` is nondeterministic requires finding and reasoning about many code paths in a monolithic kernel.

More generally, providing determinism requires tracking and mediating accesses to shared OS objects. However, many Linux kernel objects have aliased names, are named in multiple namespaces, and are accessible through multiple interfaces. For example, process IDs are exposed through system calls, the `/proc` filesystem interface, and in some cases, thread-local storage variables in the address space of a multithreaded process. If we consider PIDs to be a source of internal nondeterminism, dOS must correctly track and reconcile PIDs through all of these channels, for instance, by virtualizing PID numbers before they are exposed to a program so that PID assignment is deterministic and consistent across processes within a DPG. Even if we consider PIDs a source external nondeterminism (the choice made by dOS), for record/replay to work correctly a shim program must interpose on all of these different channels for accessing PIDs, so that PIDs can be recorded and during replay the same PIDs can be reassigned.

An OS kernel implemented "from scratch" to support DPGs would benefit from design principles advocated by exokernels and microkernels. A minimal kernel interface combined with a libOS would push many of the aliased interfaces and complex code paths out of the kernel and inside the user-space deterministic box, making it easier to reason about determinism at the system call layer. The protection domains of a microkernel could further simplify many of these issues, since reasoning about nondeterminism would largely reduce to detecting messages that cross the boundary of a deterministic box. In the `mmap` example, this might be a message to the page-allocation server.

5. Shim Applications

To demonstrate the usefulness of the shim layer, we have implemented three shims: deterministic filesystem services; record/replay by logging just external input; and replicated execution of a multithreaded server. The deterministic filesystem service and record/replay shims can be used with unmodified application binaries, while the replicated execution shim is application specific. We note that the shim layer allowed us to quickly prototype the shims described in this section.

5.1 Deterministic Filesystem Services

FSSHIM provides applications with a *deterministic file hierarchy*. All reads and writes to files within this hierarchy are deterministic; accesses to files outside of this hierarchy are considered sources of external nondeterminism, as before. There are two sources of nondeterminism FSSHIM must eliminate: the latency of each operation, and the number of bytes operated on by the `read` and `write` system calls. FSSHIM eliminates the first by deciding, *a priori*, that each operation will block for a fixed and deterministic amount of logical time. For the second, FSSHIM guarantees that all `reads` and `writes` operate on a deterministic number of bytes by always performing the maximum amount of work requested (up to an end-of-file, for `reads`). FSSHIM can make these guarantees because it performs the `read` and `write` calls on behalf of the DPG, using the pattern illustrated in Figure 5.

The deterministic blocking time selected by FSSHIM can affect performance. For example, if FSSHIM selects a logical blocking time that is too low, the DPG will stall waiting for disk operations to complete. On the other hand, if FSSHIM selects a time that is too high, the calling thread will execute artificially slowly. The logical blocking times we chose for FSSHIM are equivalent to a delay of about 5 million instructions; we did not experiment heavily with this number.

A file can exist in the deterministic file hierarchy only if it can be considered *internal* to the DPG, which is true when: (1) the initial contents of the file are deterministic; (2) the file is not written by any threads outside the DPG; and (3) operations on that file complete in a finite time. In practice, the third assumption implies fail-stop. FSSHIM relies on the user to explicitly indicate the parts of the filesystem for which these assumptions are valid. This typically includes the directories containing program inputs, as well as directories shared system-wide that are rarely updated, such as `/usr` and `/etc`.

5.2 Record/Replay

RECSHIM records all external nondeterminism introduced through the system call interface and signals, enabling deterministic program replay. RECSHIM needs to record only the external nondeterminism because DPGs eliminate all forms of internal nondeterminism. Further, RECSHIM can be combined with FSSHIM, reducing what needs to be logged since accesses to files within the deterministic file hierarchy would be deterministic.

RECSHIM utilizes the shim layer to interpose on system calls and to intercept external signals. System calls that touch user-space memory are executed by RECSHIM on behalf of the DPG. RECSHIM produces a log containing the logical time the event occurred and any other event-specific information needed during replay. For system calls, this includes the return value and logical blocking time, as well as any side-effects of the system call, such as the contents of a buffer after performing a socket read. For signals, a copy of the `siginfo` is saved. Logs are compressed on-the-fly with zlib.

We have implemented a proof-of-concept replay shim to verify that the shim layer offers all the hooks necessary to implement a replay component. The major challenges in faithfully replaying system call traces are orthogonal to the main body of our work and have been explored by prior work [19, 36, 37].

5.3 Replicated Execution

REPLICASHIM supports replication of a multithreaded webserver running inside a DPG by guaranteeing that the order of messages and their logical arrival time is kept consistent across all replicas. Given the same inputs and the same logical arrival times, the DPG abstraction guarantees that all replicas will evolve deterministically.

Our target application is `nullhttpd` [12], a small, simple, multithreaded webserver that uses a thread-per-request model. Our design splits the functionality of the basic server into three separate process types: a single arbiter process, and a set of replicas, each composed of a shim process along with a DPG that hosts `nullhttpd`.

The arbiter process operates nondeterministically, outside of any DPG, and accepts incoming HTTP requests from the network. The arbiter broadcasts requests to the replicated shims, which queue the requests locally. We modified `nullhttpd` to read new requests by making a direct call to its shim via `dpg_callshim`, rather than reading from the network. This shows a case where the programmer defines the interface via which nondeterministic inputs are received.

When the arbiter broadcasts a request, it must ensure that all replicas see that request at the same logical time. It does this by performing a two-phase commit to determine a logical time that no replica has advanced beyond. The protocol works as follows. When the arbiter receives a new HTTP request from the network, it asks all replicated shims to set a barrier and report their current logical time. The arbiter uses the maximum value reported by any replica as the logical arrival time of the new request. The arbiter then broadcasts the new request and asks each replica's shim to set a second barrier for this arrival time; once this barrier is reached at a replica, the replica's shim makes the new request available to `nullhttpd` and the replicas continue to evolve deterministically.

6. Evaluation

The goal of our evaluation is to understand the performance of DPGs in comparison to ordinary nondeterministic execution (`Nondet`). We include evaluations of the three shim programs we built, namely FSSHIM, RECSHIM, and REPLICASHIM.

Correctness We tested our dOS implementation by running the `racey` [20, 45] deterministic stress test 500 times and verifying that `racey` always produces the same output. In addition to the basic `racey` program, we tested `racey` variants that exercise the various components of our implementation, such as communication via pipes, signals, and multiprocess shared-memory.

Workloads We evaluated the following parallel workloads: the `PARSEC` [8] and `SPLASH2` [44] benchmark suites; `pbzip2` [18] to compress a Linux ISO image; and `make -j` to perform a parallel build of the Linux kernel. The `PARSEC` and `SPLASH2` are workloads optimized for parallelism; we scaled their inputs to run for about a minute with a single nondeterministic thread. We present a representative subset of the `PARSEC` and `SPLASH2` benchmarks that was selected to showcase both the best-case and worst-case performance of dOS.

We also evaluated three reactive applications: the Apache and `nullhttpd` webservers and the Chromium web browser. Apache and Chromium are especially interesting because they use multiple processes with multiple threads per process. We evaluated the webservers using httperf [26] to simulate a constant stream of requests for static pages. We evaluated Chromium with two experiments: first, we measured the load time of `nytimes.com` (without any local caching); and second, we used Chromium's debugging facilities to execute a scripted user session that opened 5 tabs and navigated to 12 URLs in rapid succession. All Chromium experiments used the process-per-tab model [34].

We ran our experiments on 8-core 2.8GHz Intel Xeon E5462 machines with 10GB of RAM using `rundet`, a small utility that constructs a single DPG and then executes an unmodified application binary inside that DPG. We used a relatively aggressive machine configuration to adequately explore the scalability of our parallel workloads. All results shown are the average over ten executions, with the highest and lowest values removed.

Benchmark	Config: Num Proc	Config: Threads per Proc	Throughput: Nondet	Throughput: DPG only	Throughput: DPG + FSSHIM
apache 10KB	16	1	10.1K	3.6K	1.7K req/s
apache 10KB	4	4	10.1K	6.6K	2.2K req/s
apache 10KB	1	16	10.2K	7.4K	2.4K req/s
apache 100KB	4	8	1.1K	1.1K	0.9K reqs/s
nullhttpd 10KB	1	16	1.0K	1.0K	1.0K req/s
chromium	nytimes		1.8 s	2.4 s	3.8 s
chromium	scripted		22 s	37 s	40 s

Table 2. Reactive Workload Evaluation

Benchmark	Overheads (relative to Nondet): DPG Only 2-th	DPG Only 4-th	DPG Only 8-th	DPG + FSSHIM 2-th	DPG + FSSHIM 4-th	DPG + FSSHIM 8-th	Speedup (8-th over 2-th): Nondet	Speedup (8-th over 2-th): FSSHIM
blackscholes	1.2	1.2	1.3	1.2	1.3	1.3	3.4	3.2
dedup	2.3	3.6	4.0	4.0	5.8	6.4	1.6	1.0
fmm	2.6	6.1	10.1	2.6	6.0	10.1	2.4	0.6
lu	2.0	2.3	2.3	2.0	2.3	2.3	2.1	1.7
pbzip2	2.0	2.7	3.0	2.1	2.8	3.4	2.6	1.6
make	2.3	4.1	5.9	3.2	5.7	8.2	2.8	1.1

Table 3. Parallel Workload Evaluation

6.1 DPG Overheads

We start with two questions: what are the overheads of DPGs for typical workloads, relative to nondeterministic execution, and how much overhead is added by FSSHIM? To answer these, we ran our workloads in DPGs with no shim attached and in DPGs with FSSHIM attached.

Table 2 summarizes this evaluation for reactive workloads. The first few rows evaluate Apache for workloads of 10KB and 100KB static pages. For the 100KB workload, both the `Nondet` and the DPG-only case are able to saturate the gigabit network of the Apache server, in spite of the extra overhead of using the DPG. FSSHIM adds some additional overhead, enough to shift the system bottleneck to the CPU.

For the 10KB workload, the `Nondet` case is still able to saturate the network link. However, this workload involves a significantly higher rate of system calls and other nondeterministic events; each system call incurs a context switch from the DPG to its shim. As a result, both the DPG-only and the FSSHIM cases experience serialization and overhead that slows the request rate between 1.4x and 5.9x.

Throughput generally decreases as the number of processes (Column 2) increases. We suspect this is because interprocess communication is more costly when executing in a DPG. Note that scaling can be achieved by running multiple smaller instances of Apache in separate DPGs. Overall, we consider these throughputs reasonable for all but the most high-traffic web sites.

The last two rows show the execution time of Chromium. For the scripted session, latency increases by 1.7× for DPGs alone and by 1.8× for DPGs with FSSHIM. Latency increases from 1.8 seconds to just 2.4 seconds when loading `nytimes.com`. We also performed this test for a Google search results page (not shown). All execution times in the Google search results test were less than a second, and informally, the differences "felt" negligible when we interacted with the browser.

Benchmark	Config: Num Proc	Config: Num Thread	Exec Breakdown: % Serial Mode	Exec Breakdown: % Single Stepping	Serialization Reasons: % Pgfault	Serialization Reasons: % Syscall
apache 10KB	16	1	72%	< 1%	< 1%	99%
apache 10KB	4	4	80%	< 1%	< 1%	99%
apache 10KB	1	16	82%	< 1%	< 1%	99%
apache 100KB	4	8	26%	< 1%	2%	98%
nullhttpd 10KB	1	16	11%	0%	2%	98%
chromium	nytimes		58%	13%	61%	39%
chromium	scripted		25%	13%	72%	28%
blackscholes	1	8	3%	27%	99%	1%
dedup	1	8	54%	12%	77%	23%
fmm	1	8	90%	18%	100%	0%
lu	1	8	45%	35%	95%	5%
pbzip2	1	8	35%	39%	100%	0%
make	8	1	79%	3%	0%	100%

Table 4. DPG Execution Characterization

Table 3 shows execution overheads for our parallel workloads with 2, 4, and 8 threads. Overheads are generally below 3×, often lower than 2.5×. Columns 5-7 show the added cost of FSSHIM, which is typically small, since most of the applications do not perform a significant number of system calls (except `dedup` and `make`).

DPG scalability is closer to `Nondet` scalability when the overheads do not grow much with the number of threads. Scalability suffers for workloads like `fmm` that share frequently at finer than page-level granularity, but `blackscholes`, which does not have fine-grained sharing, has DPG scalability very close to `Nondet`.

Characterization Table 4 characterizes execution with DPGs with FSSHIM attached. Column 4 shows the fraction of time execution was serialized (*i.e.*, in serial mode). As expected, for the parallel workloads, serialization is highly correlated with overheads and scalability. `blackscholes` and `fmm` are good comparison points; `blackscholes` is 3% serialized and scales nearly ideally with DPGs, while `fmm` is 90% serialized and has poor scalability. For the reactive workloads, the relationship between serialization and performance is less clear, as shim context switch overhead and quantum imbalance are also important factors. The rightmost set of columns show the reason for serialization, broken down into ownership page faults (Column 6) and system calls (Column 7). In reactive workloads, most serialization happens due to system calls, which is expected because reactive workloads perform frequent I/O. Conversely, for parallel workloads (except `make`), most serialization is due to ownership page faults. Also, the fact that dOS uses page-level ownership tracking can lead to unnecessary serialization due to false-sharing.

Even though serialization is very low in blackscholes, the overheads are still on the order of 30%, largely because of single-stepping. Column 5 shows the fraction of execution during which at least one thread is single-

Benchmark	Overheads w/ FSSHIM	Overheads w/o FSSHIM	Log Sizes (per day) w/ FSSHIM	Log Sizes (per day) w/o FSSHIM	Log Sizes (per day) SMP-ReVirt (from [16])
fmm	6.0	6.0	1.1 MB	2.0 MB	83.6 GB
lu	2.4	2.4	11.0 MB	13.0 MB	11.7 GB
ocean	3.0	3.0	1.5 MB	3.6 MB	28.1 GB
radix	4.5	4.5	0.8 MB	2.1 MB	88.7 GB
water	4.8	4.8	5.3 MB	83.2 MB	58.5 GB
pbzip2	2.9	4.0	5.7 MB	295.7 GB	—

Table 5. RECSHIM for Parallel Workloads (4 threads)

stepping; this varies from 0% to 39%. One interesting trend is that reactive applications single-step less often; these applications perform system calls frequently, which triggers an optimization to end quanta early (Section 4.2). Note that single-stepping does not necessarily correlate with performance because serialization and quantum imbalance dominate. In addition to data shown here, we measured the increase in frequency of total page fault events due to ownership changes. While the frequency is often higher, it was not directly correlated with performance. Serialization, quantum imbalance, and single-stepping are the dominant factors.

In summary, DPG overheads are reasonable for several applications, including some parallel applications and most reactive applications. Broadly, overhead tends to increase with sharing, especially as the number of threads grows. We did not attempt to optimize applications for more "determinism friendly" sharing, which could improve performance.

Microbenchmark To more closely understand the overhead of intercepting system calls with a shim, we wrote a simple benchmark that does nothing but call `getpid` in a loop. We ran this benchmark both in a DPG without a shim, and in a DPG with a "null-shim." The null-shim configuration ran 5× slower, suggesting that dOS imposes an overhead of 5× on system call entry.

6.2 RECSHIM: Execution Recorder Shim

We next evaluated the overhead of using RECSHIM, and its resulting log sizes. Table 5 characterizes RECSHIM for parallel workloads. Columns 2-3 show the overheads for RECSHIM with and without a deterministic file hierarchy, respectively. These overheads are essentially identical to execution without RECSHIM (Table 3). Columns 4-5 show log sizes for a full day of execution. RECSHIM's log sizes are very small because DPGs eliminate internal nondeterminism; the remaining nondeterminism is due to a few system calls such as `gettimeofday`.

Not making filesystem accesses deterministic (Column 5) increases the sources of nondeterminism, leading to larger logs. This is especially true for `pbzip2`, which must log the entire ISO image. These log sizes, however, are still orders of magnitude lower than the sizes reported by SMP-ReVirt [16] (Column 6). This is because SMP-ReVirt needs to record internal nondeterminism (again, especially shared-memory), which

Benchmark	Config Num Proc	Config Threads per Proc	Throughput w/ FSSHIM	Throughput w/o FSSHIM	Log Sizes w/ FSSHIM
apache 10KB	16	1	1.7K req/s	1.6K req/s	48.6 B/req
apache 10KB	4	4	2.3K req/s	2.1K req/s	51.3 B/req
apache 10KB	1	16	2.2K req/s	2.2K req/s	50.4 B/req
chromium	nytimes		4.2 s	3.9 s	600 KB
chromium	scripted		40 s	43 s	3.3 MB

Table 6. RECSHIM for Reactive Workloads

Config	Throughput 1 replica	Throughput 2 replicas
`Nondet`	386 req/s	373 req/s
REPLICASHIM	369 req/s	372 req/s

Table 7. Replicated Execution Overheads

can be massive. Since SMP-ReVirt is a hypervisor, it logs nondeterminism internal to the OS, adding overhead for `radix` that a process-level implementation of SMP-ReVirt might be able to avoid. This is also an indication that determinism enforcement at the hypervisor level is likely to have a higher performance cost than when enforced at the process level.

Table 6 shows overheads and log sizes for RECSHIM when running reactive applications. Columns 4-5 show the throughput while recording, both with and without FSSHIM enabled. With FSSHIM enabled, RECSHIM did not reduce the throughput of the webservers from the results shown in Table 2. However, disabling FSSHIM resulted in a small performance decrease. The decreased performance is due to the overhead of logging additional input. The overheads for Chromium are about the same as those seen in Table 2. Column 6 shows log sizes normalized to the number of requests for the Apache runs, as well as total log sizes for Chromium sessions.

6.3 REPLICASHIM: Replicated Execution Shim

We end our evaluation by investigating whether we can we quickly build on DPGs to enable replication of an existing multithreaded application. To answer this, we built and tested REPLICASHIM, which replicates our modified `nullhttpd`. For a performance comparison, we also ran replicas outside DPGs but still using the same arbiter and replication protocol (`Nondet`). This configuration does not provide any deterministic guarantees. Table 7 shows the throughput for 1 and 2 replicas with 16 threads per replica; each replica ran on a separate machine while the arbiter ran on a third machine. In both cases, the throughput is essentially matched. Note we did not spend much time optimizing the arbiter or its simplistic protocol, as REPLICASHIM is only as a proof-of-concept; the arbiter is the major bottleneck in these experiments.

6.4 Summary

Our evaluation illuminated the impact of determinism on application performance and scalability. Workload is fundamental factor: applications with frequent inter-

thread or inter-process sharing will encounter more overhead and worse scalability when executed deterministically, since this communication must be tracked and controlled. Implementation choices also have a large impact. We suspect that much of the overhead in dOS is not fundamental and might be mitigated by using sharing-aware memory allocation, by fine-tuning integration with the Linux scheduler, or by using potential upcoming hardware support for transactional memory [2].

The choice of deterministic execution algorithm is another factor. Algorithms like DMP-O that provide a strict memory model or make heavy use of barriers will likely perform worse than those that that loosen the memory model or rely on alternative mechanisms such as speculation. dOS could have implemented the DMP-TM and DMP-B algorithms we developed in earlier work [6, 14]. Both algorithms have better demonstrated scalability than DMP-O and can both be implemented at the kernel level, but both algorithms are more complex. The key idea of DMP-B is to relax the memory model by using a store buffer, which allows concurrent writes in the same quantum round and therefore improves scalability. The key idea of DMP-TM is to use transactional memory to speculate that each quantum round is conflict-free and thus can be executed in completely parallel.

7. Related Work

Deterministic Execution There are a few recent proposals for removing internal nondeterminism in multithreaded execution. DMP [14] is a hardware proposal that includes two approaches for deterministic execution: DMP-O uses ownership tracking at a cache-line granularity; DMP-TM uses transactional memory [33] to further reduce the cost of determinism by speculating that there is no communication between threads. Kendo [29] proposes a software-only library that provides a set of deterministic synchronization operations that offer some deterministic guarantees for race-free programs. CoreDet [6] proposed DMP-B and used compiler and runtime system to provide determinism for arbitrary C/C++ programs. Grace [7] uses speculative execution to provide determinism for fork-join parallel programs. These proposals all describe algorithms for *execution-level determinism*, as used by DPGs. Unlike these prior proposals, however, DPGs support determinism beyond shared-memory in arbitrary binary programs and also provide a way to precisely control external nondeterminism.

Another approach is *language-level determinism*, which uses a parallel language that is deterministic by construction, such as StreamIt [41], SHIM [17], NESL [10], Jade [35], or DPJ [11]. The prime trade-off between execution-level and language-level determinism is one of generality and controllability. In language-level determinism, the programmer must use specific language constructs but gets explicit control of which deterministic executions are possible; in execution-level determinism the programmer can use any language (*i.e.*, determinism is fully transparent) but cannot control which deterministic executions will happen, making behavior less predictable at program construction time. While deterministic languages are a promising long-term solution, the majority of today's programs are written in mainstream languages such as C++ or Java, and this will likely remain the case for the foreseeable future. Additionally, parallel languages are often domain-specific and not well suited to general purpose, reactive applications; in contrast, we have used dOS to demonstrate how reactive applications can benefit from execution-level determinism.

Determinator [3] proposes to enforce determinism using a custom microkernel. Like dOS, Determinator supports multiple processes and uses page protection to enforce determinism of shared-memory accesses. Determinator supports both standard pthreads programs, via an implementation of DMP-B, as well as programs written using specialized parallel programming constructs that are designed to be deterministic. Unlike dOS, however, Determinator does not explore the separation between internal and external nondeterminism, and further, Determinator has no equivalent of the DPG shim layer interface for precisely controlling external nondeterminism.

Record/Replay Record and replay is a natural way to cope with internal nondeterminism during debugging. There are many proposals for software-based implementations of record and replay. Some record all shared accesses that lead to communication [22]; others assume uniprocessor execution and record only scheduling decisions [13]; others record only synchronization operations [36]. The high overheads of logging shared-memory communication motivated several proposals for hardware-supported recording [24, 45, 46], including some recent OS work on virtualization of hardware mechanisms for recording [25].

More recent work [1, 31, 47] relaxes the guarantees of replay by recording just a subset of the information required for faithful deterministic replay. The result is a smaller log at the cost of requiring a potentially impractical search of the execution space during replay. ESD [48] uses symbolic execution to reconstruct thread schedules given only a core dump, without requiring any execution logs to begin with. Unfortunately, ESD suffers from the incompleteness problems faced by symbolic execution, and thus cannot guarantee that a suitable execution will be found during replay.

Two recent and notable record/replay systems are SMP-ReVirt [16] and Scribe [21]. Both systems use page ownership similarly to dOS but record ownership transitions rather than imposing a single deterministic order, as in dOS. SMP-ReVirt is a hypervisor, and so it sup-

ports full-system replay only, while Scribe is a kernel extension, allowing it to support replay of process groups much like dOS. Additionally, Scribe and dOS use similar strategies to track ownership changes of kernel objects.

In contrast to all record/replay systems, the determinism guaranteed by DPGs enables precise replay without needing to record any internal nondeterminism.

Replicated Execution Most prior work in multithreaded replicas has taken the approach of recording and replicating internal nondeterminism. Examples include systems that assume a uniprocessor [28, 40]; that assume race-freedom [4, 5]; and that conservatively replicate all potential shared-memory nondeterminism [39]. Recently, Replicant [32] proposed a limited form of deterministic execution specifically for the purpose of deterministic replication, but this approach requires programmer annotations. Most recently, Respec [23] executes replicas independently while periodically verifying consistency; when consistency is violated, replicas are rolled back to a consistent state and execution proceeds more conservatively. Respec does not support replication across more than one machine, limiting its usefulness. In contrast to prior systems, the determinism offered by DPGs naturally enables replication.

There are some parallels between how dOS provides deterministic execution within a process group and how toolkits like Isis [9] and Horus [42] provide virtually synchronous execution to a distributed process group. Isis provides totally ordered multicast primitives that guarantee all processes see messages in the same order, a powerful building block for consistent updates of distributed replicas; dOS implements DMP-O to enforce a deterministic order on both implicit shared-memory and explicit OS-channel communications between threads and processes. Unlike dOS, Isis does not guarantee the deterministic execution of a process or the deterministic timing of message delivery relative to processes' instruction sequence. Unlike Isis, dOS does not provide fault tolerance, distributed group membership services, or state transfer to new group members.

8. Conclusions

We introduced the DPG abstraction, which allows programmers to define a deterministic box inside which all communication happens deterministically. We described the shim layer, an interface through which external nondeterminism can be observed and controlled by user-space programs. We developed dOS, an implementation of DPGs in Linux. We demonstrated the shim layer with three applications: record/replay, multithreaded replication, and deterministic filesystem services.

Our evaluation showed that DPGs have reasonable cost in reactive applications such as Apache and Chromium, and also in several parallel workloads. This conceivably enables deterministic execution in deployment, which would fully leverage the benefits of determinism in testing, reliability and debugging.

9. Acknowledgments

We thank Karin Strauss, Dan Grossman, John Zahorjan, and the members of the UW Sampa and systems research groups for their feedback and help. We also thank our anonymous reviewers and our shepherd, Ed Nightingale, for their guidance. This work was supported in part by the National Science Foundation under grants CNS-0627367, CNS-0430477, and CAREER award 0846004, a Torode Family Endowed Career Development Professorship, a Microsoft Faculty Fellowship, and gifts from Nortel Networks and Intel Corporation.

References

[1] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, 2009.

[2] AMD. Advanced Synchronization Facility: Proposed Architectural Specification. `http://developer.amd.com/cpu/ASF/Pages/default.aspx`, March 2009.

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

[4] C. Basile, Z. Kalbarczyk, and R. Iyer. A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. In *International Symposium on AUtomated Analysis-driven Debugging*, 2005.

[5] C. Basile, Z. Kalbarczyk, and R. Iyer. Active Replication of Multithreaded Applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5), 2006.

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

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

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

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

[10] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, CMU.

[11] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[12] D. Cahill. NullLogic HTTPd. `http://www.nulllogic.ca/httpd/`.

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

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

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

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

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

[18] J. Gilchrist. pbzip2: parallel bzip2. `http://compression.ca/pbzip2`.

[19] Z. Gu, X. W, J. Tang, X. Liu, Z. Xu, M. Wu, F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI*, 2008.

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

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

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

[23] D. Lee, B. Wester, J. Flinn, S. Narayanasamy, and P. Chen. Respec: Efficient Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, 2010.

[24] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.

[25] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, 2009.

[26] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(4), 1998.

[27] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[28] P. Narasimhan, L. Moser, and P. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Symposium on Reliable Distributed Systems*, 1999.

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

[30] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design an Implementation of Zap: A System for Migrating Computing Environments. In *OSDI*, 2002.

[31] S. Parka, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. Do You Have to Reproduce the Bug at the First Replay Attempt? – PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, 2009.

[32] J. Pool, I. Wong, and D. Lie. Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical. In *HotOS*, 2007.

[33] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA*, 2005.

[34] C. Reis and S. Gribble. Isolating Web Programs in Modern Browser Architectures. In *EuroSys*, 2009.

[35] M. Rinard and M. Lam. The Design, Implementation, and Evaluation of Jade. *ACM TOPLAS*, 20(3), 1988.

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

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

[38] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[39] J. Slember and P. Narasimhan. Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication With Nondeterminism. In *HotDep*, 2006.

[40] J. Slye and E. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *FTCS*, 1996.

[41] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, 2002.

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

[43] V. Weaver and S. McKee. Can Hardware Performance Counters be Trusted? In *IISWC*, 2008.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[45] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, 2003.

[46] M. Xu, M. Hill, and R. Bodik. A Regulated Transitive Reduction for Longer Memory Race Recording. In *ASPLOS*, 2006.

[47] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues From Run-time Logs. In *ASPLOS*, 2010.

[48] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *EuroSys*, 2010.