Deterministic Process Groups in 

Tom Bergan
Nicholas Hunt, Luis Ceze, Steven D. Gribble

University of Washington
A Nondeterministic Program

global x=0

Thread 1
\[
\begin{align*}
t & := x \\
x & := t + 1
\end{align*}
\]

Thread 2
\[
\begin{align*}
t & := x \\
x & := t + 1
\end{align*}
\]

What is x?

\[
\begin{align*}
x & == 2 \\
x & == 2 \\
x & == 1
\end{align*}
\]
Nondeterministic IPC

Process 0

send(msg=A)
send(msg=B)

Process 1
recv(..)
recv(msg=A)
recv(msg=B)
recv(msg=B)

Process 2
recv(..)
recv(msg=A)
recv(msg=B)

Who gets msg A?
recv(msg=A)
recv(msg=B)
recv(msg=A)
recv(msg=B)

3
Nondeterminism In Real Systems

**shared-memory**
why nondeterministic:
multiprocessor hardware is unpredictable

**disks**
why nondeterministic:
drive latency is unpredictable

**IPC (e.g. pipes)**
why nondeterministic:
multiprocessor hardware is unpredictable

**network**
why nondeterministic:
packets arrive from external sources

**posix signals**
why nondeterministic:
unpredictable scheduling, also can be triggered by users
The Problem

• Nondeterminism makes programs . . .
  ➡ hard to test
    ‣ same input, different outputs
  ➡ hard to debug
    ‣ leads to heisenbugs
  ➡ hard to replicate for fault-tolerance
    ‣ replicas get out of sync

• Multiprocessors make this problem much worse!
Our Solution

- OS support for deterministic execution
  - of arbitrary programs
  - attack all sources of nondeterminism (not just shared-memory)
  - even on multiprocessors

New OS abstraction: *Deterministic Process Group* (DPG)

deterministic box
Key Questions

1. What can be made deterministic?

2. What can we do about the remaining sources of nondeterminism?
Key Questions

1. What can be made deterministic?
   - distinguish internal vs. external nondeterminism

2. What can we do about the remaining sources of nondeterminism?
Internal nondeterminism

• arises from scheduling artifacts (hw timing, etc)

NOT Fundamental can be eliminated!

External nondeterminism

• arises from interactions with the external world (networks, users, etc)

Fundamental can not be eliminated
Internal Determinism

External Nondeterminism

deterministic box

network

users
real time
Internal Determinism

- shared memory
- pipes
- private files

External Nondeterminism

- users
- real time

Process 1
Process 2
Process 3

network

a programmer-defined process group

deterministic box
Internal Determinism

- shared memory
- pipes
- private files

External Nondeterminism

- network
- pipe
- shared file

Process 4

Deterministic box

Process 1

Process 2

Process 3

users

real time
**Internal Determinism**

- shared memory
- pipes
- private files

**External Nondeterminism**

- shim program
- network
- Precisely controls all external inputs
  - value of input data
  - time input data arrives

**Deterministic box**

- Process 1
- Process 2
- Process 3

**Users**

**Real time**
An entire virtual machine could go inside the deterministic box!
- too inflexible
- too costly
Deterministic Process Groups

OS ensures:

- **internal** nondeterminism is eliminated (for shared-memory, pipes, signals, local files, ...)
- **external** nondeterminism funneled through shim program

Shim Program:

- user-space program that precisely controls all external nondeterministic inputs
Contributions

Conceptual:
- identify *internal* vs. *external* nondeterminism
- key: *internal* nondeterminism can be eliminated!

Abstraction:
- Deterministic Process Groups (DPGs)
- control *external* nondeterminism via a *shim program*

Implementation:
- dOS, a modified version of Linux
- supports arbitrary, unmodified binaries

Applications:
- deterministic parallel execution
- record/replay
- replicated execution
Outline

• Example Uses
  ➔ a parallel computation
  ➔ a webserver

• Deterministic Process Groups
  ➔ system interface
  ➔ conceptual model

• dOS: our Linux-Based Implementation

• Evaluation
This program executes deterministically!

- even on a multiprocessor
- supports parallel programs written in any language
  - no heisenbugs!
  - test input files, not interleavings
A Webserver

Deterministic Record/Replay

- implement in shim program
- requires no webserver modification

Advantages

- significantly less to log (internal nondeterminism is eliminated)
- log sizes 1,000x smaller!
**A Webserver**

**Fault-tolerant Replication**
- implement replication protocol in **shim programs** (paxos, virtual synchrony, etc)

**Advantage**
- easy to replicate multithreaded servers *(internal* nondeterminism is eliminated)
Using DPGs to construct applications

- Deterministic part (in a DPG)
- Nondeterministic part (in a shim)

Webserver

- Behaves deterministically w.r.t. requests rather than packets

Shim program defines the nondeterministic interface
• Example Uses
  ➔ a parallel computation
  ➔ a webserver

• Deterministic Process Groups
  ➔ system interface
  ➔ conceptual model

• dOS: our Linux-Based Implementation

• Evaluation
Deterministic Process Groups

System Interface

• New system call creates a new DPG: `sys_makedet()`
  ‣ DPG expands to include all child processes

• Just like ordinary Linux processes
  ‣ same system calls, signals, and hw instruction set
  ‣ can be multithreaded
Two questions:

• What are the semantics of *internal* determinism?

• How do shim programs work?
Deterministic Process Groups

Internal Determinism

- OS guarantees **internal** communication is scheduled **deterministically**
- Conceptually: executes as if serialized onto a **logical timeline**
  - implementation is parallel
Internal Determinism

Each DPG has a **logical timeline**

- instructions execute as if serialized onto the logical timeline
- **internal** events are deterministic

### Logical Timeline

<table>
<thead>
<tr>
<th>Thread</th>
<th>Logical Timeline</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wr x t=1</td>
<td>rd x t=2 always reads same value of x</td>
</tr>
<tr>
<td></td>
<td>rd y t=4</td>
<td>read(pipe) t=3 blocking call</td>
</tr>
<tr>
<td></td>
<td>rd z t=5</td>
<td>read(pipe) t=6 always returns same data</td>
</tr>
<tr>
<td></td>
<td>wr y t=7</td>
<td></td>
</tr>
</tbody>
</table>

...
**Physical time is not deterministic**

- Deterministic results, but not deterministic performance
External Nondeterminism

Two sources of nondeterminism:

- **data** returned by `read()`
- **blocking time** of `read()`
External Nondeterminism

Two sources of nondeterminism:

- **data** returned by `read()`
- **blocking time** of `read()`
Two sources of nondeterminism:

- *data* returned by `read()`
- *blocking time* of `read()`
External Nondeterminism

Two sources of nondeterminism:

- data returned by `read()` → the what
- blocking time of `read()` → the when
Shim Example: Read Syscall

Shim can either . . .

① Monitor call (e.g., for record)
② Control call (e.g., for replay)
Shim Example: Read Syscall

Shim can either . . .

1. Monitor call (e.g., for record)
2. Control call (e.g., for replay)
Key idea:
- protocol delivers (time, msg) pairs to replicas
- ensure replicas see same input at same logical time

We have implemented this idea (see paper)

Shim Example: Replication

DPG Replica 1

DPG Replica 2

DPG Replica 3
Outline

• Example Uses
  ➔ a parallel computation
  ➔ a webserver

• Deterministic Process Groups
  ➔ system interface
  ➔ conceptual model

• dOS: our Linux-Based Implementation

• Evaluation
dOS Overview

Modified version of Linux 2.6.24/x86_64
- ~8,000 lines of code added or modified
- ~50 files changed or modified
- transparently supports unmodified binaries

Support for DPGs:
- implement a **deterministic scheduler**
- implement an API for writing **shim programs**
- subsystems modified:
  - thread scheduling
  - virtual memory
  - system call entry/exit

Talk focus

Paper describes challenges in depth
dOS: Deterministic Scheduler

Which deterministic execution algorithm?
• DMP-O, from prior work [Asplos09, Asplos10]
  - other algorithms have better scalability, but
  - … Dmp-O is easiest to implement

How does DMP-O work?
How does dOS implement DMP-O?
Deterministic Execution with DMP-O

Key idea:
- serialize all communication deterministically
Deterministic Execution with DMP-O

parallelize until there is communication
Deterministic Execution with DMP-O

Ownership table
- assigns ownership of data to threads
- communication: thread wants data it doesn’t own

parallelize until there is communication
serialize communication
dOS: Changes for DMP-O
dOS: Changes for DMP-O

Ownership Table

must instrument the system interface

- **loads/stores**
  - for shared-memory

- **system calls**
  - for in-kernel channels
  - *explicit*: pipes, files, signals, ...
  - *implicit*: address space, file descriptor table, ...
dOS: Changes for DMP-O

Ownership Table

for shared-memory

- must instrument loads/stores
  - use page-protection hw
- each thread has a shadow page table
  - permission bits denote ownership
  - page faults denote communication
  - page granularity ownership
dOS: Changes for DMP-O

Ownership Table

for in-kernel channels (pipes, etc.)

- must instrument system \textit{calls}
- on syscall entry:
  - decide what channels are used
    \begin{itemize}
    \item \texttt{read}(): pipe or file being read
    \item \texttt{mmap}(): the thread’s address space
    \end{itemize}
  - acquire ownership
    ownership table is just a hash-table
  - any external channels?
    \textbf{if yes:} forward to shim program

Many challenges and complexities (see paper)
• Example Uses
  ➔ a parallel computation
  ➔ a webserver

• Deterministic Process Groups
  ➔ system interface
  ➔ conceptual model

• dOS: our Linux-Based Implementation

• Evaluation
Evaluation Overview

Setup

- 8-core 2.8GHz Intel Xeon, 10GB RAM
- Each application ran in its own DPG

Verifying determinism

- used the racey deterministic stress test [ISCA02, MarkHill]

Key questions

- How much internal nondeterminism is eliminated? (log sizes for record/replay)
- How much overhead does dOS impose?
- How much does dOS affect parallel scalability?
## Eval: Record Log Sizes

### dOS

- implemented an “execution recorder” shim

### SMP-ReVirt (a hypervisor) [VEE 08]

- also uses page-level ownership-tracking
- … but has to record internal nondeterminism

### Log size comparison

<table>
<thead>
<tr>
<th></th>
<th>dOS</th>
<th>SMP-ReVirt (log size per day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmm</td>
<td>1 MB</td>
<td>83 GB</td>
</tr>
<tr>
<td>lu</td>
<td>11 MB</td>
<td>11 GB</td>
</tr>
<tr>
<td>ocean</td>
<td>1 MB</td>
<td>28 GB</td>
</tr>
<tr>
<td>radix</td>
<td>1 MB</td>
<td>88 GB</td>
</tr>
<tr>
<td>water</td>
<td>5 MB</td>
<td>58 GB</td>
</tr>
</tbody>
</table>

8,800x bigger!
Eval: dOS Overheads

Possible sources of overhead
- deterministic scheduling
- shim program interposition

Ran each benchmark in three ways:
- without a DPG (ordinary, nondeterministic)
  scheduling overheads
- with a DPG only
  shim overheads
- with a DPG and an “execution recorder” shim program
Eval: dOS Overheads

Apache

- 16 worker threads
- serving 100KB static pages
  - DPGs saturate 1 gigabit network
- serving 10 KB static pages
  - Nondet (no DPG) saturates 1 gigabit network
  - DPG (no shim): 26% throughput drop
  - DPG (with record shim): 78% throughput drop (over Nondet)

Chromium

- process per tab
- scripted user session (5 tabs, 12 urls)
  - DPG (no shim): 1.7x slowdown
  - DPG (with record shim): 1.8x slowdown (over Nondet)
Eval: dOS Overheads

Parallel application slowdowns

- DPG only
- relative to nondeterministic execution

*5x* = 5 times slower with DPGs

Preserves scalability

Fine-grained loses scalability

2 threads
4 threads
8 threads
Wrap Up

Deterministic Process Groups
- new OS abstraction
- eliminate or control sources of nondeterminism

dOS
- Linux-Based implementation of DPGs
- use cases demonstrated: deterministic execution, record/replay, and replicated execution

Also in the paper . . .
- many more implementation details
- a more thorough evaluation
- thoughts on a “from scratch” implementation
Thank you!

Questions?

http://sampa.cs.washington.edu

C:\DOS
C:\DOS\RUN
C:\DOS\RUN\DETERM~1.EXE
(backup slides)
Performance?

Already good enough for some workloads!

• infrequent system calls
• infrequent fine-grained sharing
  - examples: Apache 100KB static pages, blackscholes, pbzip, etc.

Improvements possible:

• better scheduling algorithm ($DMP-TM$, $DMP-B$) [Asplos09, Asplos10]
• binary instrumentation (to support arbitrary data granularity)
• implement shims as kernel modules (lower context switch overhead)

Research question:

• how much does determinism **fundamentally** impact performance?
### Overheads Breakdown

#### Deterministic scheduler

<table>
<thead>
<tr>
<th></th>
<th>% serialization</th>
<th>% single-stepping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 100KB</td>
<td>26%</td>
<td>0%</td>
</tr>
<tr>
<td>Apache 10KB</td>
<td>60%</td>
<td>0%</td>
</tr>
<tr>
<td>Chromium</td>
<td>25%</td>
<td>13%</td>
</tr>
<tr>
<td>blackscholes</td>
<td>3%</td>
<td>27%</td>
</tr>
<tr>
<td>fmm</td>
<td>54%</td>
<td>18%</td>
</tr>
<tr>
<td>dedup</td>
<td>90%</td>
<td>12%</td>
</tr>
</tbody>
</table>

#### Shim context-switching

- microbenchmark: **5x** overhead on system call traps
Why are DPGs awesome?

DPGs give you determinism, which helps:
- testing
- debugging
- fault-tolerant replication
- security
  - can eliminate internal timing channels [Aviram et al, CCSW10]

DPGs give you determinism flexibly:
- user-defined process group
  - keeps separate apps isolated in their own determinism domain
- shim programs can customize:
  - the interface to the nondeterministic external world
  - the set of deterministic services
    (more details in paper)
Internal Determinism

Design Choices

- A single thread
  - current systems
  - massively nondeterministic on multiprocessors

- A single multithreaded process

- A group of multithreaded processes
  - our choice
  - most flexible

- A virtual machine
  - too costly, too inflexible

- A local area network cluster?

deterministic box
Right Place For Determinism?

Language?
✓ more robust determinism, enables static analysis (lower cost)
➡ must rewrite program with specialized constructs

Operating System?
✓ support arbitrary, unmodified binaries
➡ high overheads for some workloads

Compiler?
✓ lower overheads than OS for some workloads (finer-grained tracking)
➡ can’t resolve communication via the kernel

Hardware?
✓ low-overhead shared-memory determinism
➡ must build custom hardware
SMP-ReVirt?

Advantages of SMP-ReVirt
✓ full-system record/replay
   - includes OS code
   - via a hypervisor implementation

Advantages of dOS
✓ process level
   - cheaper than full-system?
   - don’t need to resolve kernel-level shared-memory
     (up to 50% of sharing for some benchmarks [VEE 08])
✓ no internal nondeterminism
   - smaller logs (by 1,000x)
Prior Work: Record/Replay

Record internal nondeterminism

- in software [SMP-ReVirt, Scribe, DejaVu, ...]
- in hardware [FDR, DeLorean, ...]
  - big logs, high runtime overheads for software

Search execution space during replay

- record a few bits of internal nondeterminism [PRES, ODR]
- record nothing [ESD]
  - cannot guarantee replay (might fail to find an execution)

Advantages of dOS

✓ small logs (no internal nondeterminism)
✓ replay is guaranteed
Prior Work: Deterministic Execution

References

- DMP [ASPLOS 09] custom hardware
- Kendo [ASPLOS 09] custom runtime (race-free programs only)
- CoreDet [ASPLOS 10] custom compiler/runtime
- Grace [OOPSLA 10] custom runtime (fork-join programs only)

Advantages of dOS

✓ supports:
  - multiple processes
  - communication other than shared-memory (pipes, etc.)
  - arbitrary binaries

✓ does not require:
  - custom hardware
  - recompilation

✓ shims for external nondeterminism
(extra slides)
Key: *make communication explicit*

- implicit communication makes Linux complicated
  *example:* touching a shared data structure on an obscure code path

Design ideas

- compartmentalize shared resources
  *idea:* build a microkernel?
  message passing makes communication explicit

- minimal system interface
  *idea:* push a lot of code out of the OS, into user-space, and “inside-the-deterministic-box”
Many code paths in each system call

- How do we know which channels a system call uses?

Wide system interface

- How do we reconcile aliased interfaces?

The big questions

- How do we know our instrumentation is complete?
- How do we know when a system call is actually deterministic?

Answer: careful inspection (whitelist)

- everything else assumed nondeterministic! (conservative)
- some special cases
  - e.g., `mmap()`
    - usually returns a deterministic address
    - can fail with a nondeterministic error code (e.g. ENOMEM)
dOS: Complexities

Consequence . . .

→ Cannot guarantee as much determinism as we’d like

*Example*: local file reads

- not deterministic in dOS (due to disk latencies, etc.)
- what about our deterministic parallel execution use case?

![Diagram](attachment:image)

*Solution*: use a **shim program** to implement deterministic local file reads (described in paper)
Shim Example: Record

Logical Timeline

DPG Thread

Shim Program

OS

Read() at Time t=3

Block thread at Time t=4

(blocked)
Shim Example: Record

Logical Timeline

| t=3 | | | |
| t=4 | | | |
| t=5 | (blocked) | | |
| t=10 | | | |

DPG Thread

Shim Program

OS

unblock thread

“hello”
Shim Example: Record

Logical Timeline

<table>
<thead>
<tr>
<th>t=3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t=4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t=5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t=10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t=11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DPG Thread

Shim Program

OS

record(t=10,"hello")
return("hello")

"hello"
Shim Example: Replication

Key idea:
- arbiter delivers \( \text{time}, \text{msg} \) pairs to replicas
- ensure replicas see same input at same logical time

Arbiter

DPG Replica 1
- Logical Time = 3

DPG Replica 2
- Logical Time = 2

DPG Replica 3
- Logical Time = 5
Shim Example: Replication

Arbiter

5 msg block_time() 5 msg

3 shim server

Logical Time = 3

stalled

DPG Replica 1

2 shim server

Logical Time = 2

stalled

DPG Replica 2

5 shim server

Logical Time = 5

stalled

DPG Replica 3
DPG Use Cases

Deterministic Parallel Execution
- no heisenbugs!
- test input files, not interleavings

Record / Replay Debugging
- shim does record/replay
- significantly less nondet to record (~1,000x)

Fault-tolerant Replication
- naturally enables multithreaded replicas