Deterministic Process Groups in dDS

Tom Bergan
Nicholas Hunt, Luis Ceze, Steven D. Gribble
University of Washington
A Nondeterministic Program

```
global x=0

Thread 1

\[ t := x \]
\[ x := t + 1 \]

Thread 2

\[ t := x \]
\[ x := t + 1 \]

What is x?

x == 2  x == 2  x == 1
```
Nondeterministic IPC

Process 0

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

Process 1

recv(..)

Process 2

recv(..)

Who gets msg A?

recv(msg=A)
recv(msg=B)
recv(msg=A)
recv(msg=B)
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)
Key Questions

1. What can be made deterministic?

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

users

real time

network
Internal Determinism

External Nondeterminism

- shared memory
- pipes
- private files

network

Process 1

Process 2

Process 3

users

real time

a programmer-defined process group

deterministic box
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
Internal Determinism

External Nondeterminism

An entire virtual machine could go inside the deterministic box!
- too inflexible
- too costly

Deterministic box
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
A Parallel Computation

parallel program

local input files

deterministic box

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!
Fault-tolerant Replication

- implement replication protocol in shim programs (paxos, virtual synchrony, etc)

Advantage

- easy to replicate multithreaded servers (internal nondeterminism is eliminated)
A Webserver

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
Outline

- 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
Each DPG has a **logical timeline**

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

Thread 1

Logical Timeline

Thread 2

wr x

| t=1
| t=2
| t=3
| t=4
| t=5
| t=6
| t=7

rd x

read(pipe)

rd y

rd z

blocking call

read(pipe)

wr y

arbitrary delays in physical time are possible

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()`
External Nondeterminism

Two sources of nondeterminism:

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

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

Logical Timeline

| t=2  |  |  |
|------|  |  |
| t=3  |  |  |
| t=4  |  |  |
|      |  |  |
| t=10 |  |  |
|      |  |  |
| t=11 |  |  |

DPG Thread

- read()

Shim Program

- return("hello")

OS

- "hello"

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)
Shim Example: Replication

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

We have implemented this idea (see paper)

Replication protocol

DPG Replica 1

DPG Replica 2

DPG Replica 3

shim

multithreaded server

shim

multithreaded server

shim

multithreaded server
• 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

Paper describes challenges in depth
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

parallelize until there is communication

serialize communication

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

Thread₁ Thread₂ Thread₃ Logical Timeline

Ownership table

parallelize until there is communication

serialize communication

Ownership table

thread 1

thread 2

thread 3

Logical Timeline

$t=1$

$t=2$

$t=3$

$t=4$

...
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, ...
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
 Ownership Table

for in-kernel channels (pipes, etc.)

• must instrument system calls
• on syscall entry:
  - decide what channels are used
    read(): pipe or file being read
    mmap(): the thread’s address space
  - acquire ownership
    ownership table is just a hash-table
  - any external channels?
    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</th>
<th>(log size per day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmm</td>
<td>1 MB</td>
<td>83 GB</td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>11 MB</td>
<td>11 GB</td>
<td></td>
</tr>
<tr>
<td>ocean</td>
<td>1 MB</td>
<td>28 GB</td>
<td></td>
</tr>
<tr>
<td>radix</td>
<td>1 MB</td>
<td>88 GB</td>
<td>8,800x bigger!</td>
</tr>
<tr>
<td>water</td>
<td>5 MB</td>
<td>58 GB</td>
<td></td>
</tr>
</tbody>
</table>
Eval: dOS Overheads

Possible sources of overhead
  ‣ deterministic scheduling
  ‣ shim program interposition

Ran each benchmark in three ways:
  ‣ without a DPG (ordinary, nondeterministic)
  ‣ with a DPG only
  ‣ 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)
Parallel application slowdowns

- DPG only
- relative to nondeterministic execution

5x = 5 times slower with DPGs

<table>
<thead>
<tr>
<th>Application</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pbzip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>make</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

fine-grained stacking loses scalability
preserves scalability
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
Discussion
How can we “constructively” make use of DPG?
Is OS the right place to provide determinism? How else can we provide deterministic program execution? Language, Compiler, Hardware? What are the pros and cons of each approach?

ewm87: “each source of non-determinism should handle itself”
wysem: “do we really want/need deterministic execution for everything?”
danyangz: “the cost of making the scheduling deterministic is quite large...better to use some invariance reasoning”
Why do we need deterministic processes?

bornholt: “The demand for determinism seems like a side effect of terrible abstractions for concurrency.”
Is DPG a perfect solution for debugging/testing?

osandov: “make data race bugs harder to find”
naveenks: “since many multi-threaded bugs are due to race-conditions and concurrency, how does debugging inside a DPG help catch those bugs?”
lijl: “when customers encounter a bug, the developers should be able to reproduce the bug even on a completely different machine”
How robust is the determinism enforced inside a DPG? What if the programmer add a single debug print statement?

billzorn: “It's like not getting the best of either world: the determinism is fragile and complicated.”
What are the preconditions to use DPG for your application? What are the properties that are not compulsory but good to have?

unmodified
performance-insensitive
no randomness

share as few things as possible
have a small number of external communications
Are there any constraints/assumptions that can be relaxed in DPG to give us better performance?

antoinek “the false-sharing problem technically is the exact same thing as for cache lines, but with two major differences:
1) the 4K pages on x86 are generally 64x larger than cache lines
2) even false sharing at a relatively low rate can quickly become really expensive, because execution has to switch to serialized”