Efficient System-Enforced Deterministic Parallelism

Ammittai Aviram, Shu-Chun Weng,
Sen Hu, Bryan Ford (Yale)
OSDI 2010

https://www.usenix.org/conference/osdi10/efficient-system-enforced-deterministic-parallelism
Dan Tsafrir, 25/5/2016, slightly modified version of Bryan’s slides
Pervasive Parallelism

Uniprocessor

Multiprocessor

Multicore

“Many-core”

Industry shifting from “faster” to “wider” CPUs
Today's Grand Software Challenge

Parallelism makes programming harder.

Why? Because parallelism introduces

• **Nondeterminism** (in general)
  - Execution behavior subtly depends on timing

• **Data Races** (in particular)
  - Unsynchronized concurrent state changes

=> **Heisenbugs**: sporadic, difficult to reproduce
  - A bug that seems to disappear or alter its behavior when one attempts to study it
Races are Everywhere

- **Memory Access**
  - Write/Write: $x = 1 \rightarrow x = 2$
  - Read/Write: $x = 2 \rightarrow y = x$

- **File Access**
  - open() \(\xrightarrow{\text{rename()}}\)

- **Synchronization**
  - lock;
  - $x++$;
  - unlock;
  - lock;
  - $x *= 2$;
  - unlock;

- **System APIs**
  - malloc() \(\rightarrow\) ptr
  - malloc() \(\rightarrow\) ptr
  - open() \(\rightarrow\) fd
  - open() \(\rightarrow\) fd
Living With Races

• “Don't write buggy programs”
• Logging/replay tools (BugNet, IGOR, …)
  - Reproduce bugs that manifest while logging
• Race detectors (RacerX, Chess, …)
  - Analyze/instrument program to help find races
• Deterministic schedulers (DMP, Grace, CoreDet)
  - Synthesize a repeatable execution schedule

All of the above help manage races but don't eliminate them
“Heisenbug papers” at SOSP/OSDI (detecting, replaying, avoiding, recovering from...)
Must We Live With Races?

• **Ideal**: a parallel programming model in which
  - Races don't arise in the first place

• Already possible with **restrictive languages**
  - Pure functional languages (Haskell)
  - Deterministic value/message passing (SHIM)
  - PL that adopts deterministic message passing as its sole communication mechanism [DATE'08]
  - Separation-enforcing type systems (DPJ)
    - A Type and Effect System for Deterministic Parallel Java [OOPSLA'09]

• But then we'd have to use these languages.. :-)  
  - What about race-freedom for **any language**?
Introducing **Determinator**

• New OS offering race-free parallel programming model
  - Pervasively throughout the system APIs

• OS-level => compatible with arbitrary existing languages
  - C, C++, Java, assembly, ...

• Avoids races at multiple abstraction levels
  - Shared memory, file system, sync, ... (discussed earlier)

• Takes clean-slate approach for simplicity
  - Ideas could be later retrofitted into existing OSes

• Focus: compute-bound applications
  - Early prototype, many limitations
Talk Outline

✓ Introduction: parallelism and data races
  • Determinator's programming model
  • Prototype kernel/runtime implementation
  • Performance evaluation
A deterministic programming model

• Explore the “purist” approach of pervasive determinism
  - Tradeoffs are possible and may improve efficiency
  - We always favor determinism

• Identify 4 aspects that must be dealt with
  1. Explicit nondeterministic input
  2. Race-free model of shared state
  3. Race-free synchronization API
  4. Race-free system namespace
Explicit nondeterministic input (item #1)

• Inputs
  - Incoming messages, timers, random numbers
  - Useful, we want to keep them
  - All replay mechanism deal with this (record, e.g. gettimeofday)

• Can be done efficiently on a uniprocessor VM
  - Becomes much more expensive when need to handle nondeterminism due to parallelism (shared memory races...)

• So...
  - Determinator transforms useful sources of nondeterminism into explicit I/O, and...
  - Eliminates nondeterminism resulting from parallelism
Race-free model for shared-state (item #2)

• No concurrent access to memory
  - Instead, each process gets a private workplace
  - Virtual replica of all of parent's state
  - No races can arise
Determinator's Programming Model

• With pthreads, if one thread forks another
  - From that point, they're interacting through shared vars

• In determinator, “check-out/check-in” model for shared-state
  1. on fork, “check-out” a copy of all shared state
  2. thread reads/writes private working copy only
  3. on join, “check-in” and merge changes
Seen this before?

• Precedents for “check-in/check-out” model
  - DOALL in early parallel Fortran computers
    • Burroughs FMP 1980, Myrias 1988
    • Language-specific, limited to specific construct: DO loops
  - Version control systems (cvs, svn, git, hg, ...)
    • Manual check-in/check-out procedures
    • For files only, not shared memory state

• Determinator applies this model pervasively & automatically
  - To all shared state
Example 1: gaming/simulation, conventional threads 😊

```c
struct actorstate actor[NACTORS];

void update_actor(int i) {
    ...examine state of other actors...
    ...update state of actor[i] in-place...
}

int main() {
    ...initialize state of all actors...
    for (int time = 0; ; time++) {
        thread t[NACTORS];
        for (i = 0; i < NACTORS; i++)
            t[i] = thread_fork(update_actor, i);
        for (i = 0; i < NACTORS; i++)
            thread_join(t[i]);
    }
}
```

ODE – determinator
Example 1: gaming/simulation, conventional threads 😞

```c
struct actorstate actor[NACTORS];

void update_actor(int i) {
    ...
    examine state of other actors...
    ...
    update state of actor[i] in-place...
}

int main() {
    ...
    initialize state of all actors...
    for (int time = 0; ; time++) {
        thread t[NACTORS];
        for (i = 0; i < NACTORS; i++)
            t[i] = thread_fork(update_actor, i);
        for (i = 0; i < NACTORS; i++)
            thread_join(t[i]);
    }
}
```

main thread

actors [0] [1]

read

(partial) update

read

update

oops! corruption/crash due to race
Example 1: gaming/simulation, determinator threads

struct actorstate actor[NACTORS];

void update_actor(int i) {
    ...examine state of other actors...
    ...update state of actor[i] in-place...
}

int main() {
    ...initialize state of all actors...
    for (int time = 0; ; time++) {
        thread t[NACTORS];
        for (i = 0; i < NACTORS; i++)
            t[i] = thread_fork(update_actor, i);
        for (i = 0; i < NACTORS; i++)
            thread_join(t[i]);
    }
}
Example 2: parallel make, conventional Unix 😊

# Makefile for file 'result'

result: foo.out bar.out
    combine $^ >$@

%.out: %.in
    stage1 <$^ >tmpfile
    stage2 <tmpfile >$@
    rm tmpfile

$^ = all prerequisites
$@ = filename of the target of the rule

$ make
    read Makefile, compute dependencies
    fork worker shell
    stage1 <foo.in >tmpfile
    stage2 <tmpfile >foo.out
    rm tmpfile
    stage1 <bar.in >tmpfile
    stage2 <tmpfile >bar.out
    rm tmpfile
    combine foo.out bar.out >result
Example 2: parallel make, conventional Unix 😞

# Makefile for file 'result'

result: foo.out bar.out
combine $^ >$@

%.out: %.in
stage 1 < $^ > tempfile
stage 2 < tempfile > $@
rm tempfile

$ make -j
(parallel make)

read Makefile, compute dependencies
fork worker processes

read foo.out, bar.out
write result

ode – determinator
Example 2: parallel make, determinator

# Makefile for file 'result'

result: foo.out bar.out
combine $^ >$@

%.out: %.in
stage 1 <$^ >tmpfile
stage 2 <tmpfile >$@
rm tmpfile

$ make -j
(parallel make)
read Makefile, compute dependencies
fork worker processes

$ make -j
read foo.out, bar.out
write result

ODE – determinator
What happens to data races?

• Read/Write races
  - Go away entirely
  - Write ops propagate only via synchronization
  - Read ops always see last write by *same* thread, else value at last synchronization point
What happens to data races

• Write/Write races
  - Go away if threads “undo” their changes
    • rm tmpfile in make -j example
  - Otherwise become deterministic conflicts
    • Always detected at join/merge point
    • Runtime exception, just like divide-by-zero (or git merge conflict)
    • As opposed to standard threads, where one or the other silently wins
Race-free synchronization API (item #3)

• When using a “lock” sync abstraction, nondeterminism is inherent
  - Even if you always lock some shared variable...
  - Order of ops can change

• Allow only deterministic synchronization abstractions
  - Okay examples:
    • thread_join, barrier
    • Queue abstractions such as pipe between 2 threads

• Program logic alone determines when synch occurs
  - Not okay:
    • Locks, shared memory, waitpid (to any pid, not to specific pid), ...
Race-free system namespace (item #4)

• Can't have, e.g., ...
  - fork() that returns some global number as pid
  - malloc() performed by 2 threads concurrently, each getting a different pointer that depends on the scheduling
  - open() performed by 2 threads, returning nondeterministic file descriptors

• Solution
  - Have the caller specify these values
    • E.g., invoke fork with a value of the pid as a parameter, determined by the app
  - But then values must remain in private namespace of process
    • Pid not disjoint
Talk Outline

✓ Introduction: parallelism and data races
✓ Determinator's programming model
  • Prototype kernel/runtime implementation
  • Performance evaluation
Determinator OS architecture

Grandchild Space -> Parent/Child Interaction -> Child Space

Child Space -> Parent/Child Interaction -> Root Space

Root Space -> Registers (1 thread)

Root Space -> Device I/O

Root Space -> Address Space

Root Space -> Snapshot

Determinator Microkernel

Hardware
Microkernel API

• Three system calls (and a few options to each – see paper) :
  - PUT: copy data into child, snapshot, start child
  - GET: copy data or modifications out of child
  - RET: return control to parent

• No kernel support for processes, threads, files, pipes, sockets, messages, shared memory, ...
User-level runtime

• Emulates familiar programming abstractions
  - C library
  - Unix-style process management
  - Unix-style file system API
  - Shared memory multithreading
  - Pthreads via deterministic scheduling

• It's a library => all facilities are optional
Threads – determinator style

Parent:
1. thread_fork(Child1): PUT
2. thread_fork(Child2): PUT
3. thread_join(Child1): GET
4. thread_join(Child2): GET

Child 1:
- read/write memory
- thread_exit(): RET

Child 2:
- read/write memory
- thread_exit(): RET

1a. copy into Child1
1b. save snapshot
2a. copy into Child2
3. copy diffs back into Parent
4. copy diffs back into parent

Multithreaded Process
Virtual memory optimizations

• Copy/snapshot quickly via copy-on-write (COW)
  - Mark all pages read-only
  - Duplicate mappings rather than pages
  - Copy pages only on write attempt

• Variable-granularity virtual diff & merge
  - If only parent or child has modified a page, reuse modified page: no byte-level work
  - If both parent and child modified a page, perform byte-granularity diff & merge
Emulating a shared file system

• Each process has a complete file system replica in its address space
  - a “distributed filesystem” w/ weak consistency
  - fork() makes virtual copy
  - wait() merges changes made by child processes
  - merges at file rather than byte granularity

• No persistence yet
  - just for intermediate results
Filesystem conflicts

• Hard conflicts
  - concurrent file creation, random writes, etc.
  - mark conflicting file => accesses yield errors

• Soft conflicts
  - concurrent appends to file or output device
  - merge appends together in deterministic order
Other features (paper)

• System enforcement of determinism
  - important for malware/intrusion analysis
  - might help with timing channels [CCSW 10]

• Distributed computing via process migration
  - forms simple distributed FS, DSM system

• Deterministic scheduling (optional)
  - backward compatibility with pthreads API
  - races still exist but become reproducible
Talk Outline

- Introduction: parallelism and data races
- Determinator's programming model
- Prototype kernel/runtime implementation
  - Performance evaluation
Evaluation goals

• Question: Can such a programming model be
  - efficient
  - scalable
  - ...enough for everyday use in real apps?

• Answer: it depends on the app (of course)
Single-Node Speedup over 1 CPU
Single-Node Performance: Determinator versus Linux

Coarse-grained

Fine-grained

Speedup over Linux (log scale)

Benchmark

ODE – determinator
Drilldown: Varying Granularity (Parallel Quicksort)

“break-even point”
Conclusions

• Determinator provides a race free, deterministic parallel programming model
  - Avoids races via “check-out, check-in” model
  - Supports arbitrary, existing languages
  - Supports thread- and process-level parallelism

• Efficiency through OS-level virtual memory optimizations
  - Minimal overhead for coarse-grained apps

• Further information: http://dedis.cs.yale.edu