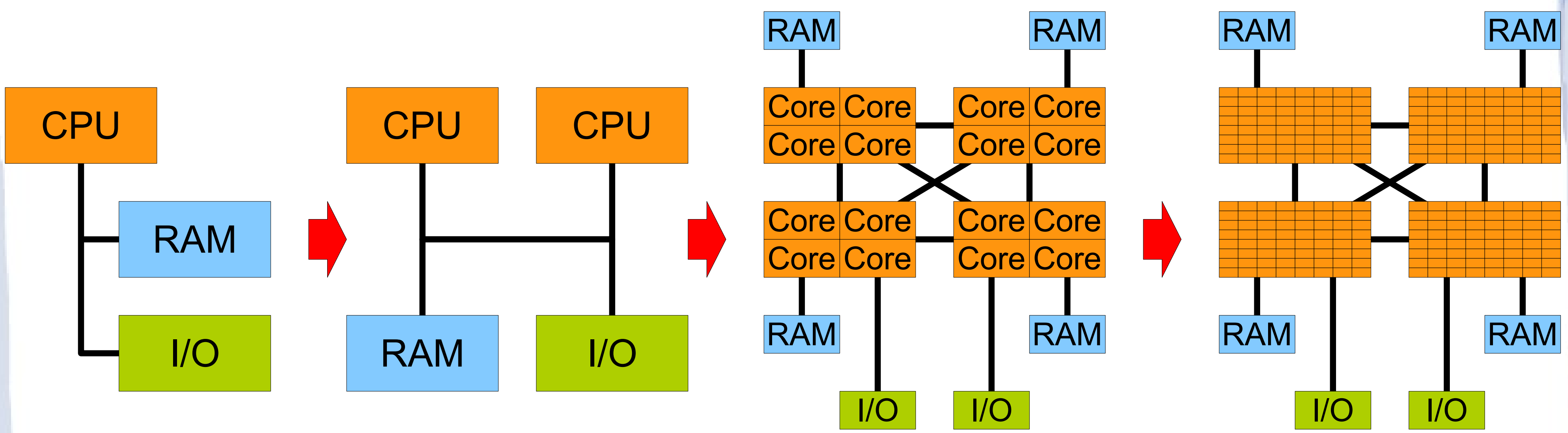
Efficient System-Enforced Deterministic Parallelism

Amittai Aviram, Shu-Chun Weng,
Sen Hu, **Bryan Ford**

*Decentralized/Distributed Systems Group,
Yale University*
http://dedis.cs.yale.edu/

9th OSDI, Vancouver – October 5, 2010

Pervasive Parallelism



Uniprocessor Multiprocessor Multicore "Many-core"

Industry shifting from "faster" to "wider" CPUs

Today's Grand Software Challenge

Parallelism makes programming harder.

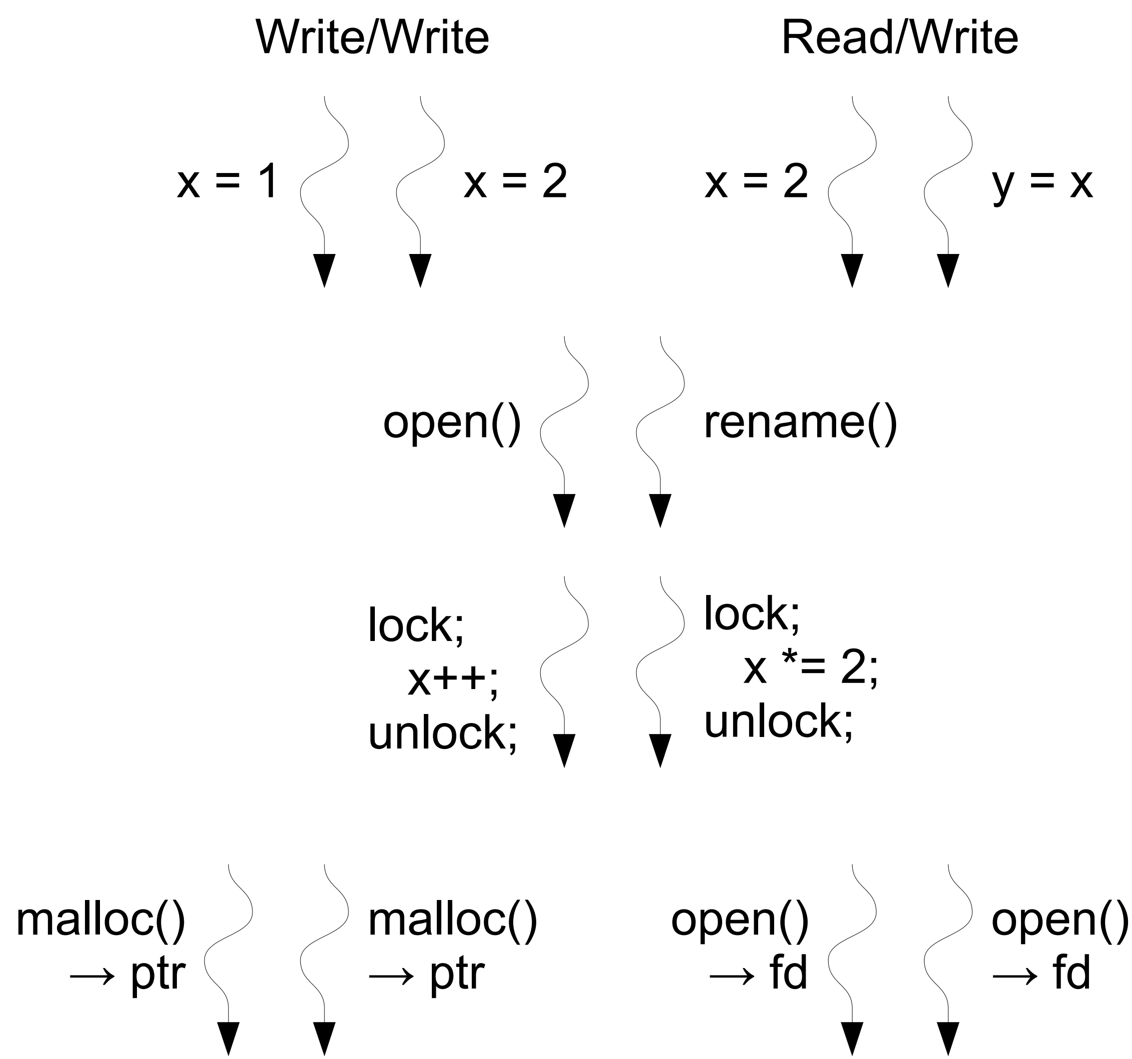
Why? Parallelism introduces:

- **Nondeterminism** (in general)
  - Execution behavior subtly depends on timing
- **Data Races** (in particular)
  - Unsynchronized concurrent state changes

→ ***Heisenbugs:*** sporadic, difficult to reproduce

Races are Everywhere

- Memory Access
- File Access
- Synchronization
- System APIs



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: help *manage* races but don't *eliminate* them

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)
- Separation-enforcing type systems (DPJ)

What about race-freedom for **any language?**

Introducing **Determinator**

New OS offering *race-free parallel programming*

- Compatible with arbitrary (existing) languages
  - C, C++, Java, assembly, …
- Avoids races at multiple abstraction levels
  - Shared memory, file system, synch, ...
- Takes *clean-slate* approach for simplicity
  - Ideas could be retrofitted into existing Oses
- Current focus: *compute-bound* applications
  - Early prototype, many limitations

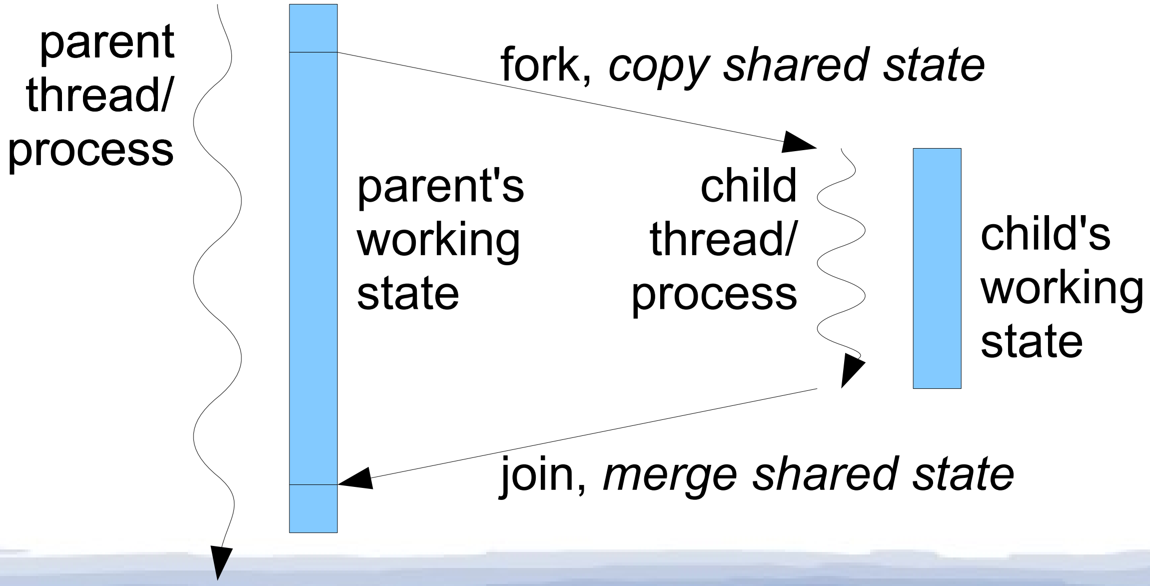
Talk Outline

- ✔ Introduction: Parallelism and Data Races
- Determinator's Programming Model
- Prototype Kernel/Runtime Implementation
- Performance Evaluation

Determinator's Programming Model

“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?

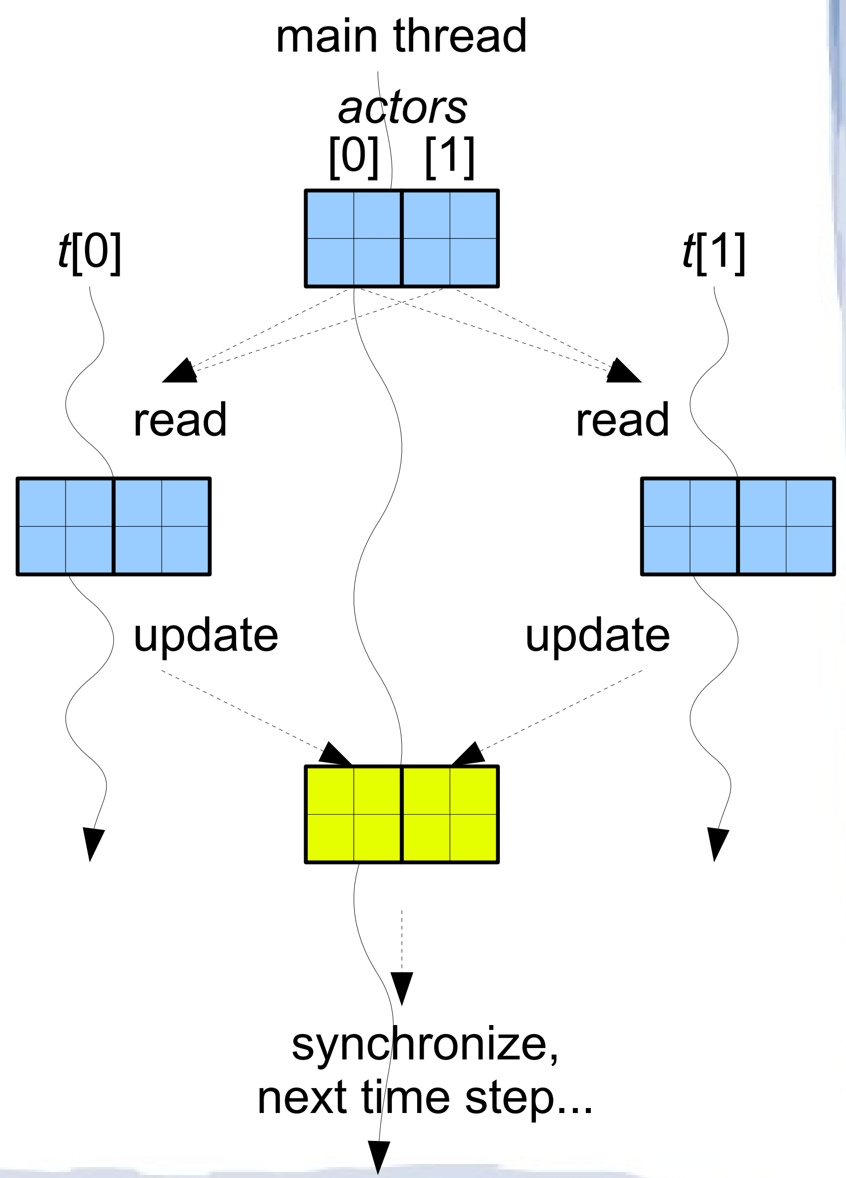
Precedents for “check-in/check-out” model:

- DOALL in early parallel Fortran computers
  - Burroughs FMP 1980, Myrias 1988
  - Language-specific, limited to DO loops
- Version control systems (cvs, svn, git, …)
  - Manual check-in/check-out procedures
  - For files only, not shared memory state

Determinator applies this model *pervasively* and *automatically* to all shared state

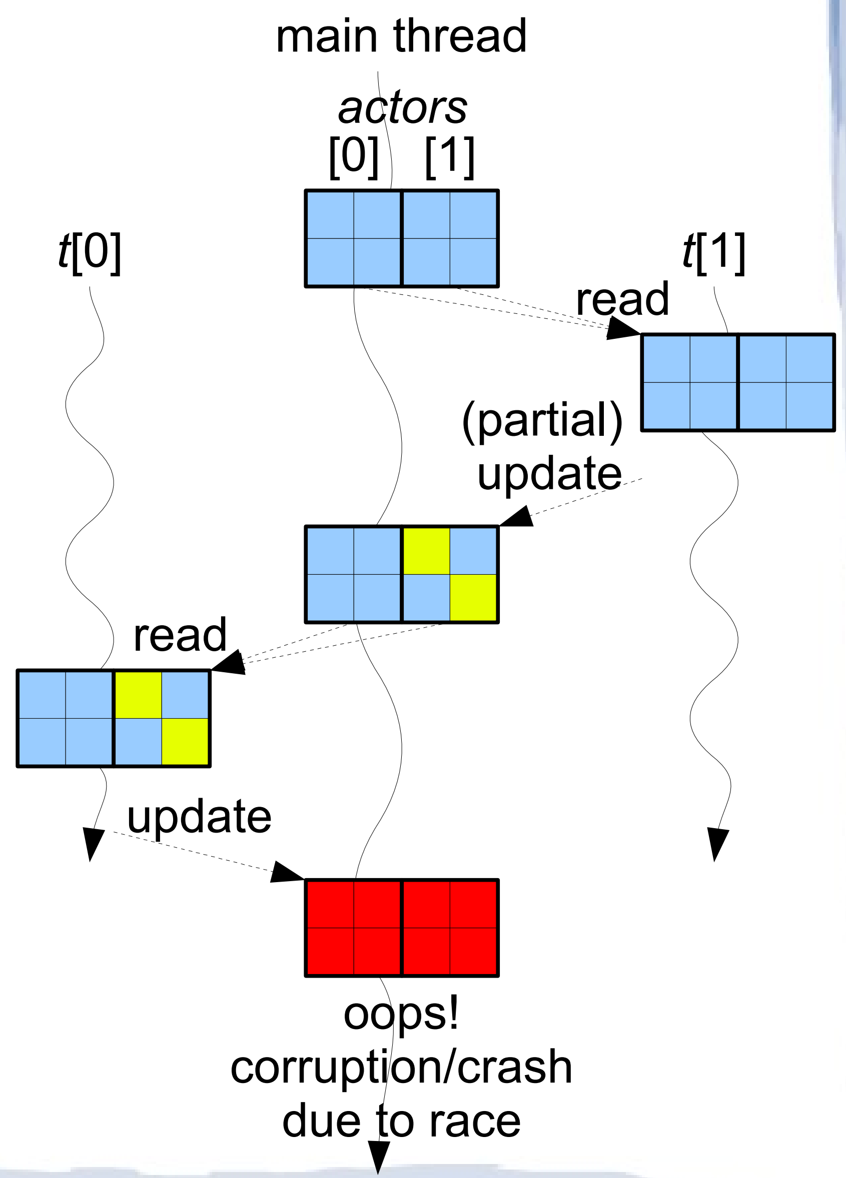
Example 1: Gaming/Simulation, Conventional Threads

```
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]);
        }
}
```



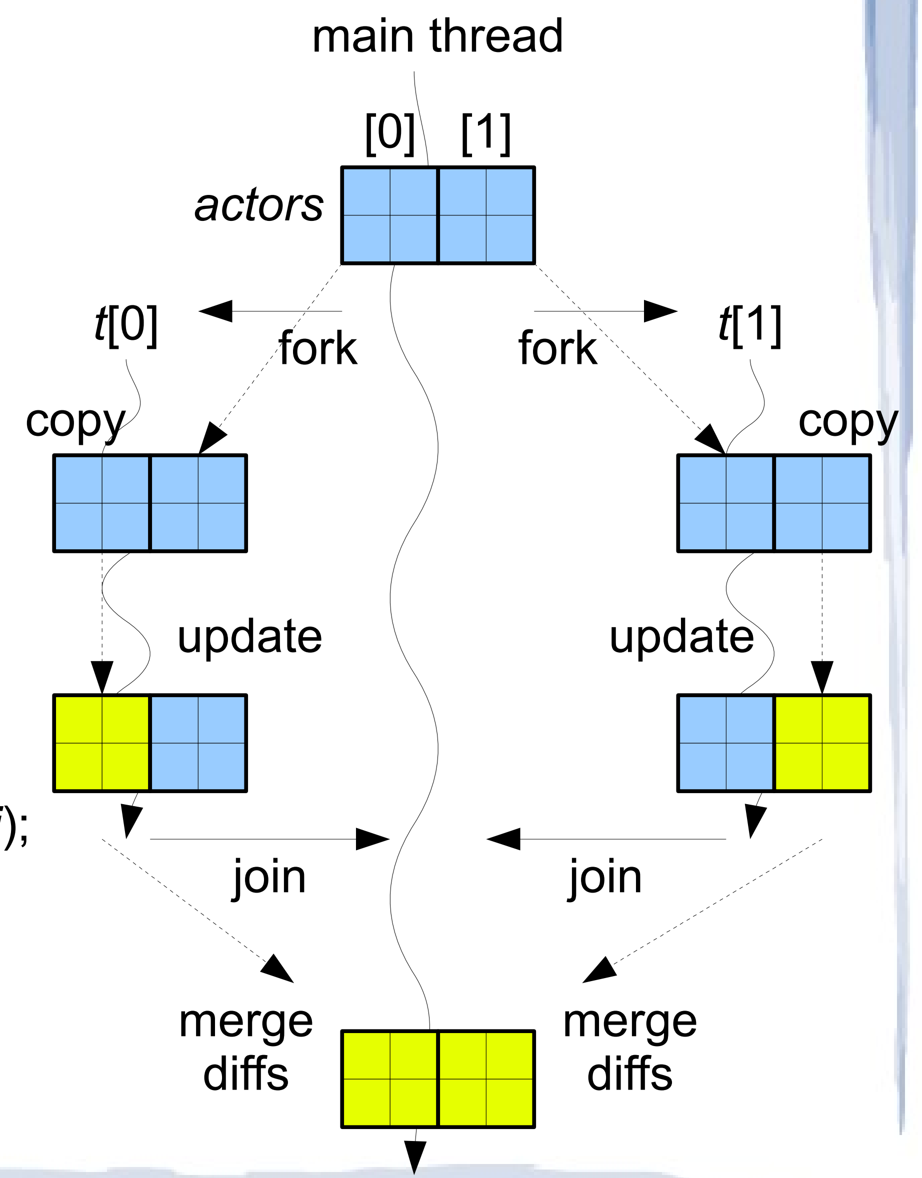
Example 1: Gaming/Simulation, Conventional Threads

```
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]);
        }
}
```



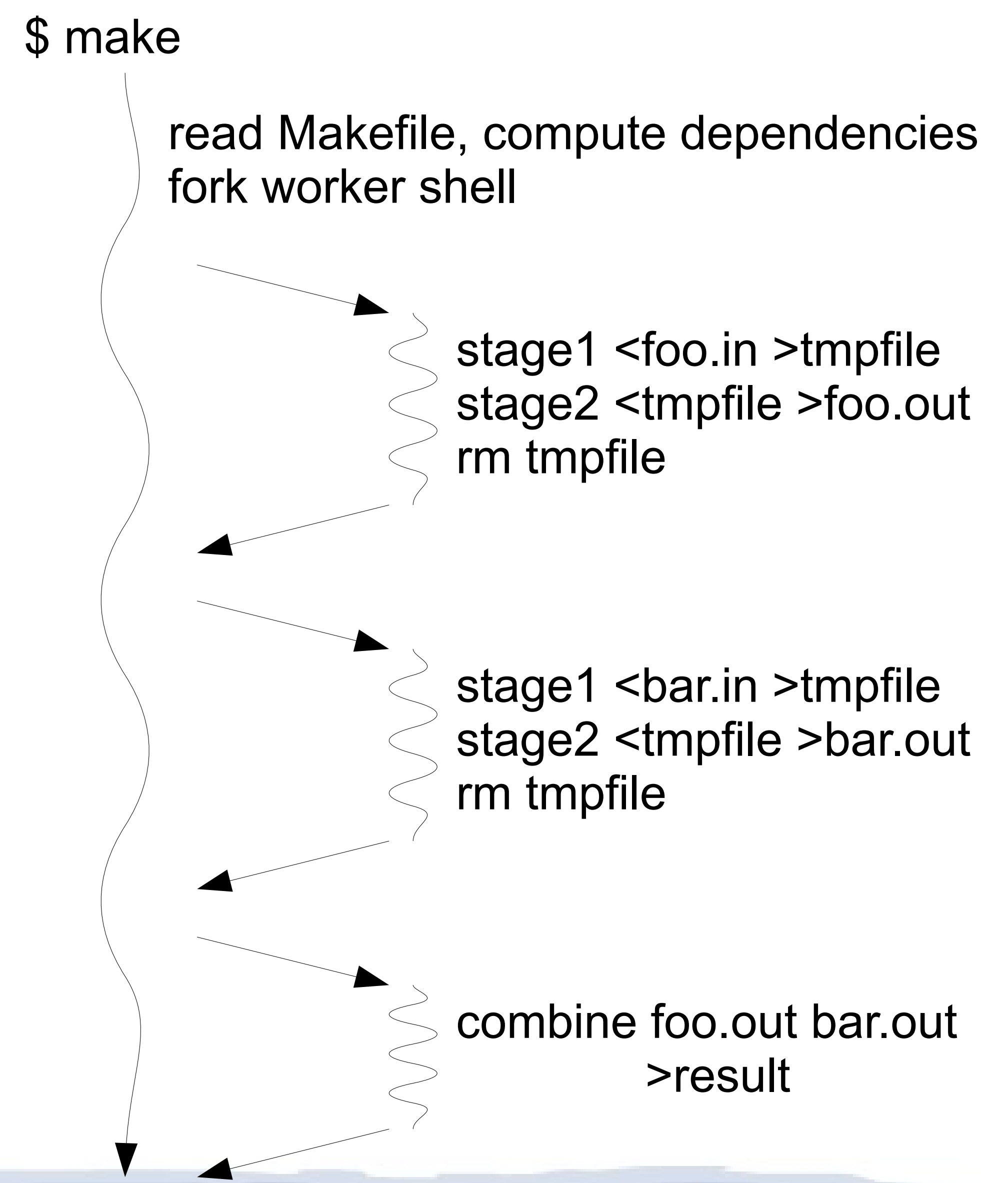
Example 1: Gaming/Simulation, **Determinator Threads**

```
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]);
        }
}
```



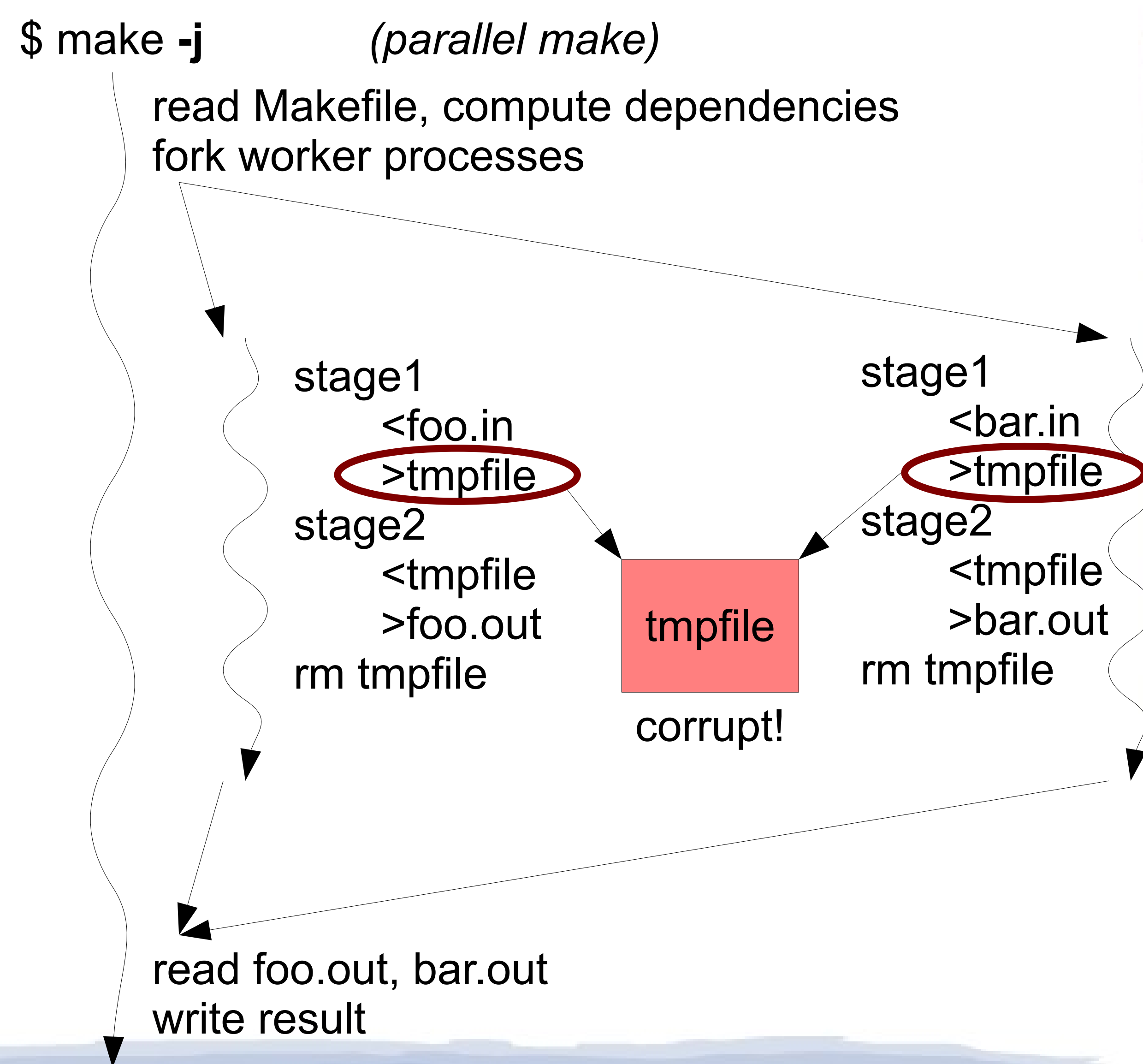
Example 2: Parallel Make/Scripts, Conventional Unix Processes

```
# Makefile for file 'result'

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

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



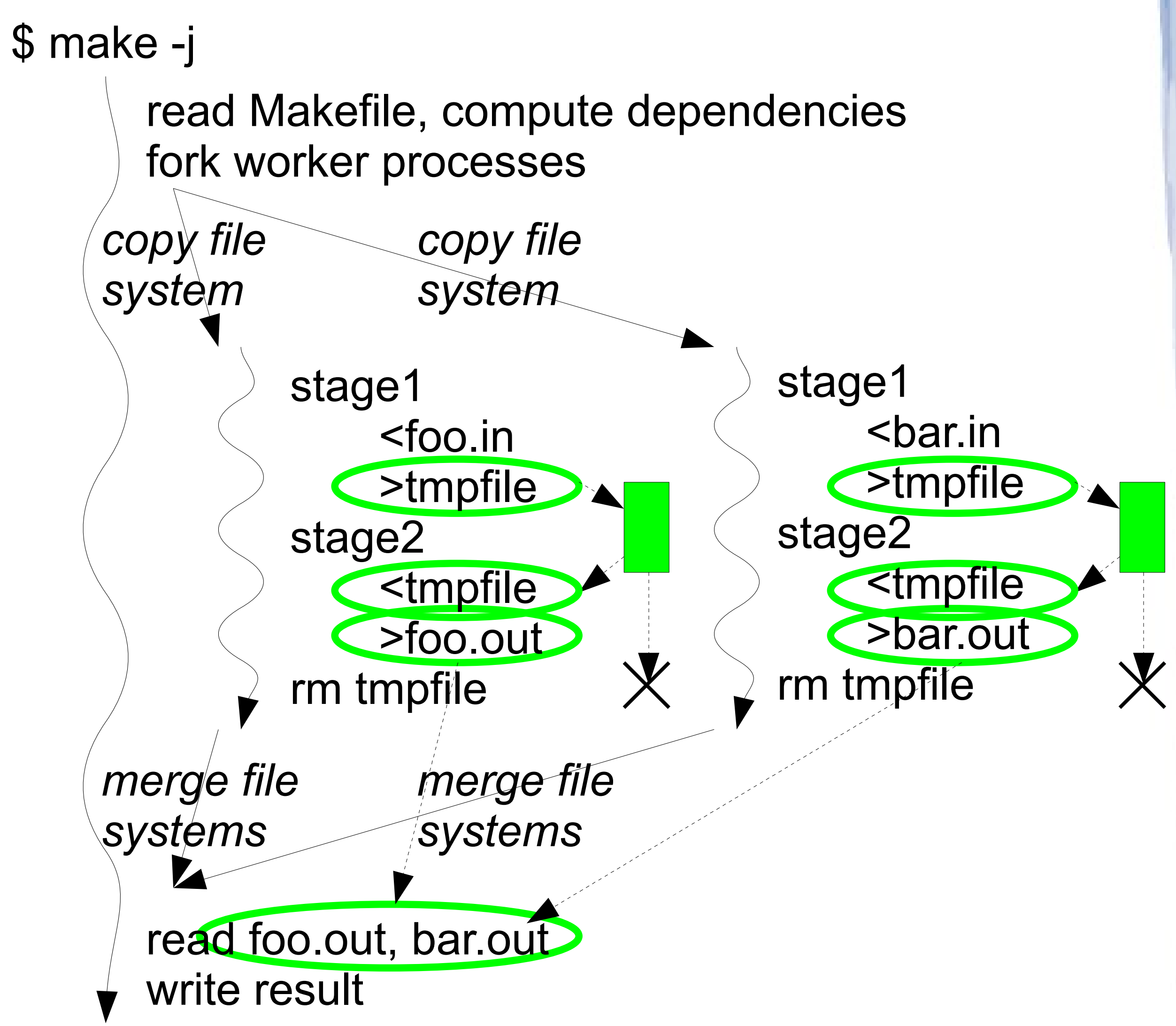
Example 2: Parallel Make/Scripts, Conventional Unix Processes

```
# Makefile for file 'result'

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

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



Example 2: Parallel Make/Scripts, **Determinator Processes**

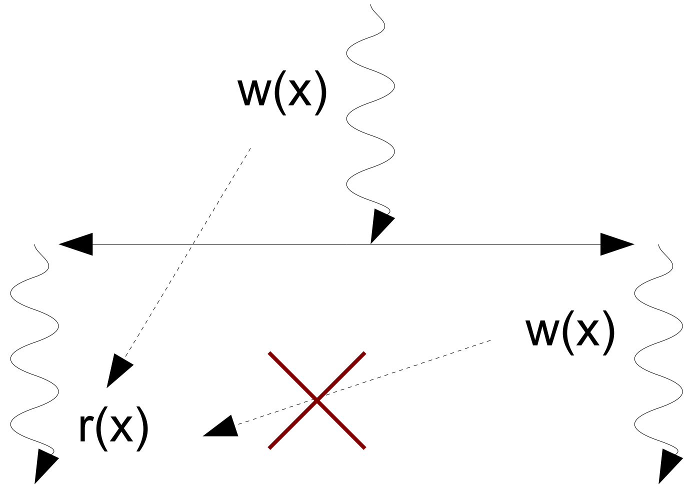
```
# Makefile for file 'result'

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

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



What Happens to Data Races?

Read/Write races: go away *entirely*

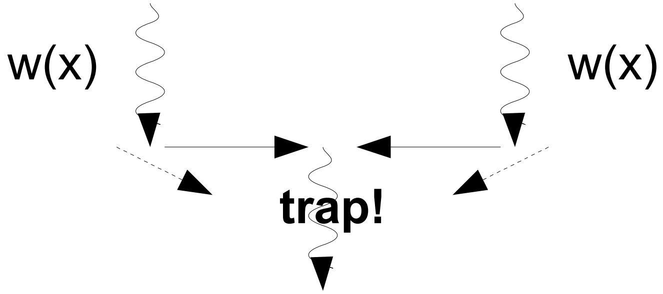
- writes propagate *only* via synchronization
- reads *always* see last write by *same* thread, else value at last synchronization point



What Happens to Data Races?

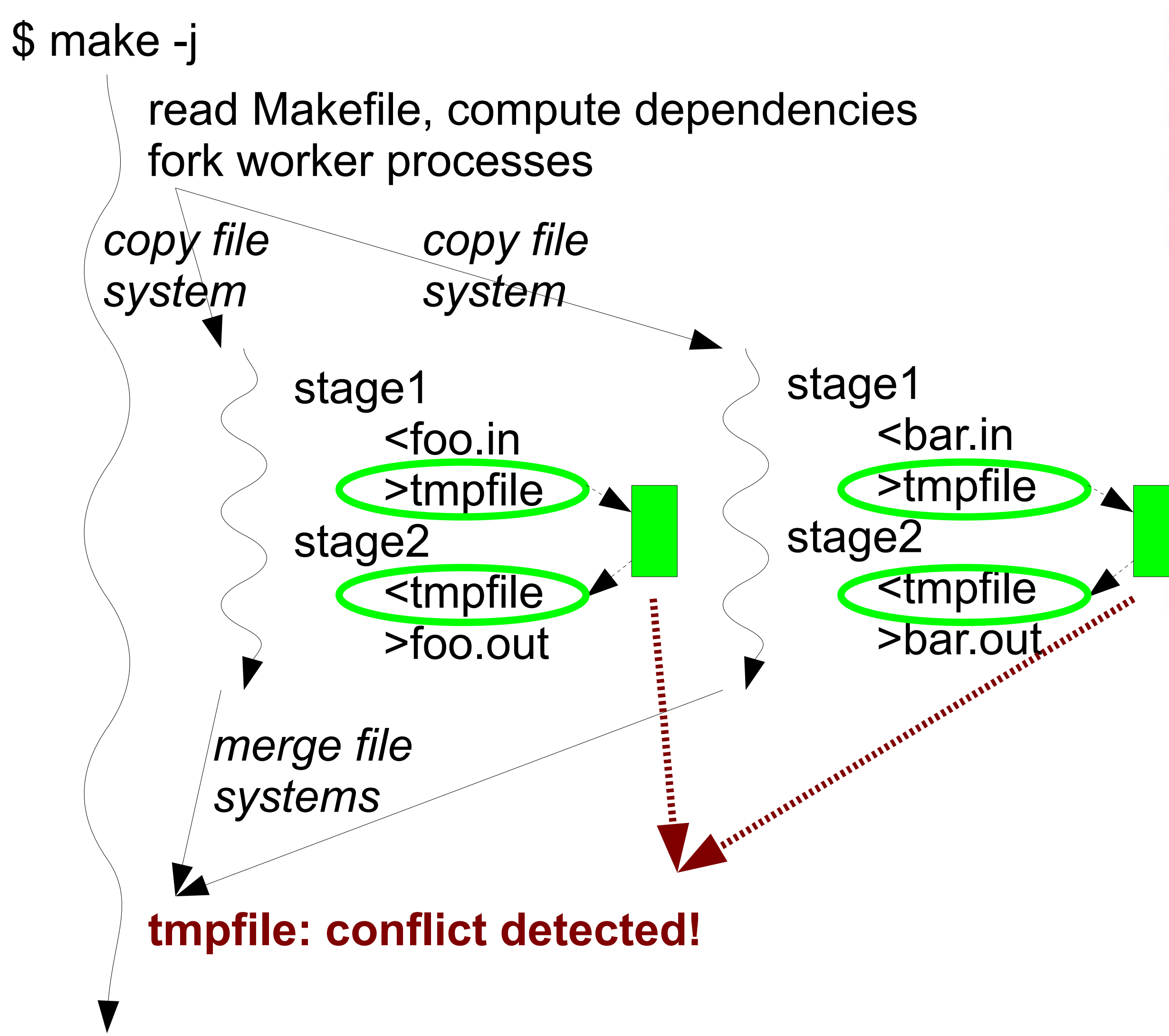
Write/Write races:

- go away if threads “undo” their changes
  - tmpfile in make -j example
- otherwise become deterministic *conflicts*
  - *always detected* at join/merge point
  - runtime exception, just like divide-by-zero



Example 2: Parallel Make/Scripts, **Determinator Processes**

```
# Makefile for file 'result'

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

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



Repeatability

Ability to replay past executions gives us:

- Bug reproducibility
- Time-travel debugging (reverse execution)
- [Byzantine] fault tolerance
- Computation accountability (PeerReview)
- Intrusion analysis/response (ReVirt, IntroVirt)

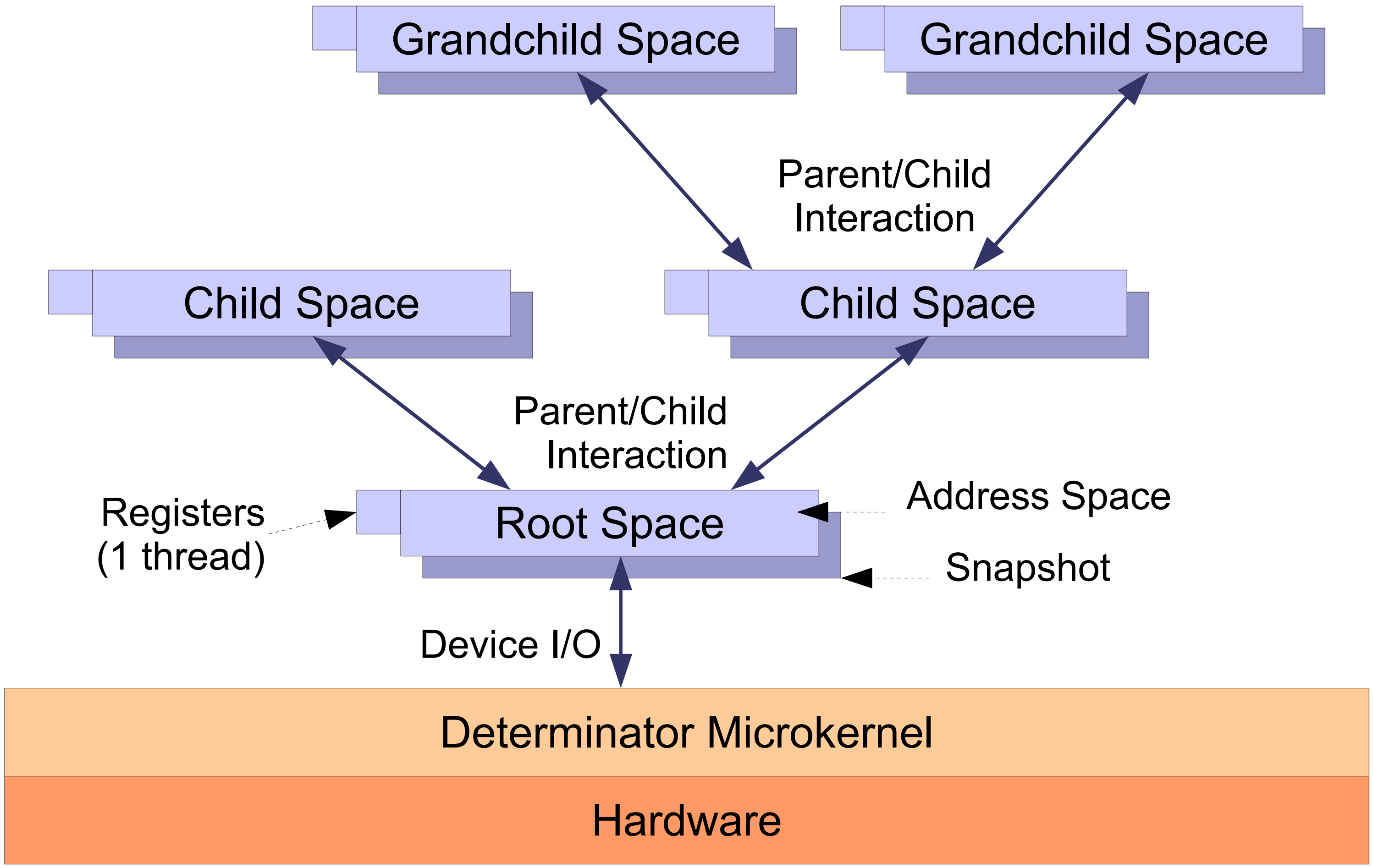
Sometimes need *system-enforced determinism*

- replay *arbitrary malicious code* exactly

Talk Outline

- ✔ Introduction: Parallelism and Data Races
- ✔ Determinator's Programming Model
- **Prototype Kernel/Runtime Implementation**
- Performance Evaluation

Determinator OS Architecture



Microkernel API

Three system calls:

- PUT: copy data into child, snapshot, start child
- GET: copy data or modifications out of child
- RET: return control to parent

(and a few options to each – see paper)

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

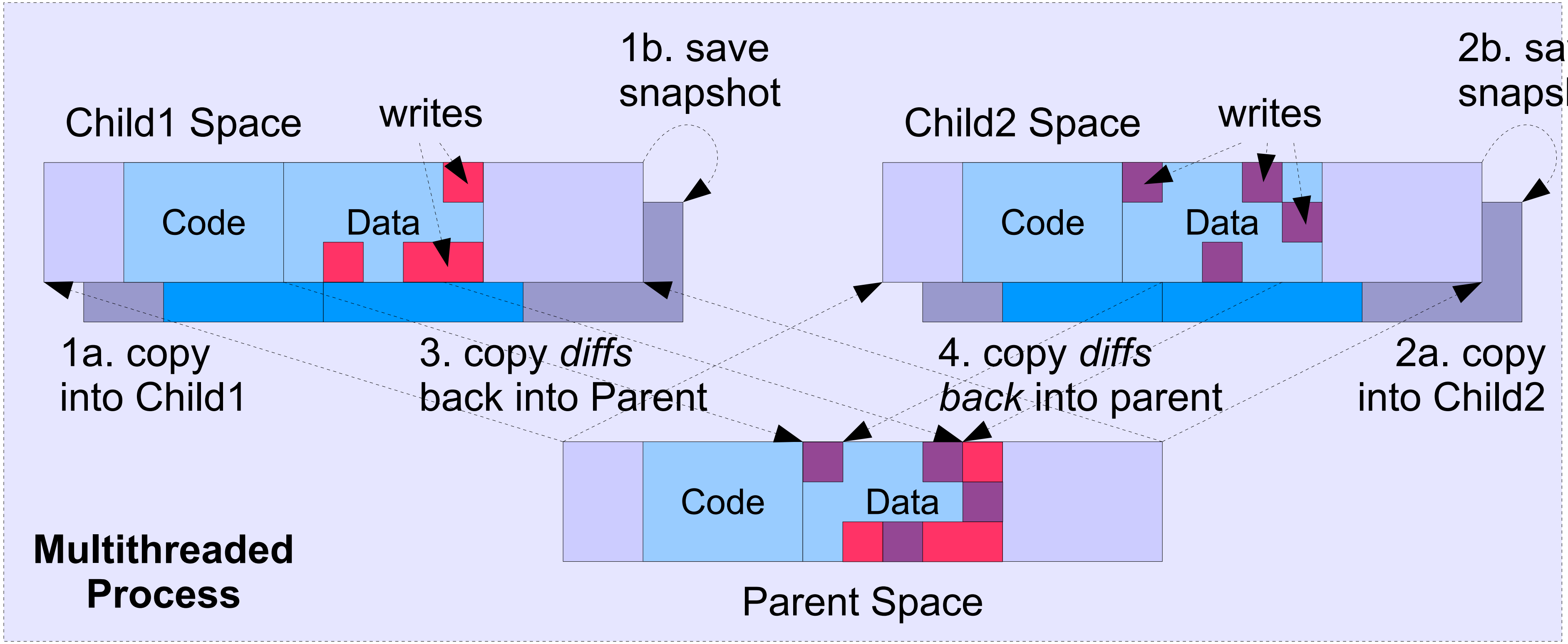
it's a library → all facilities are optional

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



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

Threads, Classic Style

Optional **deterministic scheduling**

- Backward compatible with pthreads API
- Similar to DMP/CoreDet approach
  - Quantize execution by counting instructions

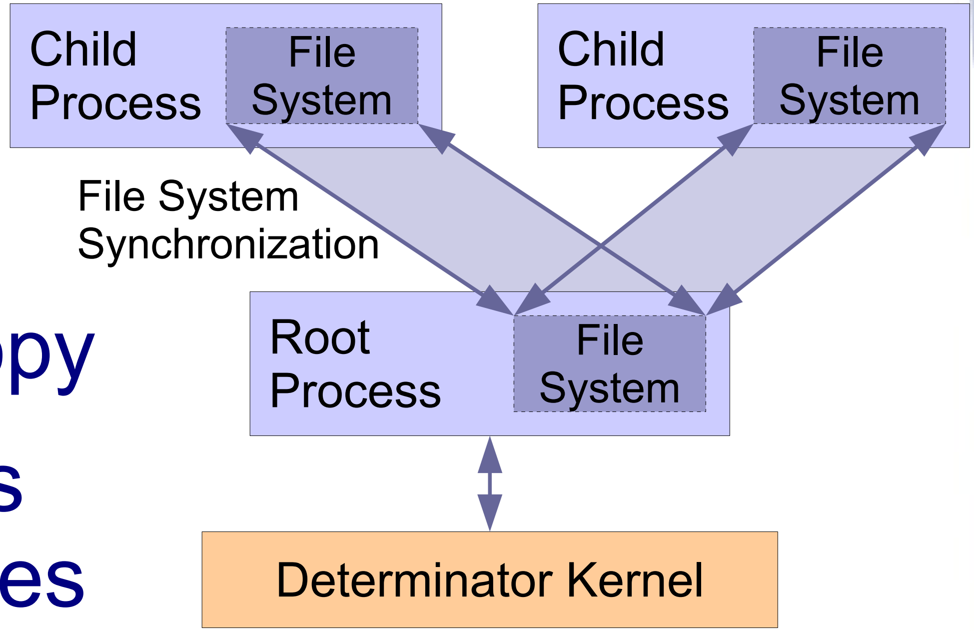
Disadvantages:

- Same old parallel programming model
  - Races, schedule-dependent bugs still possible
- Quantization incurs runtime overhead

Emulating a Shared File System

Each process has a *complete file system replica* in its address space

- a "distributed FS" 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

File System Conflicts

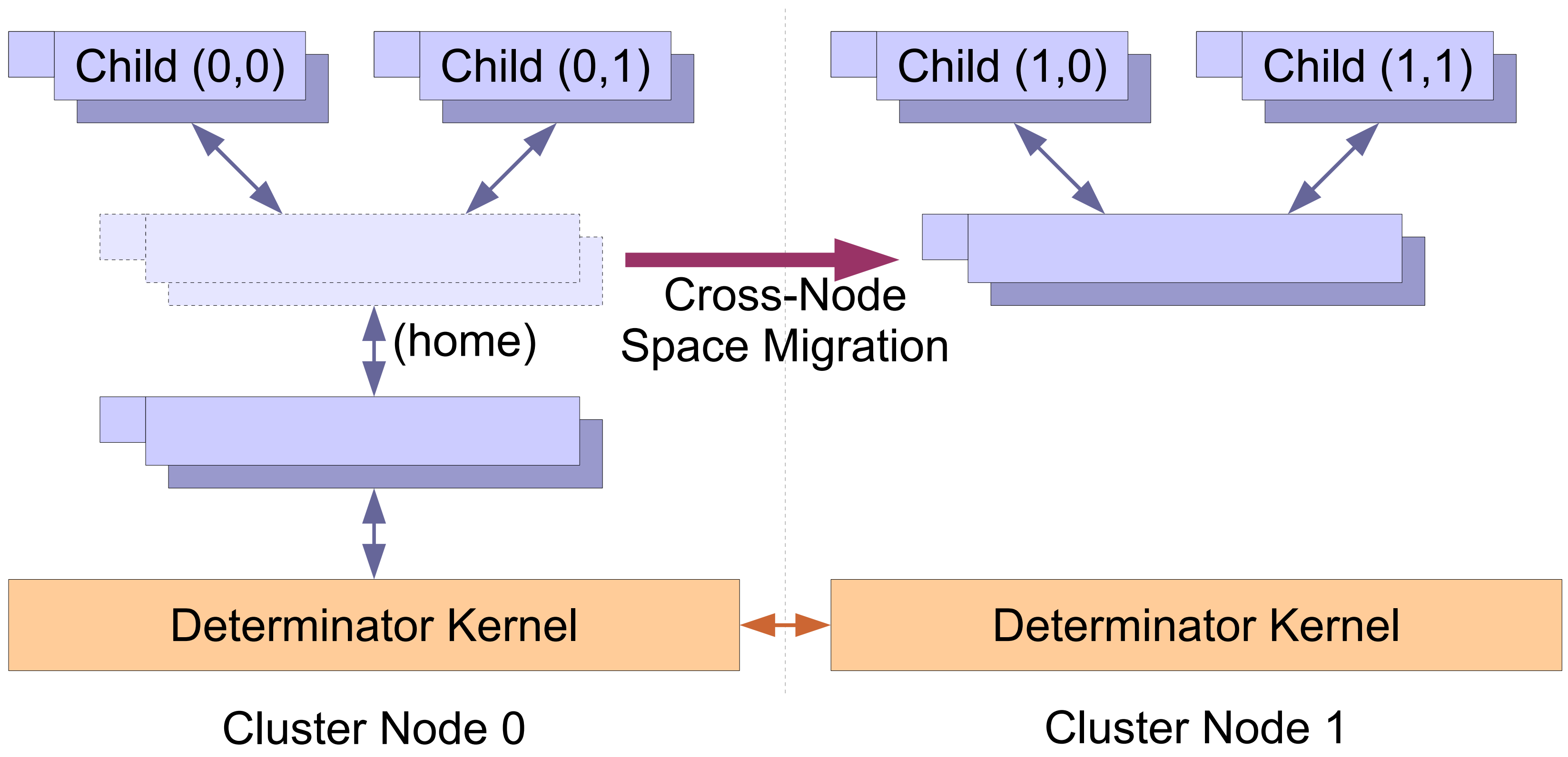
Hard conflicts:

- 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

Distributed Computing



Other Features (See 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

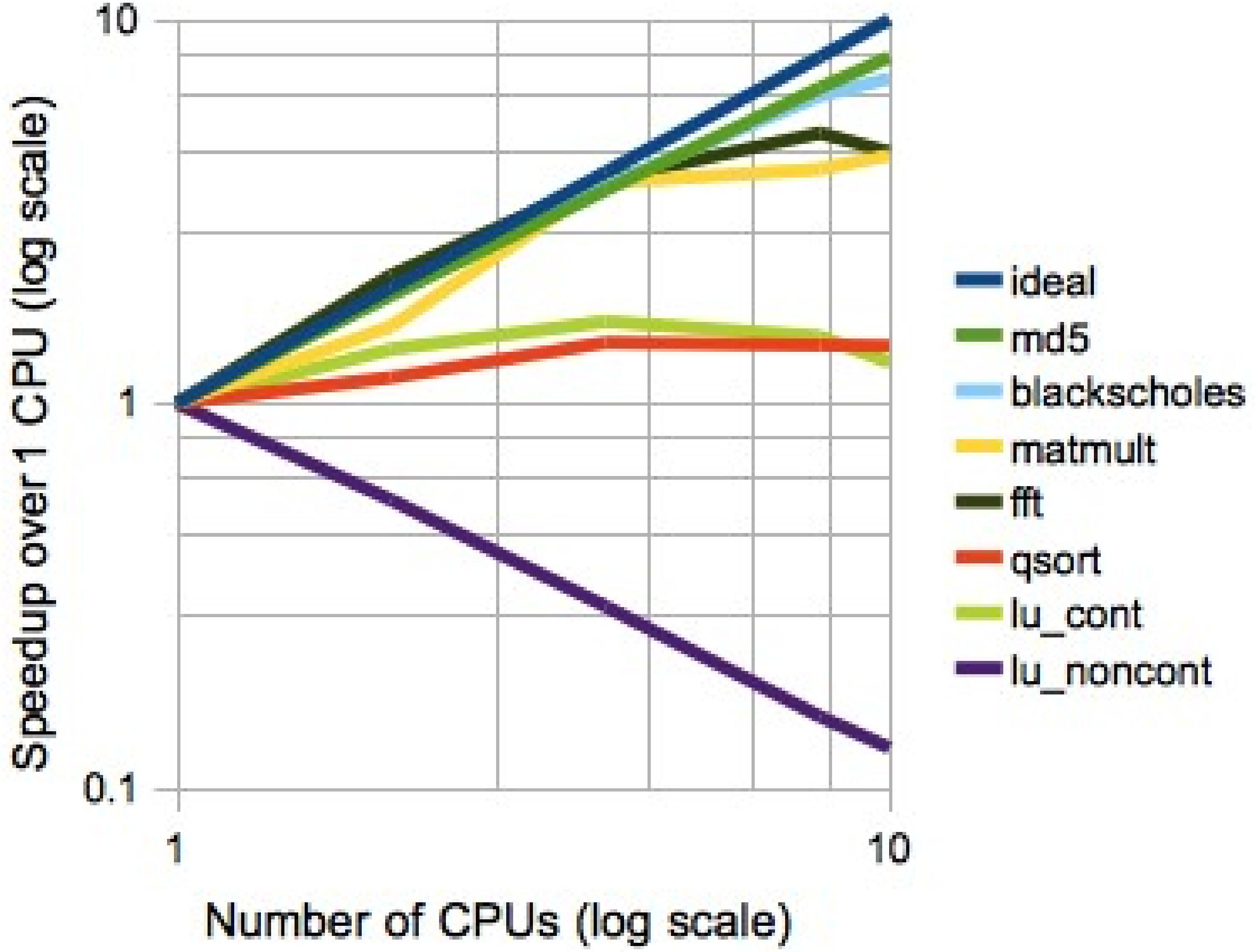
Question: Can such a programming model be:

- efficient
- scalable

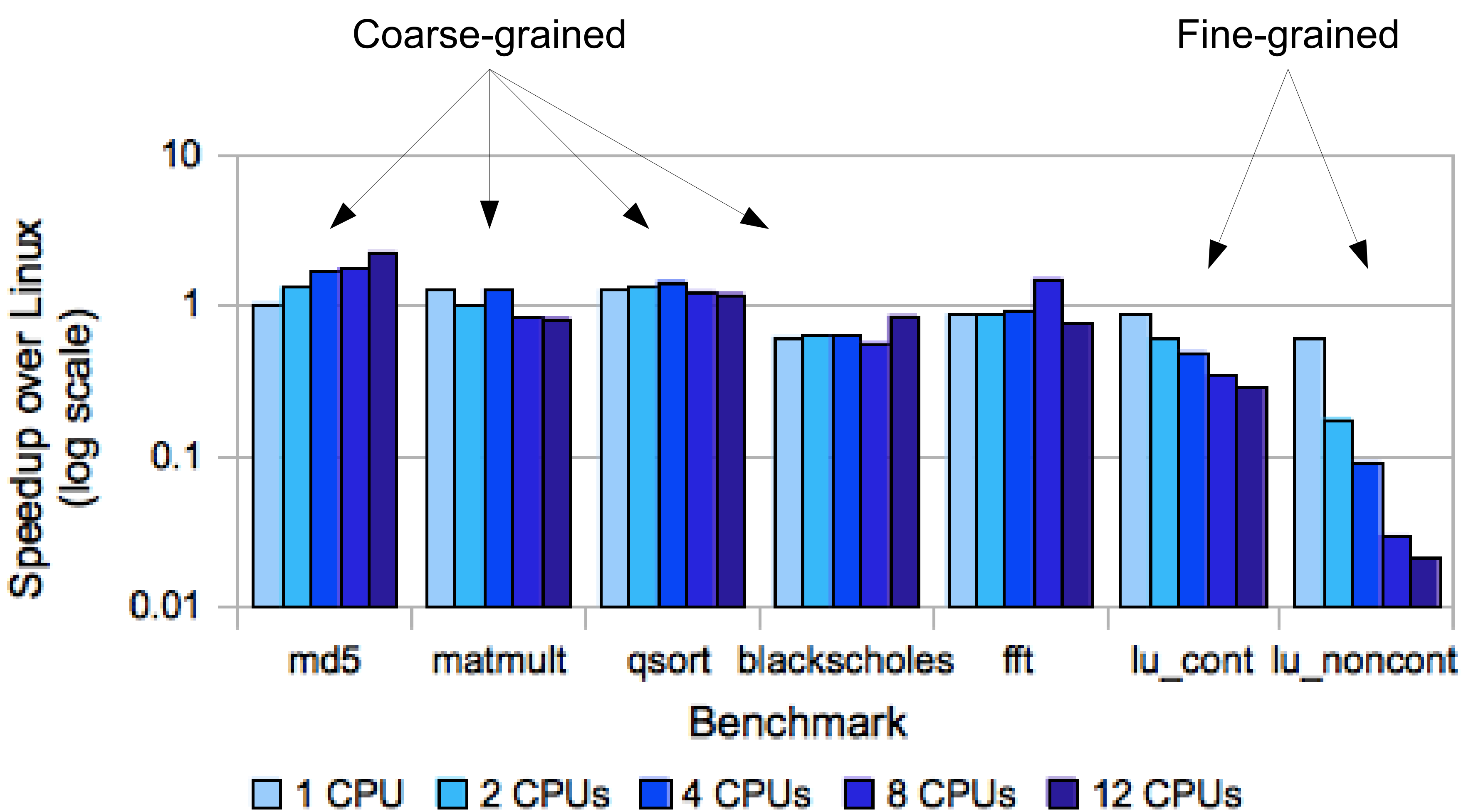
...enough for everyday use in real apps?

Answer: *it depends on the app* (of course).

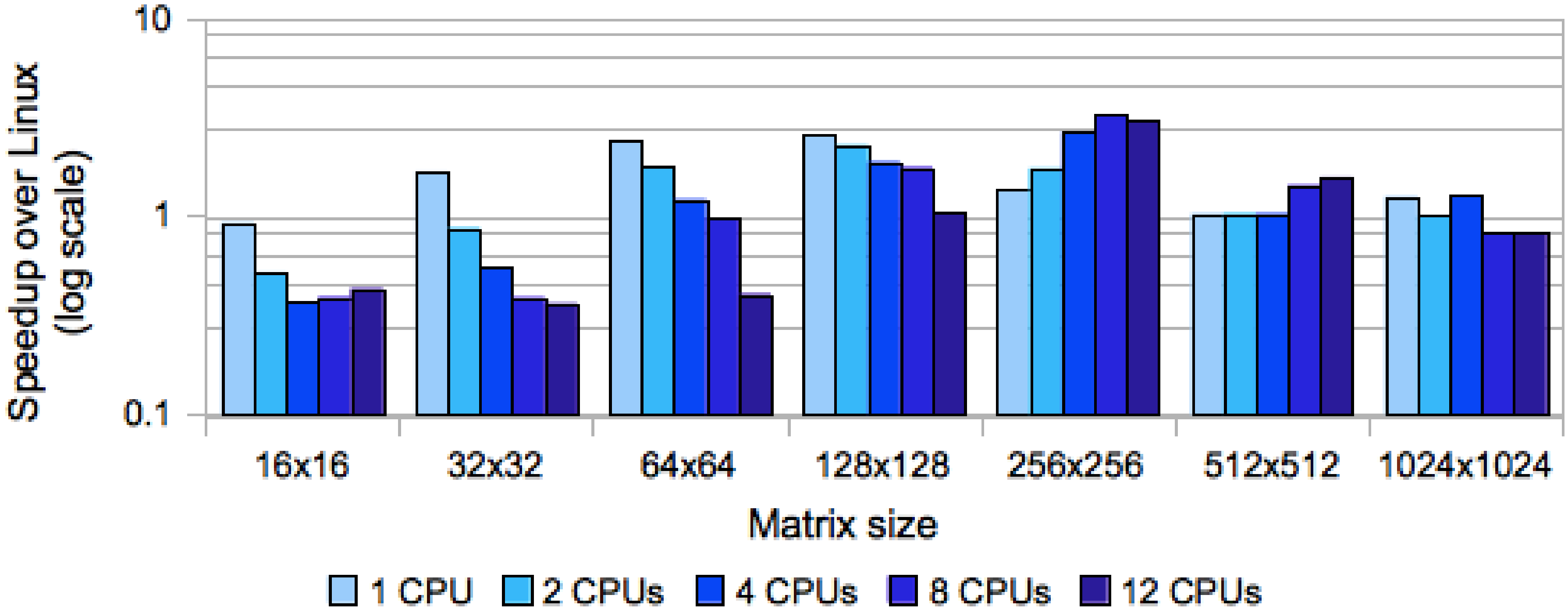
Single-Node Speedup over 1 CPU



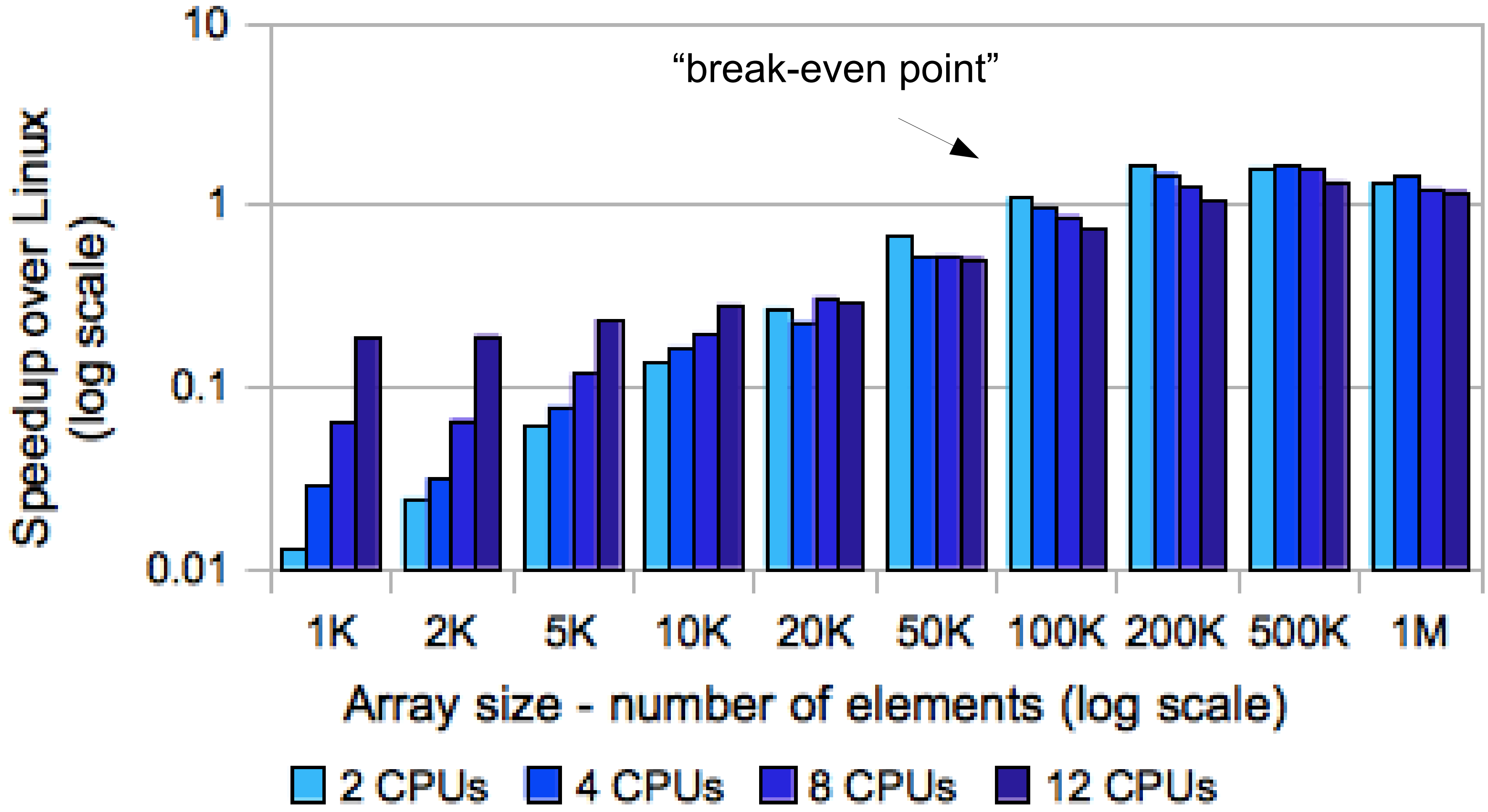
Single-Node Performance: Determinator versus Linux



Drilldown: Varying Granularity (Parallel Matrix Multiply)



Drilldown: Varying Granularity (Parallel Quicksort)



Future Work

Current early prototype has many limitations left to be addressed in future work:

- Generalize hierarchical fork/join model
- Persistent, deterministic file system
- Richer device I/O and networking (TCP/IP)
- Clocks/timers, interactive applications
- Backward-compatibility with existing OS
- …

Conclusion

- 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 VM optimizations
  - Minimal overhead for coarse-grained apps

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

Acknowledgments

Thank you:
Zhong Shao, Rammakrishna Gummadi,
Frans Kaashoek, Nickolai Zeldovich, Sam King,
the OSDI reviewers

Funding:
ONR grant N00014-09-10757
NSF grant CNS-1017206

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