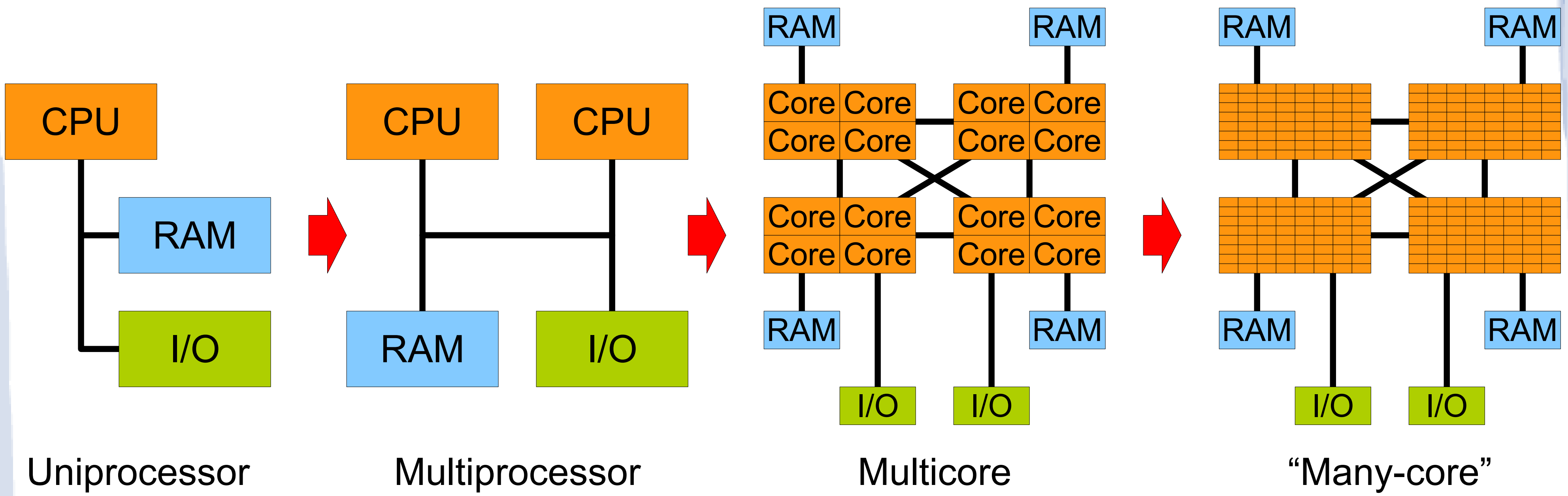
Towards Predictable, *Heisenbug-Free* Parallel Software Environments

Bryan Ford
Amittai Aviram, Yu Zhang,
Shu-Chun Weng, Sen Hu

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

Harvard University – November 3, 2011

Pervasive Parallelism



Industry shifting from "faster" to "wider" CPUs

Today's Grand Software Challenge

Parallelism makes programming harder.

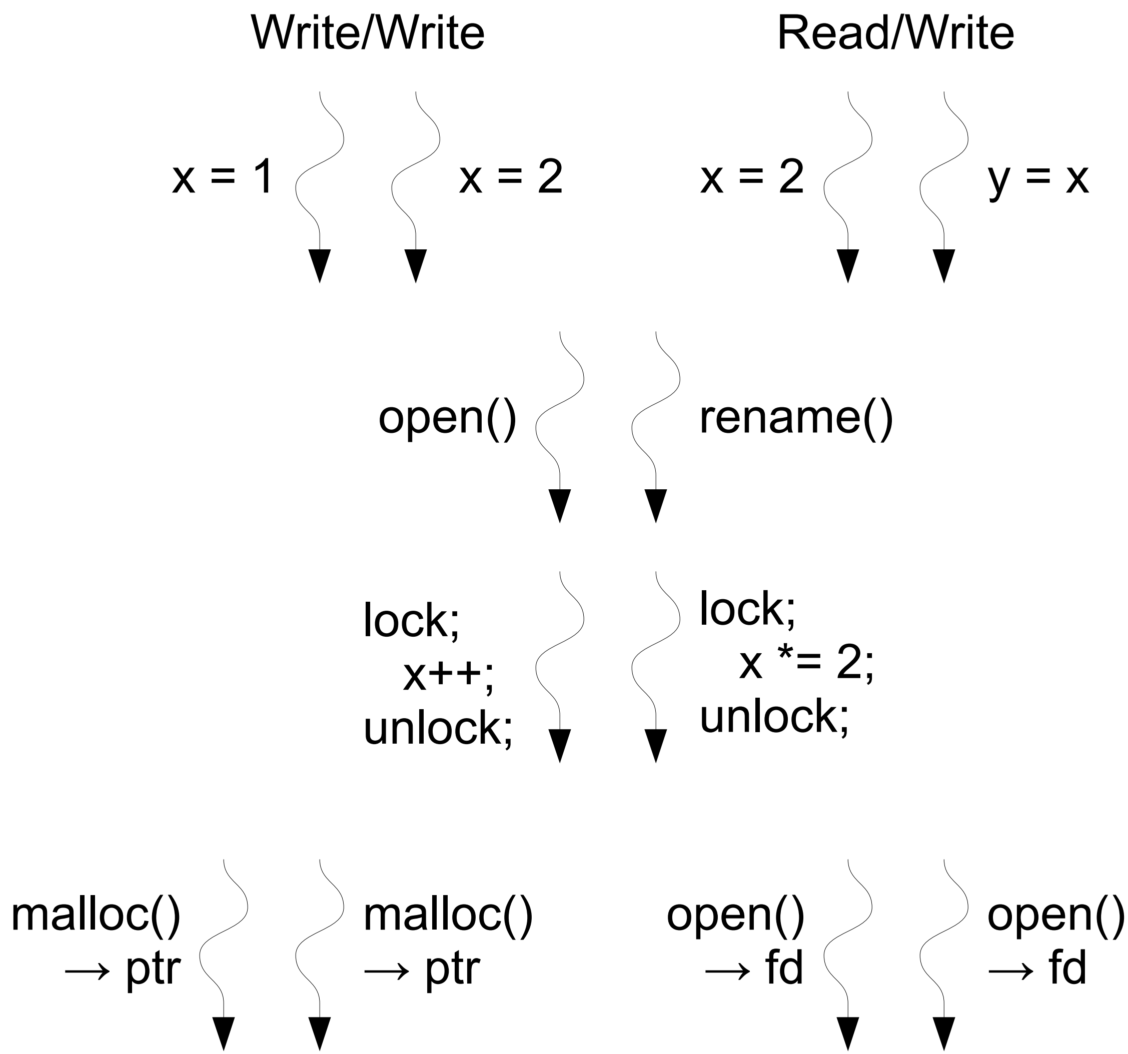
Why? Parallelism introduces:

- **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, …)

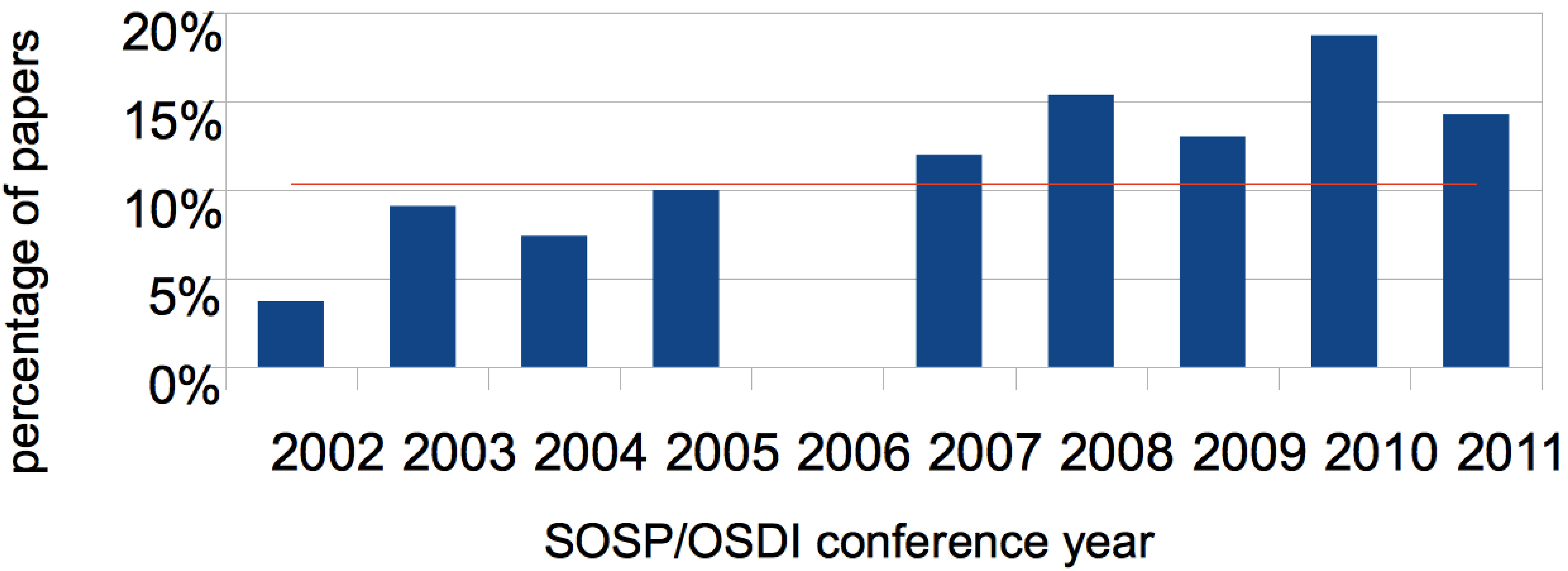
- Analyze/instrument program to help find races

Deterministic schedulers (DMP, Grace, CoreDet)

- Synthesize a repeatable execution schedule

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

“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 in **particular 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 a *race-free parallel environment*

- 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, still work-in-progress...

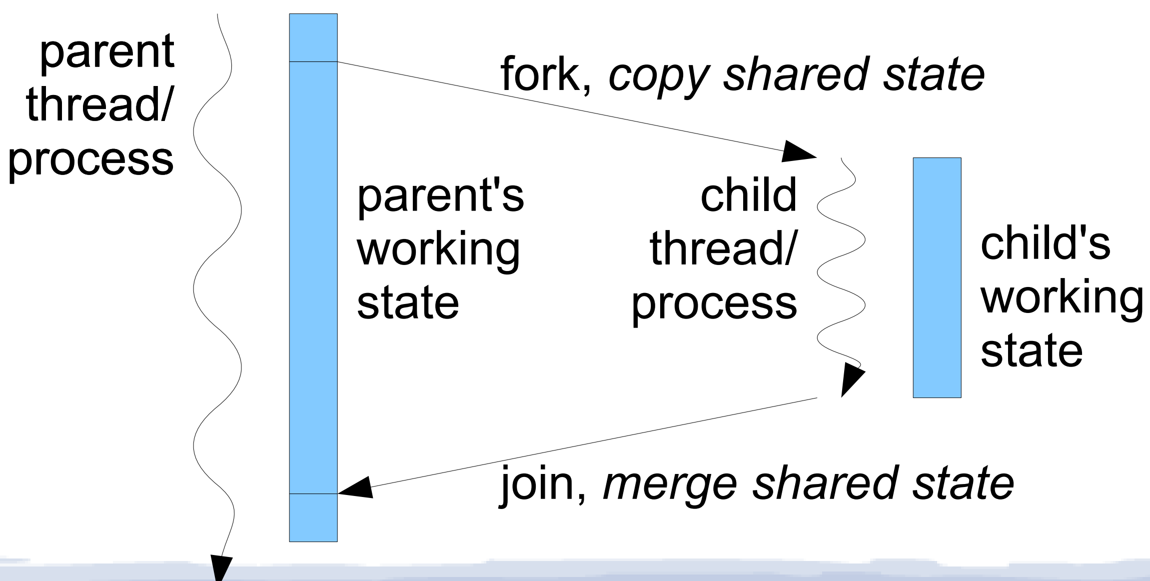
Talk Outline

- ✓ Introduction: Parallelism and Data Races
- Determinator Programming Model and Design
  - Deterministic "threads" and "shared memory"
  - Deterministic "processes" and "file systems"
- Challenges and Ongoing Work
  - New abstractions versus legacy compatibility
  - Performance and scalability
  - Deterministic distributed computing
- Conclusion

Determinator's Programming Model

Private workspace 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 private workspace 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
- Snapshot consistency in databases
  - For them it's a bug, for us it's a feature

What does this Mean?

Determinator applies private workspace model *pervasively* to all application-visible shared state

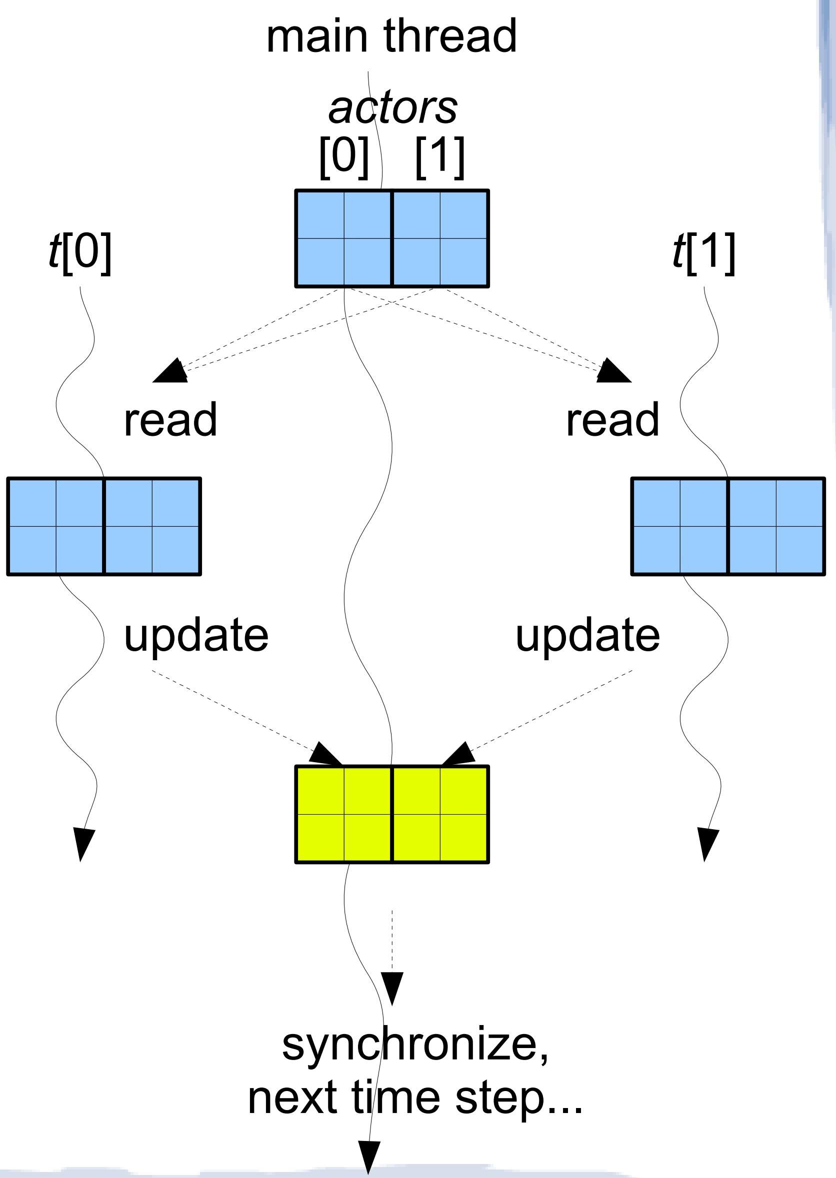
- **Threads** and **shared memory**
- **Processes** and shared **file systems**

Extensively use synchronization, reconciliation techniques developed for distributed systems...

- think **“distributed system in a box”**

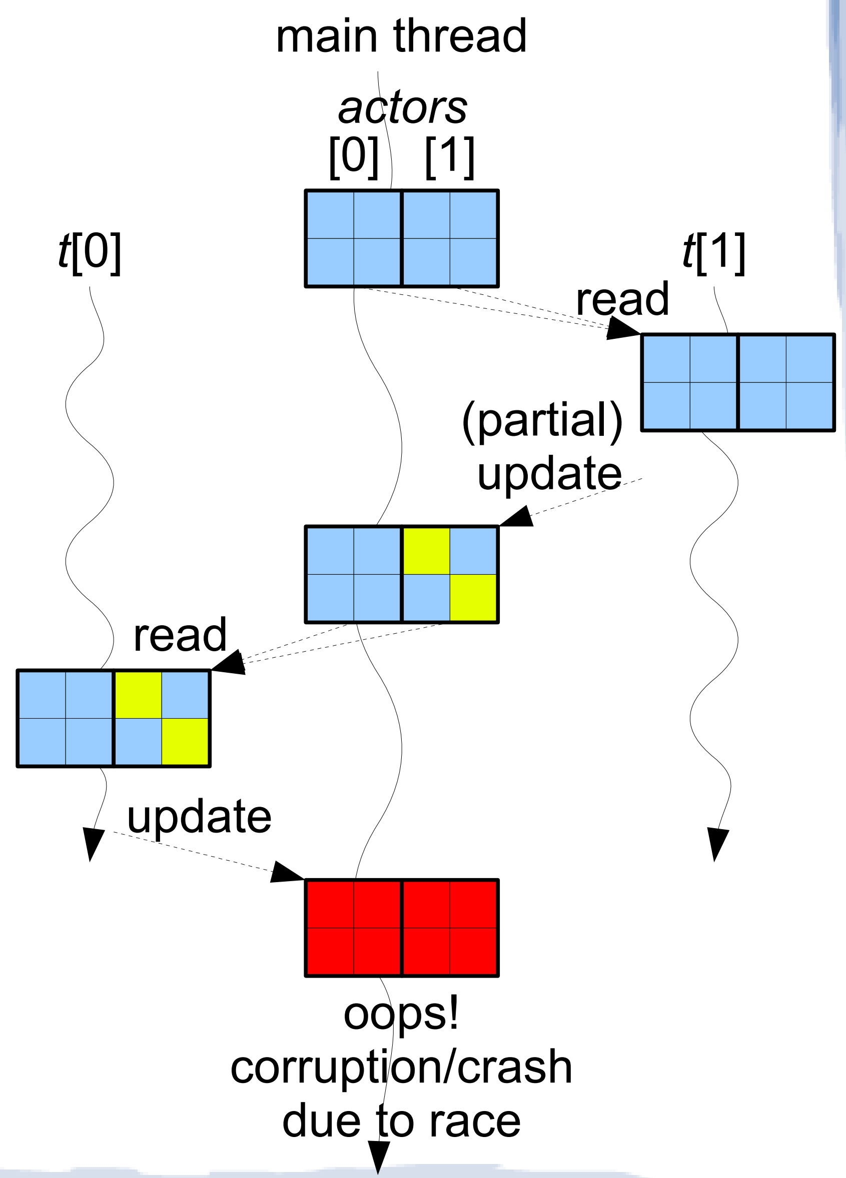
Example: Gaming/Simulation, Conventional Threads

```
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]);
        }
}
```



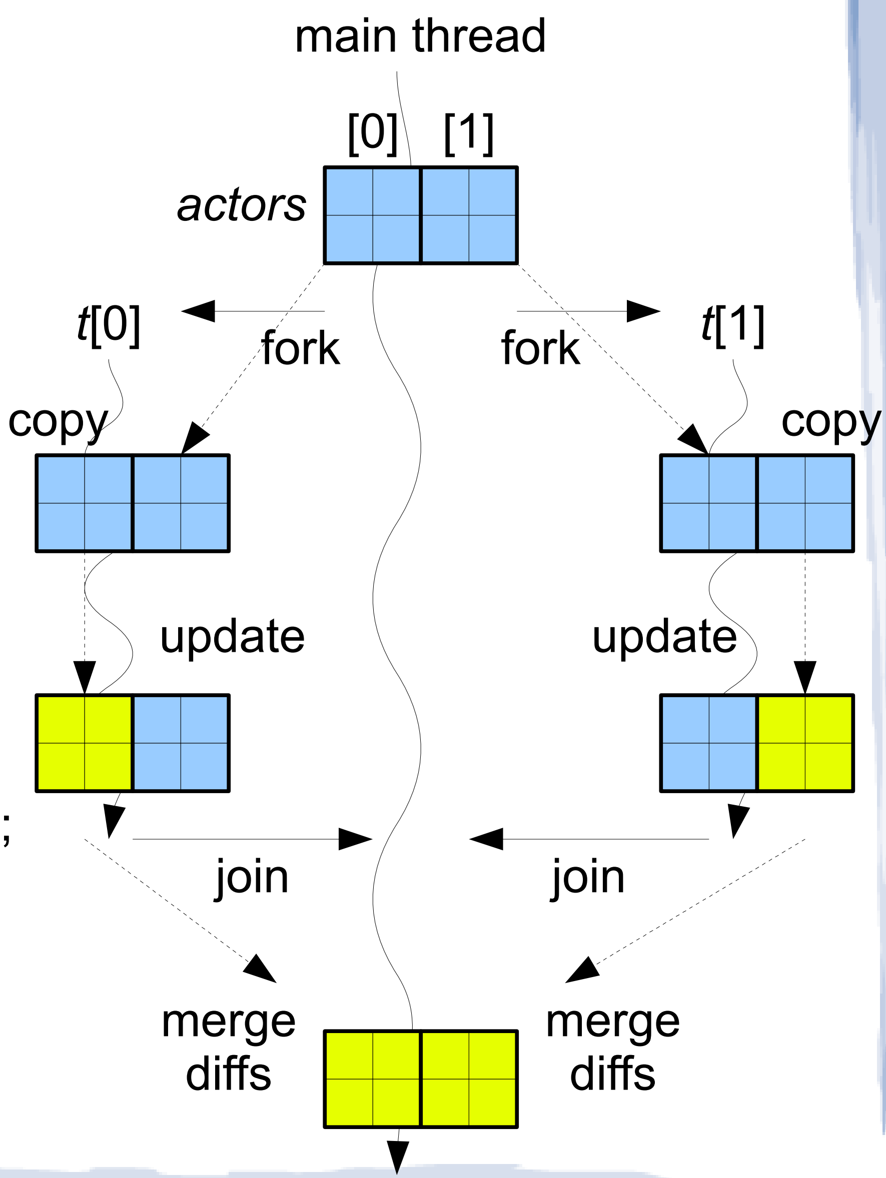
Example: Gaming/Simulation, Conventional Threads

```
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: 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]);
        }
}
```



What happened?

Buggy code (on conventional threads) became **correct** code (on Determinator threads)

Because: (informal intuition)

- Developer can *know* exactly what “version” of shared state in use at any point in code
- Synchronization defined by program logic → semantically deterministic, predictable

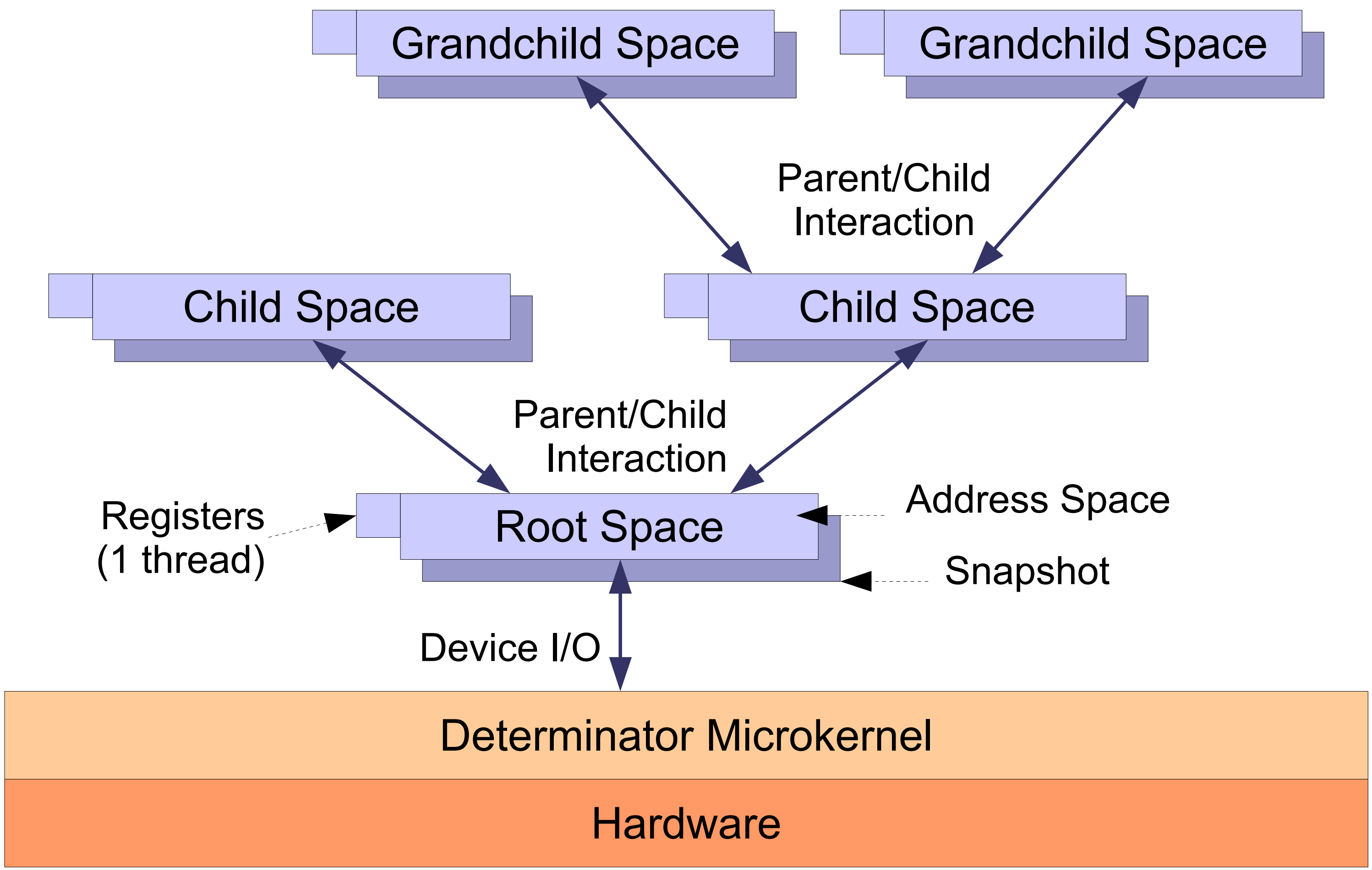
Details: [Aviram/Ford/Zhang, WoDet '11]

How Determinator Works

Determinator OS consists of:

- Minimal microkernel providing
  - 1 abstraction: hierarchy of *spaces*
  - 3 system calls: PUT, GET, RET
  - *no* files, shared memory, pipes, sockets, ...
- User-level runtime
  - emulates subset of Unix API: procs, files, etc.
  - it's a library → all facilities optional

Determinator OS Architecture

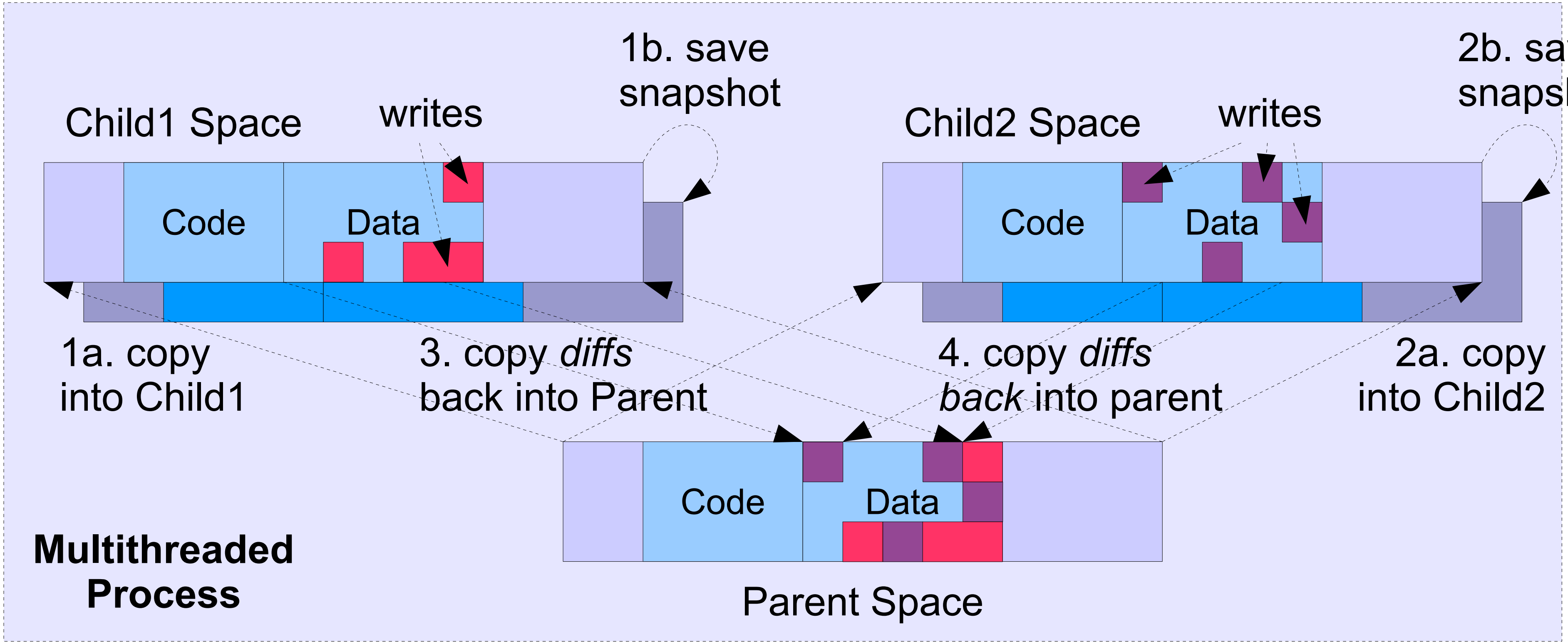


Threads, Determinator Style

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



Slow? Not necessarily...

Copy/snapshot quickly via **copy-on-write (COW)**

- Mark all pages *read-only*
- Duplicate *mappings* rather than *pages*
- Copy pages only on write attempt

Multi-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

What about File Systems?

File systems traditionally conflate two functions:

1. **Hierarchical abstraction**: files, directories
2. **Durable/persistent storage**: survives reboot

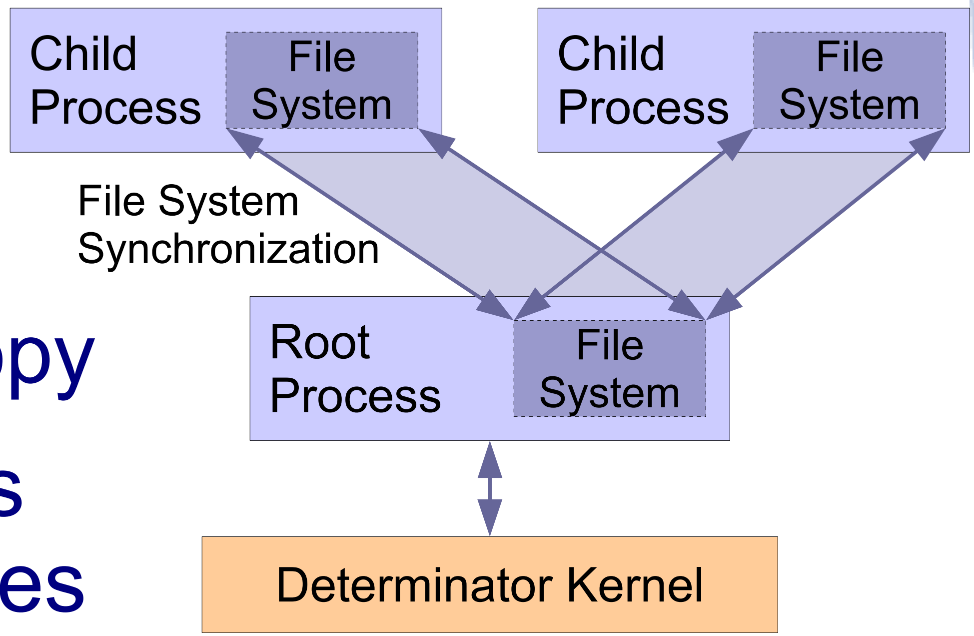
Determinator's design separates these functions

- **File system** offers abstraction, *not* persistence
- **Persistence** done by checkpointing spaces
    - *Work-in-progress.* Precedent: KeyKOS, L3

File Systems in Determinator

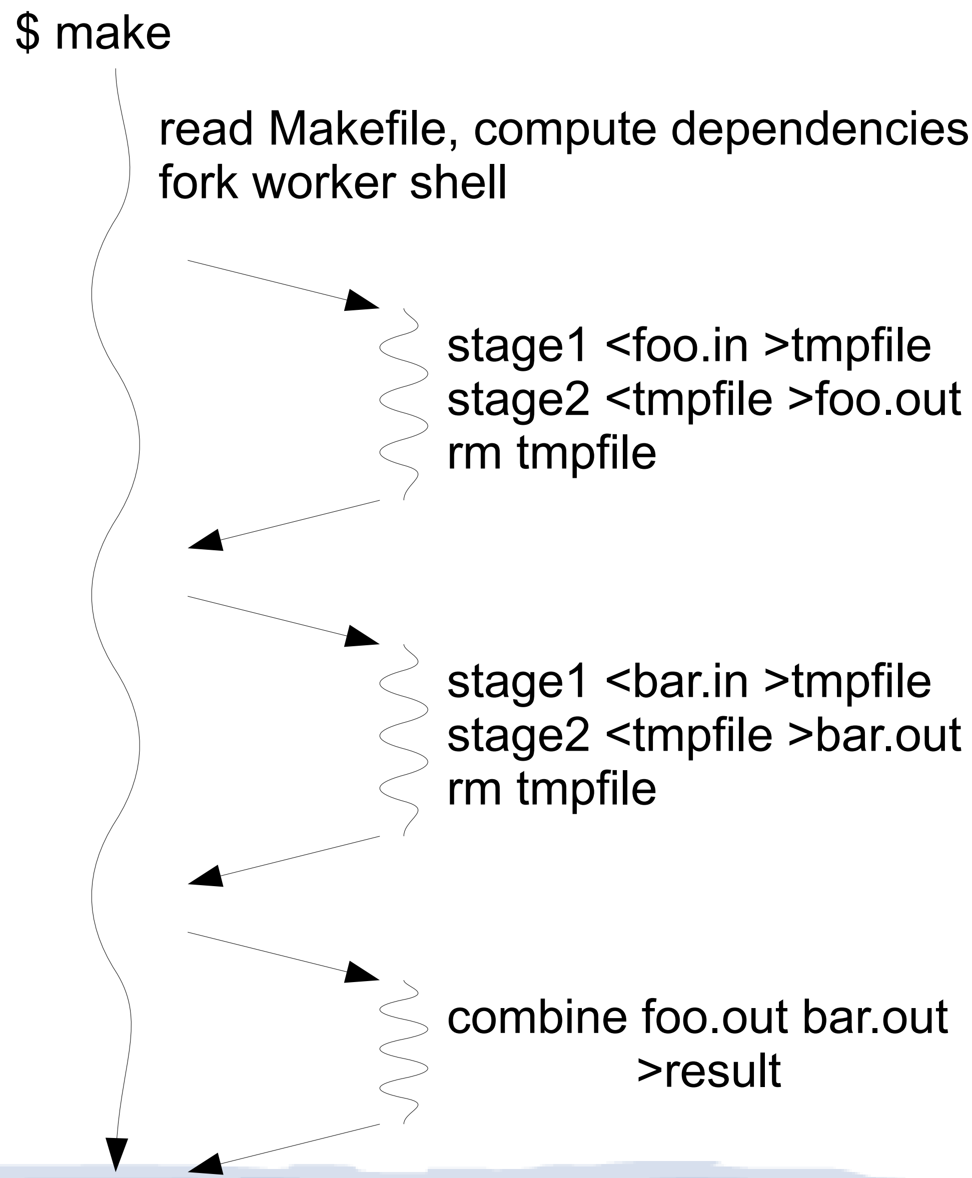
Each process has a *complete file system replica* in its address space

- a "distributed FS" w/ weak consistency
- **fork()** makes virtual copy
- **wait()** merges changes made by child processes
- merges at *file* rather than *byte* granularity



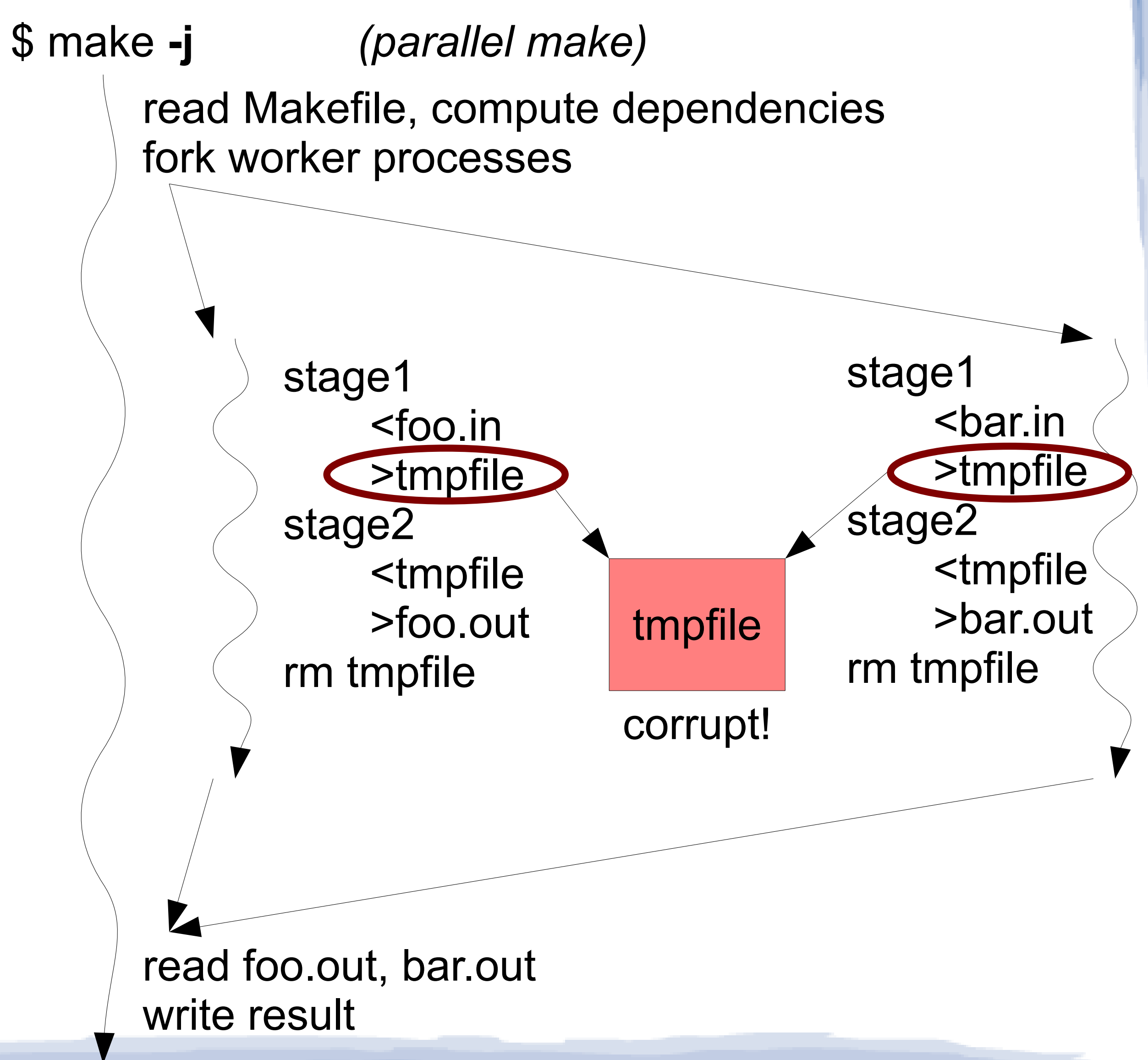
Example: Parallel Make/Scripts, Conventional Unix Processes

```
# Makefile for file 'result'

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

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



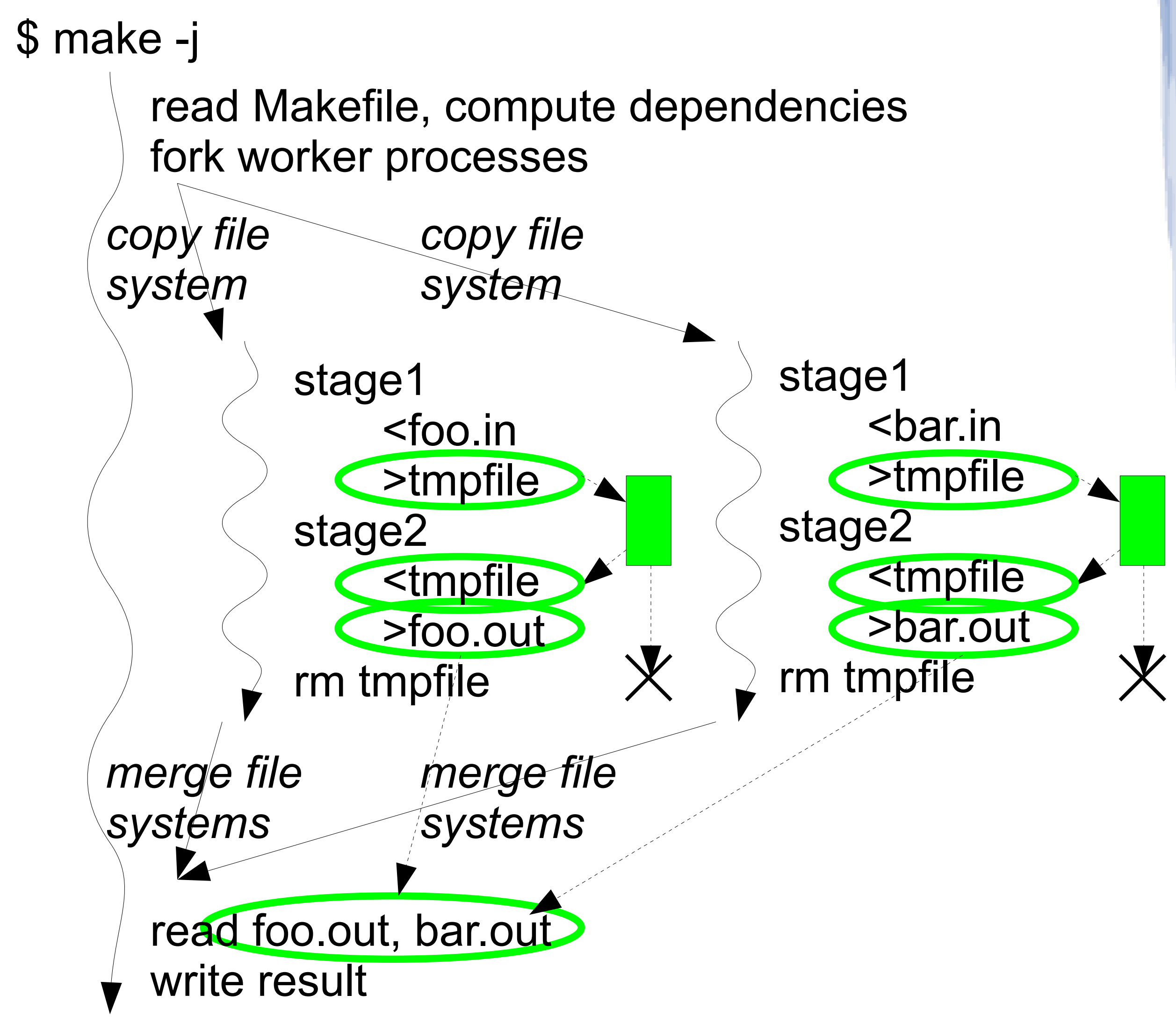
Example: Parallel Make/Scripts, Conventional Unix Processes

```
# Makefile for file 'result'

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

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



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

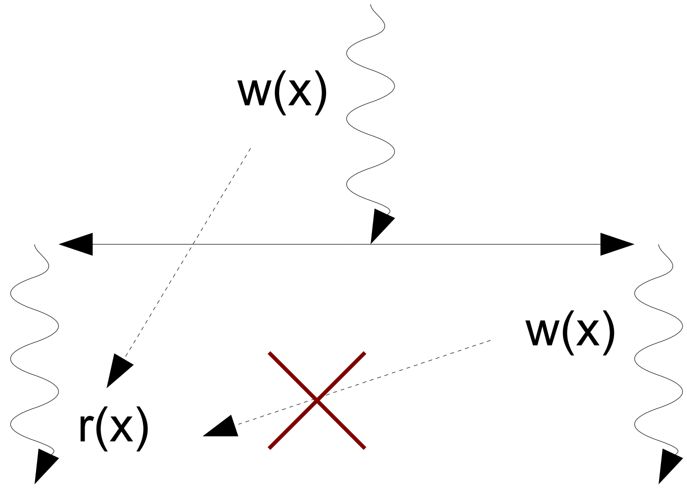
```
# Makefile for file 'result'

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

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



What Happened to the Races?

Read/Write races: go away *entirely*

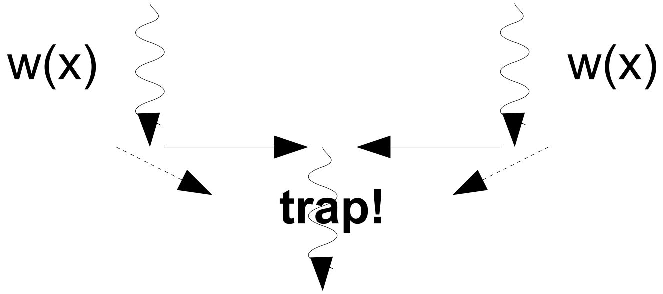
- writes propagate *only* via synchronization
- reads *always* see last write by *same* thread, else value at last synchronization point



What Happened to the 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

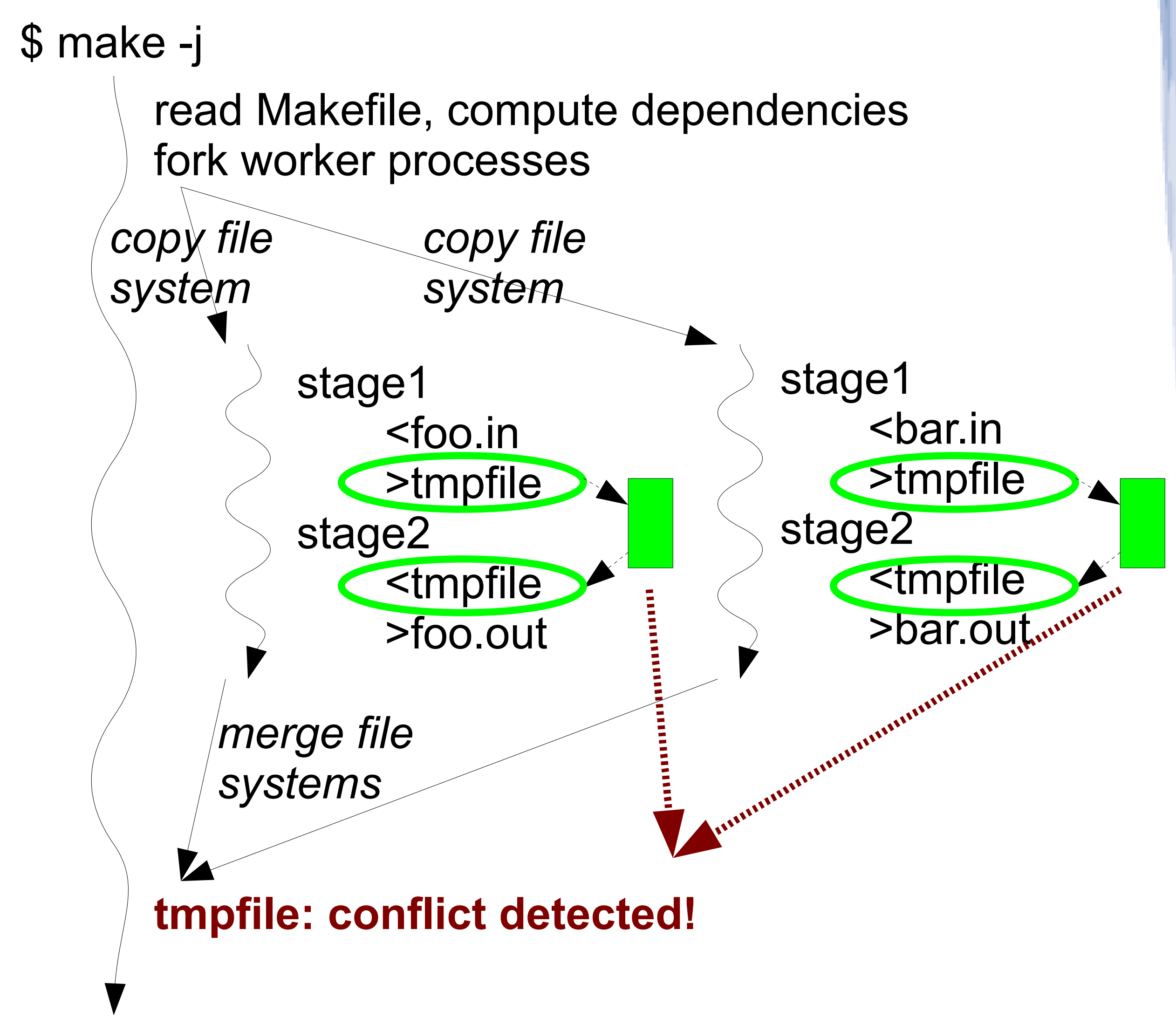


Example: Parallel Make/Scripts, **Determinator Processes**

Makefile for file 'result'

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

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



Talk Outline

- ✓ Introduction: Parallelism and Data Races
- ✓ Determinator Programming Model and Design
  - Deterministic “threads” and “shared memory”
  - Deterministic “processes” and “file systems”
- Challenges and Ongoing Work
  - New abstractions versus legacy compatibility
  - Performance and scalability
  - Deterministic distributed computing
- Conclusion

The “Pthreads Problem”

Mutex locks, condition variables, etc., have *fundamentally nondeterministic semantics*

- Lock order implicitly depends on “time” – is *not* specified by program logic
- Determinator runtime can “synthesize time” for backward compatibility with pthreads code
  - via deterministic scheduling, as in CoreDet
- But synthetic time is *still arbitrary!*
  - new inputs, new compiler, new options → new time schedule → **more heisenbugs**

Towards Deterministic Parallel APIs

To escape race-prone parallel programming, **must** wean ourselves from pthreads-like APIs!

Ongoing Determinator work is exploring:

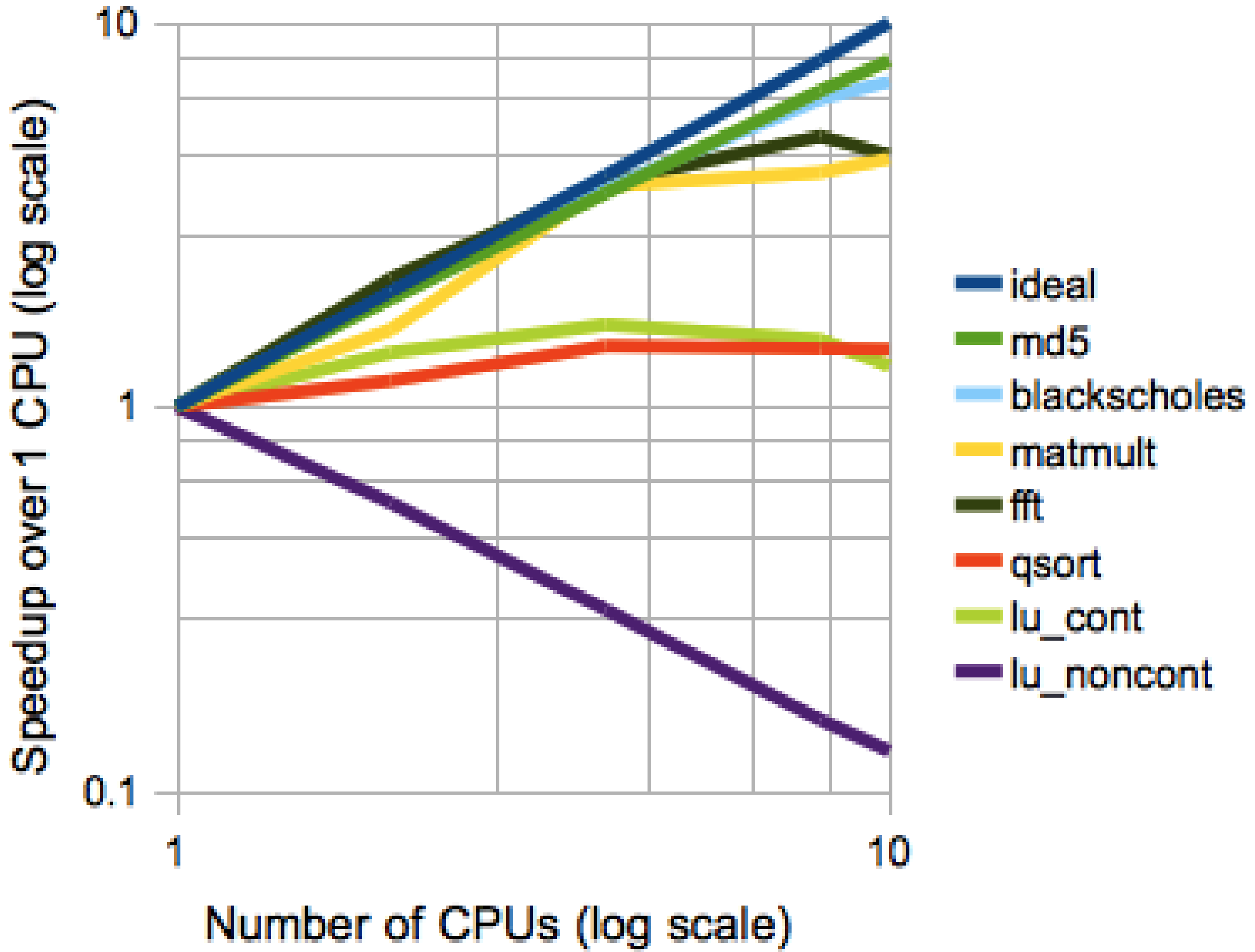
- General deterministic synchronization models
  - [Aviram/Ford/Zhang, WoDet '11]
- Deterministic OpenMP-style shared memory
  - [Aviram/Ford, HotPar '11]
- Deterministic MPI-style message passing
  - [Zhang/Ford, APsys '11]

Performance and Scalability

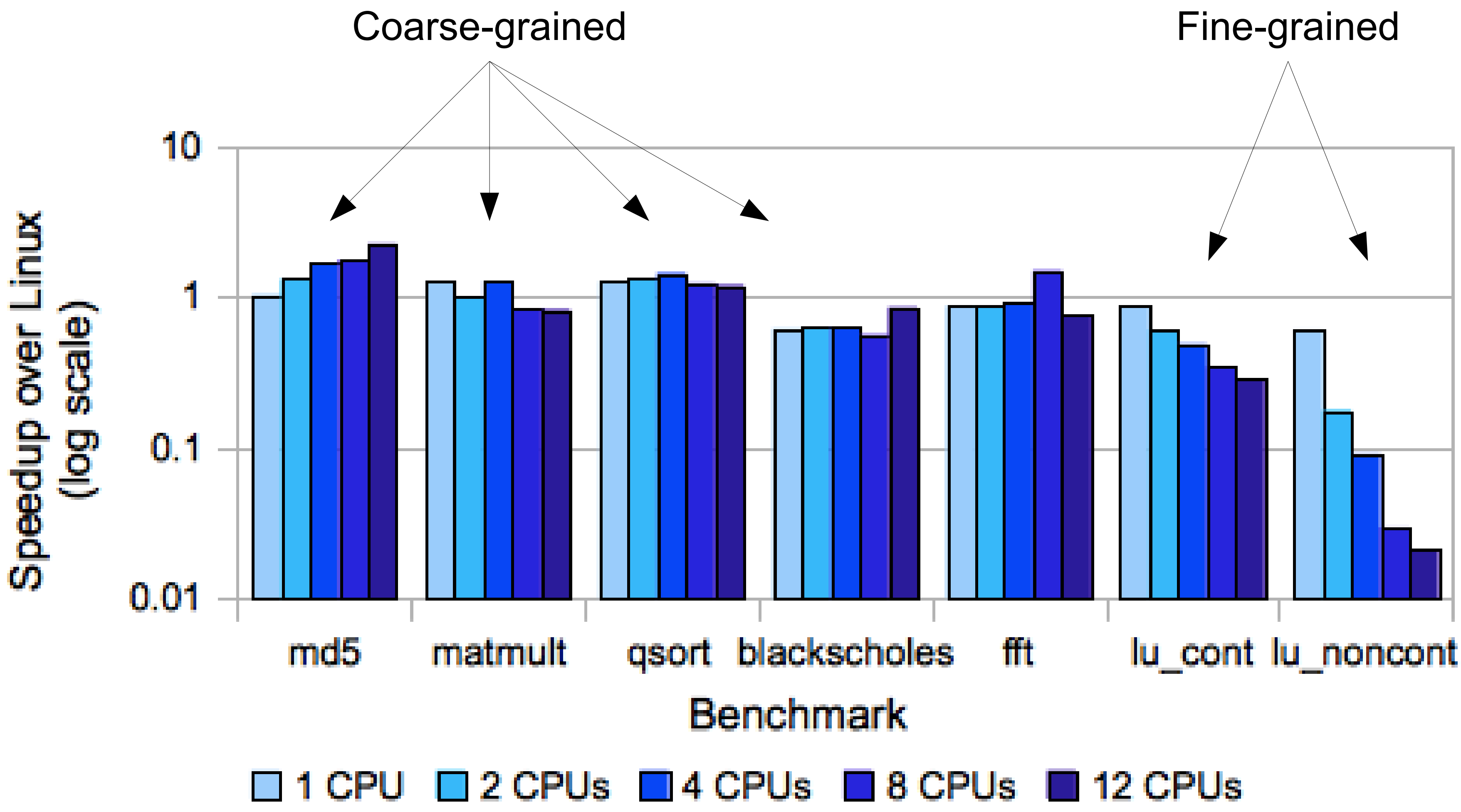
Question: can a Determinator-like model be efficient and scalable enough for everyday use?

Current answer: *it depends* (of course)

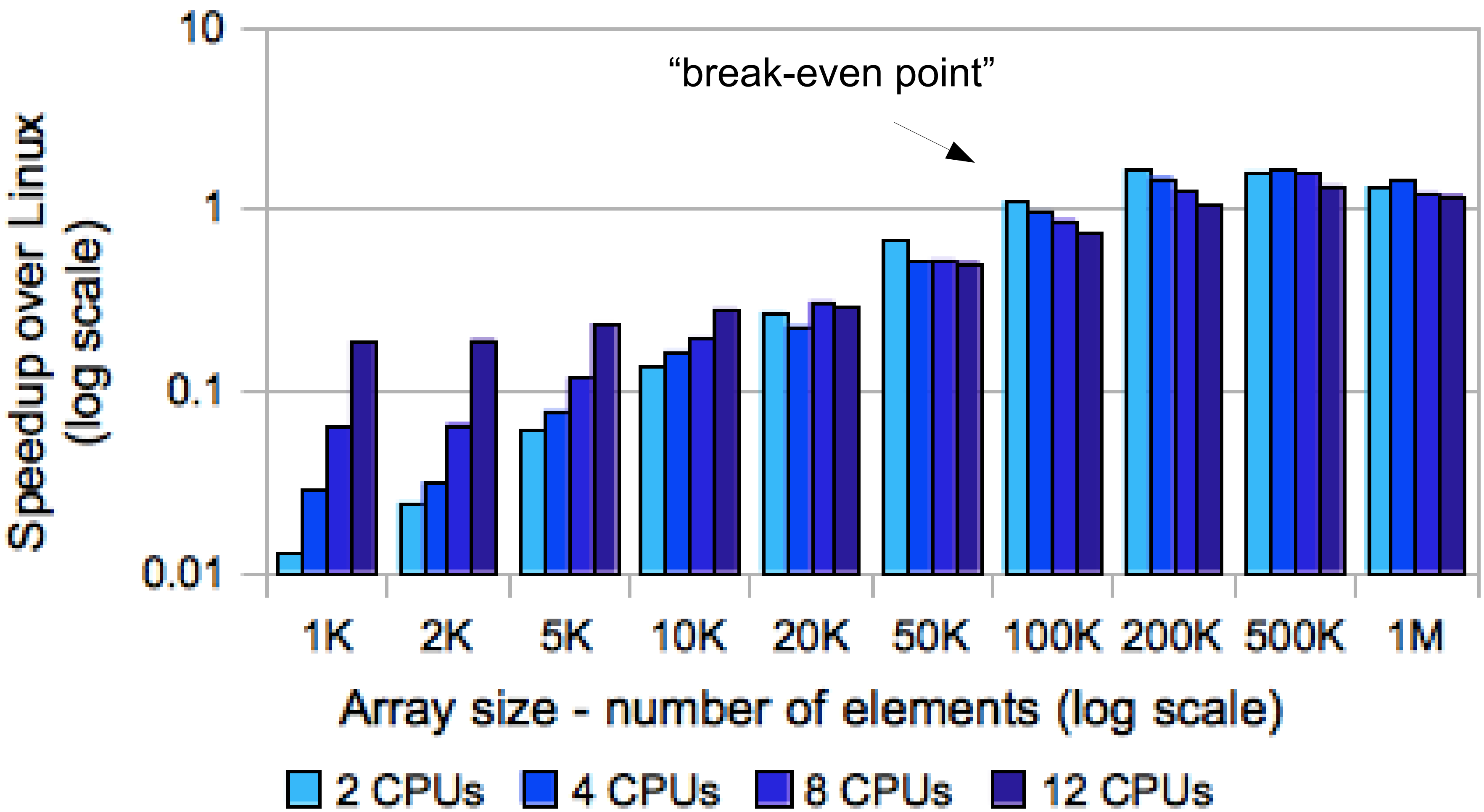
Single-Node Speedup over 1 CPU



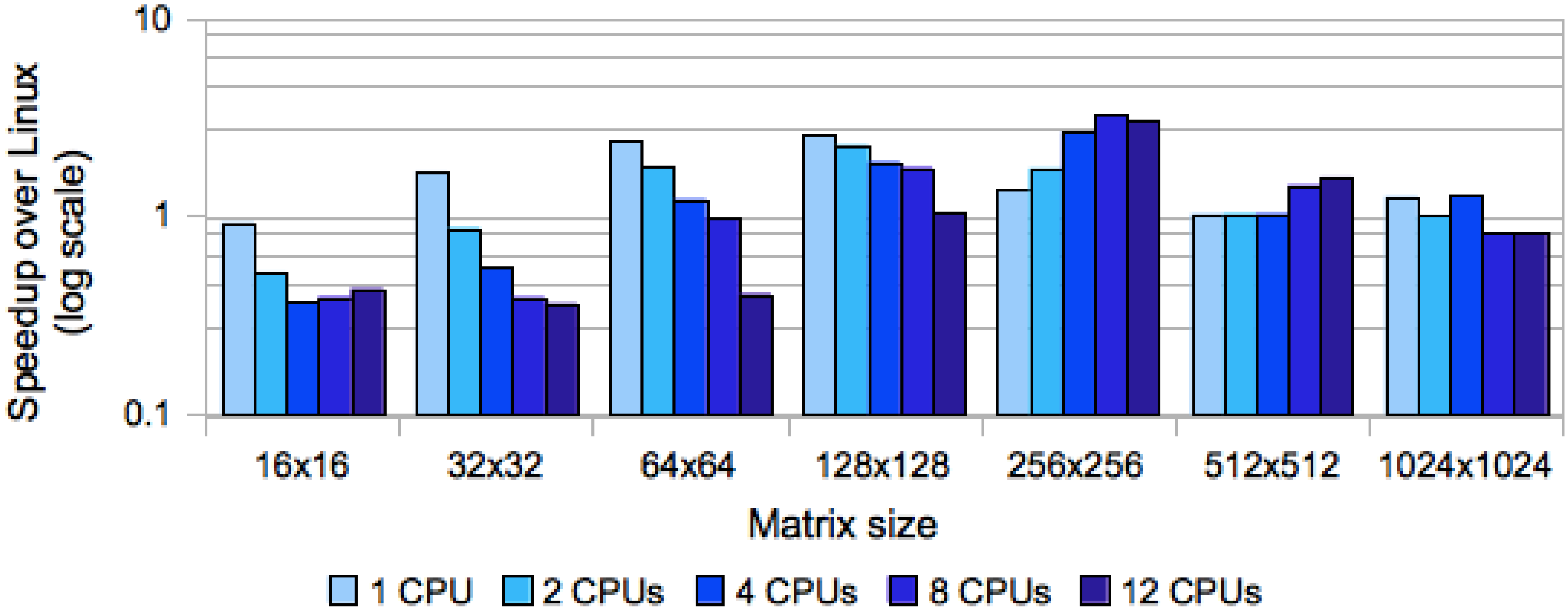
Single-Node Performance: Determinator versus Linux



Drilldown: Varying Granularity (Parallel Quicksort)



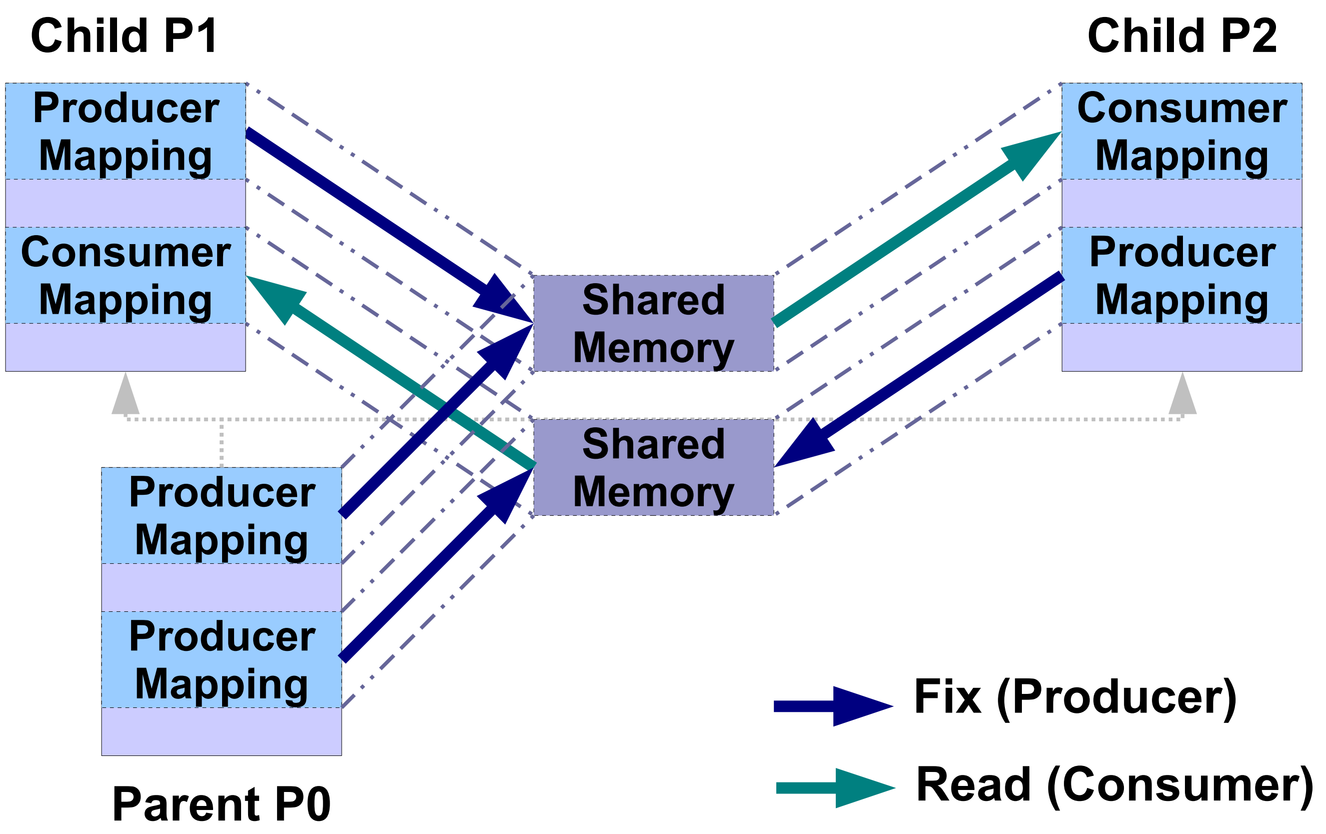
Drilldown: Varying Granularity (Parallel Matrix Multiply)



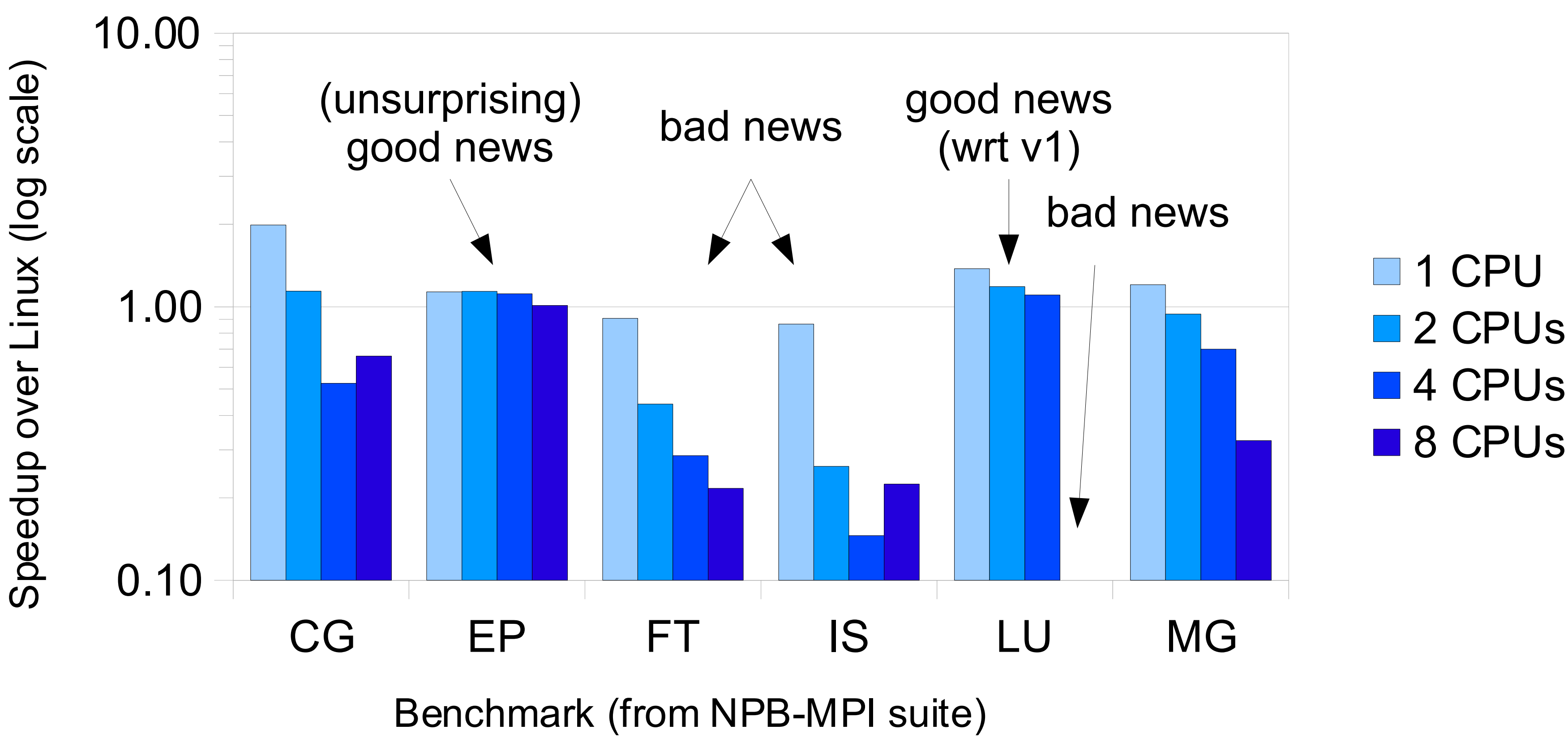
Improving Scalability with Producer/Consumer Virtual Memory

- In Determinator v1 [OSDI '10], threads could synchronize *only* via **common parent space**
  - hierarchy → fundamental scalability bottleneck
- In Determinator v2 (in-progress), threads can create peer-to-peer "**virtual memory pipes**"
  - single producer, multiple consumer (SPMC)
  - emulate unicast, broadcast communication

Producer/Consumer Virtual Memory



Determinator v2 (preliminary) Performance Relative to Linux



Distributed Determinism?

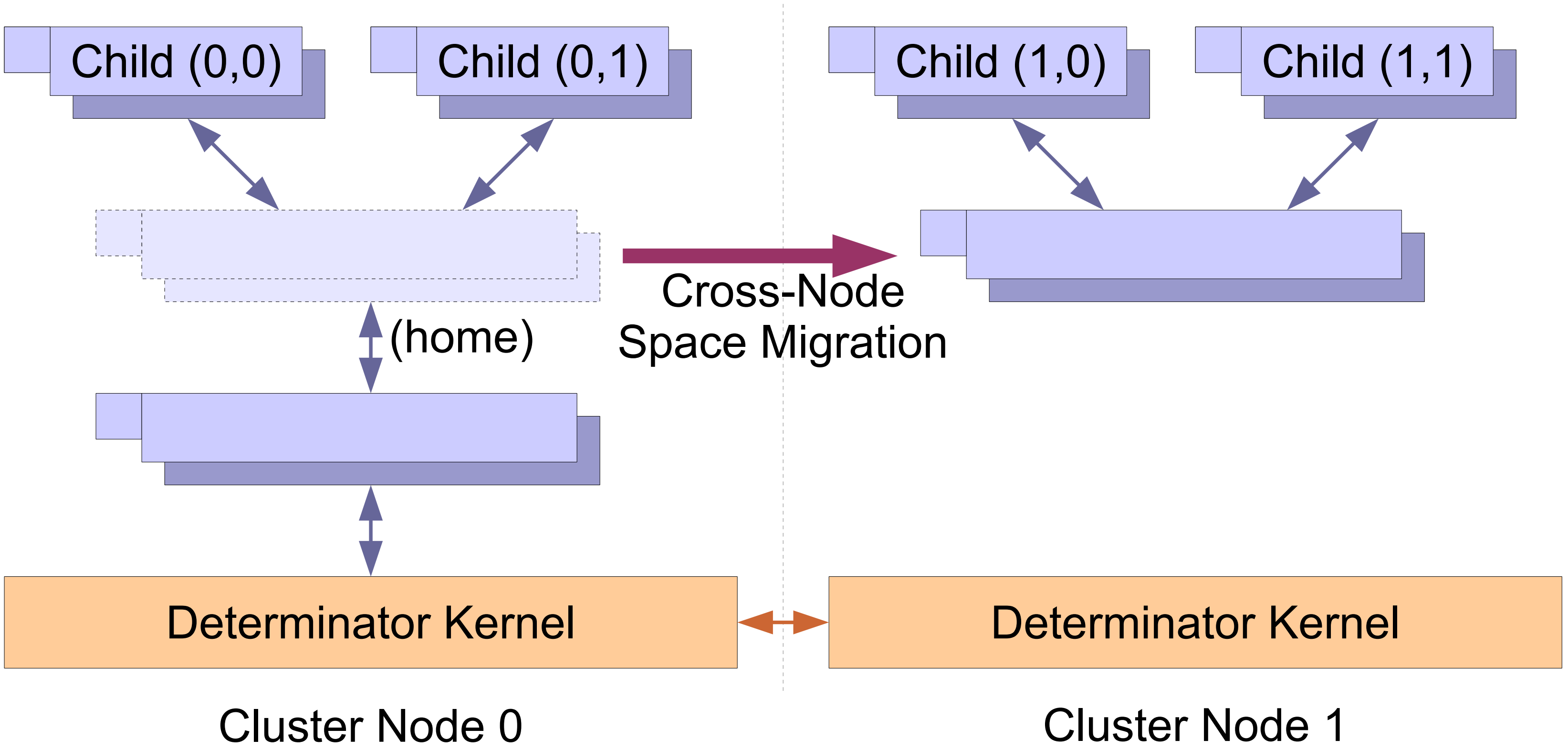
Tantalizing potential...

- Time-travel-debug 1000-node data analysis or scientific computations
- Replay-based intrusion analysis/response in large distributed systems
- New solutions to timing channels in the cloud [Aviram/Hu/Ford/Gummadi, CCSW '10]

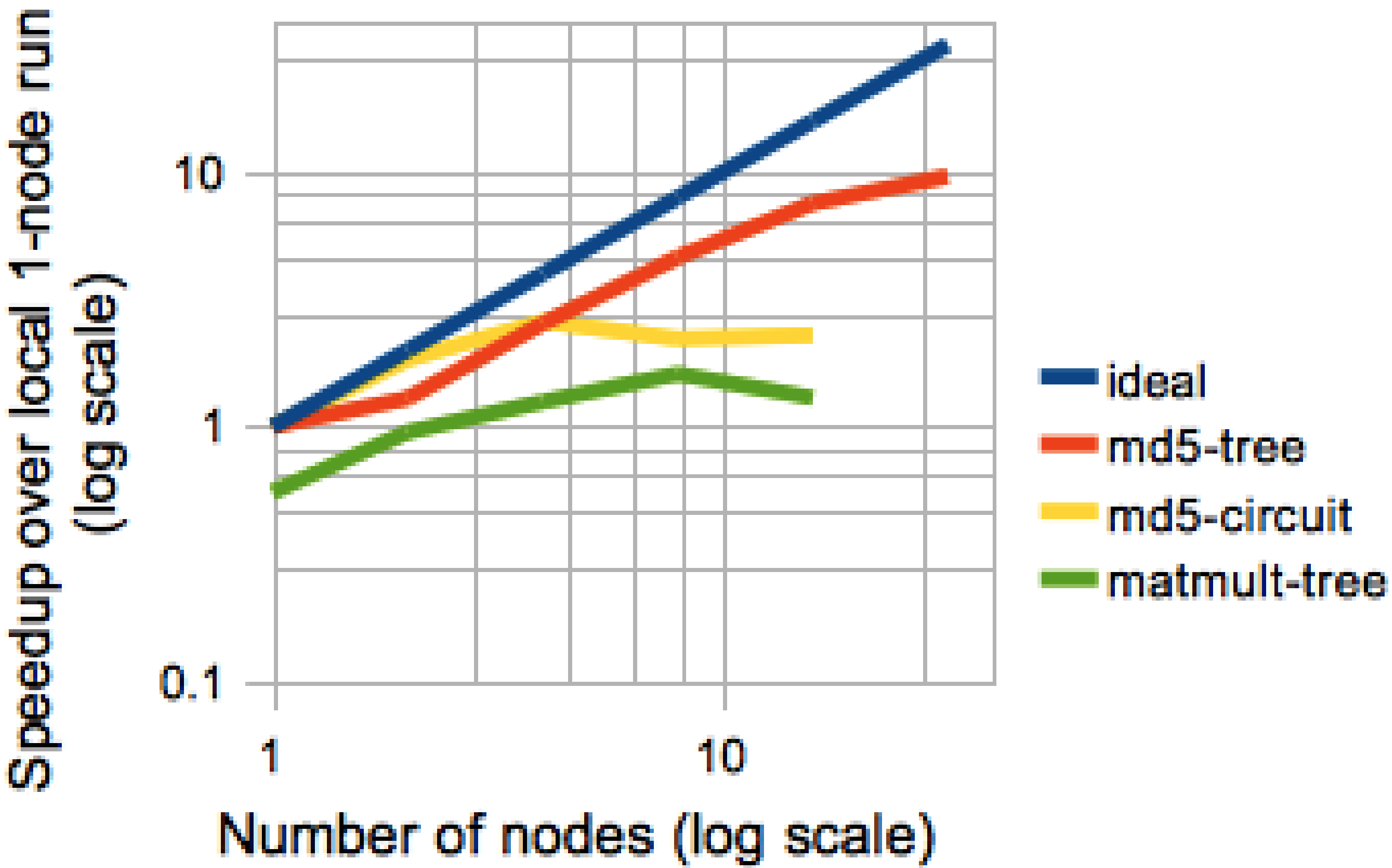
But is it practical?

A Proof-of-Concept Approach

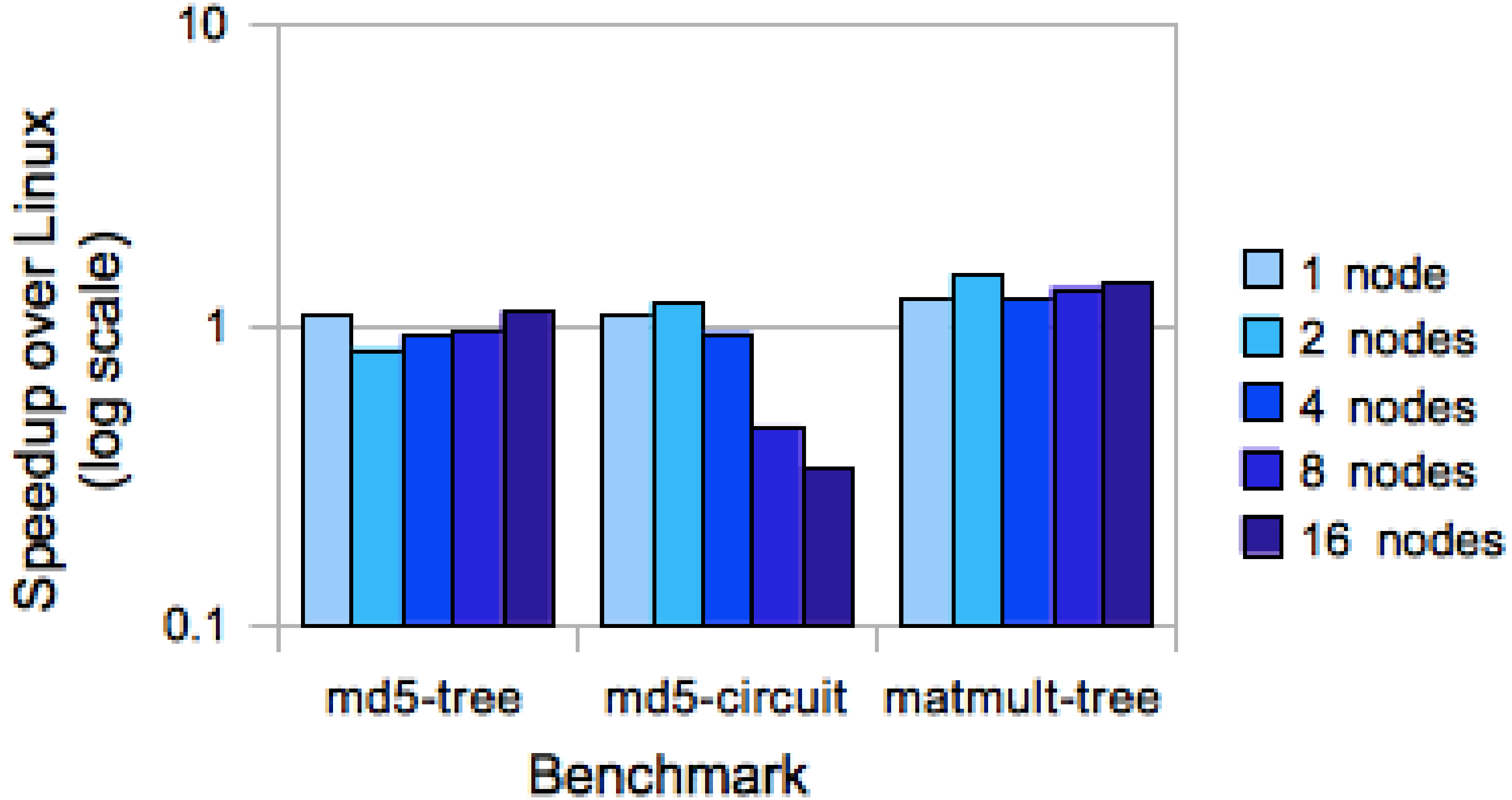
Transparent process migration among nodes



Distributed Speedup over 1 Node



Distributed Performance vs Linux



Conclusion

Determinator offers and explores a *race free, deterministic parallel programming model*

- Avoids races via *private workspace model*
- Supports existing languages
- Thread- and process-level parallelism

Many open questions for future work

- The “right” parallel abstractions?
- Can it be made efficient enough? Distributed?

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