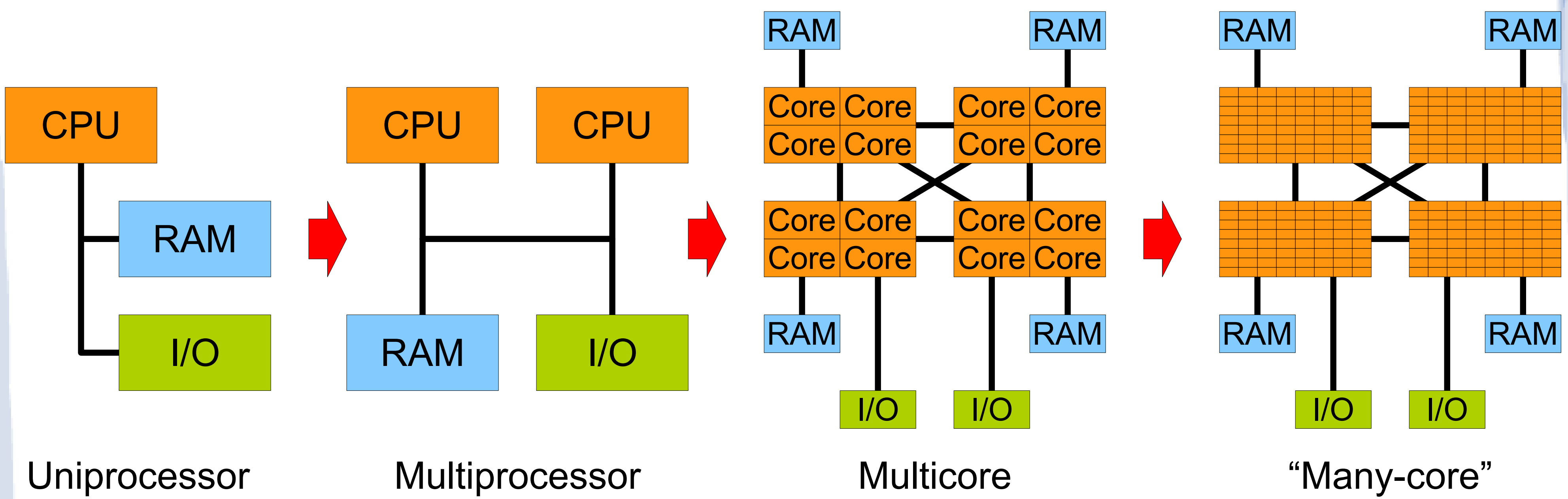
System-Enforced Determinism: What it Is, How Practical Is It, and What's It Good For?

Bryan Ford
Amittai Aviram, Weiyi Wu, Yu Zhang,
Bandan Das, Shu-Chun Weng, Sen Hu

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

University of Texas at Austin – Nov 15, 2012

Pervasive Parallelism



Industry shifting from “faster” to “wider” CPUs

Today's Grand Software Challenge

Parallelism makes everything harder.

- **Nondeterministic programming models**
  - Synchronization, concurrency challenges
- **Creates pervasive risks of data races**
  - Leads to “once-in-a-million runs” *heisenbugs*
- **Undermines execution repeatability**
  - Needed in fault tolerance, debugging, ...
- **Unintentionally leaks information**
  - Timing side-channels, IDS-evading malware

Does
Pervasive Parallelism
imply
Pervasive Nondeterminism?

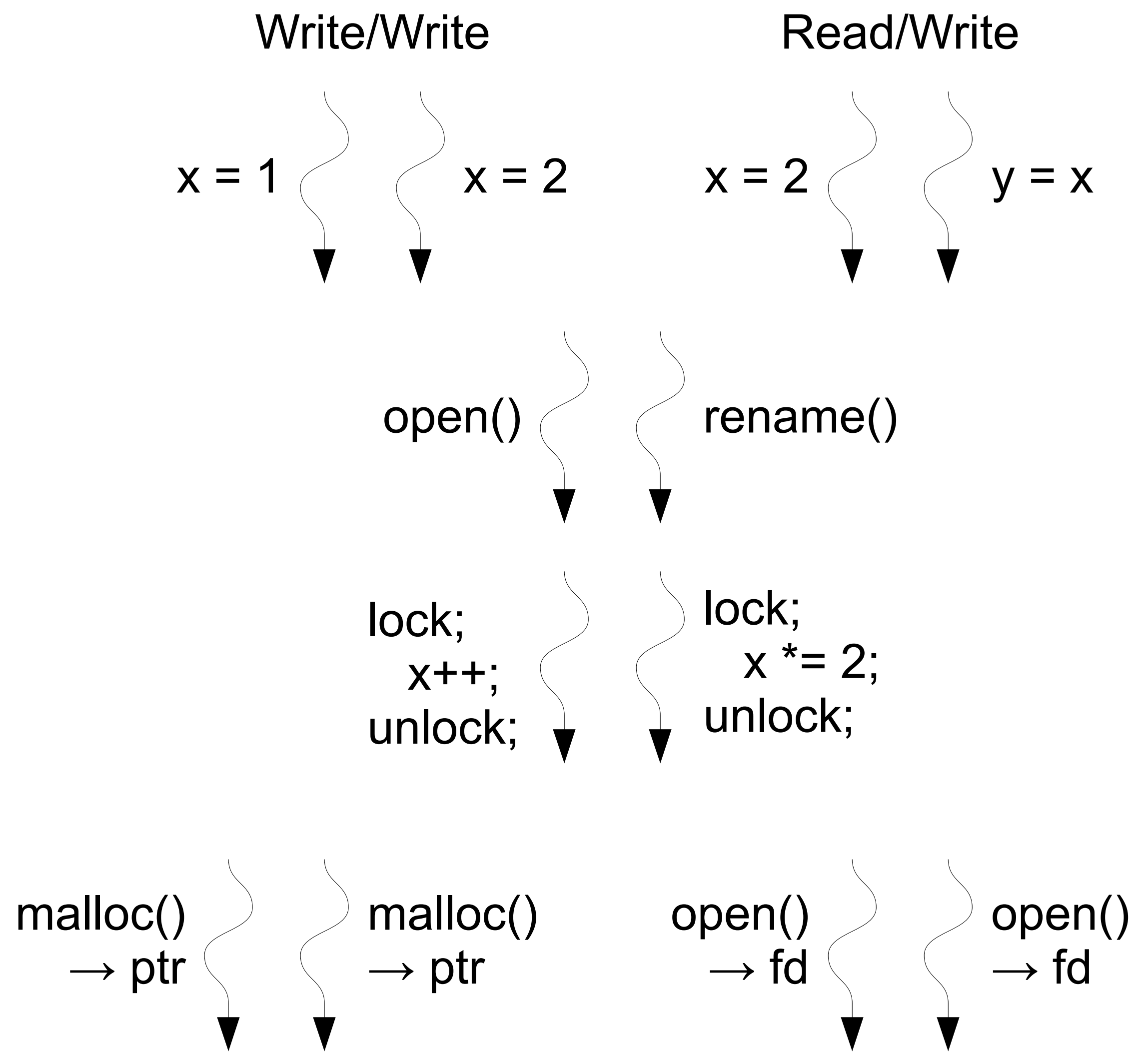
Not necessarily...

Talk Outline

- Introduction: Parallelism and Data Races
- Determinator: a Determinism-Enforcing OS
- Is Determinism *Efficient, General, Usable*?
- Why *System-Enforced* Determinism?
- Conclusion

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, …)

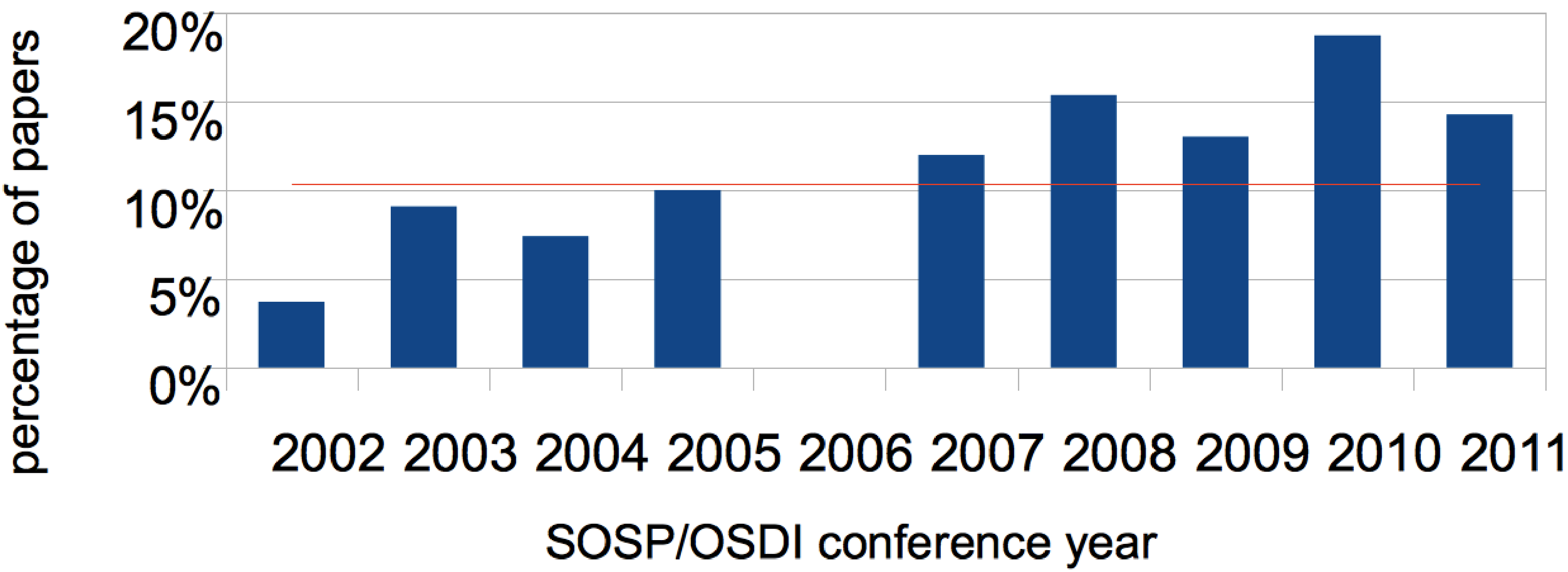
- Analyze/instrument program to help find races

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?**

Talk Outline

- ✔ Introduction: Parallelism and Data Races
- **Determinator: a Determinism-Enforcing OS**
- Is Determinism *Efficient, General, Usable*?
- Why *System-Enforced* Determinism?
- Conclusion

Introducing **Determinator**

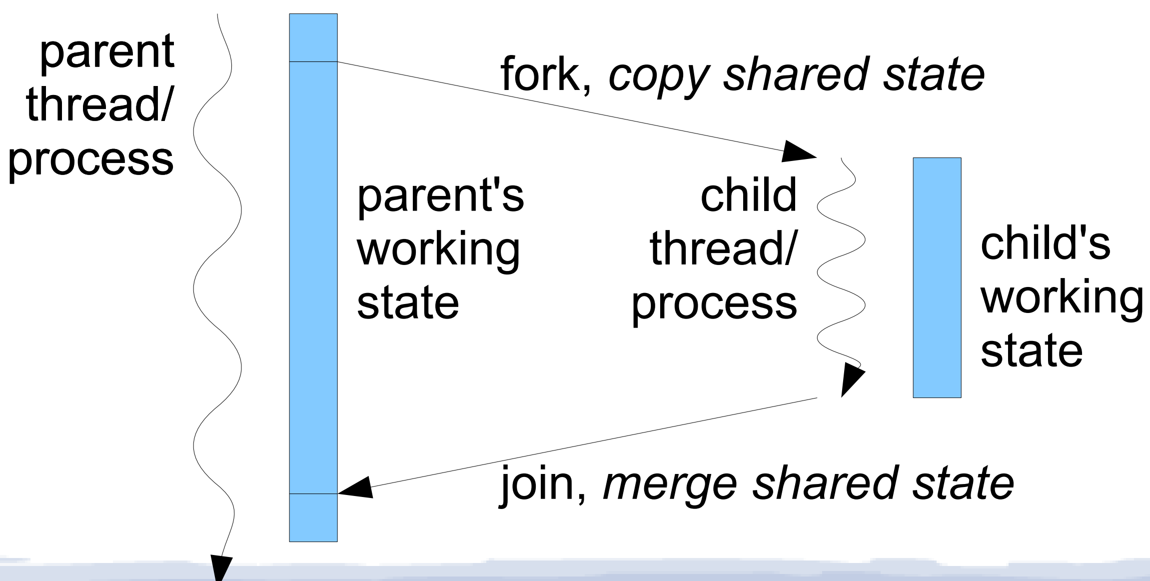
New OS offering a *race-free parallel environment*

- 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
  - But we can support interactive apps too

Determinator's Parallel 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
  - Is “weakness” a bug or a feature?

What does this mean in an OS?

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

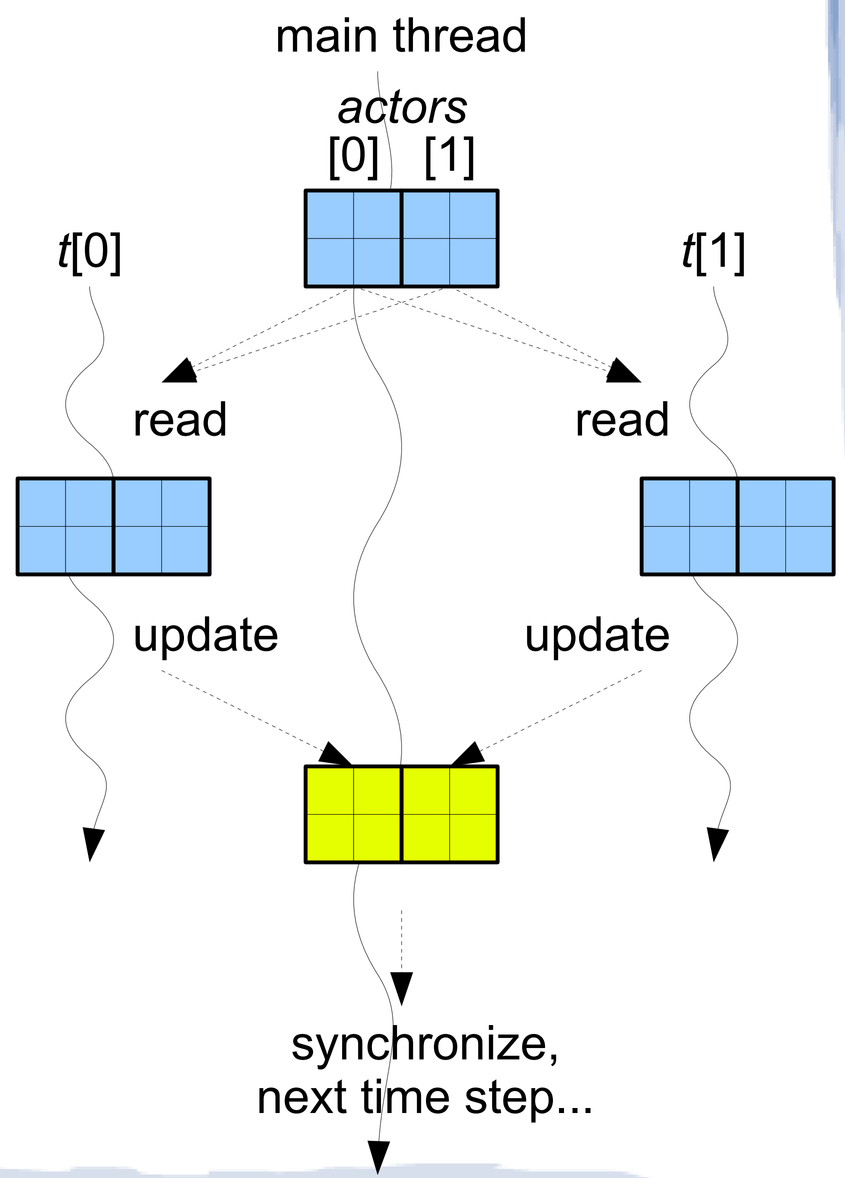
- **Threads** and **shared memory**
- **Processes** and shared **file systems**

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

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

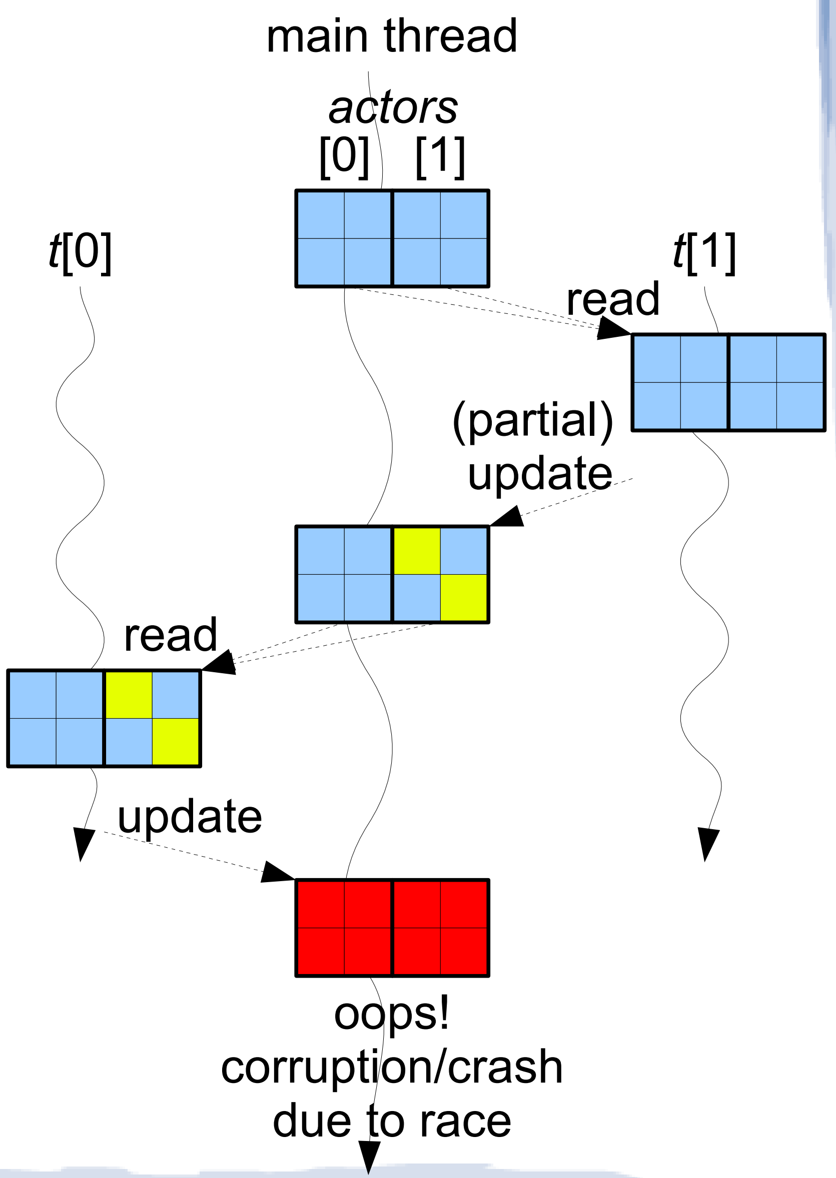
Example: Gaming/Simulation, Conventional Threads

```
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]);
        }
}
```



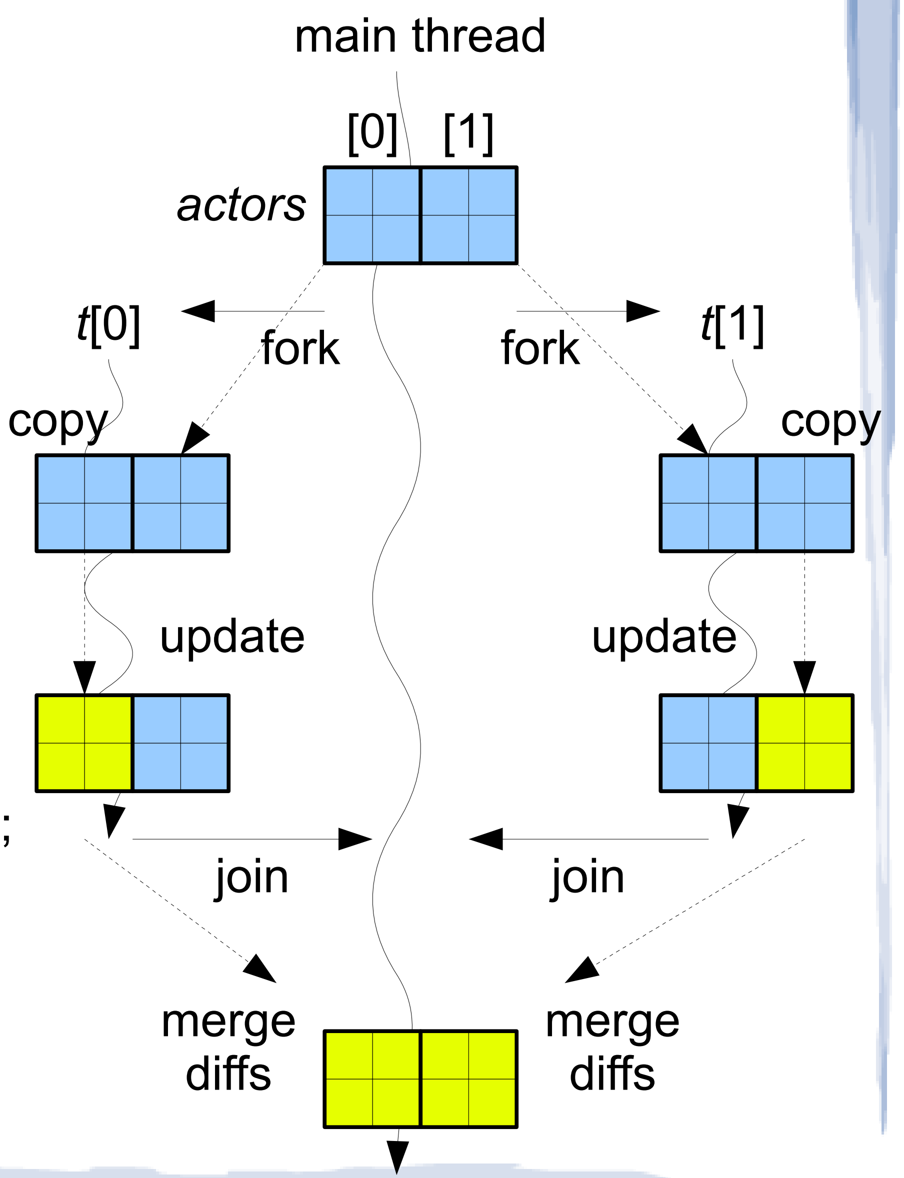
Example: Gaming/Simulation, Conventional Threads

```
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

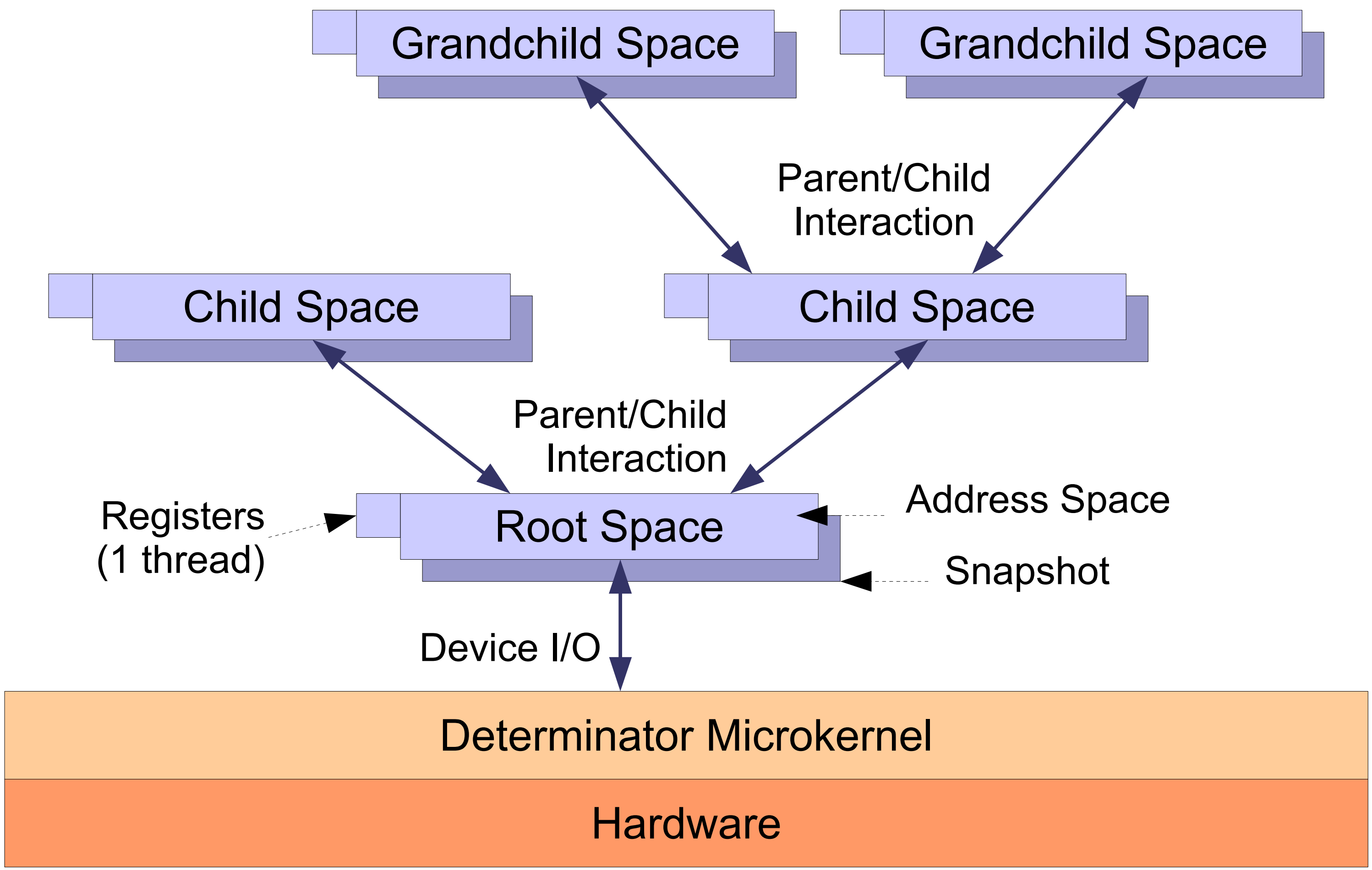
Details: [Aviram/Ford/Zhang, WoDet '11]

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

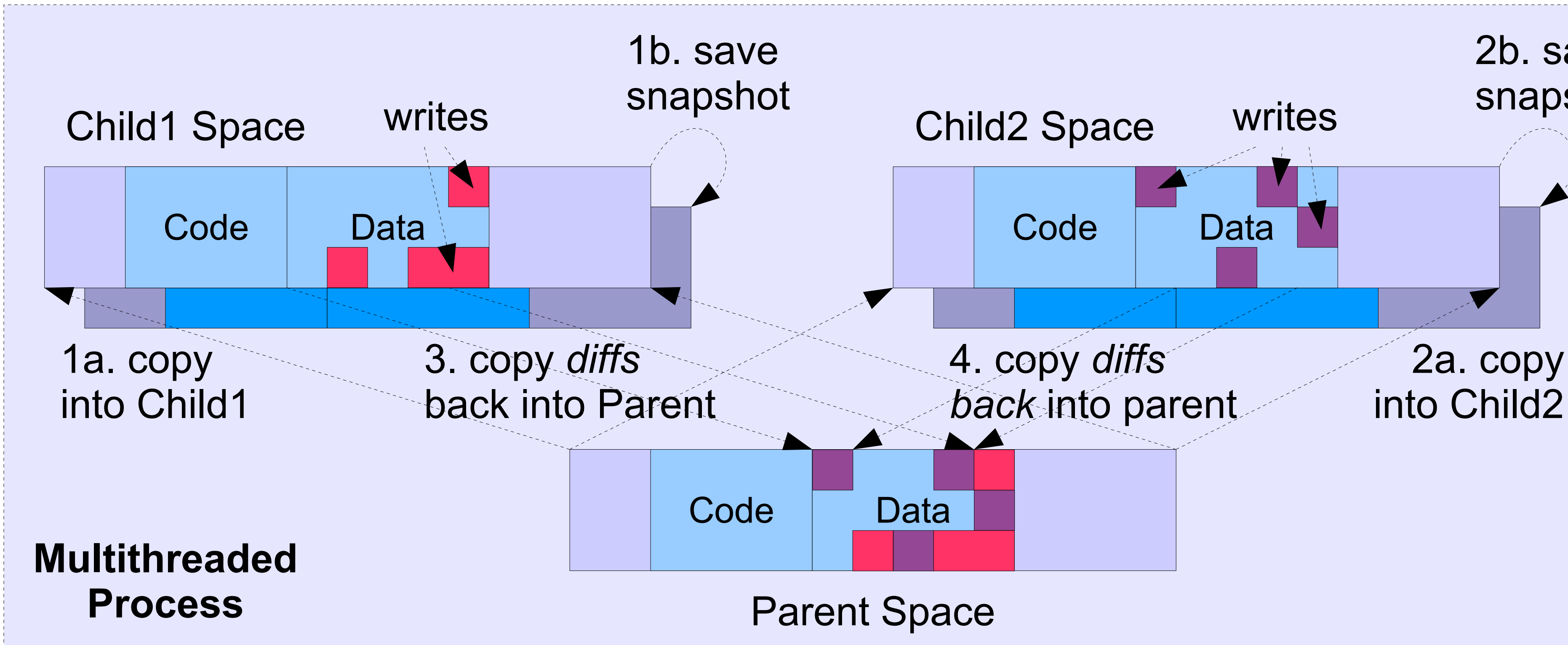


Threads, Determinator Style

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

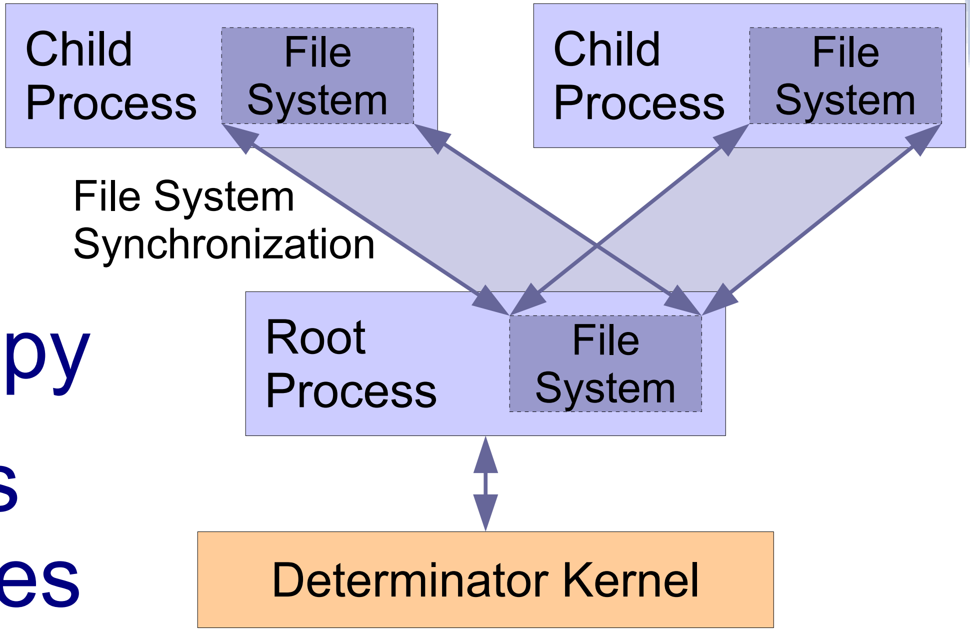
Multi-granularity **virtual diff & merge**

- 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

File Systems in Determinator

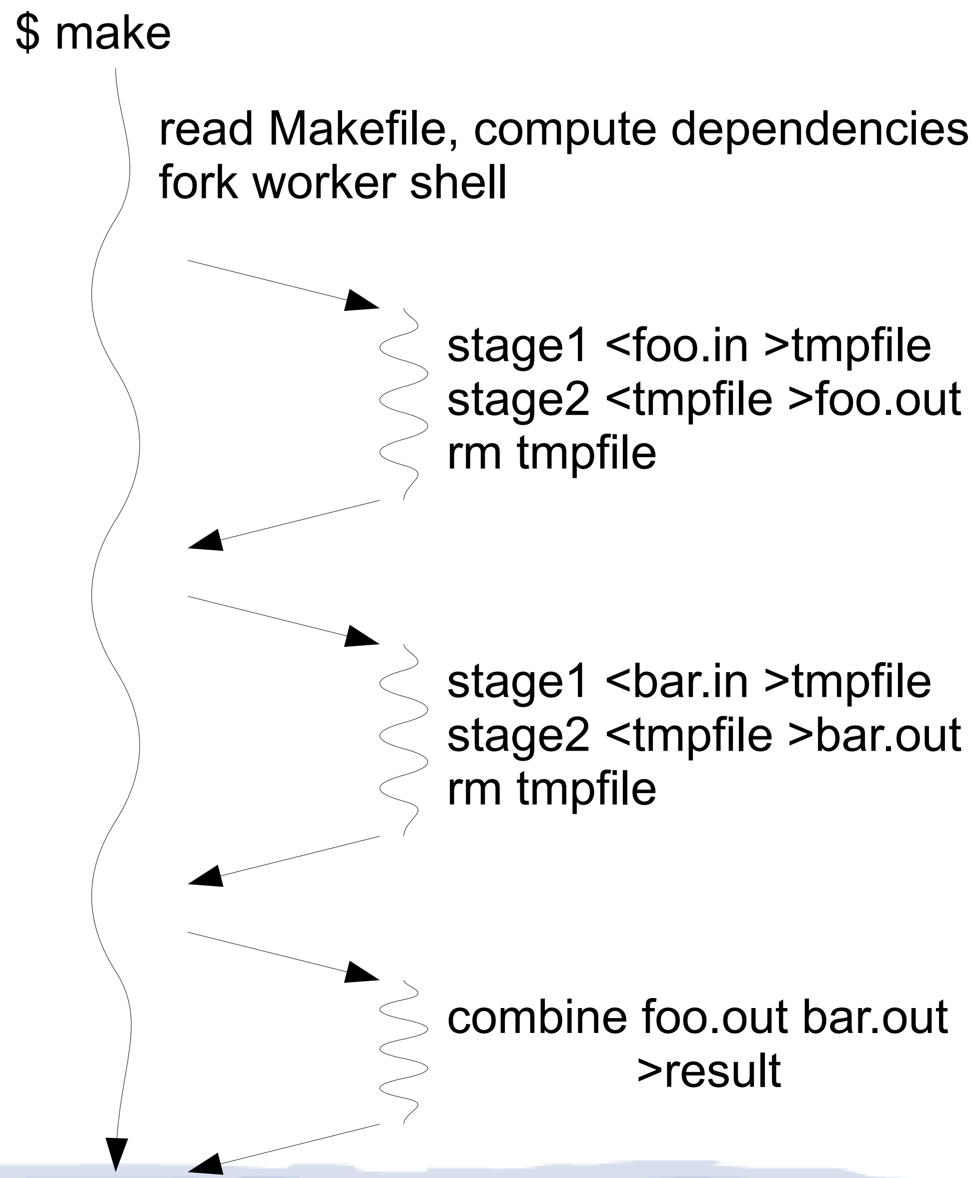
Each process has a *complete file system replica* in its address space

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



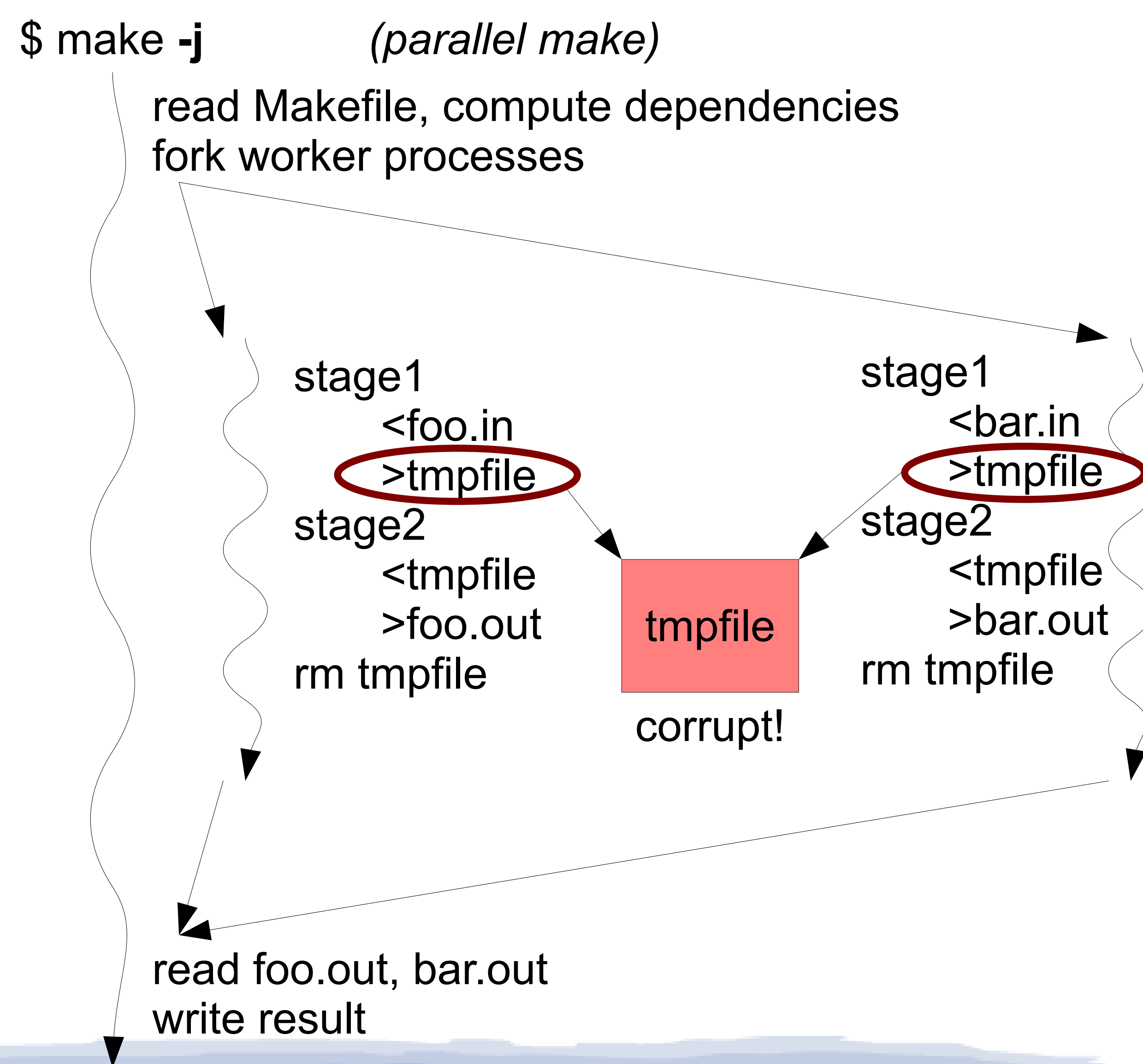
Example: Parallel Make/Scripts, Conventional Unix Processes

```
# Makefile for file 'result'

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

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



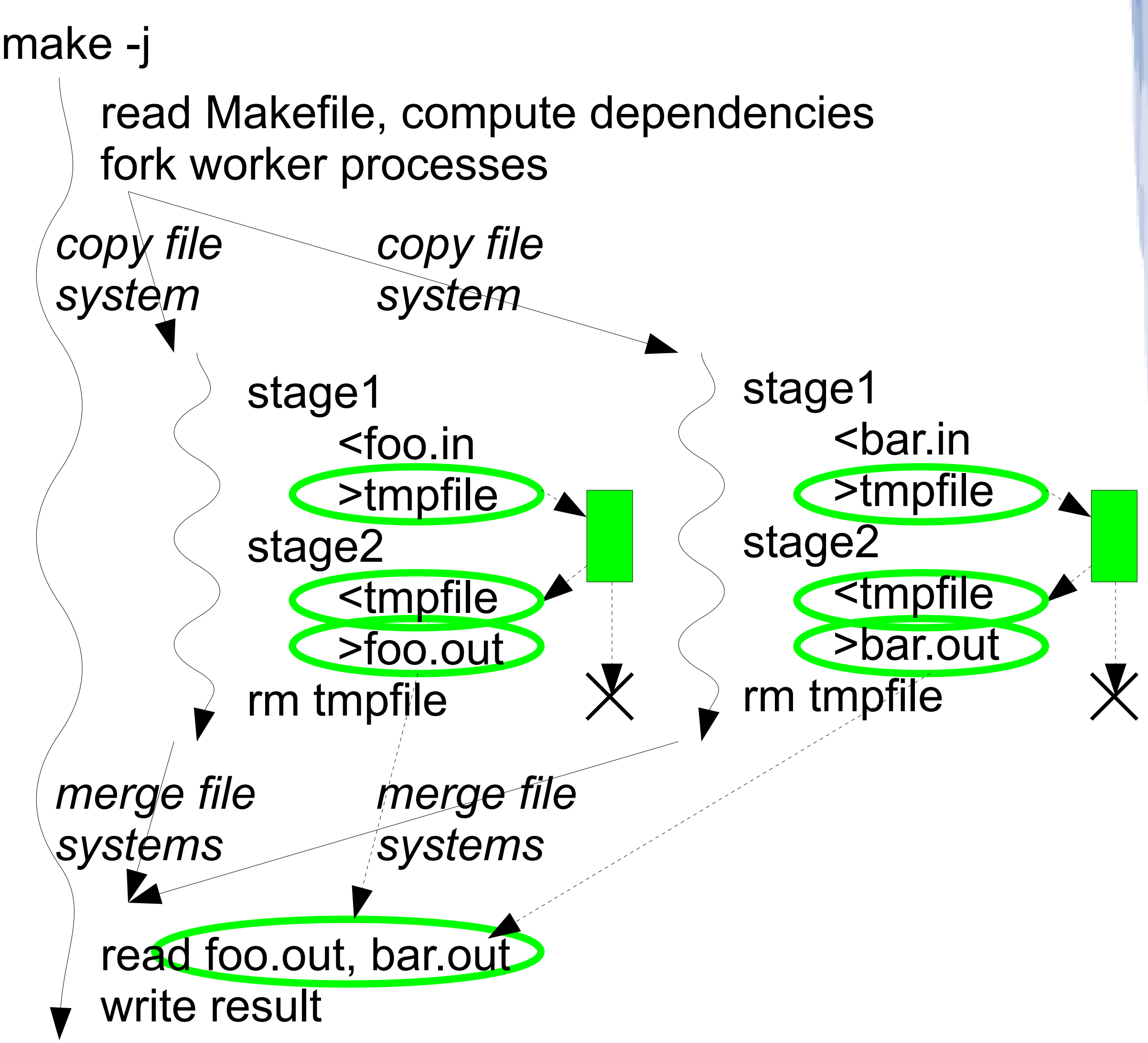
Example: Parallel Make/Scripts, Conventional Unix Processes

```
# Makefile for file 'result'

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

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



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

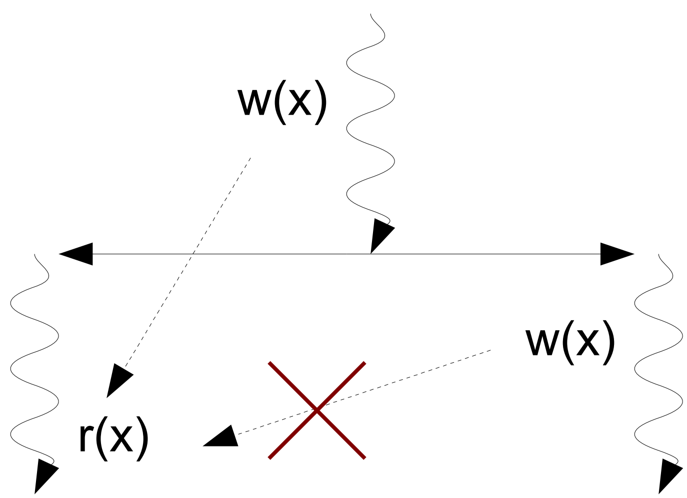
```
# Makefile for file 'result'

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

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



What Happened to Races?

Read/Write races: no longer possible

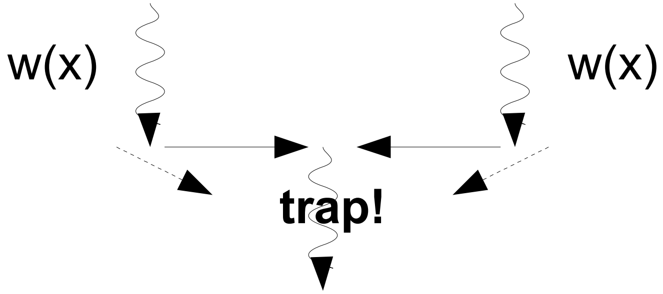
- writes propagate *only* via synchronization
- reads *always* see last write by *same* thread, else value at last synchronization point



What Happened to 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

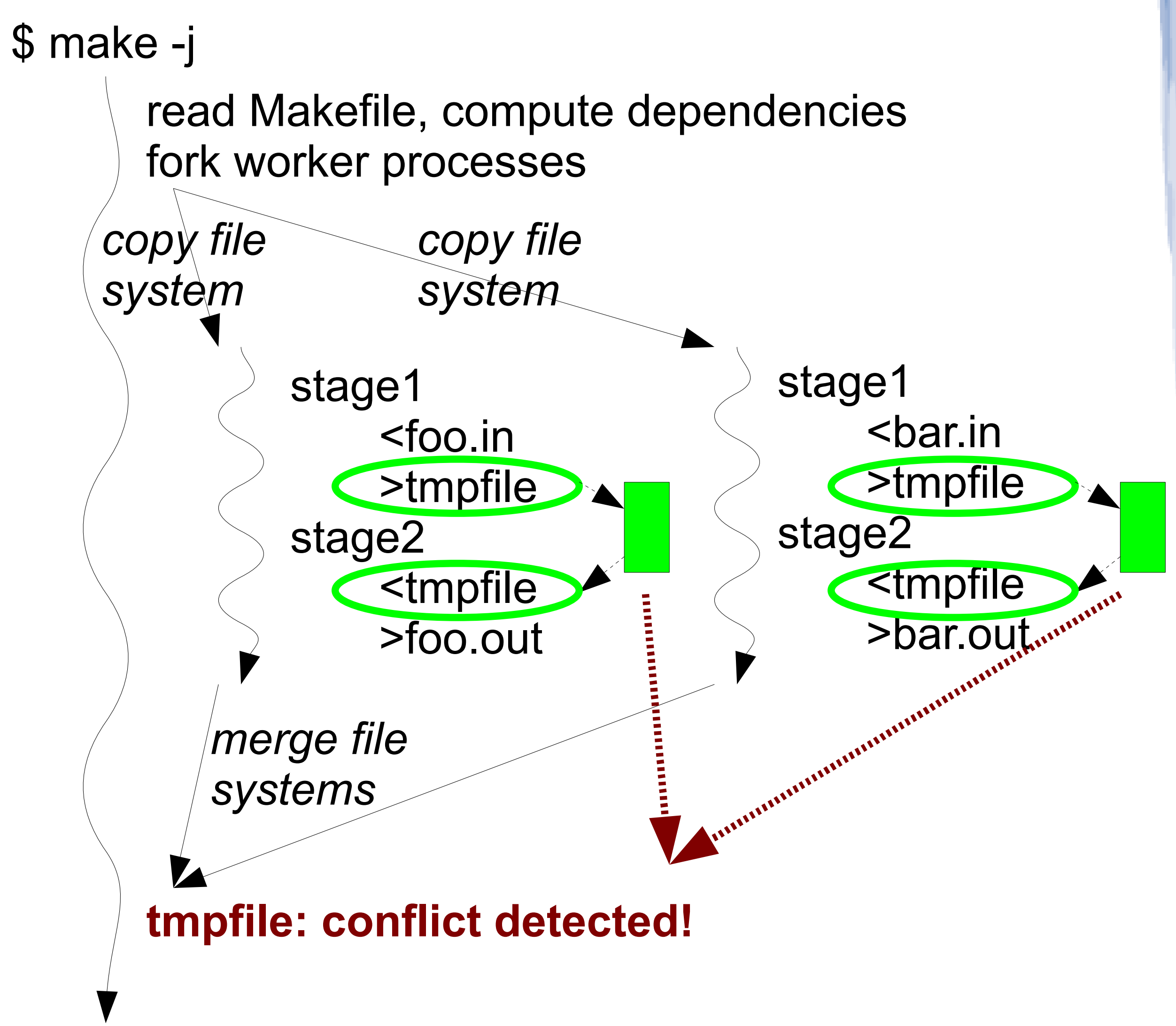


Example: Parallel Make/Scripts, **Determinator Processes**

\# 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: a Determinism-Enforcing OS
- **Is Determinism *Efficient, General, Usable*?**
- Why *System-Enforced* Determinism?
- Conclusion

Is it Efficient, General, Usable?

Can we...

- Make it **efficient enough for everyday use?**
- Support **non-hierarchical synchronization?**
- Run **nondeterministic pthreads-style code?**
- Make it **accessible to ordinary developers?**
- Support **distributed execution?**

Yes we can! *(we think)*

Determinator Performance

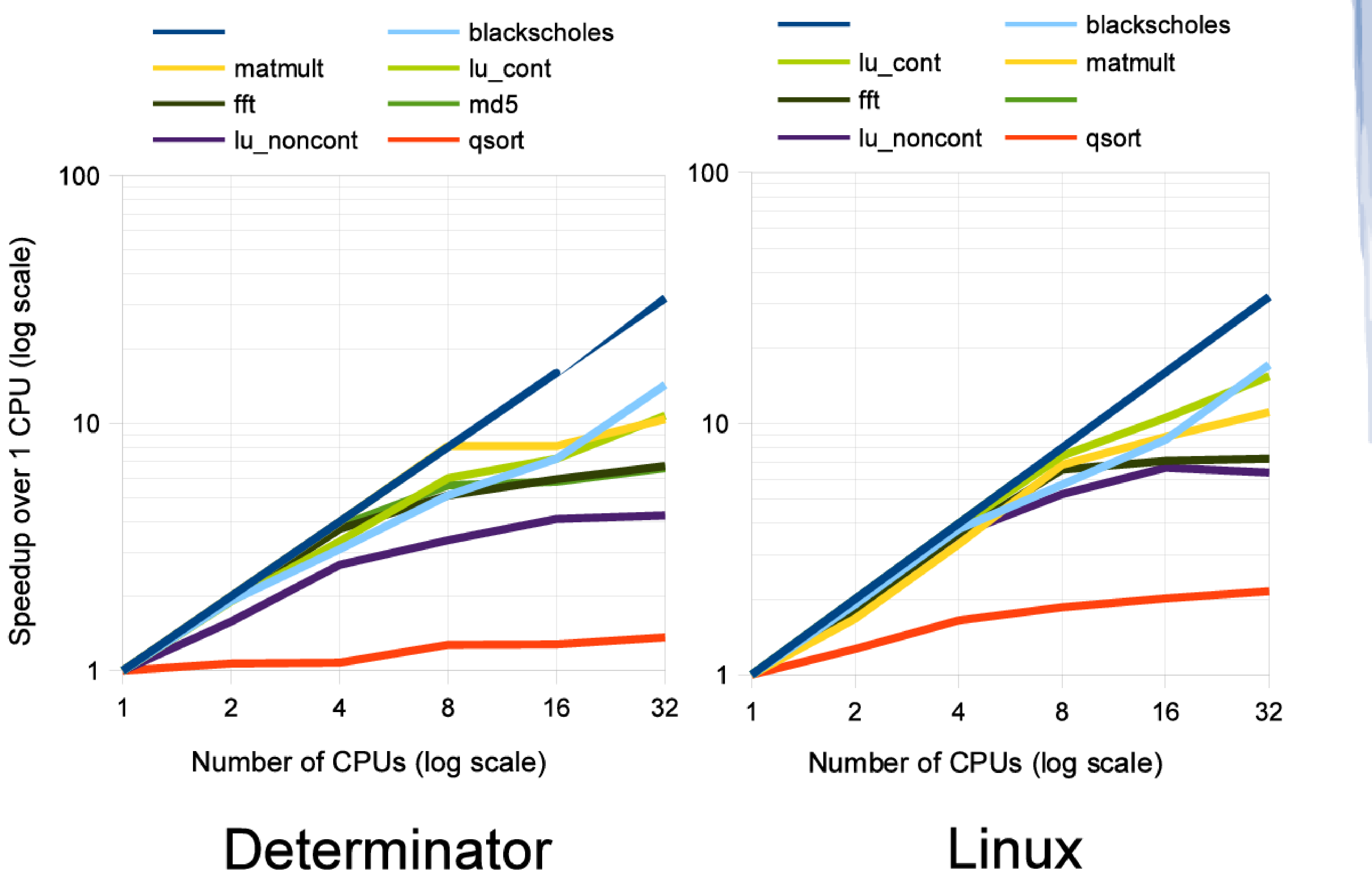
Determinator v1 for 32-bit x86 evaluated in:

- "Efficient System-Enforced Deterministic Parallelism", OSDI 2010 – Best Paper Award

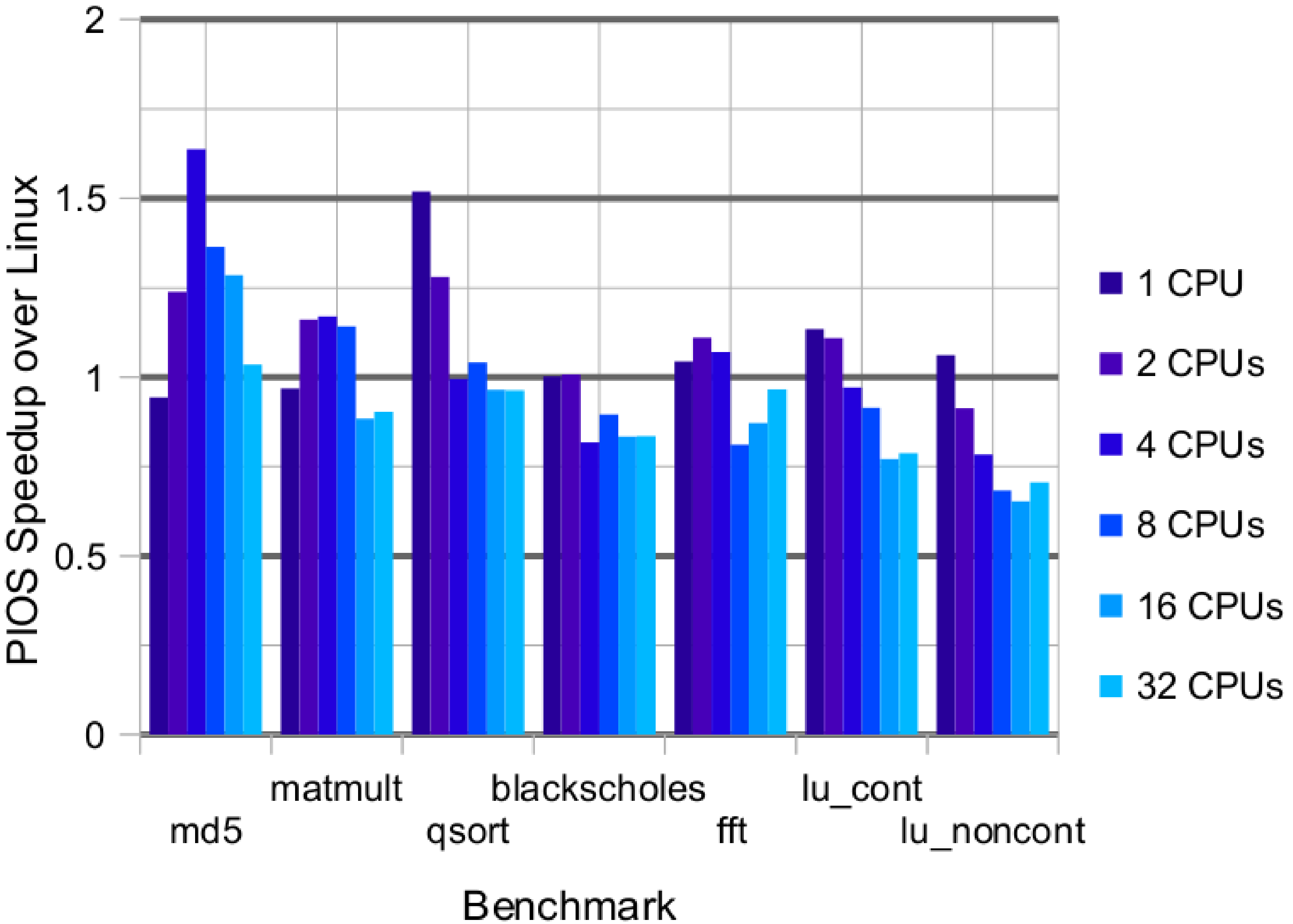
Determinator v2 for 64-bit x86 now working:

- Larger address spaces for larger benchmarks, utilize more CPU cores efficiently, ...

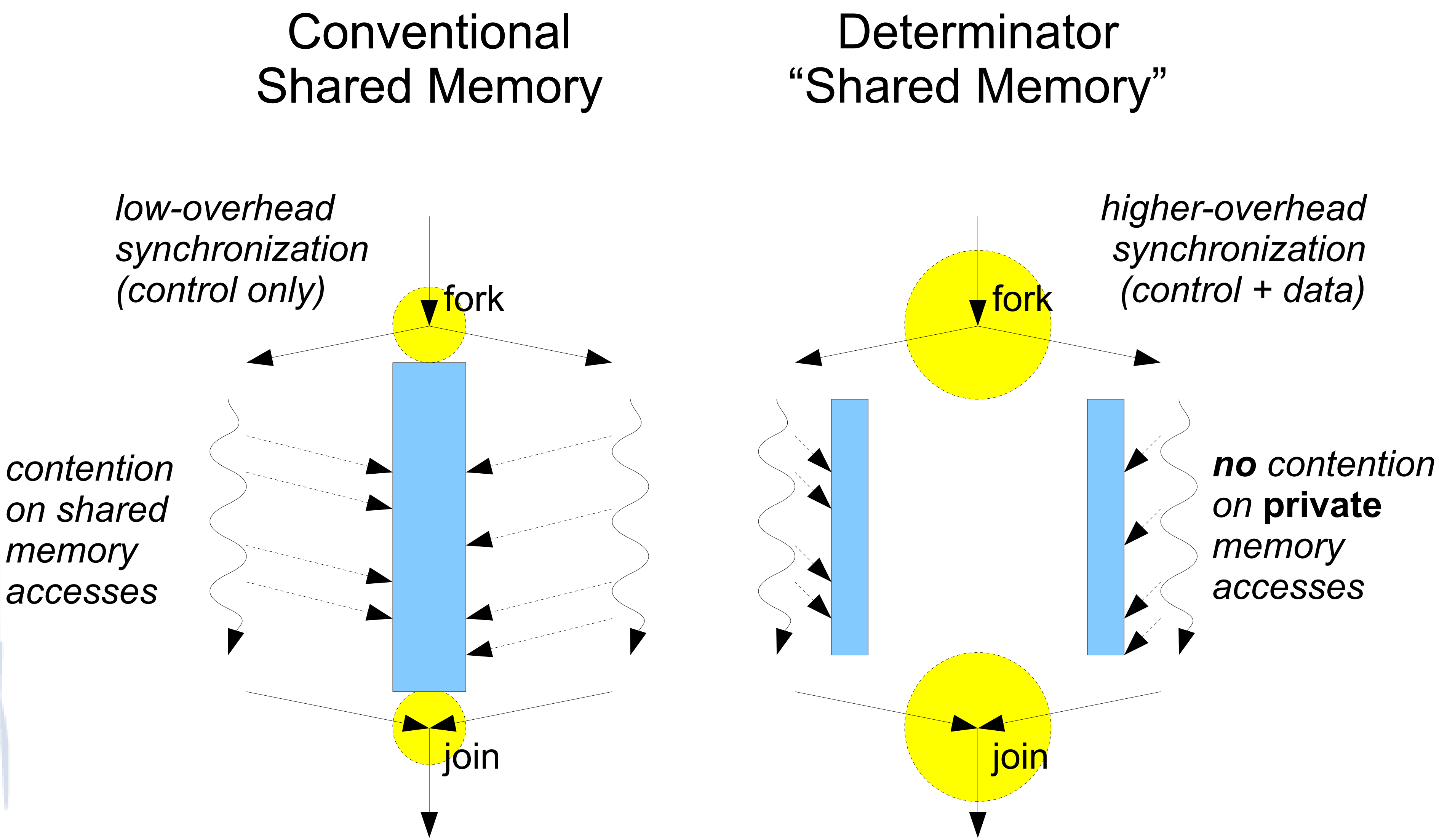
Speedup over 1 CPU



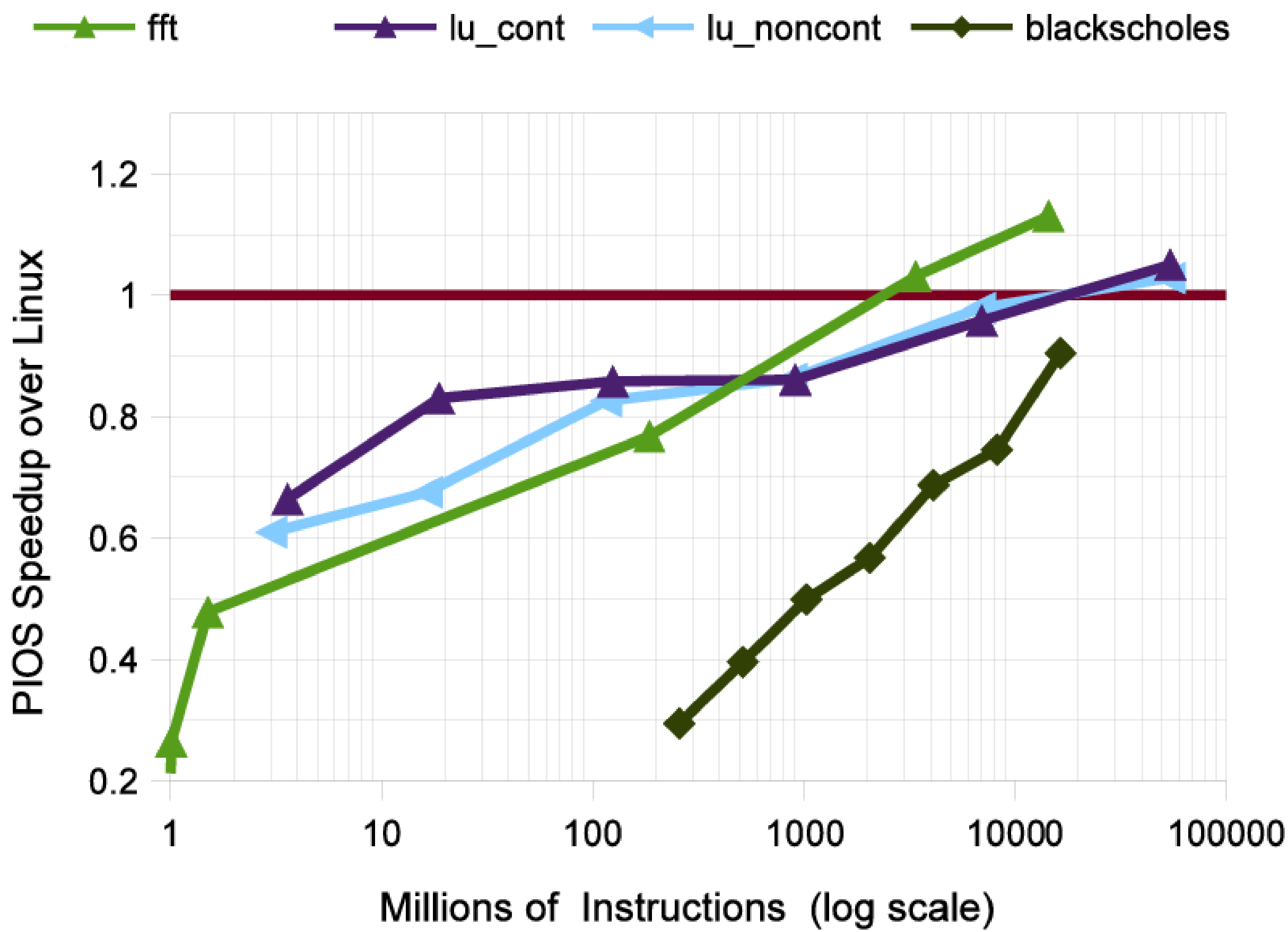
Performance Relative to Linux



Why can Performance **Improve**?



Relative Speed vs Problem Size



Is Determinator's Model **General?**

Determinator v1 directly supported only
simple hierarchical synchronization

- e.g., **fork**, **join**, **barrier**

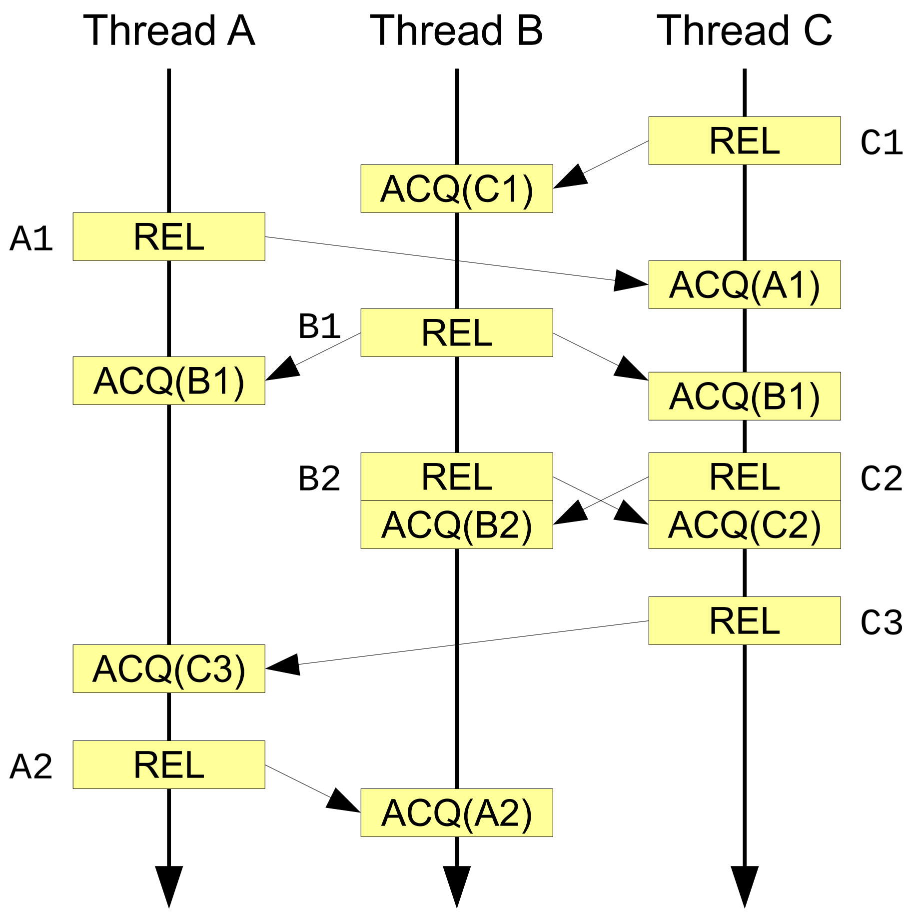
Determinator v2 generalizes to support
general non-hierarchical synchronization

- via **producer-consumer shared memory**

General “Workspace Consistency”

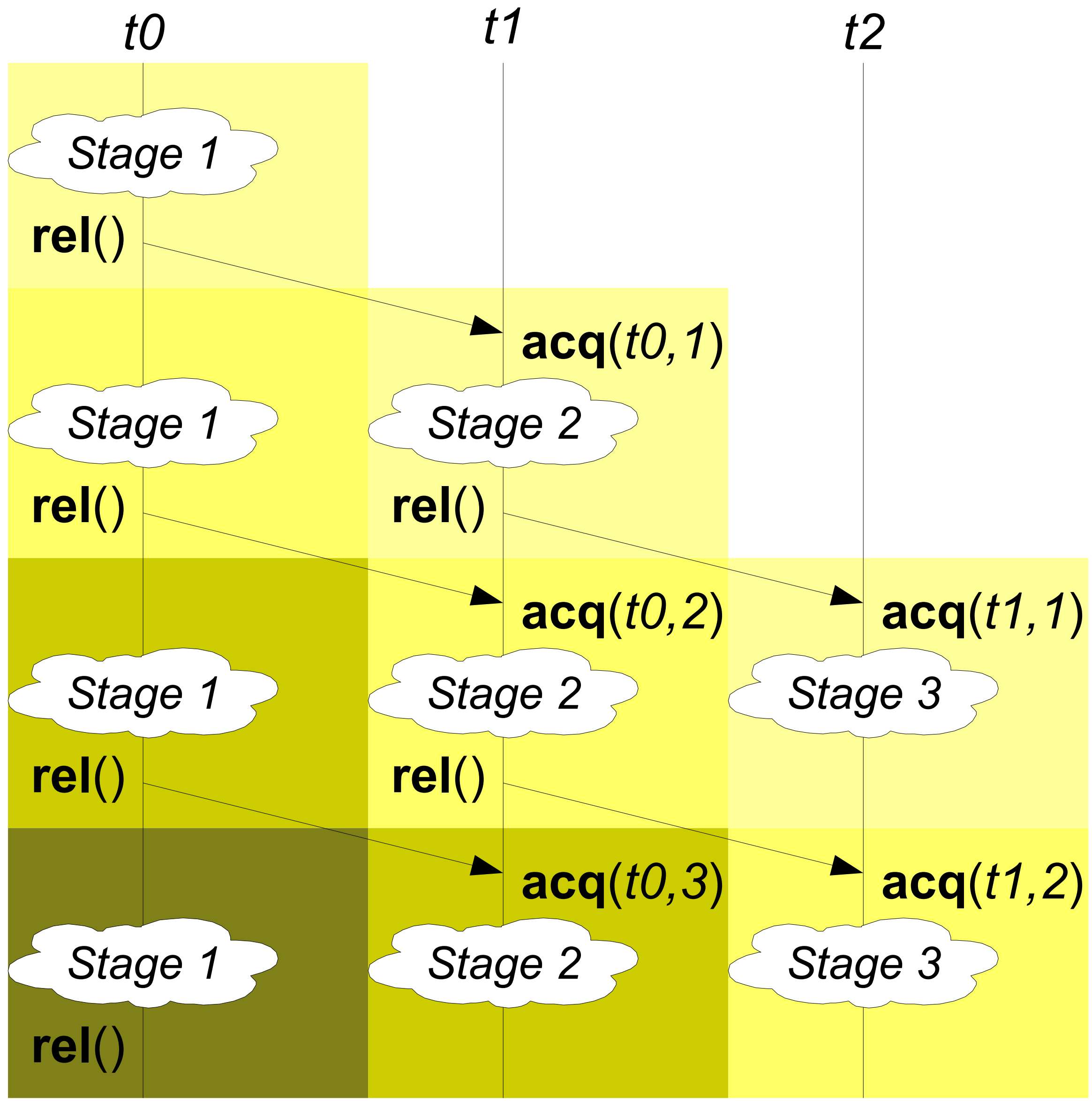
Deterministic analog of **release consistency**

- releases & acquires *explicitly paired*
- updates propagate *only when required to*

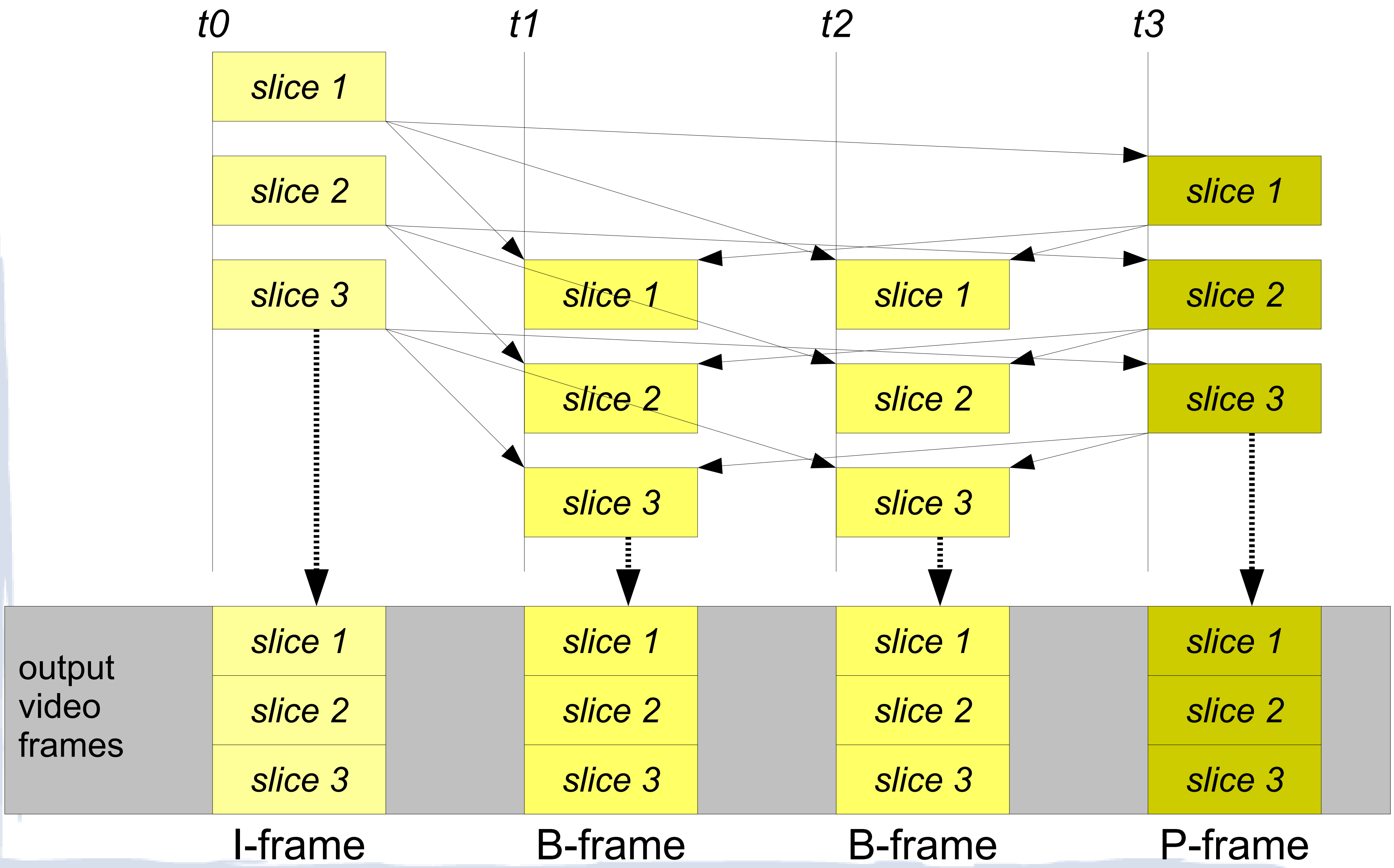


Described in [WoDet '11]

Example: Pipelines

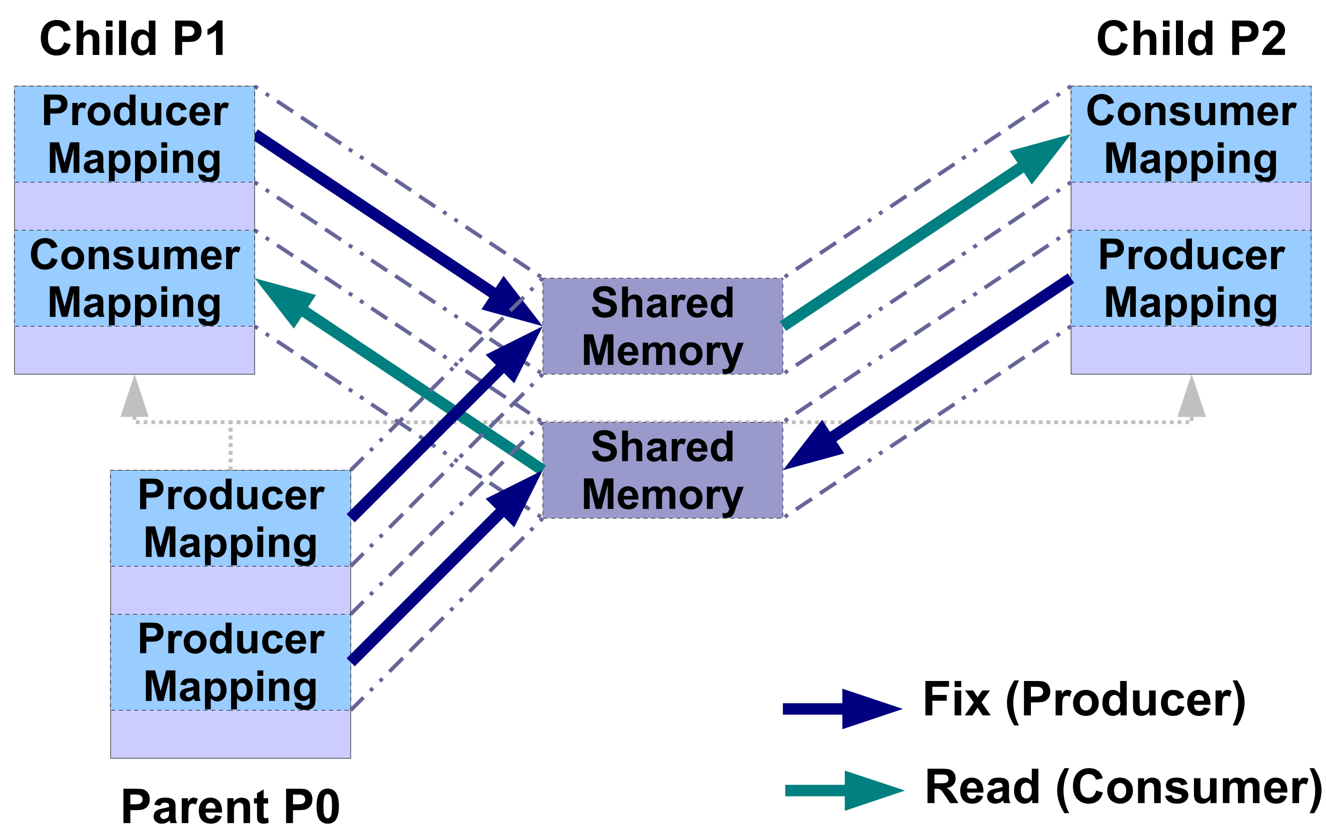


Example: Parallel Video Codec



Producer/Consumer Virtual Memory

OS analog of futures, I-structures [Arvind]



Backward Compatibility

Can we support legacy, **nondeterministic**, pthreads-style parallel code when needed?

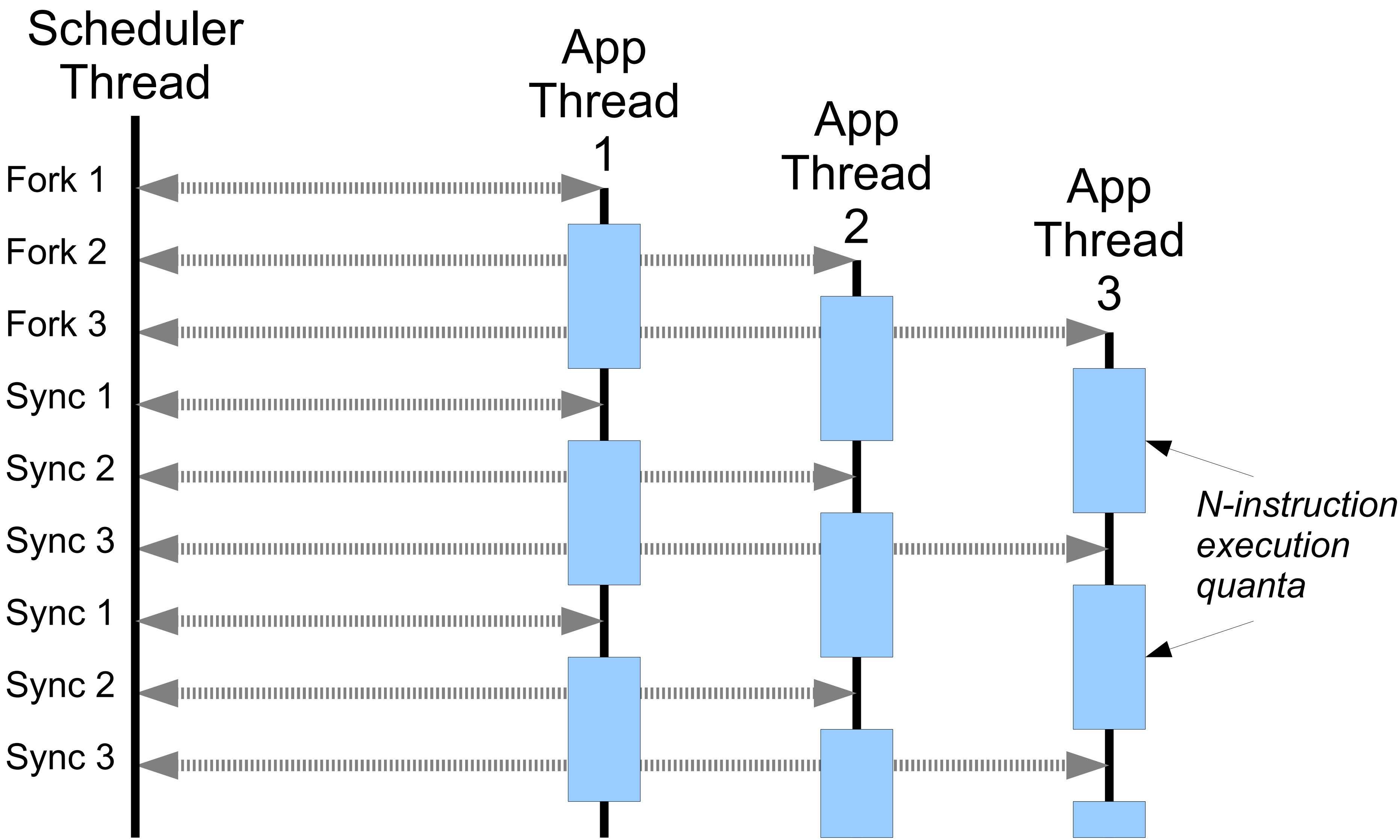
Yes – via **deterministic scheduling**

- synthesize artificial "time schedule" for threads
- similar to techniques in DMP, CoreDet, Grace

But **non-ideal in long term**

- mutexes etc still *semantically nondeterministic*
- "synthetic time" still *unpredictable to developer*
- new inputs, new compiler, new options →
  new time schedule → **new heisenbugs**

Deterministic Scheduling Example



Making Determinism Accessible

To get a **deterministic programming model,** do developers need to **relearn from scratch?**

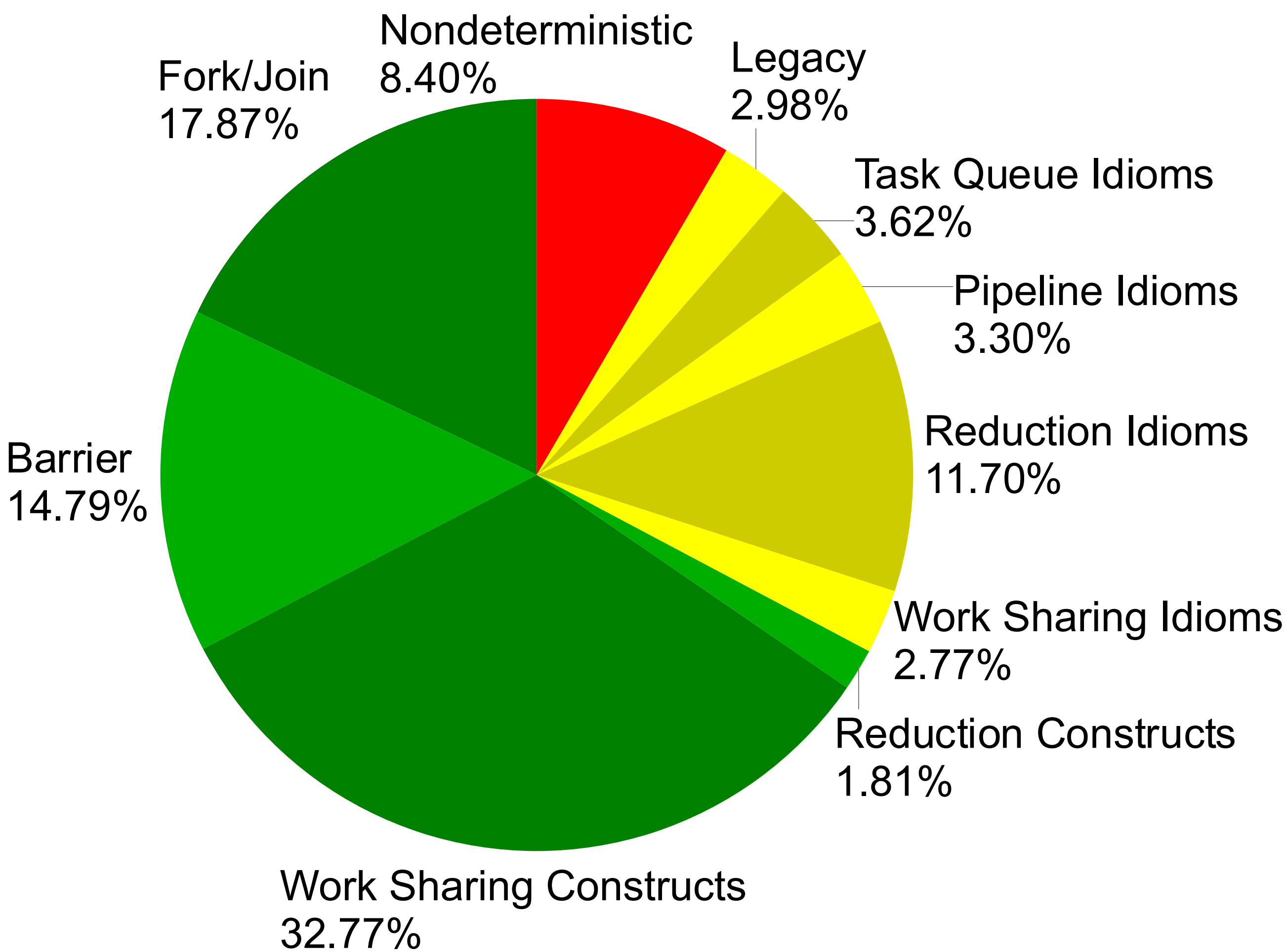
- Unfamiliar languages, parallel abstractions?

Maybe not!

- Existing *high-level* parallel frameworks such as OpenMP are already **“near-deterministic”**
- But “deterministic subsets” not yet rich enough

Uses of Synchronization Idioms

Across SPLASH, NPB, and PARSEC suites



Reduction Examples

Where OpenMP reductions **do** work: CG.f

```
!$omp parallel do reduction(+:t1,t2)
      do j = 1, lastcol-firstcol+1
         t1 = t1 + x(j)*z(j)
         t2 = t2 + z(j)*z(j)
      enddo
```

Where they **don't** work: EP.f – due to *vector* data

```
      do 155 i = 0, nq - 1
!$omp atomic
         q(i) = q(i) + qq(i)
 155  continue
```

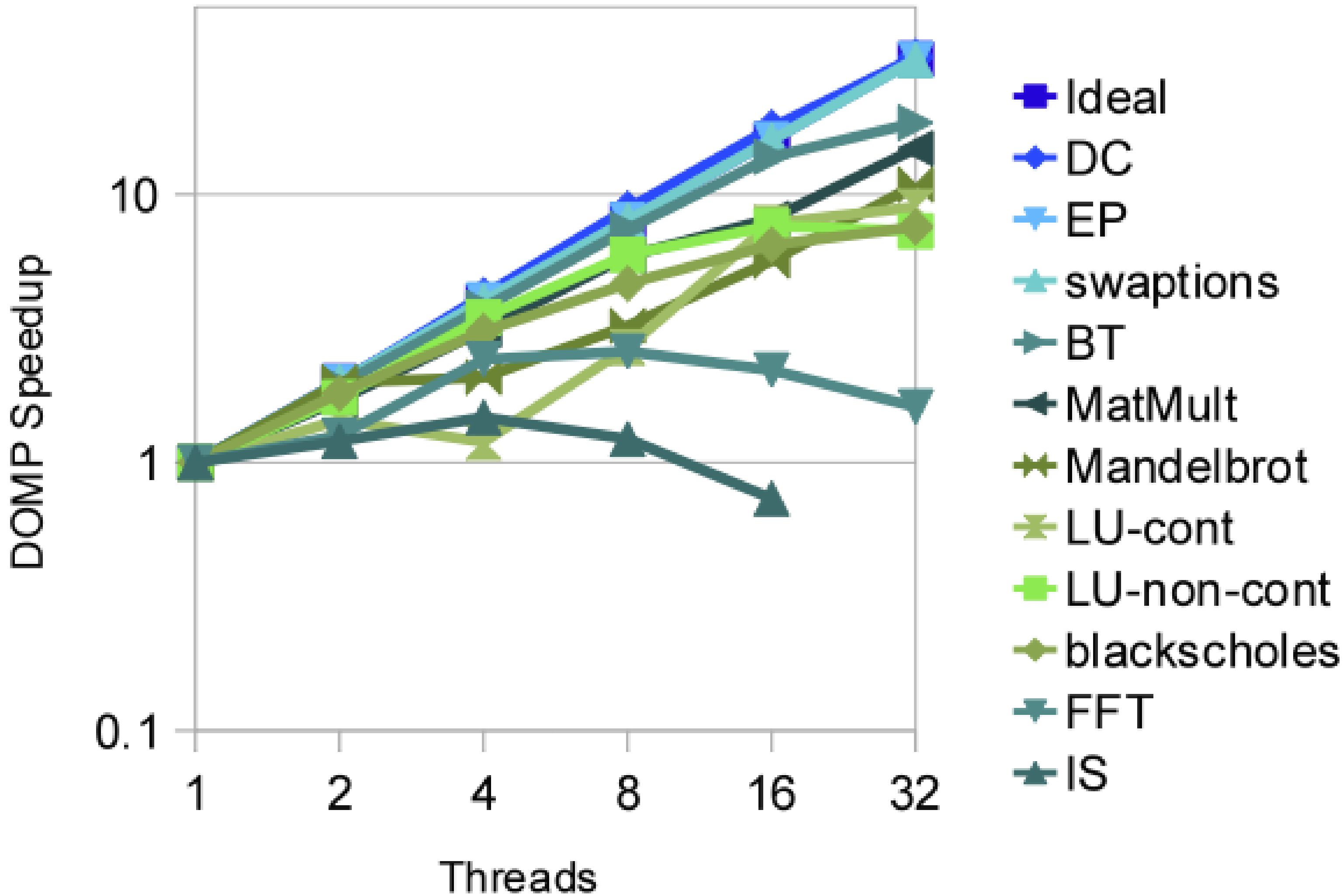
DOMP: Deterministic OpenMP

Make deterministic model more accessible by:

- Retaining familiarity, compatibility w/ OpenMP
- Enriching deterministic parallel abstractions
  - Generalized, user-customizable reductions
- Supporting execution on “vanilla” Linux OS

PhD thesis – Amittai Aviram, Oct 2012
http://dedis.cs.yale.edu/2010/det/

DOMP Speedup on Linux



Can we Distribute Determinism?

Tantalizing potential...

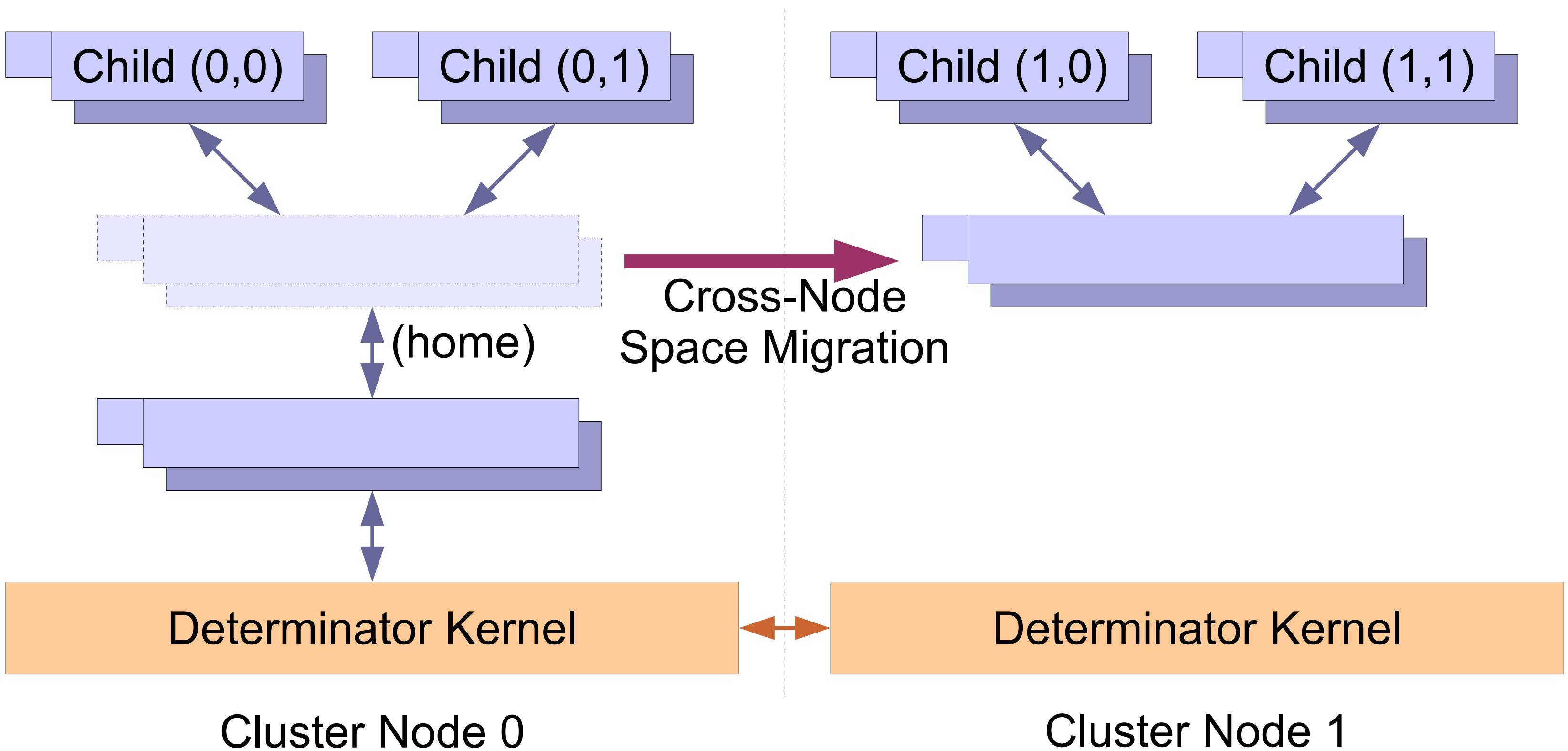
- Time-travel-debug 1000-node data analysis or scientific computations
- Replay-based intrusion analysis/response in large distributed systems

But is it practical?

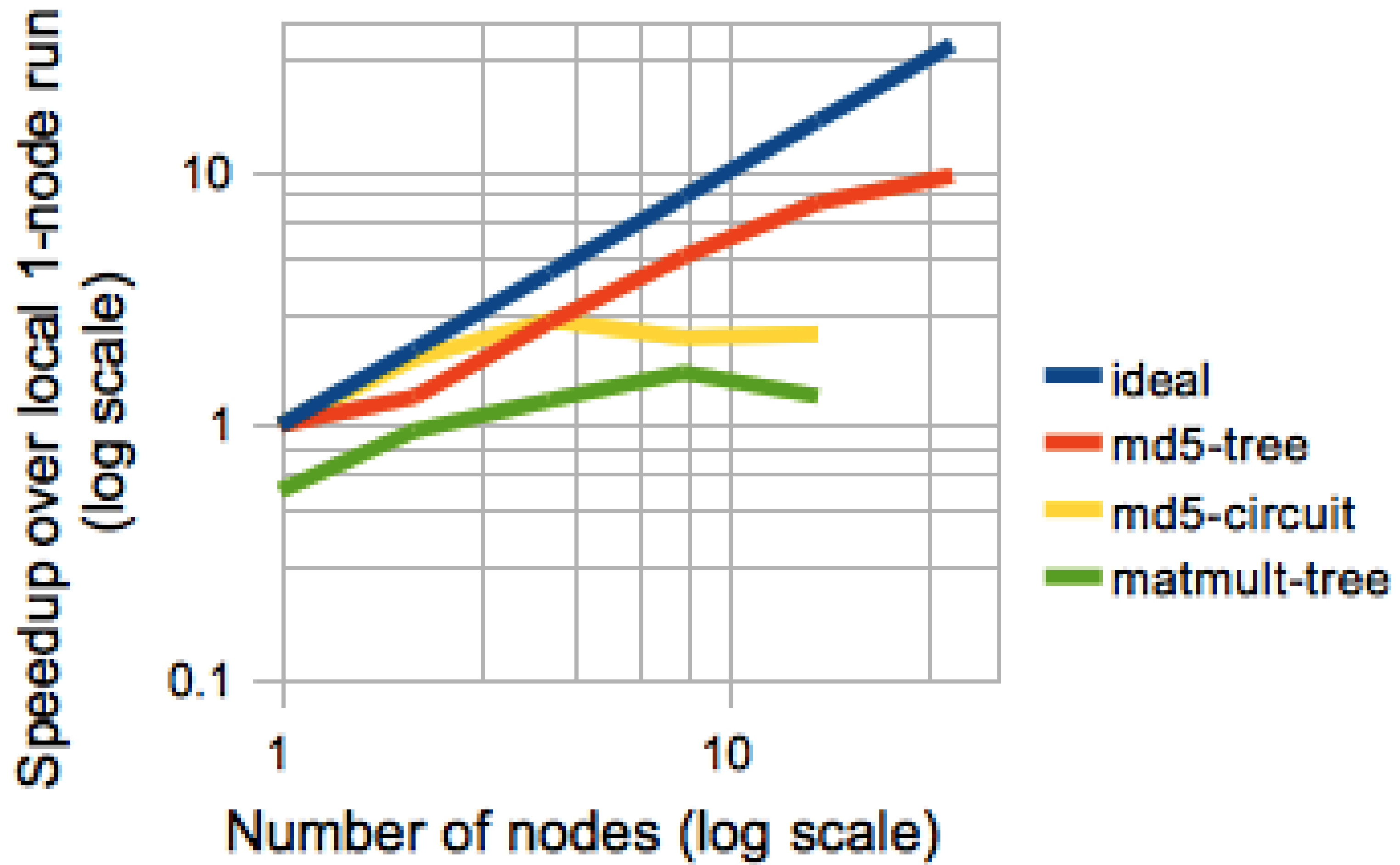
- Simple migration-based mechanism working
- General “Kahn Process Networks” messaging approach w/ MPI layer in-progress

A Proof-of-Concept Approach

Transparent process migration among nodes



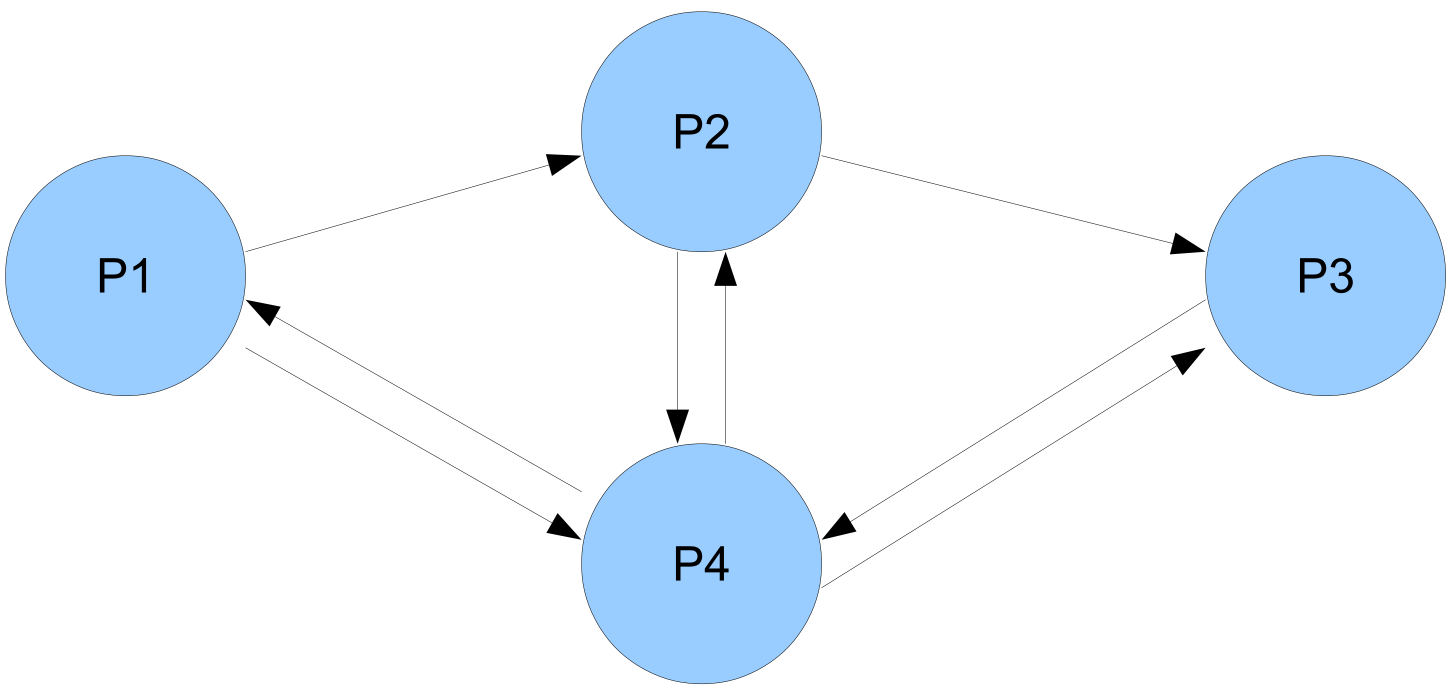
Distributed Speedup over 1 Node



Ongoing Work

Generalize to support efficient

- "Kahn Process Network" message passing
- Deterministic distributed shared memory



Talk Outline

- ✓ Introduction: Parallelism and Data Races
- ✓ Determinator: a Determinism-Enforcing OS
- ✓ Is Determinism *Efficient, General, Usable*?
- **Why *System-Enforced* Determinism?**
- Conclusion

System-Enforced Determinism

Prior deterministic environments implemented by *unprotected* code in *user-space* libraries

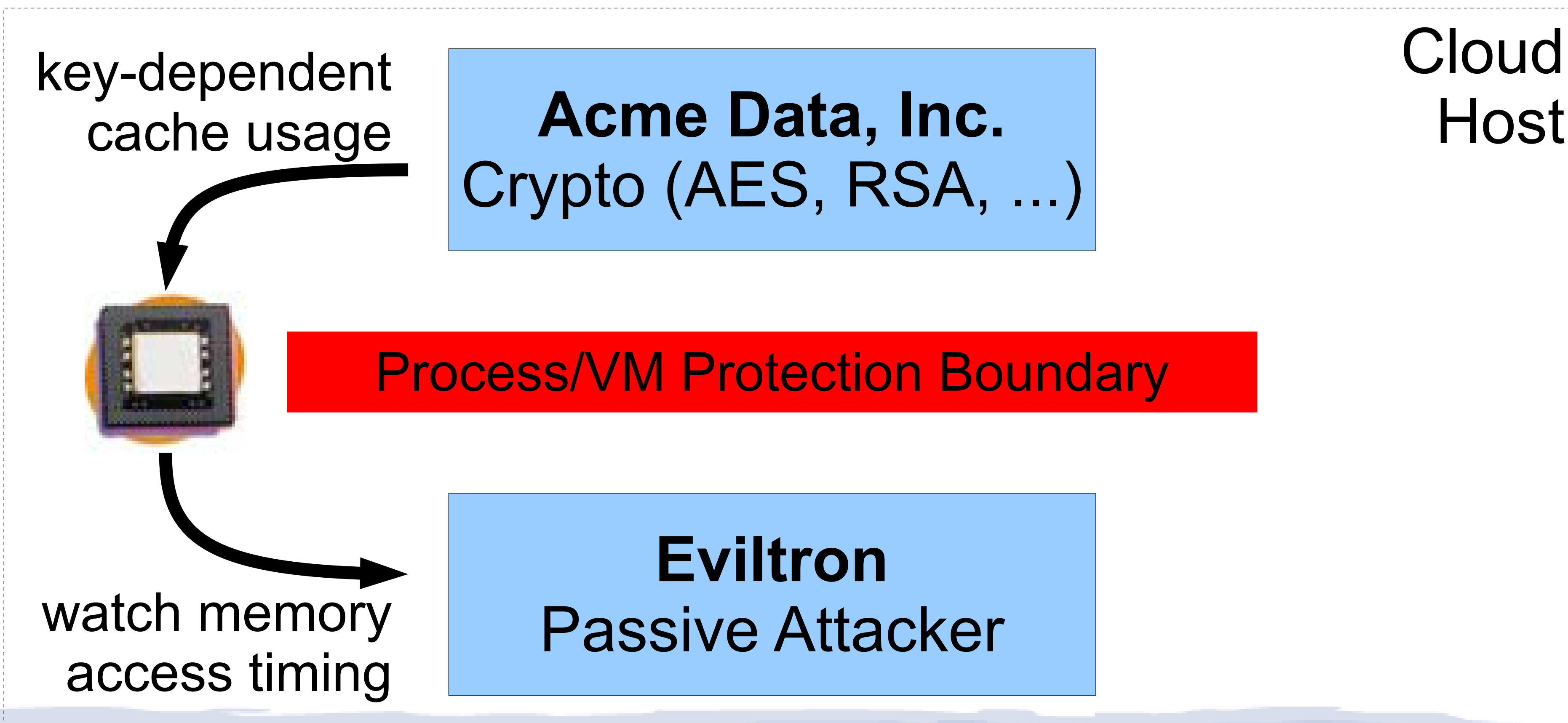
- App bugs can clobber deterministic runtime

Why should we **enforce** deterministic execution?

- *Arbitrarily* buggy code always repeatable
- Prevent malware from evading IDS, analysis
- Close timing side-channel leaks...

Key-Stealing via Timing Channels

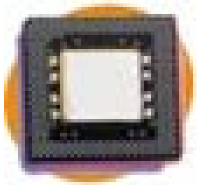
Code *unintentionally* modulate shared resources to reveal secrets when running known algorithms



Anatomy of a Timing Channel

Two elements required: [Wray 91]

- A *resource* that can be *modulated* by the signaling process (or victim)



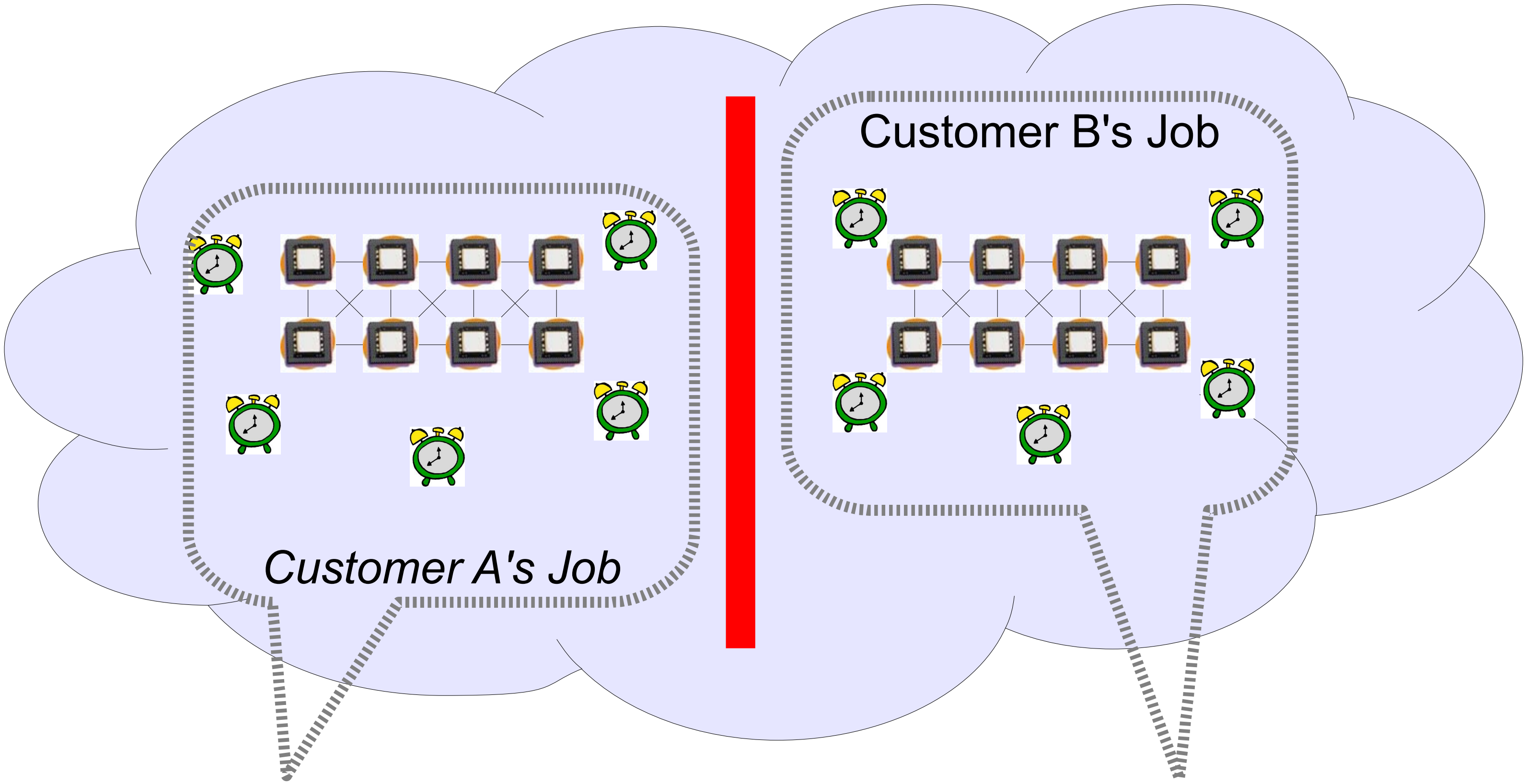
- A *reference clock* enabling the attacker to observe, extract the modulated signal

Remove either → no timing channel.

Traditional Approaches

Eliminate modulation by partitioning hardware

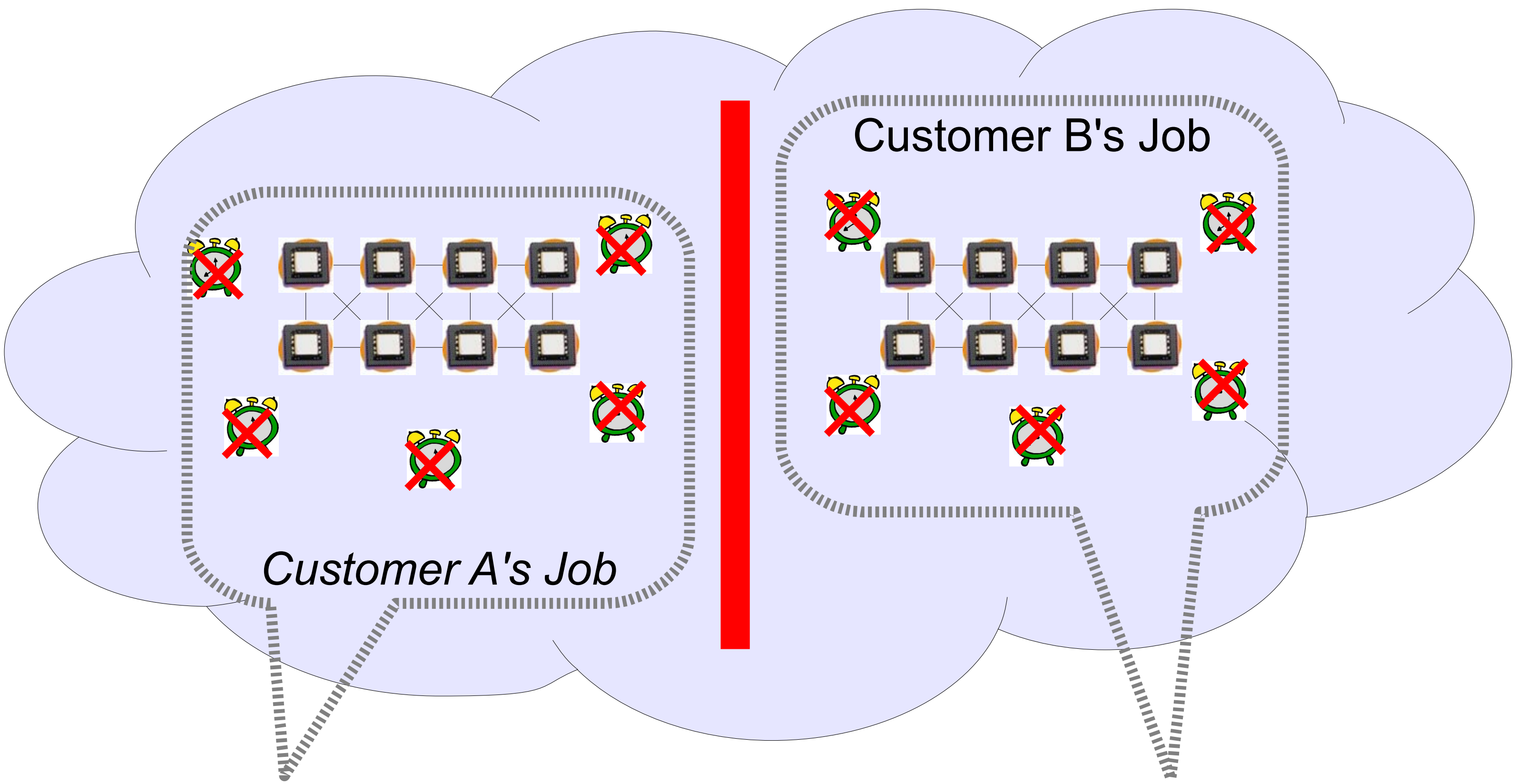
- Requires hardware modifications
- Can't stat-mux → goodbye cloud computing!



The Determinator Approach

Allow modulation, **eliminate reference clocks**

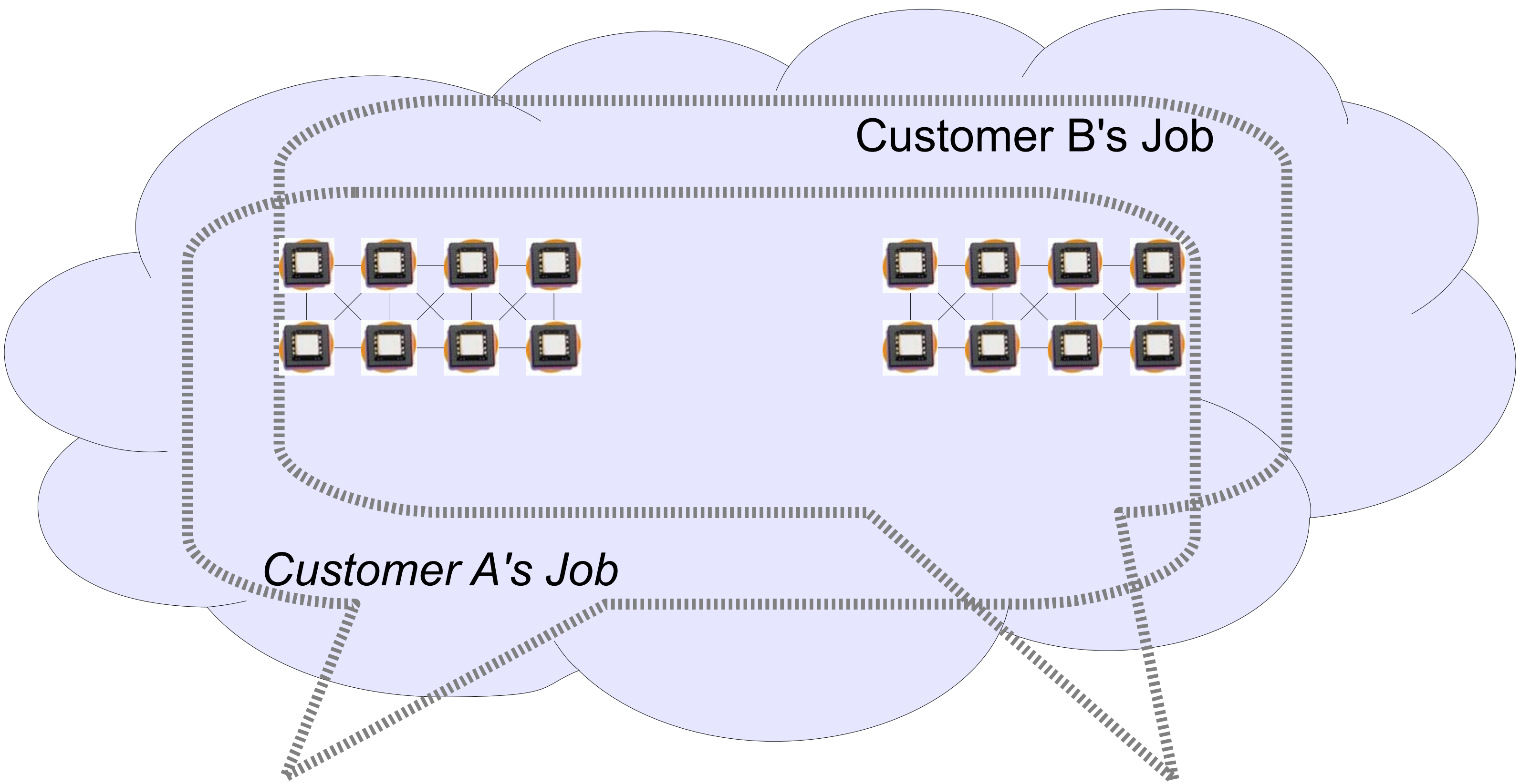
- *Works on current hardware, stat-mux allowed*



The Determinator Approach

Allow modulation, **eliminate reference clocks**

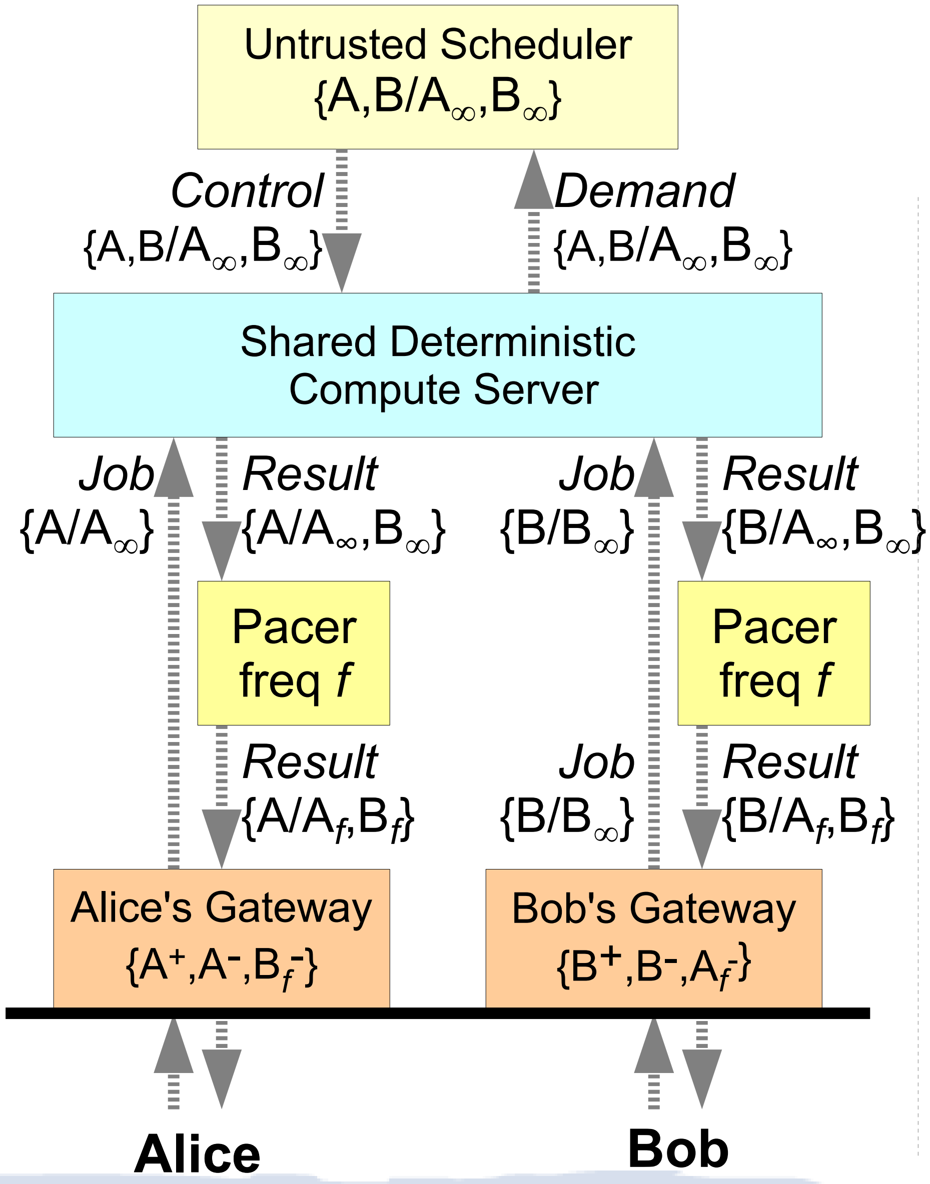
- *Works on current hardware, stat-mux allowed*



Timing Information Flow Control

Initial exploration in:

- Determinating Timing Channels in the Cloud [CCSW '10]
- Plugging Side-Channel Leaks with Timing Information Flow Control [HotCloud '12]



Talk Outline

- ✔ Introduction: Parallelism and Data Races
- ✔ Determinator: a Determinism-Enforcing OS
- ✔ Is Determinism *Efficient, General, Usable*?
- ✔ Why *System-Enforced* Determinism?
- **Conclusion**

Conclusion

In a pervasively parallel world, can we live in a **deterministic model** most–or all–the time?

Determinator suggests pervasive determinism is

- Practical even with **existing languages**
- Even **efficient**, as problem sizes increase
- Has unexpected uses, especially if **enforced**

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

Funding: *NSF CNS-1017206, DARPA CRASH*