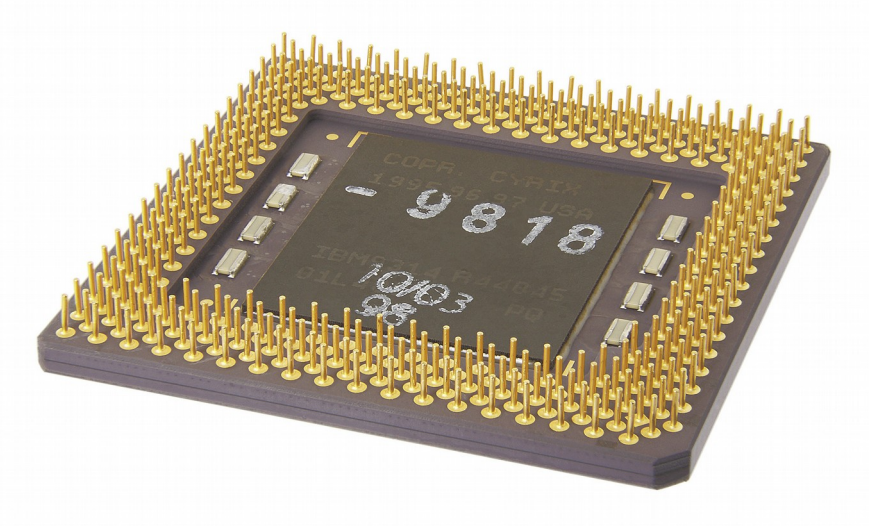
Practical Record And Replay Debugging With rr

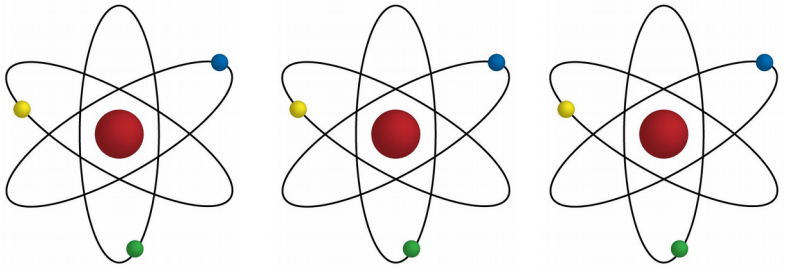
Robert O'Callahan

Debugging nondeterminism

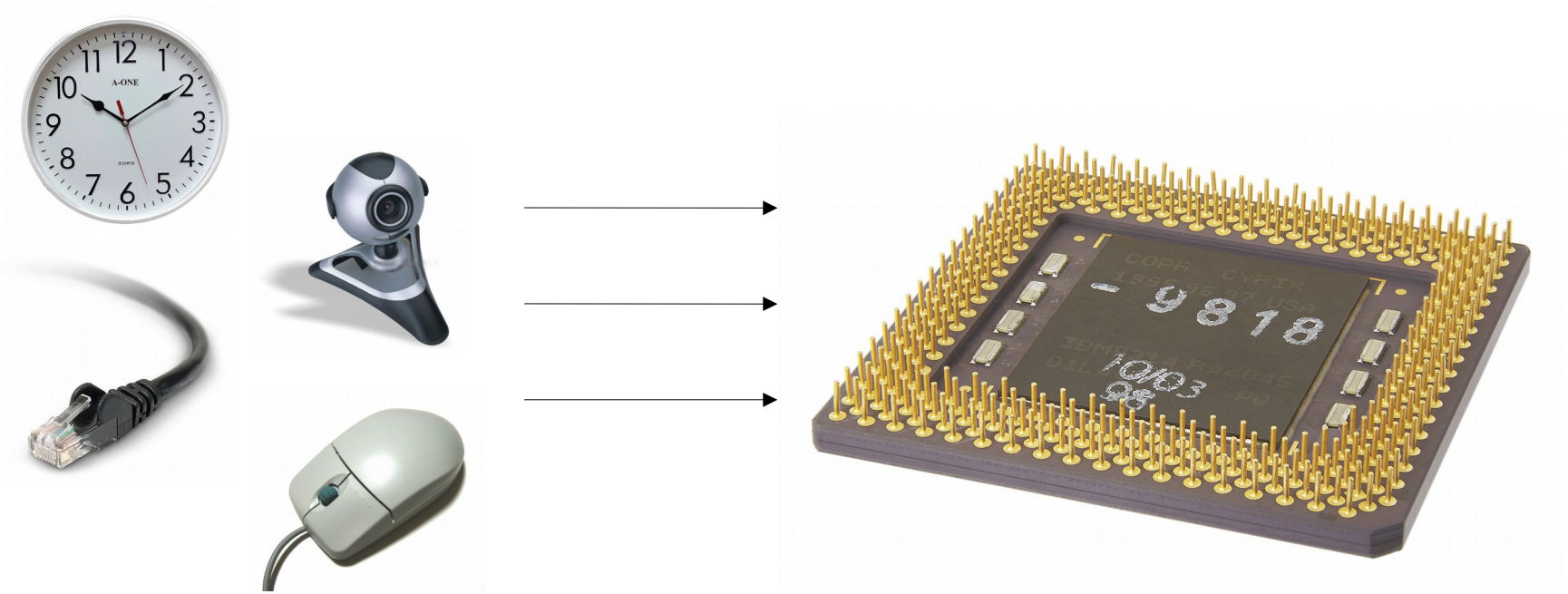
Linux opt	B Cpp Jit1 Jit2 Mn Mn-e10s Wr X M(1 2 3 4 5 JP bc1 bc2 bc3 dt gl oth p) M-e10s(1 2 3 4 5 bc1 bc2 bc3 dt) R(C J R1 R2 Ru) R-e10s(C R-e10s) T(c d g1 g2 o s tp) W(1 2 3 4)
Linux pgo	B Cpp Jit1 Jit2 Mn Mn-e10s Wr X M(1 2 3 4 5 JP bc1 bc2 bc3 dt gl oth p) M-e10s(1 2 3 4 5 bc1 bc2 bc3 dt) R(C J R1 R2 Ru) R-e10s(C R-e10s) T(c d g1 g2 o s tp) W(1 2 3 4)
Linux debug	B Cpp Jit1 Jit2 Mn X M(1 2 3 4 5 JP bc1 bc2 bc3 dt1* dt2 dt3 dt4 gl oth* p) M-e10s(1 2 3 4 5 bc1* bc1 bc2* bc3) R(C J R1 R2) R-e10s(R-e10s1 R-e10s2)
Linux x64 opt	B Cpp H Jit1 Jit2 Ld Mn V Wr X M(1 2 3 4 5 JP bc1 bc2 bc3 dt gl oth p) M-e10s(1 2 3 4 5 bc1 bc2 bc3 dt) R(C J R) R-e10s(C R-e10s) T(c d g1 g2 o s tp) W(1 2 3 4)
Linux x64 pgo	B Cpp Jit1 Jit2 Ld Mn Wr X M(1 2 3 4 5 JP bc1 bc2 bc3 dt gl oth p) M-e10s(1 2 3 4 5 bc1 bc2 bc3 dt) R(C J R) R-e10s(C R-e10s) T(c d g1 g2 o s tp) W(1 2 3 4)
Linux x64 asan	Bd Bo Cpp Jit1 Jit2 M(1 2 3 4 5 JP bc1* bc2 bc3 dt* gl oth p) M-e10s(1 2* 2 3 4 5 bc1* bc2 bc3) R(C J R*)
Linux x64 debug	B Cpp Jit1 Jit2 Mn S X M(1 2 3 4 5 JP bc1 bc2 bc3 dt1 dt2 dt3 dt4 gl oth p) M-e10s(1 2 3 4 5 bc1 bc2 bc3) R(C J R1 R2) R-e10s(R-e10s1 R-e10s2)

Deterministic hardware

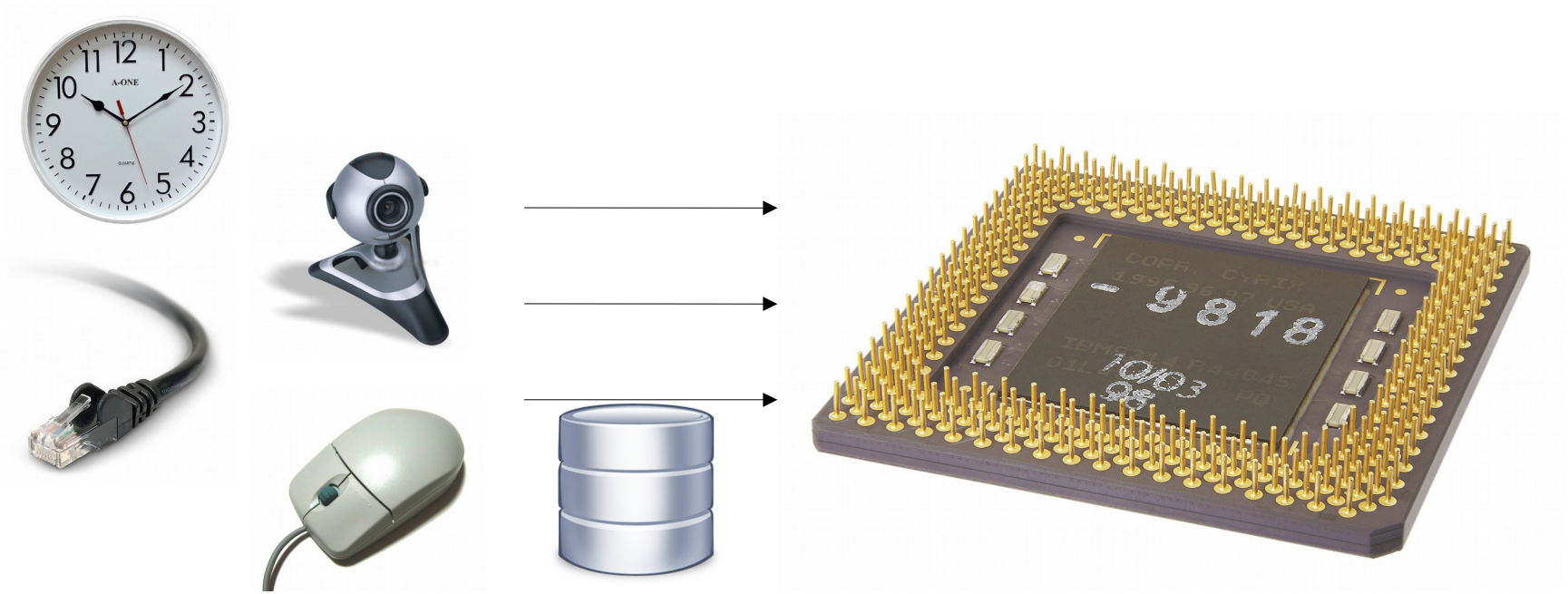




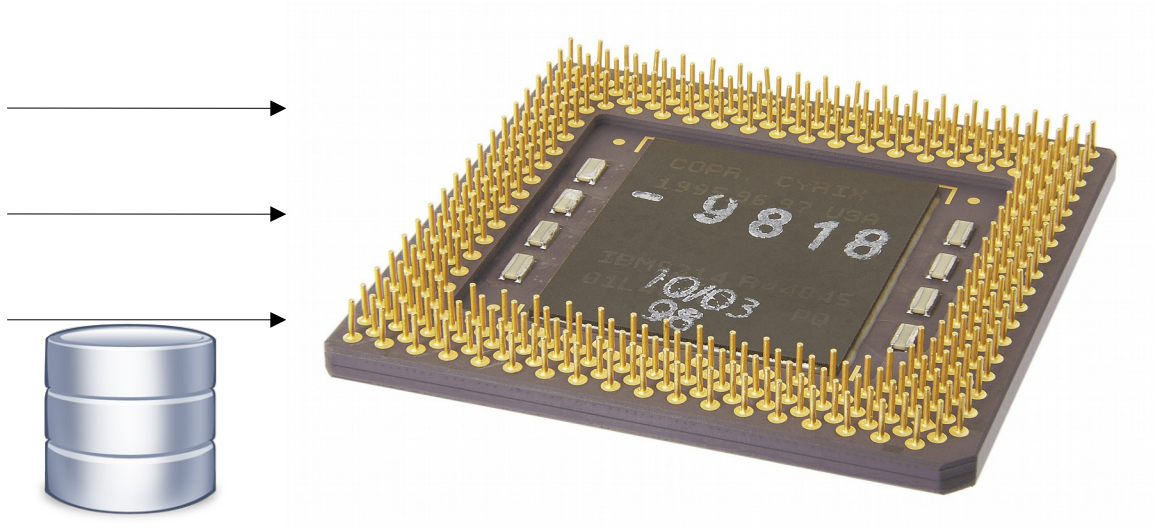
Sources of nondeterminism



Record inputs



Replay execution



“Old idea”

PinPlay

ReVirt

Nirvana

Jockey

ReSpec

Chronomancer

ODR

PANDA

Scribe

Echo

CLAP

FlashBack

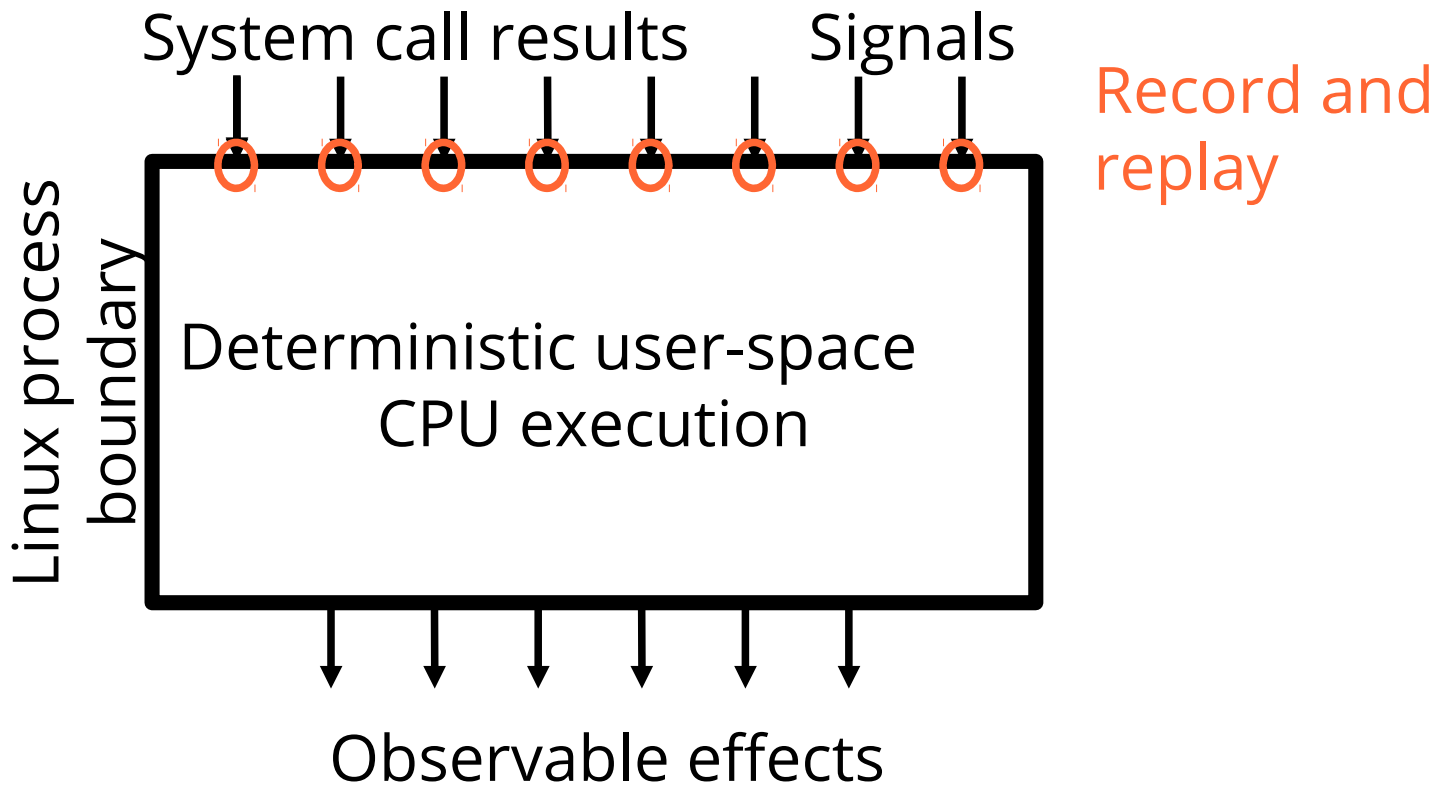
QuickRec

ReTrace

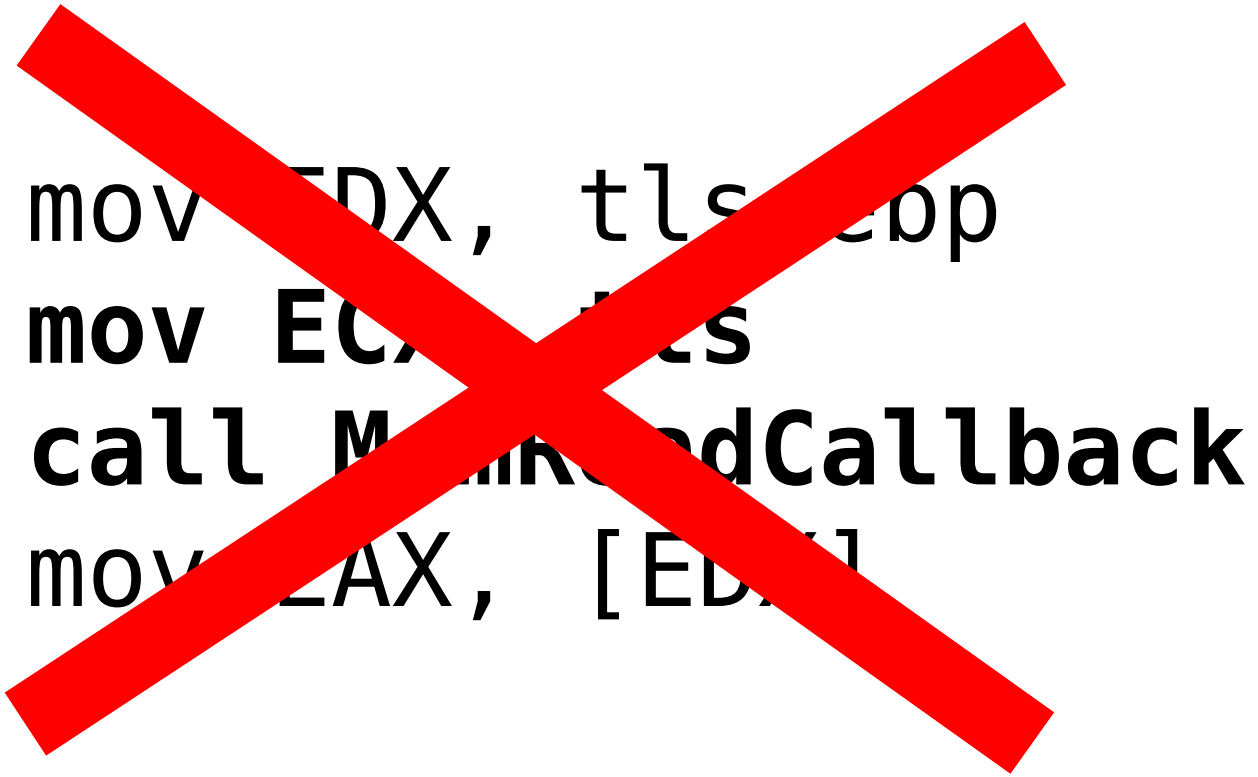
rr goals

- Easy to deploy: stock hardware, OS
- Low overhead
- Works on Firefox
- Small investment

rr design


System call results
Signals
Record and replay
Linux process boundary
Deterministic user-space CPU execution
Observable effects

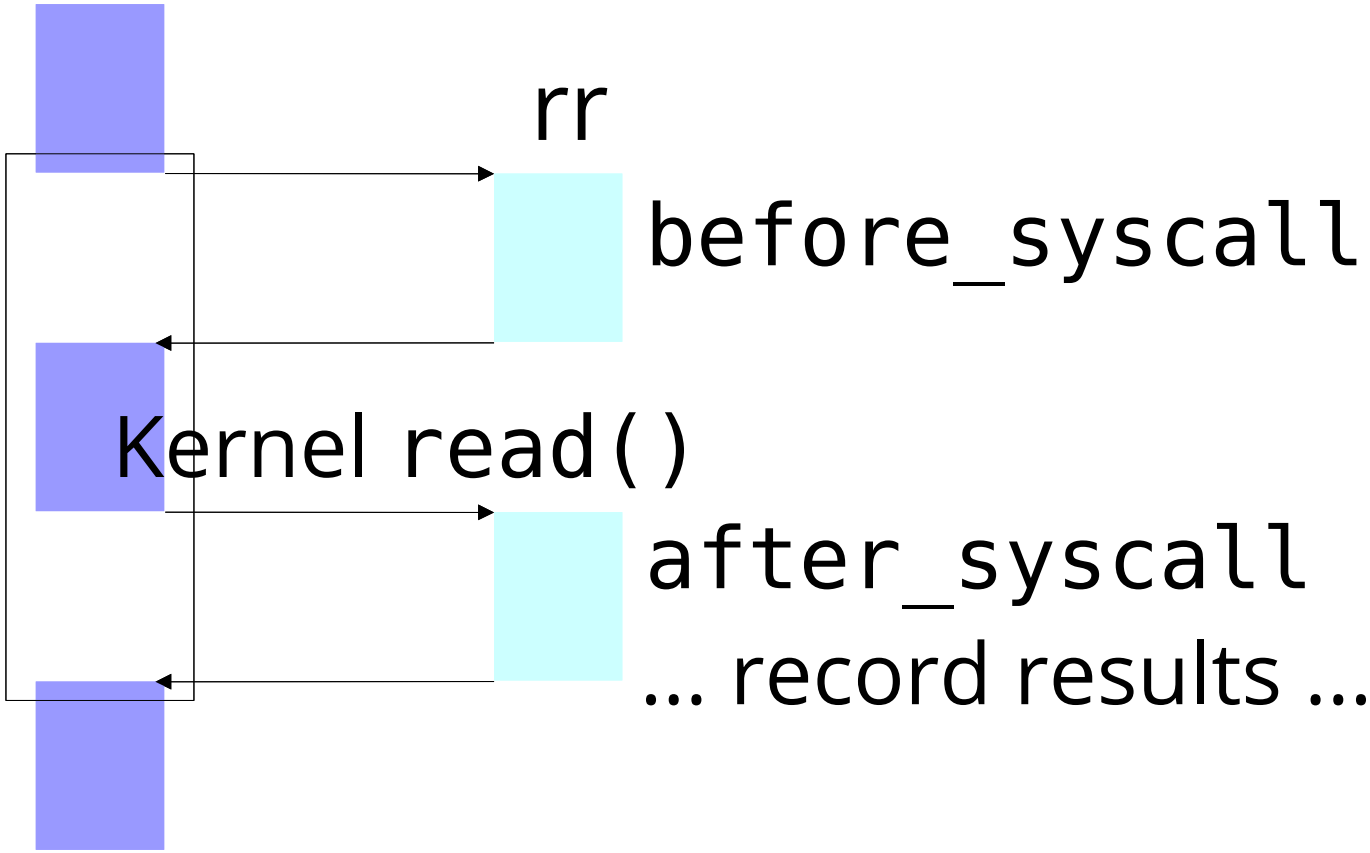
No code instrumentation



Use modern HW/OS features

System call results	`ptrace`
Signals	`ptrace`
Shared memory data races	Limit to single core
Asynchronous event timing	HW performance counters
Trap on a subset of system calls	`seccomp-bpf`
Notification when system call blocks in the kernel	`DESCHED` perf events
Cheap block copies	`FIOCLONERANGE`

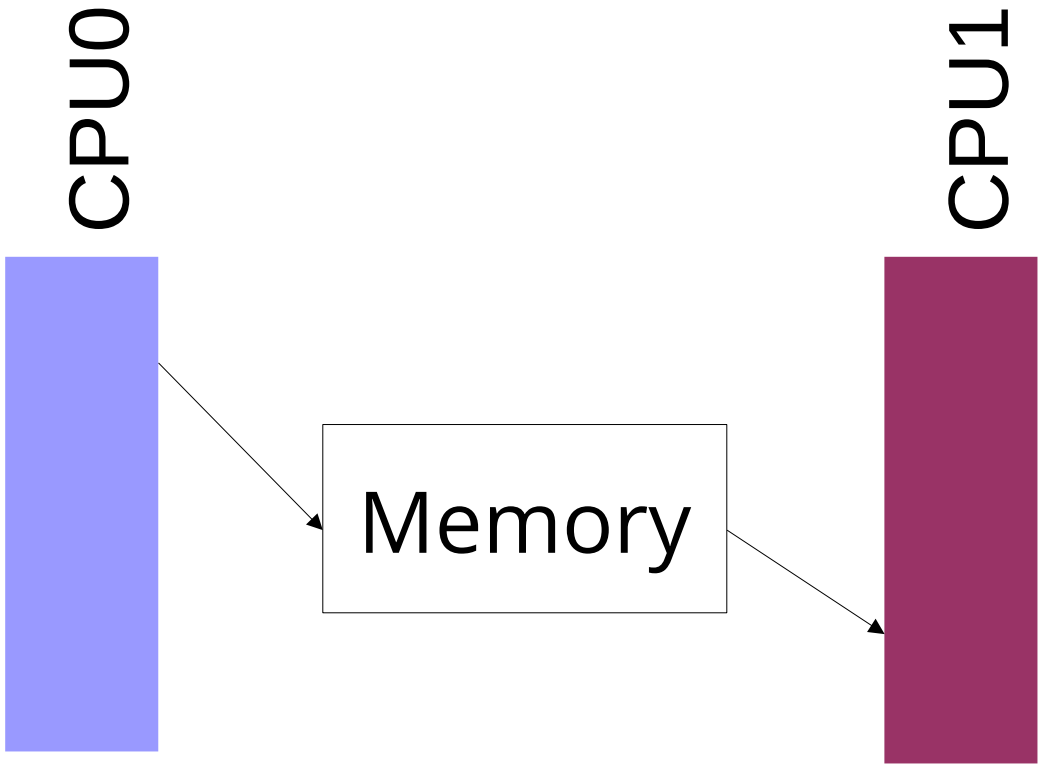
ptrace


rr
before_syscall
Kernel read()
after_syscall
... record results ...

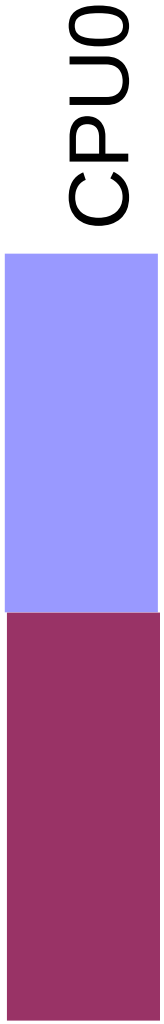
Use modern HW/OS features

System call results	`ptrace`
Signals	`ptrace`
Shared memory data races	Limit to single core
Asynchronous event timing	HW performance counters
Trap on a subset of system calls	`seccomp-bpf`
Notification when system call blocks in the kernel	`DESCHED` perf events
Cheap block copies	`FIOCLONERANGE`

Data races



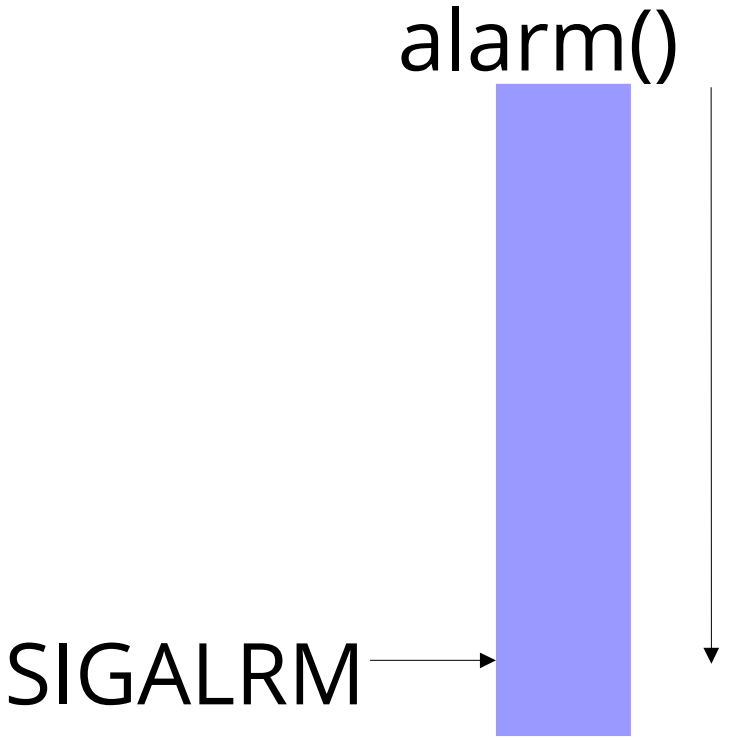
Data races



Use modern HW/OS features

System call results	`ptrace`
Signals	`ptrace`
Shared memory data races	Limit to single core
Asynchronous event timing	HW performance counters
Trap on a subset of system calls	`seccomp-bpf`
Notification when system call blocks in the kernel	`DESCHED` perf events
Cheap block copies	`FIOCLONERANGE`

Event timing: HW perf counters



Measure progress

~~Instructions executed!~~

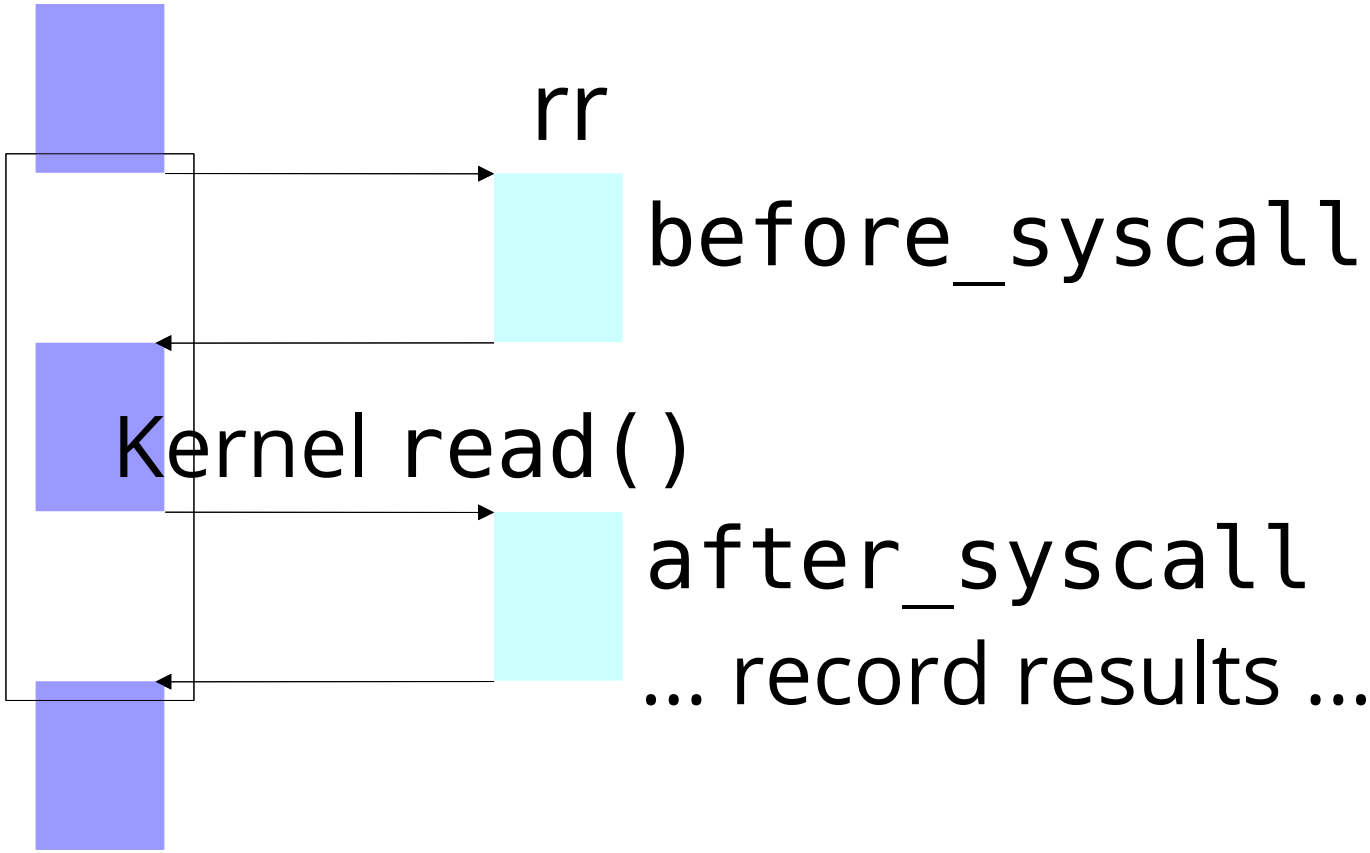
Retired conditional branches (Intel)

Zero overhead

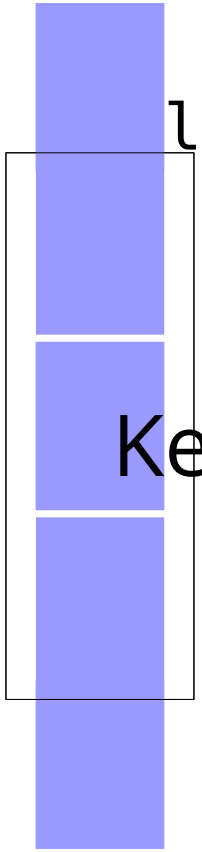
Use modern HW/OS features

System call results	`ptrace`
Signals	`ptrace`
Shared memory data races	Limit to single core
Asynchronous event timing	HW performance counters
Trap on a subset of system calls	`seccomp-bpf`
Notification when system call blocks in the kernel	`DESCHED` perf events
Cheap block copies	`FIOCLONERANGE`

Accelerating system calls



Avoid context switches



librrpreload.so
shim_read()

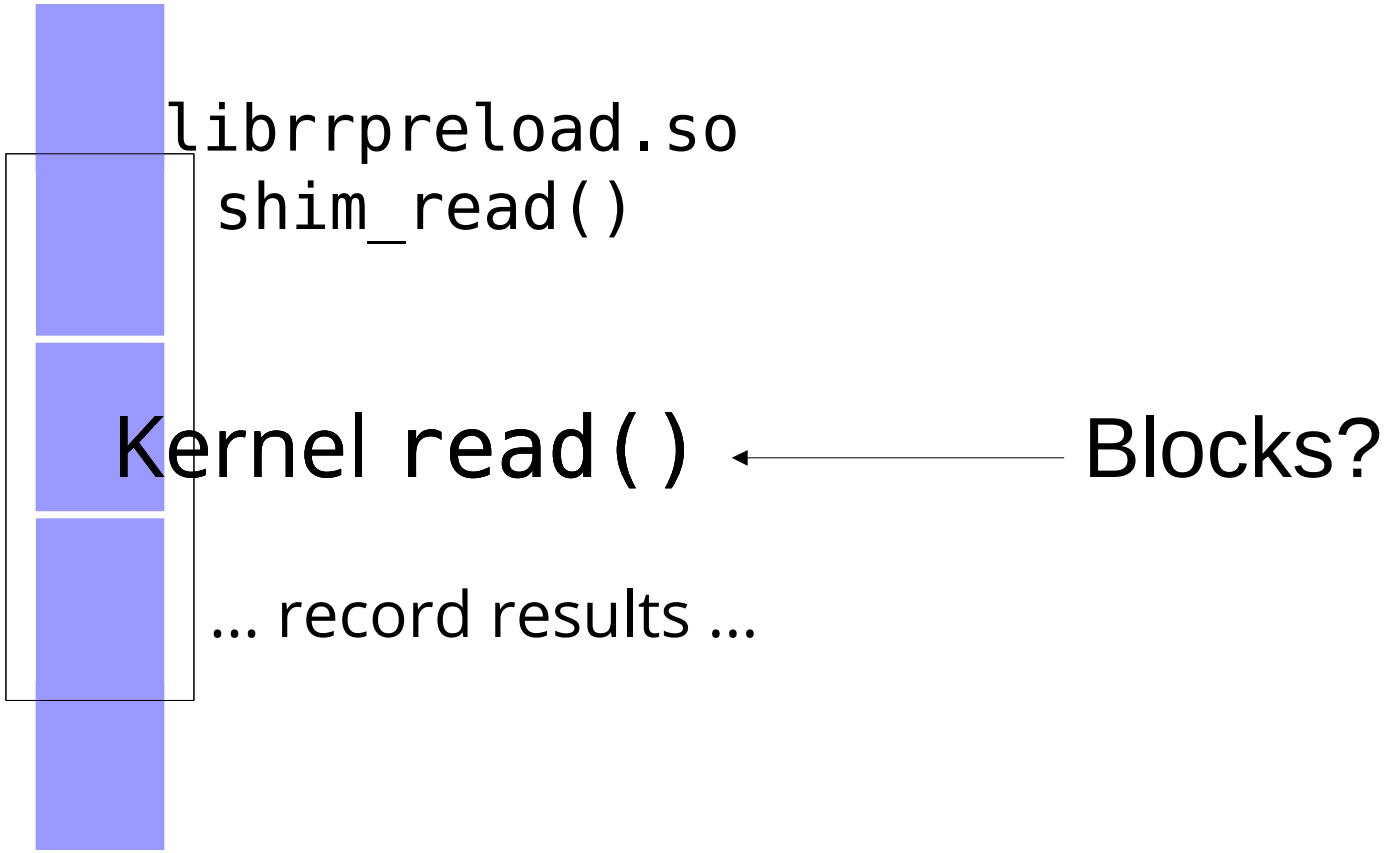
Kernel `read()`

… record results …

Suppress `ptrace` trap

Use `seccomp-bpf` predicates

Blocking system calls

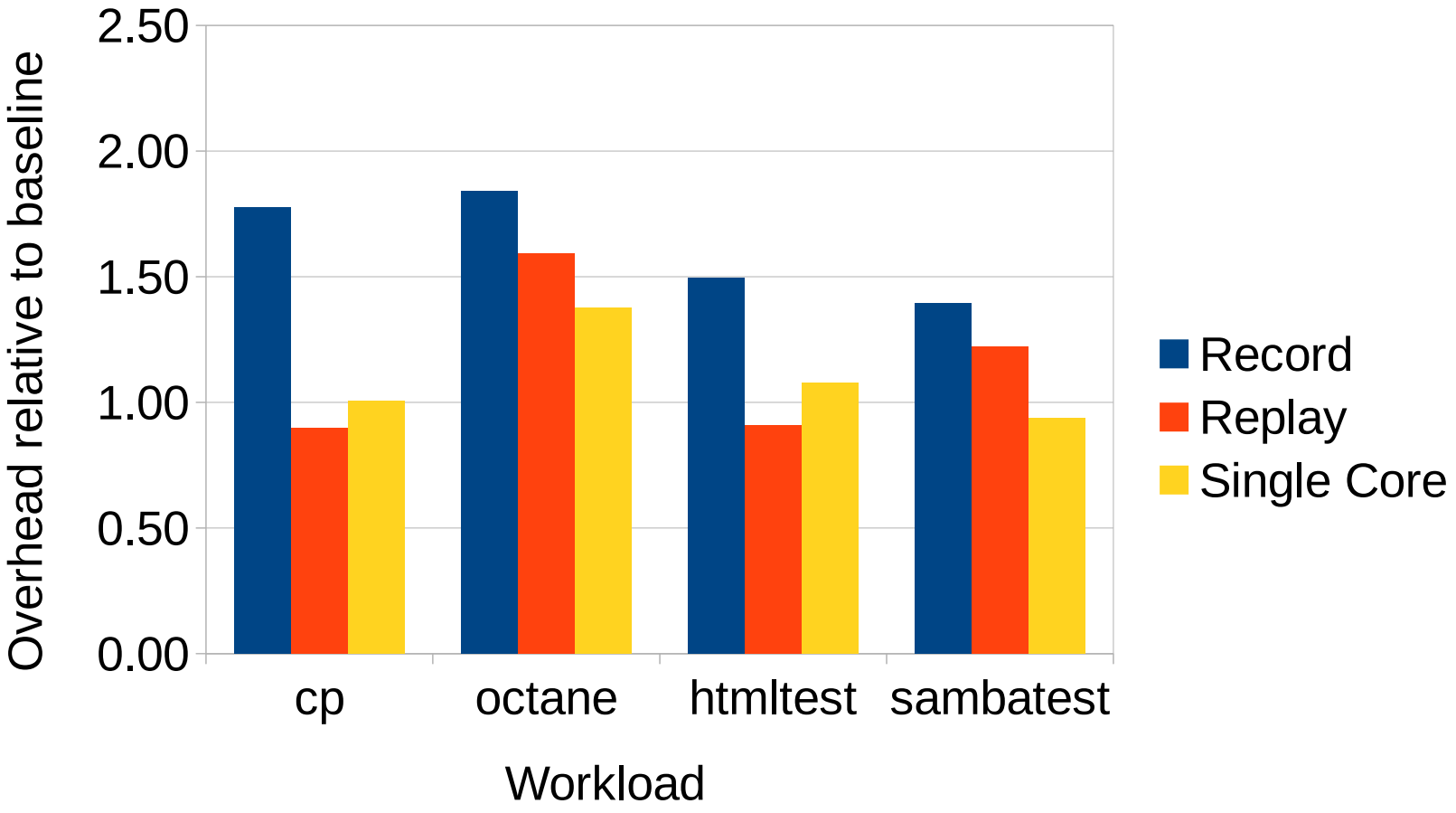


Blocking system calls

Other issues

RDTSC
RDRAND
XBEGIN/XEND
CPUID

rr Overhead


Overhead relative to baseline
2.50
2.00
1.50
1.00
0.50
0.00
cp
octane
htmltest
sambatest
Workload
Record
Replay
Single Core

Reverse-execution Debugging

Chandler Carruth
@chandlerc1024

Follow

Debug on Linux at all? Stop and go get `rr` *RIGHT NOW*. Biggest improv. to debugging for me ever. H/T Justin Lebar.

Lessons

Replay performance matters

Session-cloning performance matters
→ Cloning processes via fork() seems cheaper than e.g. cloning VM state

Lessons

In-process system-call interception is fragile
→ applications make syscalls in strange states (bad TLS, insufficient stack, etc)
→ in-process interception code could be accidentally or maliciously subverted
→ move this part into kernel?

OS design implications

Recording boundary should:
→ be stable, simple, documented API boundary
→ also be a boundary for hardware performance counter measurement

Linux kernel/user boundary is this (mostly)
Windows kernel/user boundary is not

ARM

```
retry:
LDREX r0,[addr]
ADD r0,1
hardware interrupt???
STREX r1,r0,[addr]
CMP r1,0
BNE retry
```

→ Need hardware support to detect/compensate
→ Or binary rewriting

Related work

VM-level replay … **heavyweight**
→ ReVirt, VMWare, QEMU (PANDA), Xen
Kernel-supported replay … **hard to maintain**
→ Scribe, dOS, Arnold
Pure user-space replay … **instrumentation, higher overhead**
→ PinPlay, iDNA, UndoDB
Higher-level replay … **more limited scope**
→ Chronon, Dolos, Chakra, R2
Parallel replay … **more limited scope, higher overhead**
→ SMP-ReVirt, DoublePlay, ODR, Castor
Hardware-supported parallel replay … **nonexistent hardware**
→ FDR, BugNet, DeLorean, QuickRec

Conclusions

rr's approach delivers a lot of value

More research needed for multicore approaches

Lots of unexplored applications of record+replay

http://rr-project.org
https://github.com/mozilla/rr