

Operating Systems Meet Fault Tolerance

Microkernel-Based Operating Systems

Maksym Planeta Björn Döbel

30.01.2018

“If there’s more than one possible outcome of a job or task, and one of those outcome will result in disaster or an undesirable consequence, then somebody will do it that way.” (Edward Murphy jr.)

Outline

- ▶ Murphy and the OS: Is it really that bad?
- ▶ Fault-Tolerant Operating Systems
 - ▶ Minix3
 - ▶ CuriOS
 - ▶ L4ReAnimator
- ▶ Dealing with Hardware Errors
 - ▶ Transparent replication as an OS service

Textbook terminology

Dependability threats:

- ▶ Failure
- ▶ Error
- ▶ Fault

Dependability means

- ▶ Prevention
- ▶ Removal
- ▶ Forecasting
- ▶ Tolerance

Why Things go Wrong

- ▶ Programming in C:

This pointer is certainly never going to be NULL!

Why Things go Wrong

- ▶ **Programming in C:**

This pointer is certainly never going to be NULL!

- ▶ **Layering vs. responsibility:**

Of course, someone in the higher layers will already have checked this return value.

Why Things go Wrong

- ▶ **Programming in C:**

This pointer is certainly never going to be NULL!

- ▶ **Layering vs. responsibility:**

Of course, someone in the higher layers will already have checked this return value.

- ▶ **Concurrency:**

*This struct is shared between an IRQ handler and a kernel thread.
But they will never execute in parallel.*

Why Things go Wrong

- ▶ **Programming in C:**

This pointer is certainly never going to be NULL!

- ▶ **Layering vs. responsibility:**

Of course, someone in the higher layers will already have checked this return value.

- ▶ **Concurrency:**

This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.

- ▶ **Hardware interaction:**

But the device spec said, this was not allowed to happen!

Why Things go Wrong

- ▶ **Programming in C:**

This pointer is certainly never going to be NULL!

- ▶ **Layering vs. responsibility:**

Of course, someone in the higher layers will already have checked this return value.

- ▶ **Concurrency:**

This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.

- ▶ **Hardware interaction:**

But the device spec said, this was not allowed to happen!

- ▶ **Hypocrisy:**

I'm a cool OS hacker. I won't make mistakes, so I don't need to test my code!

A Classic Study

- ▶ A. Chou et al.: *An empirical study of operating system errors*, SOSP 2001
- ▶ Automated software error detection (today: <https://www.coverity.com>)
- ▶ Target: Linux (1.0 - 2.4)
 - ▶ Where are the errors?
 - ▶ How are they distributed?
 - ▶ How long do they survive?
 - ▶ Do bugs cluster in certain locations?

Revalidation of Chou's Results

- ▶ N. Palix et al.: *Faults in Linux: Ten years later*, ASPLOS 2011
- ▶ 10 years of work on tools to decrease error counts - has it worked?
- ▶ Repeated Chou's analysis until Linux 2.6.34

Linux: Lines of Code

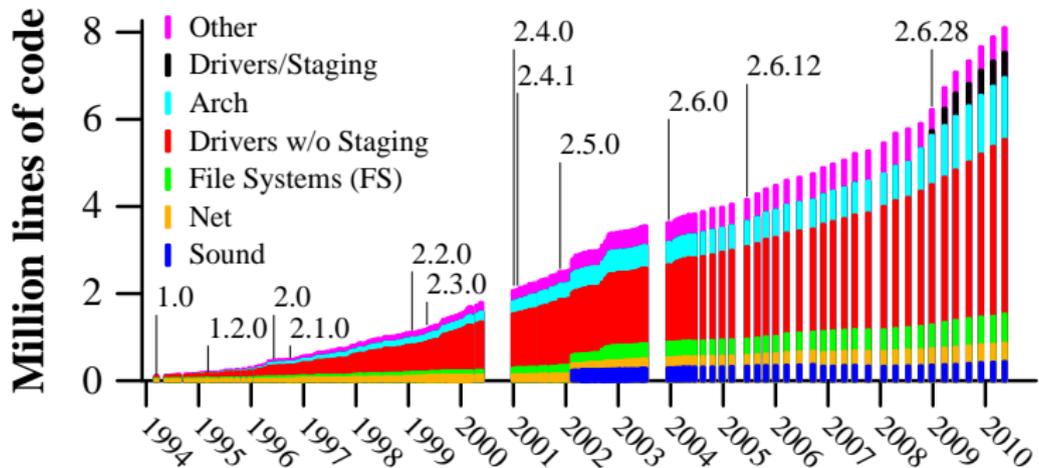


Figure: Linux directory sizes (in MLOC) [14]

Faults per Subdirectory (2001)

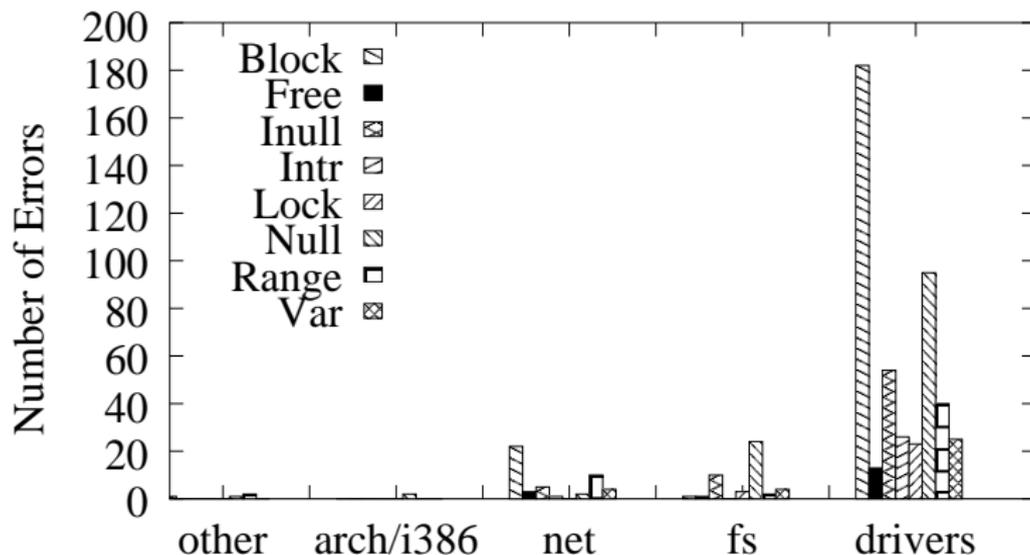


Figure: Number of errors per directory in Linux [4]

Fault Rate per Subdirectory (2001)

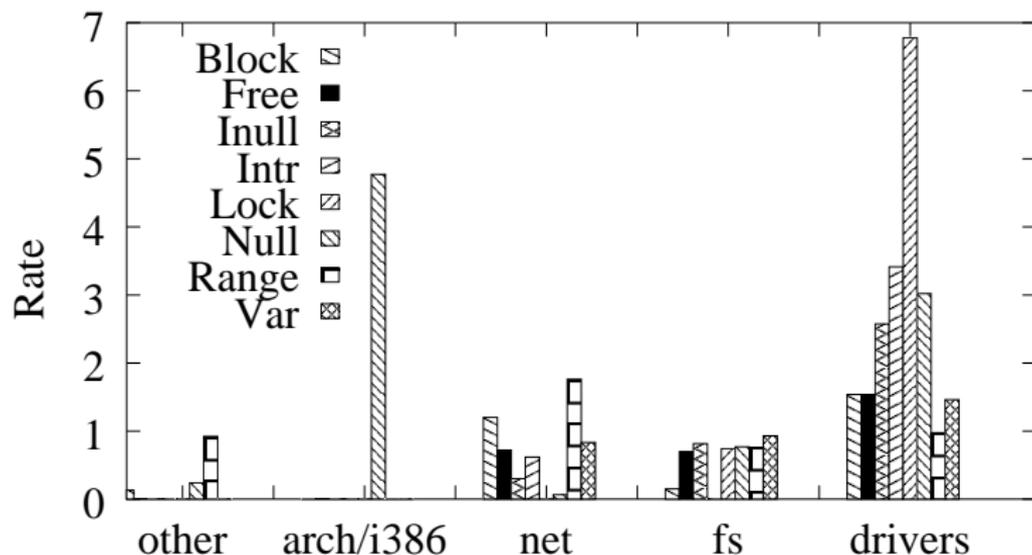


Figure: Rate of errors compared to other directories [4]

Fault Rate per Subdirectory (2011)

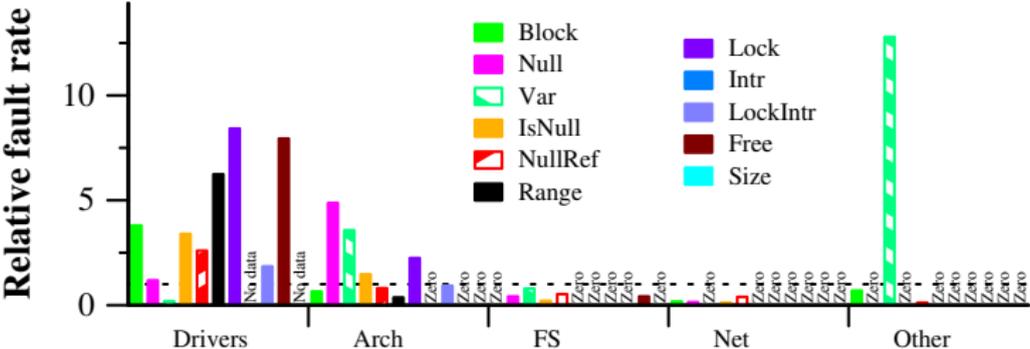
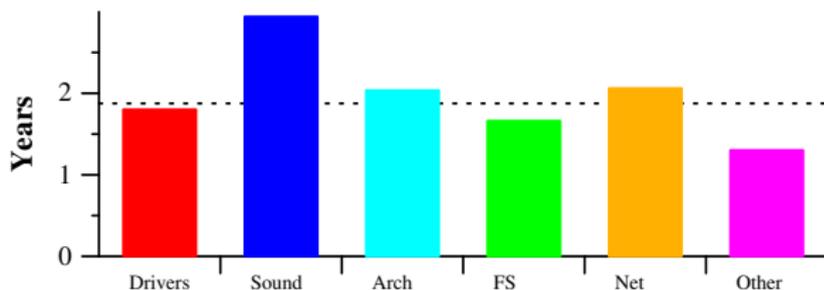
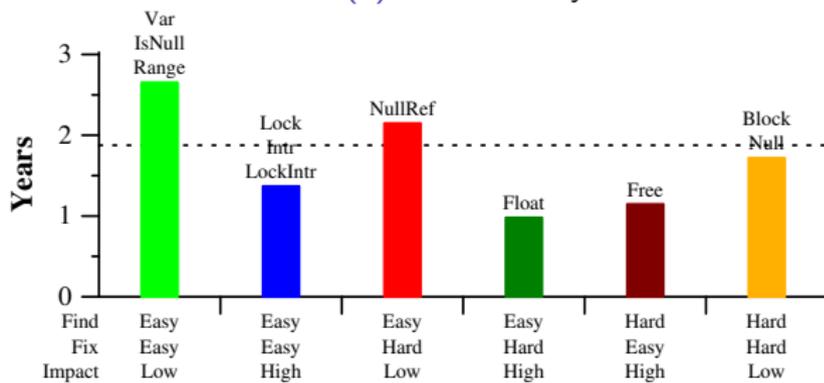


Figure: Linux directory sizes (in MLOC) [14]

Bug Lifetimes (2011)



(a) Per directory



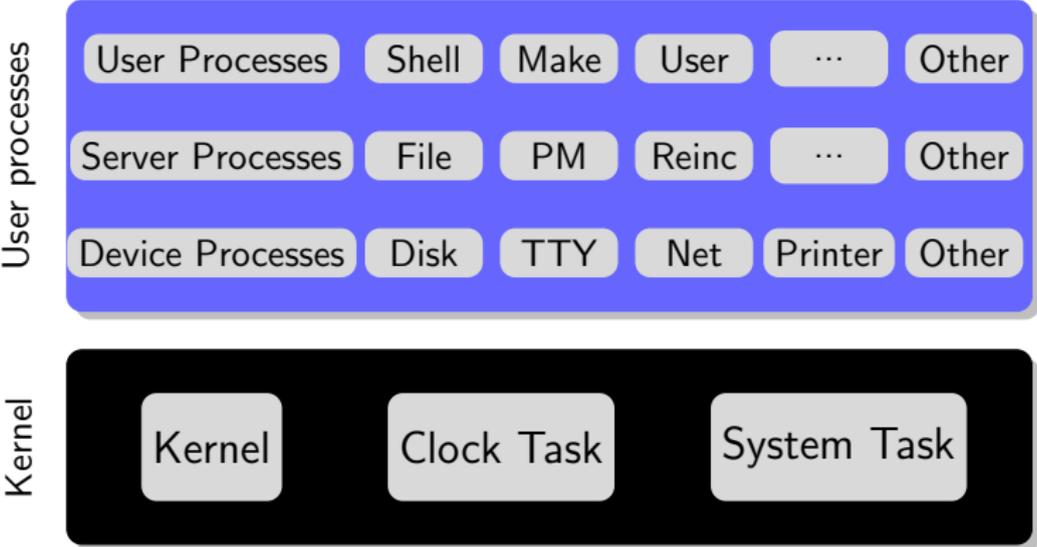
(b) Per finding and fixing difficulty, and impact likelihood

Figure: Average fault lifespans [14]

Break

- ▶ Faults are an issue.
- ▶ Hardware-related stuff is worst.
- ▶ Now what can the OS do about it?

Minix3 – A Fault-tolerant OS



Minix3: Fault Tolerance¹

- ▶ Address Space Isolation
 - ▶ Applications only access private memory
 - ▶ Faults do not spread to other components
- ▶ User-level OS services
 - ▶ Principle of Least Privilege
 - ▶ Fine-grain control over resource access
 - ▶ e.g., DMA only for specific drivers
- ▶ Small components
 - ▶ Easy to replace (micro-reboot)

¹Jorrit N Herder et al. "Fault isolation for device drivers". In: *DSN*. 2009, pp. 33–42.

Minix3: Fault Detection

- ▶ Fault model: transient errors caused by software bugs
- ▶ Fix: Component restart
- ▶ *Reincarnation server* monitors components
 - ▶ Program termination (crash)
 - ▶ CPU exception (div by 0)
 - ▶ Heartbeat messages
- ▶ Users may also indicate that something is wrong

Repair

- ▶ Restarting a component is insufficient:
 - ▶ Applications may *depend* on restarted component
 - ▶ After restart, *component state* is lost
- ▶ Minix3: explicit mechanisms
 - ▶ Reincarnation server signals applications about restart
 - ▶ Applications store state at data store server
 - ▶ In any case: program interaction needed
 - ▶ Restarted app: store/recover state
 - ▶ User apps: recover server connection

OSIRIS: Transparent recovery in MINIX²

```
/* initialization */
while (true) {
  receive(&endpoint, &request);
  switch (request.type) {
    case REQ_TYPE_x:
      reply = req_handler_x(request);
      break;
    case REQ_TYPE_y:
      reply = req_handler_y(request);
      break;
    /* ... */
  }
  if (reply) send(endpoint, reply);
}
```

- ▶ Target typical server architecture
- ▶ Local checkpoints
- ▶ Recovery windows
- ▶ Compiler assisted state recording

OSIRIS: Results

Server	Recovery coverage (%)	
	<i>Pessimistic</i>	<i>Enhanced</i>
PM	54.9	61.7
VFS	72.3	72.3
VM	64.6	64.6
DS	47.1	92.8
RS	49.4	50.5
<i>Weighted average</i>	<i>57.7</i>	<i>68.4</i>

Figure: Percentage of time inside recovery window

Recovery mode	<i>Pass</i>	<i>Fail</i>	<i>Shutdown</i>	<i>Crash</i>
Stateless	19.6%	0.0%	0.0%	80.4%
Naive	20.6%	2.4%	0.0%	77.0%
Pessimistic	18.5%	0.0%	81.3%	0.2%
Enhanced	25.6%	6.5%	66.1%	1.9%

Figure: Survivability under random fault injection

L4ReAnimator: Restart on L4Re³

- ▶ L4Re Applications
 - ▶ Loader component: ned
 - ▶ Detects application termination: parent signal
 - ▶ Restart: re-execute Lua init script (or parts of it)
 - ▶ Problem after restart: capabilities
 - ▶ No single component knows everyone owning a capability to an object
 - ▶ Minix3 signals won't work

³Dirk Vogt, Björn Döbel, and Adam Lackorzynski. “Stay strong, stay safe: Enhancing reliability of a secure operating system”. In: *Workshop on Isolation and Integration for Dependable Systems*. 2010, pp. 1–10.

L4ReAnimator: Lazy recovery

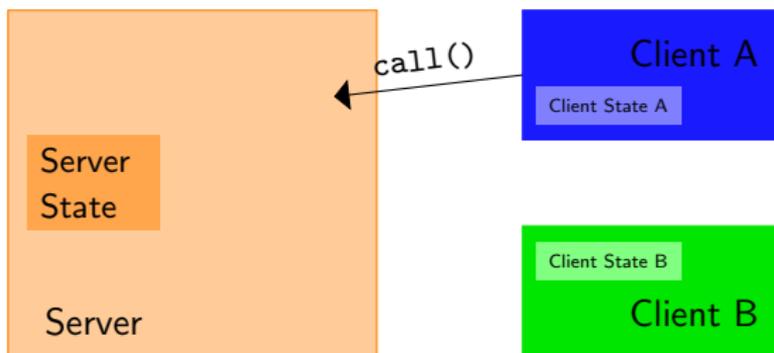
- ▶ Only the application itself can detect that a capability vanished
- ▶ Kernel raises *Capability fault*
- ▶ Application needs to re-obtain the capability: execute *capability fault handler*
- ▶ Capfault handler: application-specific
 - ▶ Create new communication channel
 - ▶ Restore session state
- ▶ Programming model:
 - ▶ Capfault handler provided by server implementor
 - ▶ Handling transparent for application developer
 - ▶ *Semi-transparency*

Break

- ▶ Minix3 fault tolerance
 - ▶ Architectural Isolation
 - ▶ Explicit monitoring and notifications
- ▶ L4ReAnimator
 - ▶ semi-transparent restart in a capability-based system
- ▶ Next: CuriOS
 - ▶ smart session state handling

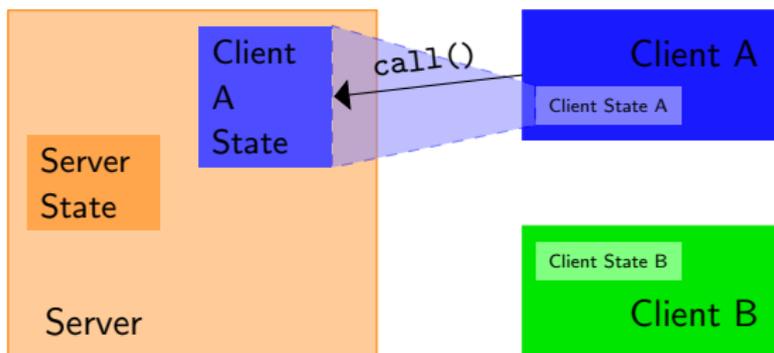
CuriOS: Protecting Sessions

- ▶ SSR gets mapped only when a client actually invokes the server
- ▶ Solves another problem: failure while handling A's request will never corrupt B's session state



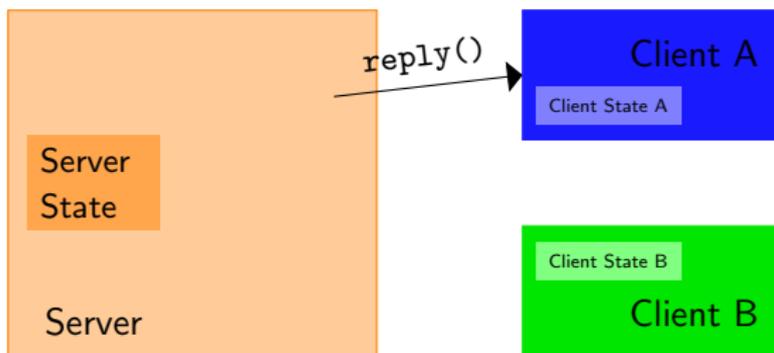
CuriOS: Protecting Sessions

- ▶ SSR gets mapped only when a client actually invokes the server
- ▶ Solves another problem: failure while handling A's request will never corrupt B's session state



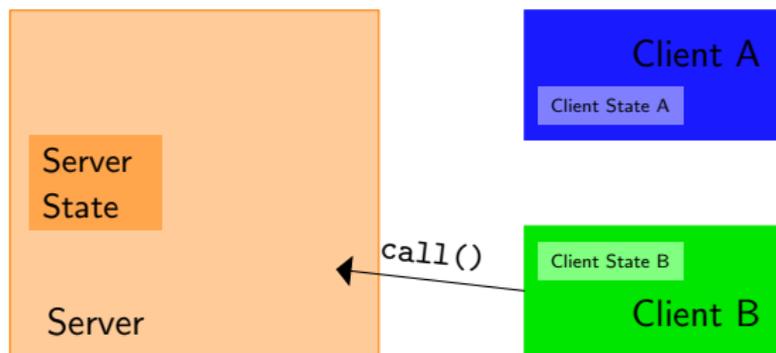
CuriOS: Protecting Sessions

- ▶ SSR gets mapped only when a client actually invokes the server
- ▶ Solves another problem: failure while handling A's request will never corrupt B's session state



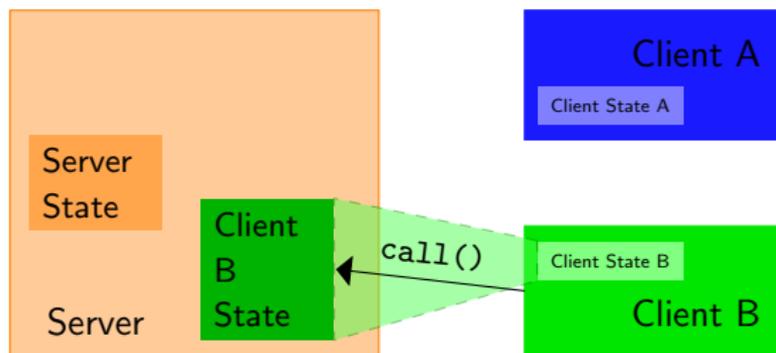
CuriOS: Protecting Sessions

- ▶ SSR gets mapped only when a client actually invokes the server
- ▶ Solves another problem: failure while handling A's request will never corrupt B's session state



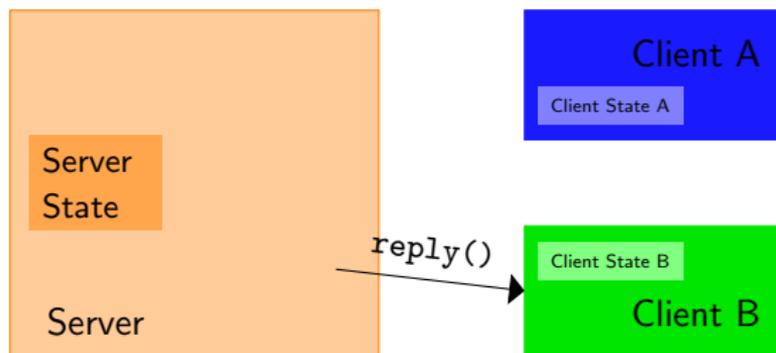
CuriOS: Protecting Sessions

- ▶ SSR gets mapped only when a client actually invokes the server
- ▶ Solves another problem: failure while handling A's request will never corrupt B's session state



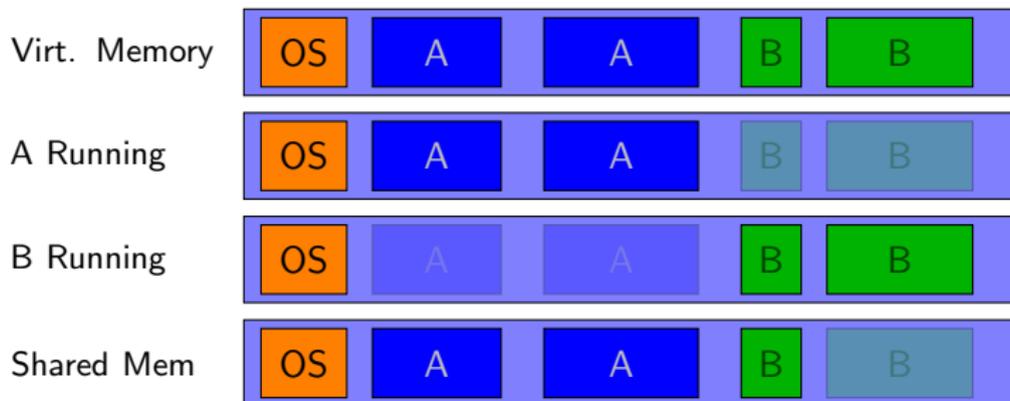
CuriOS: Protecting Sessions

- ▶ SSR gets mapped only when a client actually invokes the server
- ▶ Solves another problem: failure while handling A's request will never corrupt B's session state



CuriOS: Transparent Restart

- ▶ CuriOS is a *Single-Address-Space OS*:
 - ▶ Every application runs on the same page table (with modified access rights)



Transparent Restart

- ▶ Single Address Space
 - ▶ Each object has unique address
 - ▶ Identical in all programs
 - ▶ Server := C++ object
- ▶ Restart
 - ▶ Replace old C++ object with new one
 - ▶ Reuse previous memory location
 - ▶ References in other applications remain valid
 - ▶ OS blocks access during restart

seL4: Formal verification of an OS kernel⁵

- ▶ seL4: <https://sel4.systems/>
- ▶ Formally verify that system adheres to specification
- ▶ Microkernel design allows to separate components easier
- ▶ Hence verification process is easier

⁵Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *SOSP. 2009*, pp. 207–220.

Verification of a microkernel

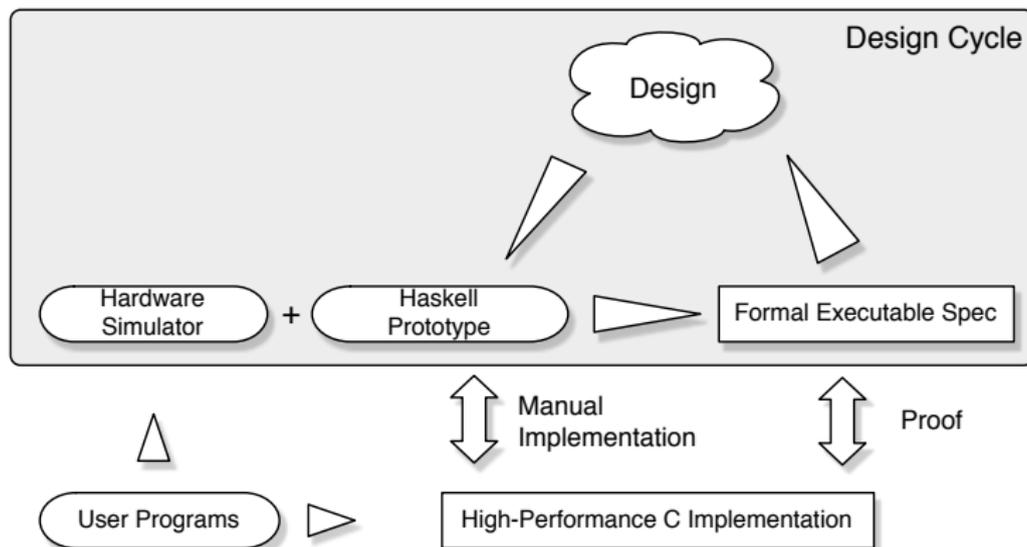


Figure: The seL4 design process [12]

Refinement of verification

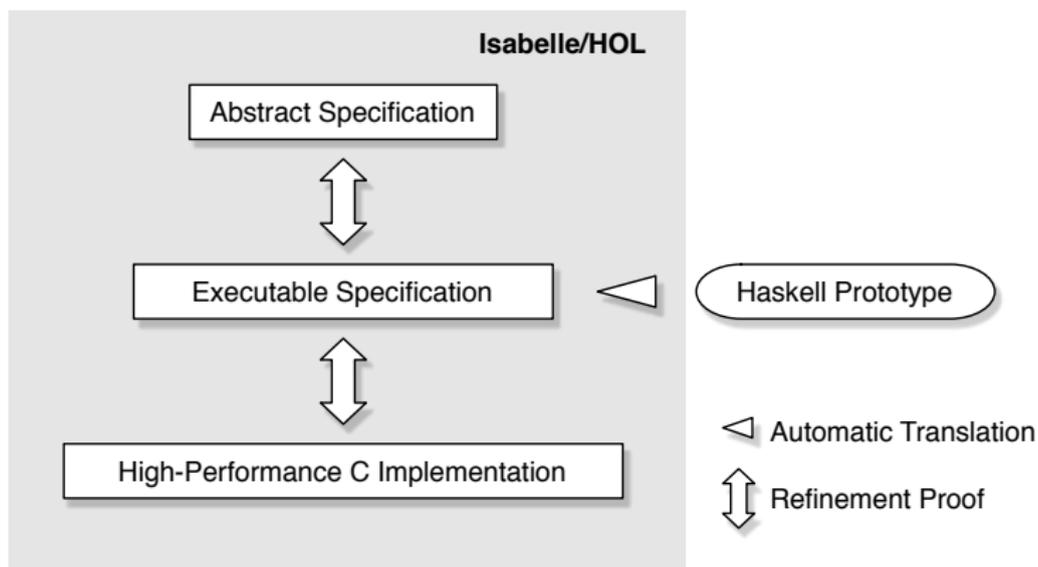


Figure: Refinement layers in the verification of seL4 [12]

Break

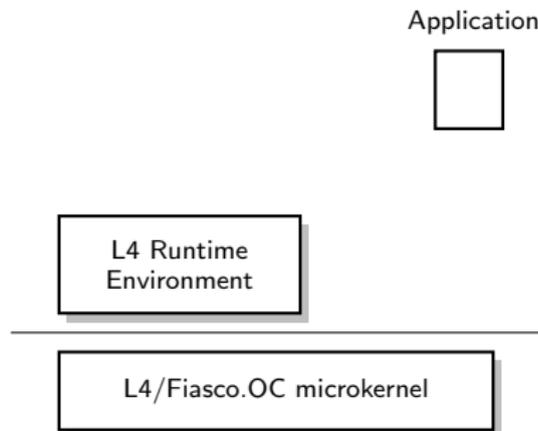
- ▶ seL4
 - ▶ Assumes correctness of compiler, assembly code, and hardware
 - ▶ DMA over IOMMU
 - ▶ Architectures: arm, x86
 - ▶ Virtualization
 - ▶ Future: Verification on multicores

- ▶ All these frameworks only deal with software errors.
- ▶ What about hardware faults?

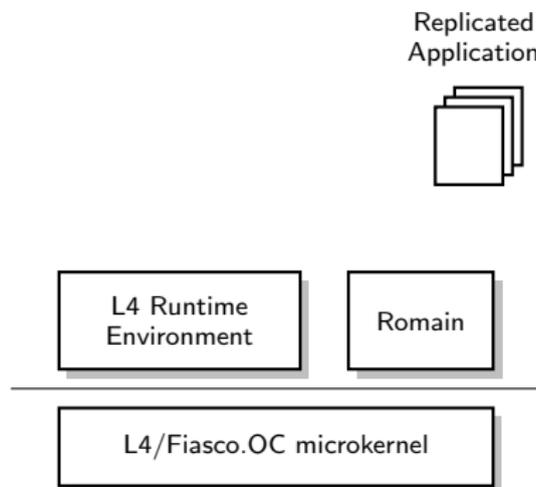
Transient Hardware Faults

- ▶ Radiation-induced soft errors
 - ▶ Mainly an issue in avionics+space?
- ▶ DRAM errors in large data centers
 - ▶ Google study: >2% failing DRAM DIMMs per year [15]
 - ▶ ECC insufficient [11]
- ▶ Decreasing transistor sizes → higher rate of errors in CPU functional units [6]

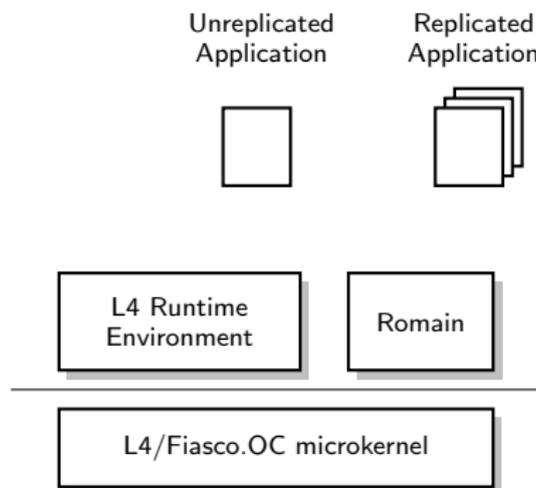
Transparent Replication as OS Service [8, 7]



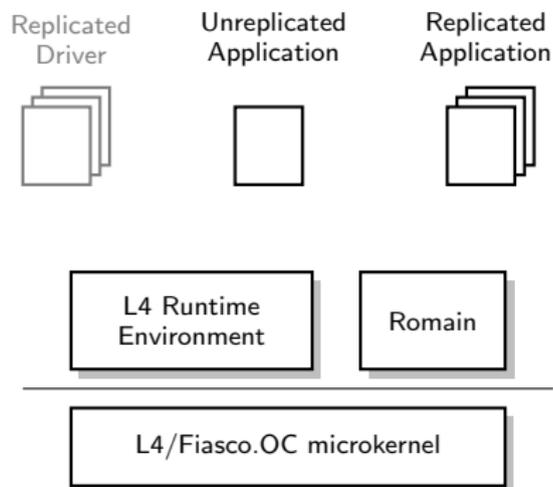
Transparent Replication as OS Service [8, 7]



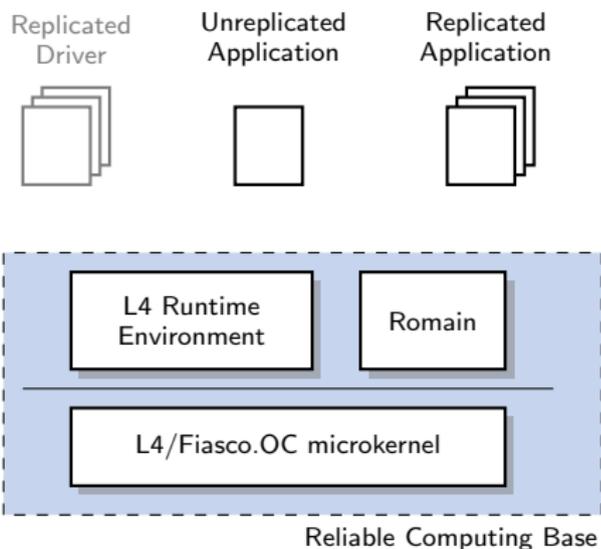
Transparent Replication as OS Service [8, 7]



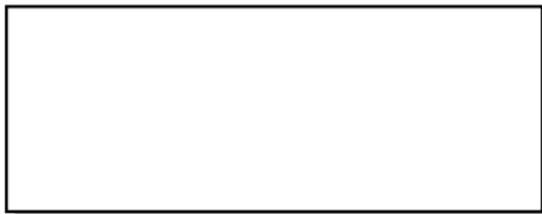
Transparent Replication as OS Service [8, 7]



Transparent Replication as OS Service [8, 7]

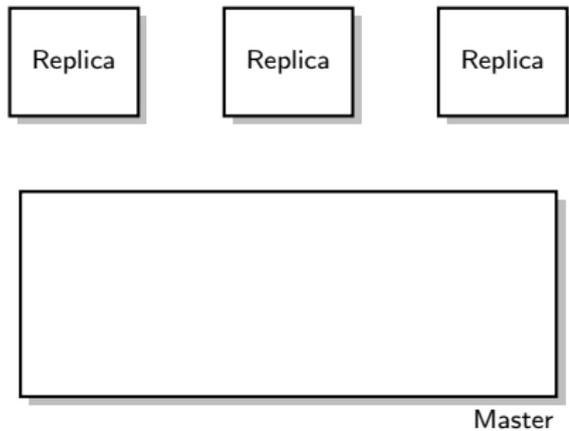


Romain: Structure

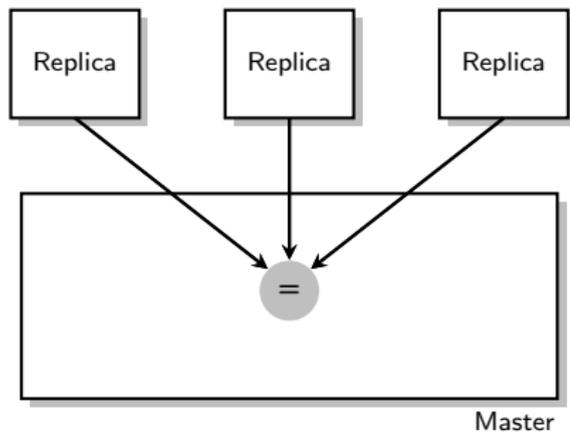


Master

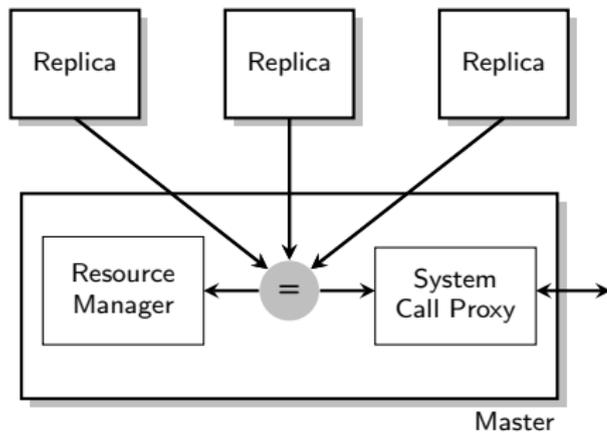
Romain: Structure



Romain: Structure



Romain: Structure

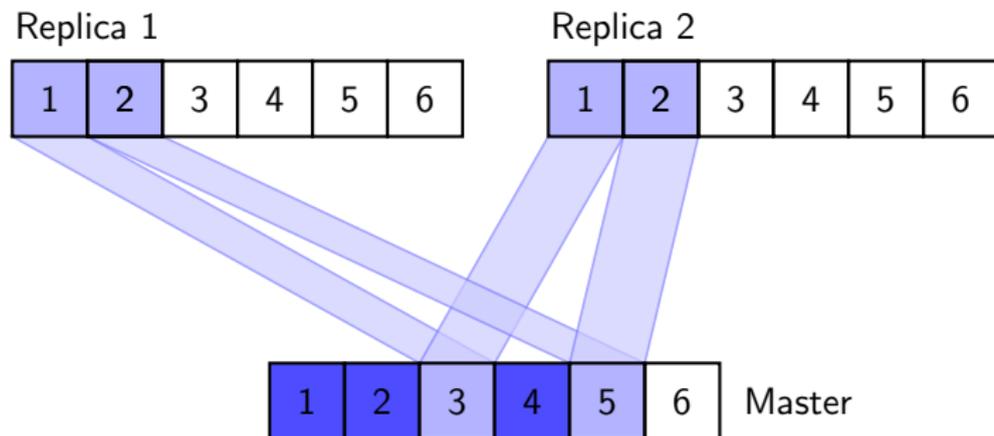


Resource Management: Capabilities

Replica 1

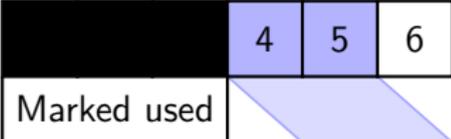
1	2	3	4	5	6
---	---	---	---	---	---

Resource Management: Capabilities

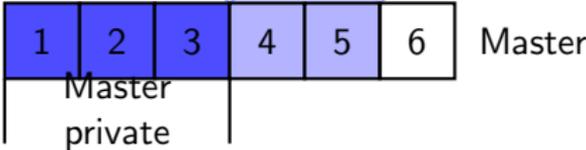


Partitioned Capability Tables

Replica 1

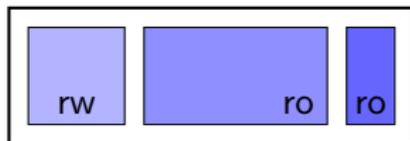


Replica 2

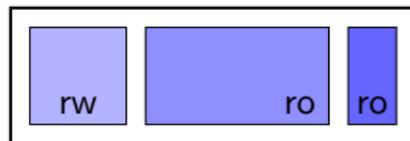


Replica Memory Management

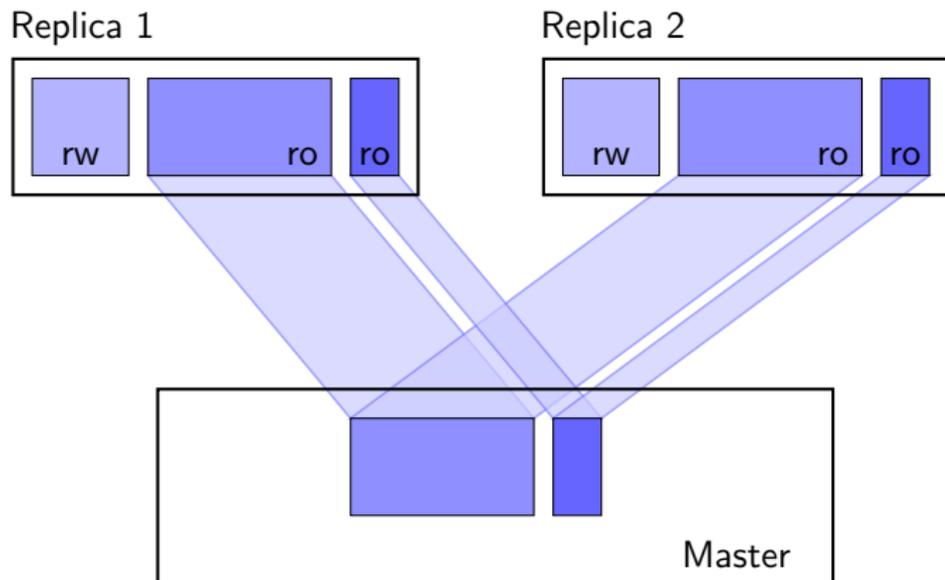
Replica 1



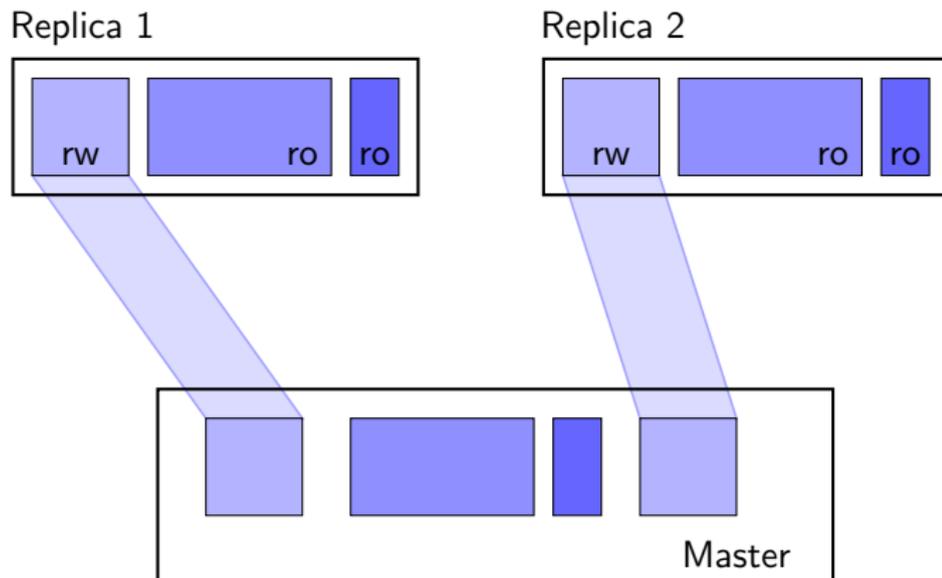
Replica 2



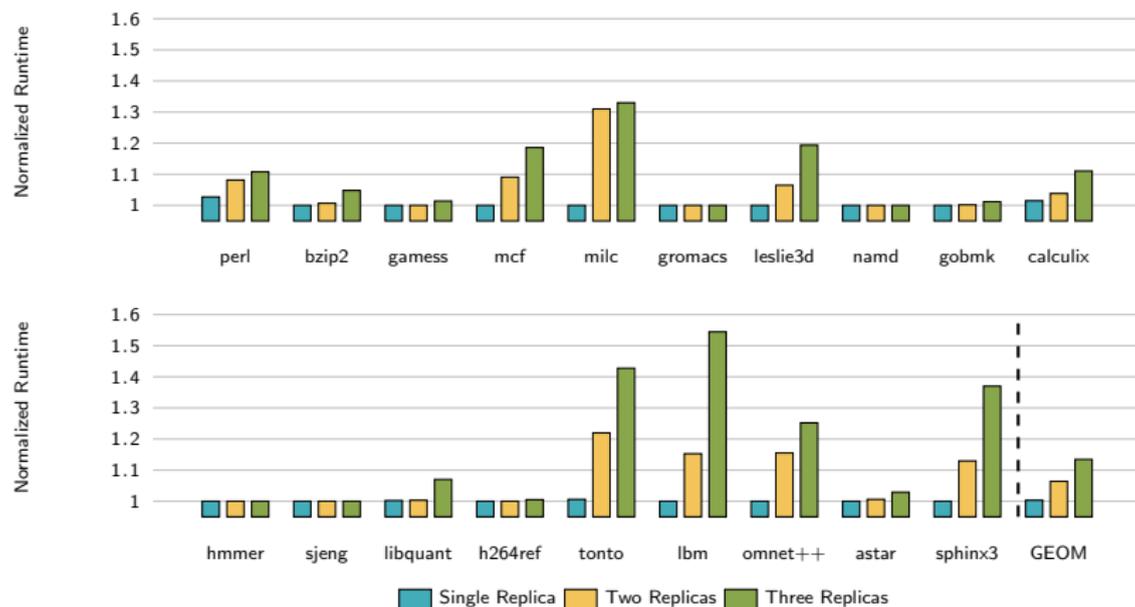
Replica Memory Management



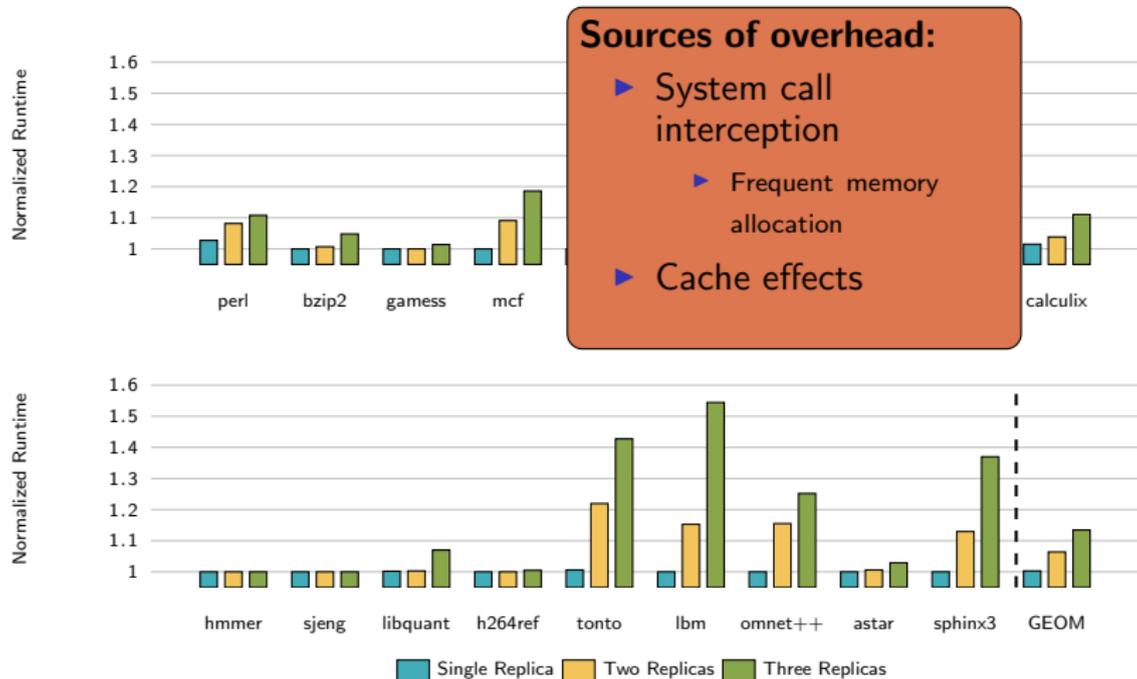
Replica Memory Management



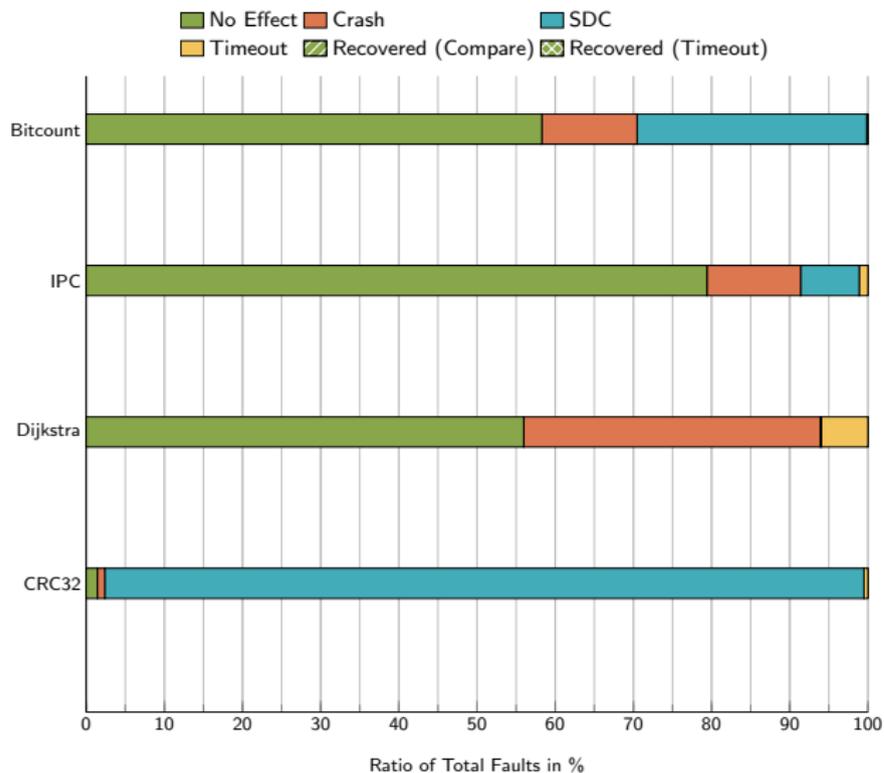
Replicating SPEC CPU 2006 [9]



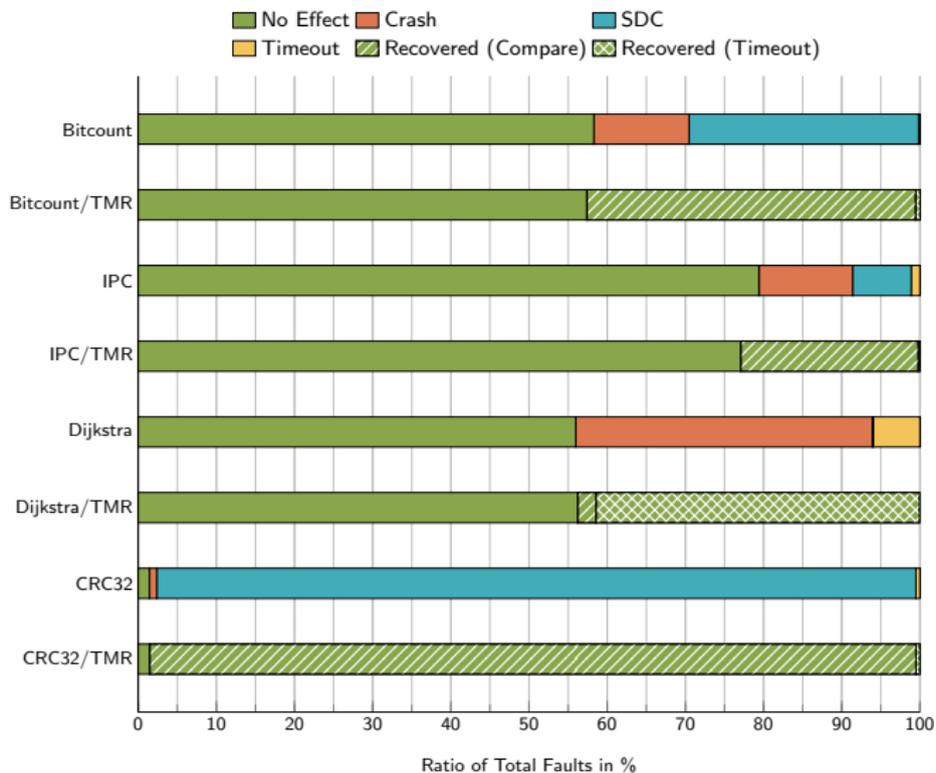
Replicating SPEC CPU 2006 [9]



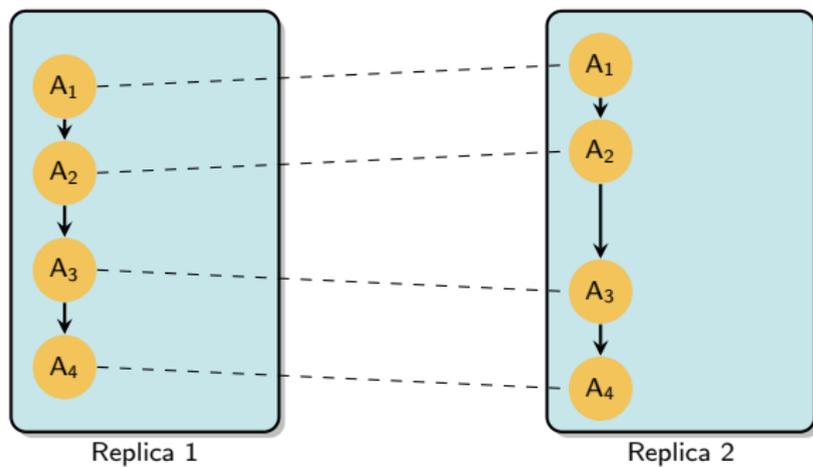
Error Coverage [9]



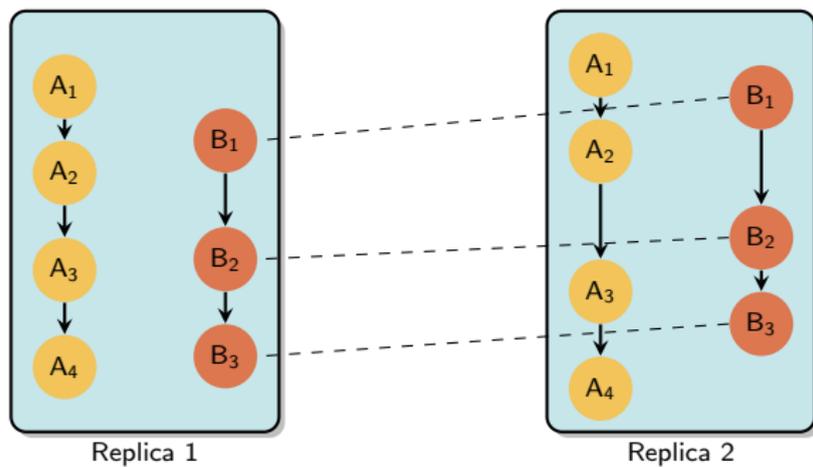
Error Coverage [9]



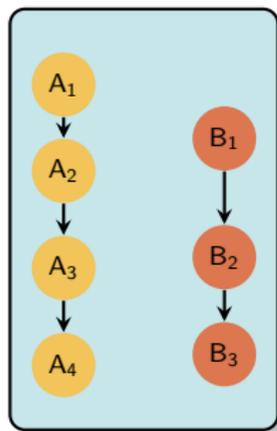
How About Multithreading?



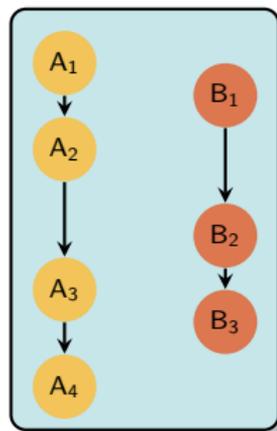
How About Multithreading?



How About Multithreading?

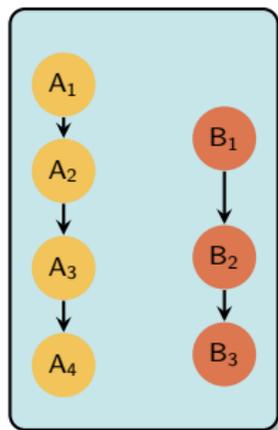


Replica 1

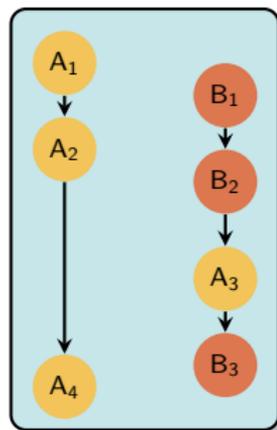


Replica 2

Problem: Nondeterminism



Replica 1



Replica 2

Solution: Deterministic Multithreading

- ▶ Related work: debugging multithreaded programs
- ▶ **Compiler solutions** [2]:
No support for binary-only software

Solution: Deterministic Multithreading

- ▶ Related work: debugging multithreaded programs
- ▶ **Compiler solutions [2]:**
No support for binary-only software
- ▶ **Workspace-Consistent Memory [1]:**
Requires per-replica and per-thread memory copies

Solution: Deterministic Multithreading

- ▶ Related work: debugging multithreaded programs
- ▶ **Compiler solutions [2]:**
No support for binary-only software
- ▶ **Workspace-Consistent Memory [1]:**
Requires per-replica and per-thread memory copies
- ▶ **Lock-Based Determinism**
 - ▶ Reuse ideas from Kendo [13]

Solution: Deterministic Multithreading

- ▶ Related work: debugging multithreaded programs
- ▶ **Compiler solutions** [2]:
No support for binary-only software
- ▶ **Workspace-Consistent Memory** [1]:
Requires per-replica and per-thread memory copies
- ▶ **Lock-Based Determinism**
 - ▶ Reuse ideas from Kendo [13]
 - ▶ **Only for lock-based software!**

Enforced Determinism

- ▶ Adapt libpthread
 - ▶ `pthread_mutex_lock`
 - ▶ `pthread_mutex_unlock`
 - ▶ `--pthread_lock`
 - ▶ `--pthread_unlock`
- ▶ Lock operations reflected to Romain master
- ▶ Master enforces lock ordering

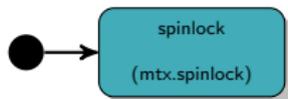
Cooperation: Lock Acquisition

```
lock_rep(mtx)
```

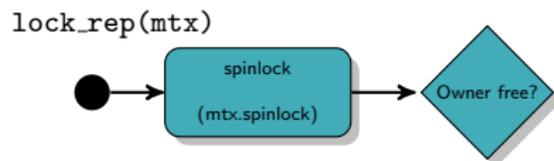


Cooperation: Lock Acquisition

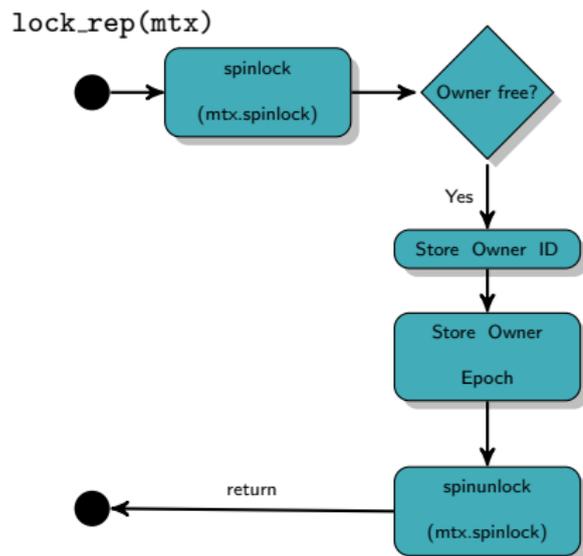
`lock_rep(mtx)`



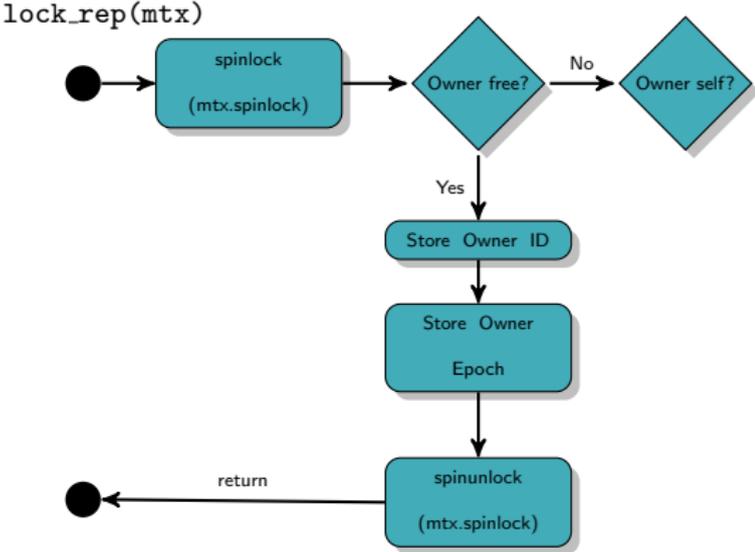
Cooperation: Lock Acquisition



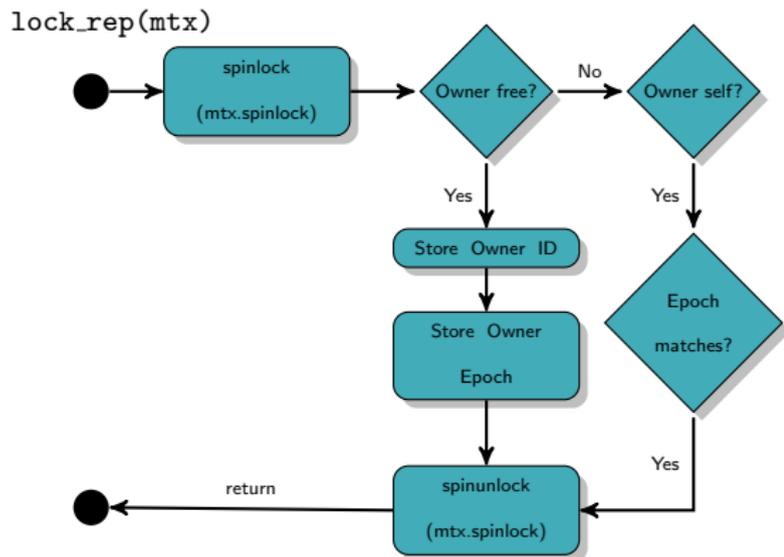
Cooperation: Lock Acquisition



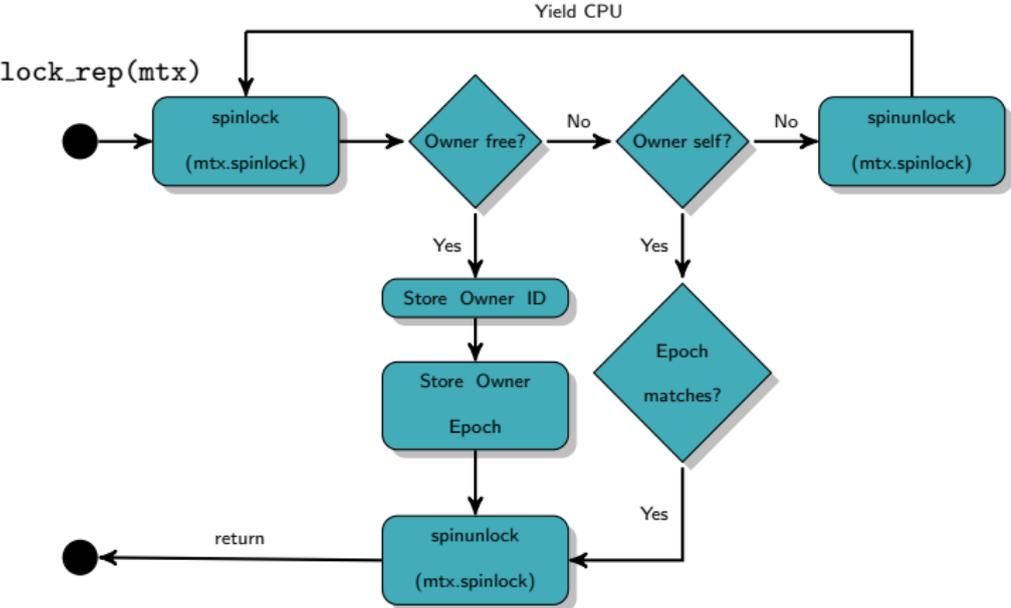
Cooperation: Lock Acquisition



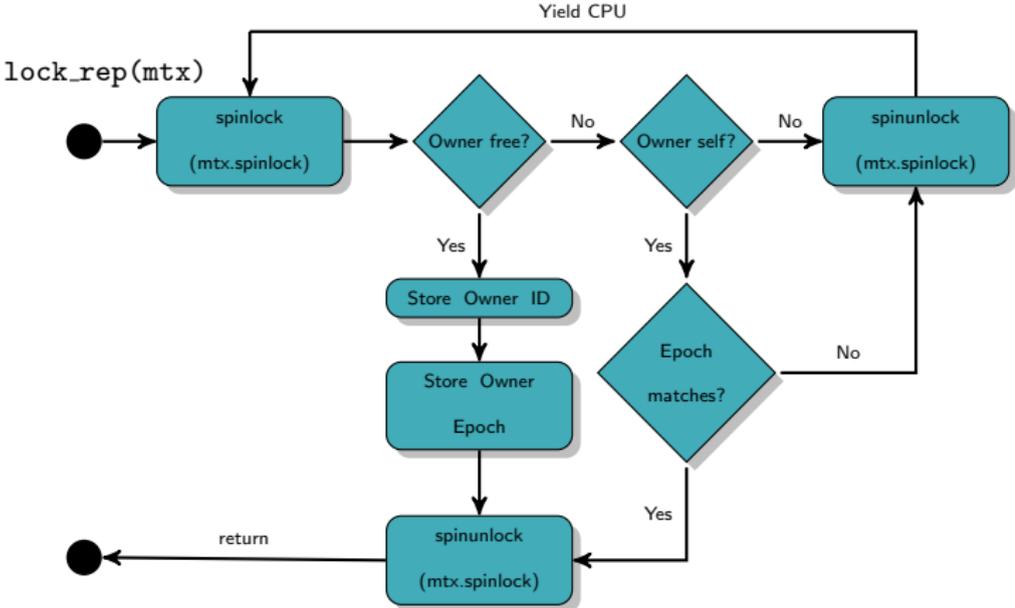
Cooperation: Lock Acquisition



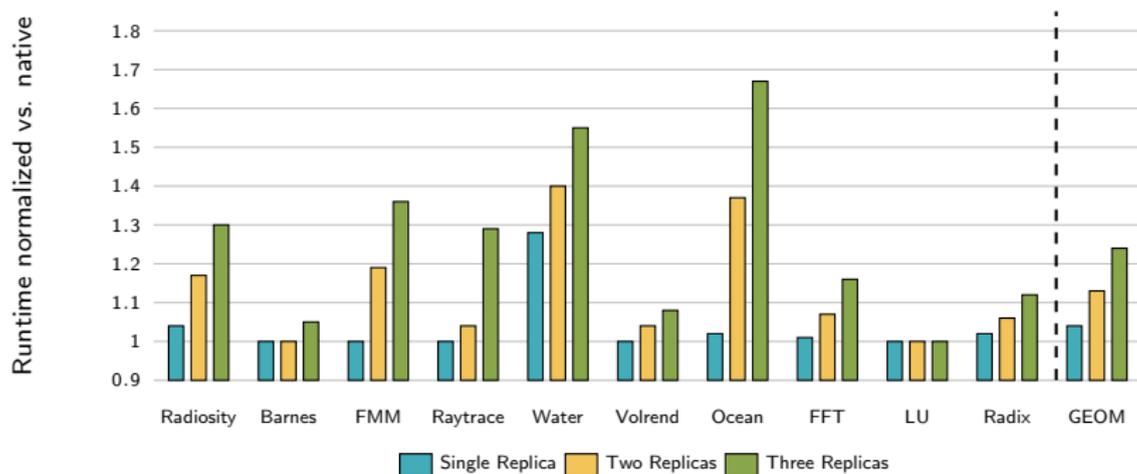
Cooperation: Lock Acquisition



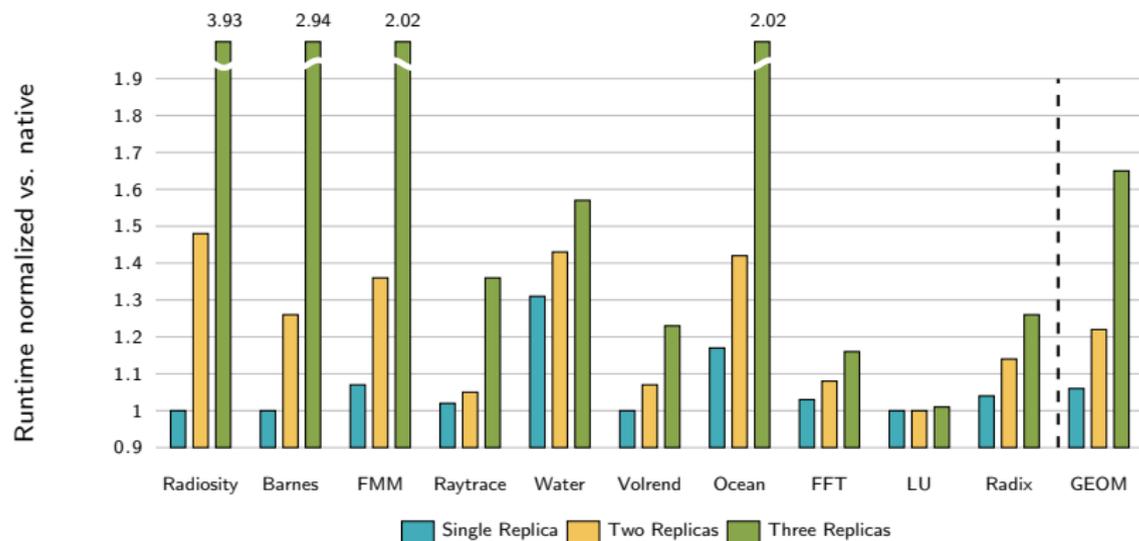
Cooperation: Lock Acquisition



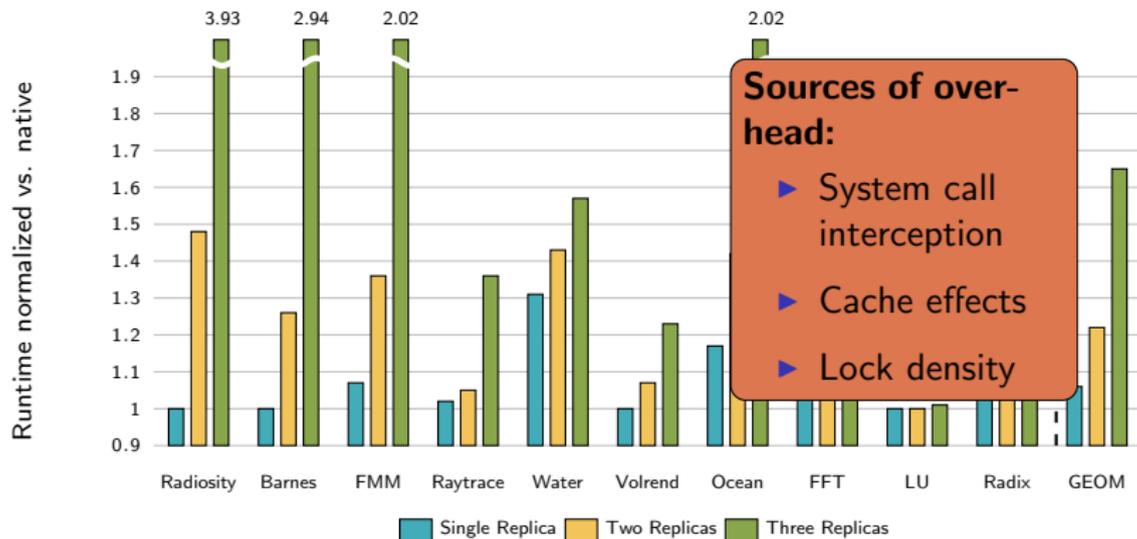
Overhead: SPLASH2, 2 workers [9]



Overhead: SPLASH2, 4 workers



Overhead: SPLASH2, 4 workers



Summary

- ▶ OS-level techniques to tolerate SW and HW faults
- ▶ Address-space isolation
- ▶ Microreboots
- ▶ Various ways of handling session state
- ▶ Replication against hardware errors

Further Reading

- ▶ **Minix3**: Jorrit Herder, Ben Gras,, Philip Homburg, Andrew S. Tanenbaum: *Fault Isolation for Device Drivers*, DSN 2009
- ▶ **CuriOS**: Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle and Roy H. Campbell *CuriOS: Improving Reliability through Operating System Structure*, OSDI 2008
- ▶ **Qmail**: Daniel Bernstein: *Some thoughts on security after ten years of qmail 1.0*
- ▶ **seL4**: Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick and others *Formal verification of an OS kernel*, SOSP 2009
- ▶ **Romain**:
 - ▶ Björn Döbel, Hermann Härtig, Michael Engel: *Operating System Support for Redundant Multithreading*, EMSOFT 2012
 - ▶ Björn Döbel, Hermann Härtig: *Can We Put Concurrency Back Into Redundant Multithreading?*, EMSOFT 2014

Bibliography I



Amittai Aviram et al. “Efficient system-enforced deterministic parallelism”. In: *OSDI*. 2012, pp. 111–119.



Tom Bergan et al. “CoreDet: a compiler and runtime system for deterministic multithreaded execution”. In: *ACM SIGARCH Computer Architecture News*. 2010, pp. 53–64.



Koustubha Bhat et al. “OSIRIS: Efficient and consistent recovery of compartmentalized operating systems”. In: *DSN. IEEE*. 2016, pp. 25–36.



Andy Chou et al. “An empirical study of operating systems errors”. In: *SOSP*. 2001, pp. 73–88.



Francis M David et al. “CuriOS: Improving Reliability through Operating System Structure.”. In: *OSDI*. 2008, pp. 59–72.



Anand Dixit and Alan Wood. “The impact of new technology on soft error rates”. In: *International Reliability Physics Symposium (IRPS)*. 2011, 5B–4.

Bibliography III



Nicolas Palix et al. “Faults in Linux: Ten years later”. In: *ASPLOS*. 2011, pp. 305–318.



Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. “DRAM errors in the wild: a large-scale field study”. In: *SIGMETRICS/Performance*. 2009, pp. 193–204.



Dirk Vogt, Björn Döbel, and Adam Lackorzynski. “Stay strong, stay safe: Enhancing reliability of a secure operating system”. In: *Workshop on Isolation and Integration for Dependable Systems*. 2010, pp. 1–10.