TECHNISCHE UNIVERSITÄT DRESDEN

OPERATING SYSTEMS MEET FAULT TOLERANCE

Microkernel-Based Operating Systems

Björn Döbel

Dresden, 21.01.2015

“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

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: `http://www.coverity.com`)
- Target: Linux (1.0 - 2.4)
  - Where are the errors?
  - How are they distributed?
  - How long do they survive?
  - Du 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

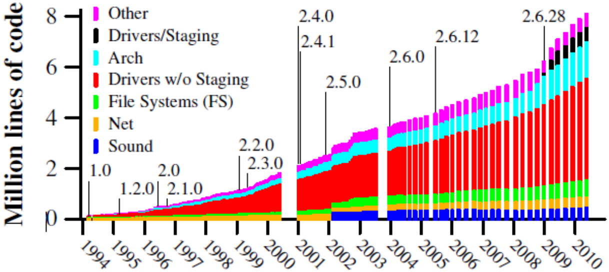
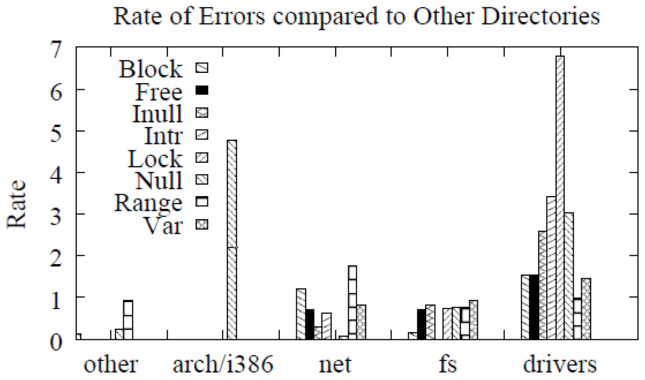
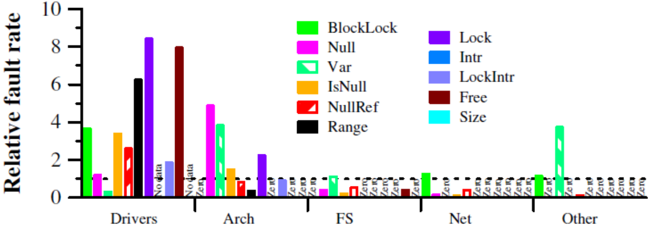


Figure 1. Linux directory sizes (in MLOC)

Fault Rate per Subdirectory (2001)



Fault Rate per Subdirectory (2011)



(b) Rate of faults compared to all other directories

Bug Lifetimes (2011)

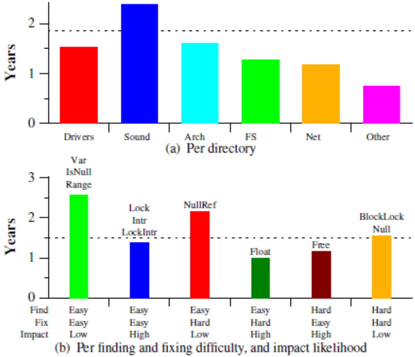
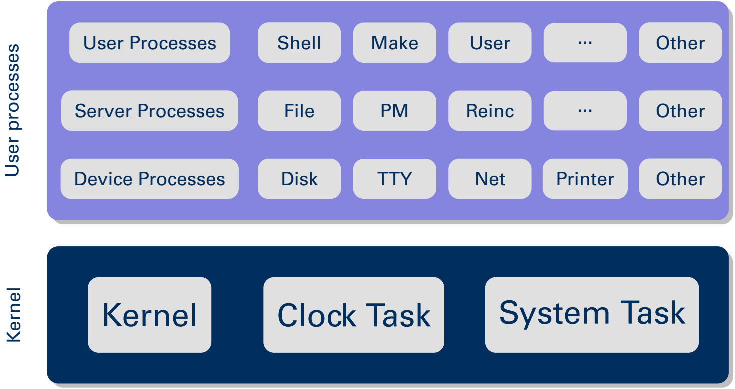


Figure 13. Average fault lifespans (without staging)

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 Tolerance

- 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)

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

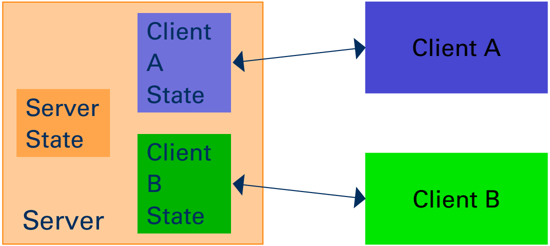
- Restarting a component is insufficient:
  - Applications may *depend* on restarted component
  - After restart, *component state* is lost
- Minix3: explicit mechanisms
  - Reincarnation server signals applications about restart
  - Applications store state at data store server
  - In any case: program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

Break

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

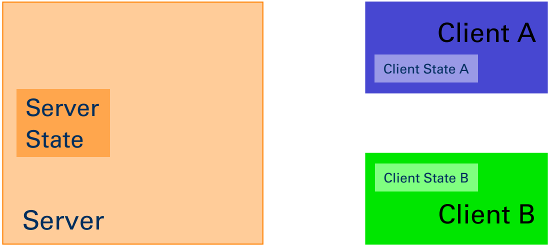
CuriOS: Servers and Sessions

- State recovery is tricky
  - Minix3: Data Store for application data
  - But: applications interact
    - Servers store *session-specific* state
    - Server restart requires potential rollback for every participant



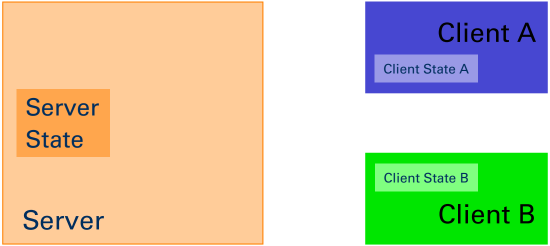
CuriOS: Server State Regions

- CuriOS kernel (CuiK) manages dedicated session memory: *Server State Regions*
- SSRs are managed by the kernel and attached to a client-server connection



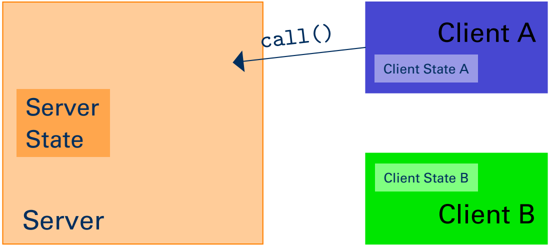
CuriOS: Protecting Sessions

- 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



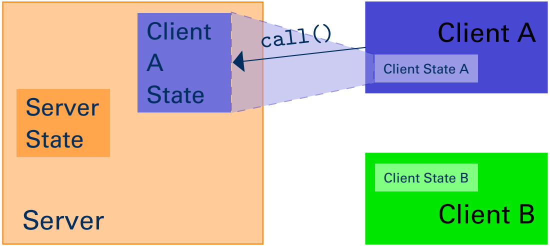
CuriOS: Protecting Sessions

- 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



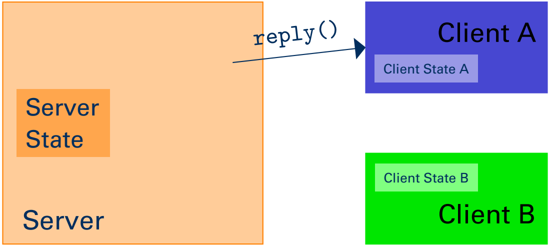
CuriOS: Protecting Sessions

- 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



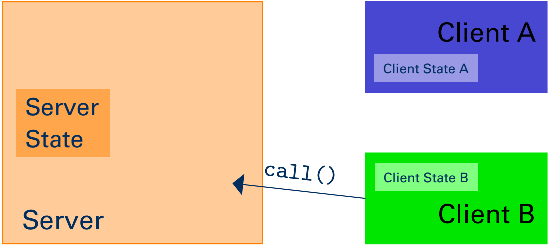
CuriOS: Protecting Sessions

- 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



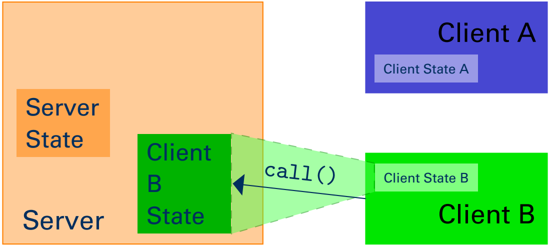
CuriOS: Protecting Sessions

- 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



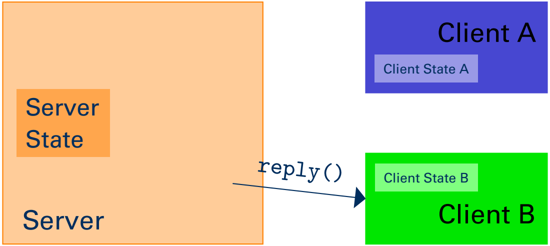
CuriOS: Protecting Sessions

- 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: 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

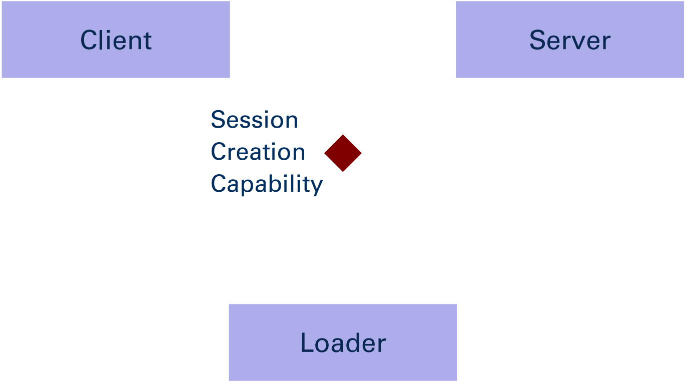
- Single Address Space
  - Each object has unique address
  - Identical in all programs
  - Server := C++ object
- Restart
  - Replace old C++ object with new one
  - Reuse previous memory location
  - References in other applications remain valid
  - OS blocks access during restart

L4ReAnimator: Restart on L4Re

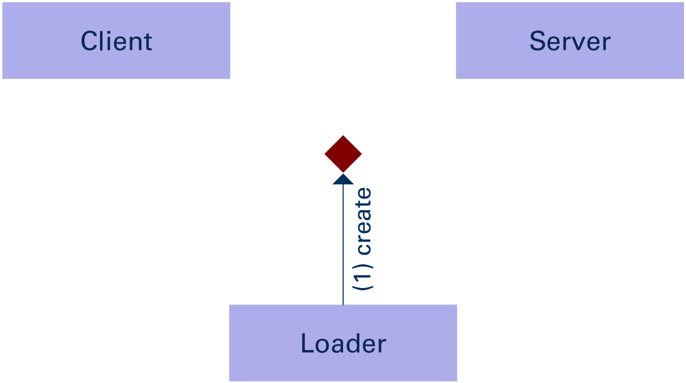
- L4Re Applications
  - Loader component: ned
  - Detects application termination: parent signal
  - Restart: re-execute Lua init script (or parts of it)

  - Problem after restart: capabilities
    - No single component knows everyone owning a capability to an object
    - Minix3 signals won't work

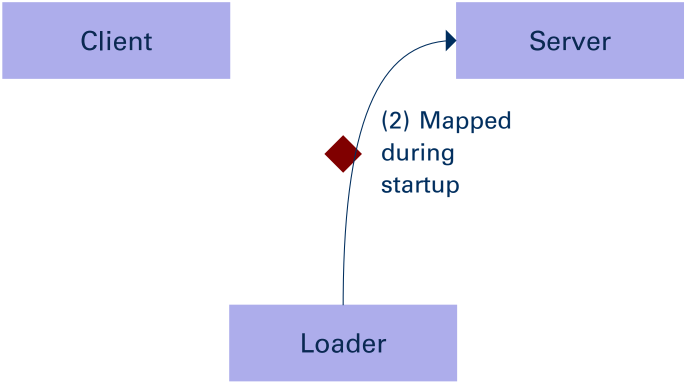
L4Re: Session Creation



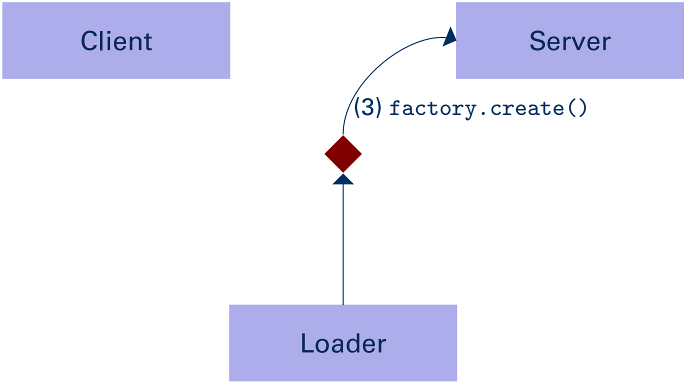
L4Re: Session Creation



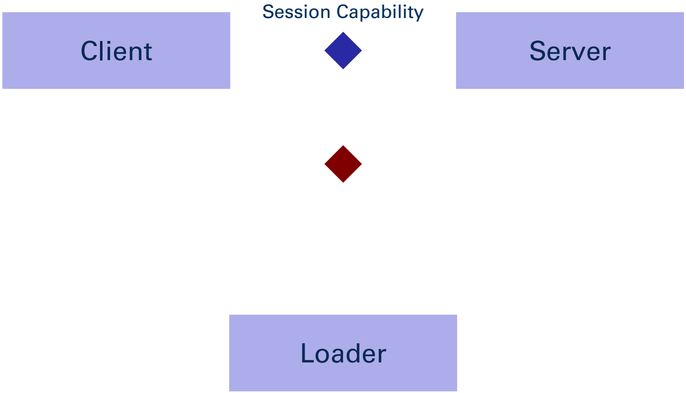
L4Re: Session Creation



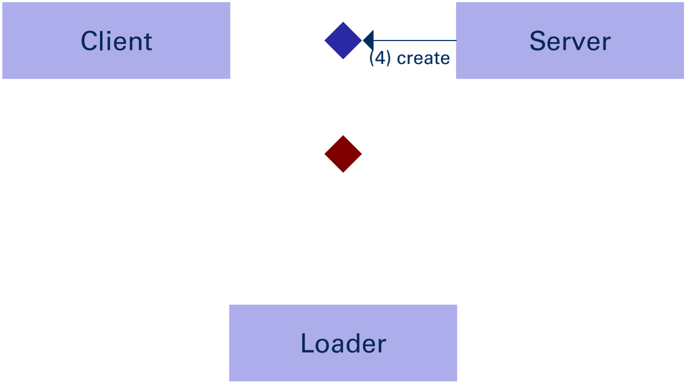
L4Re: Session Creation



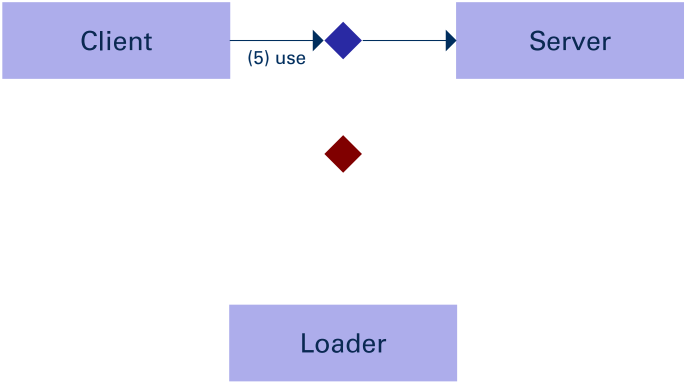
L4Re: Session Creation



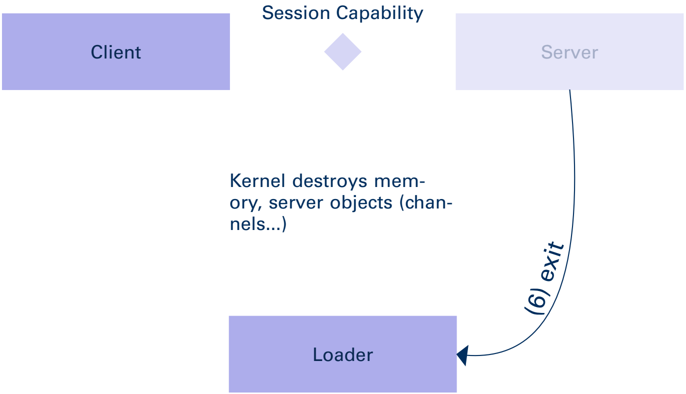
L4Re: Session Creation



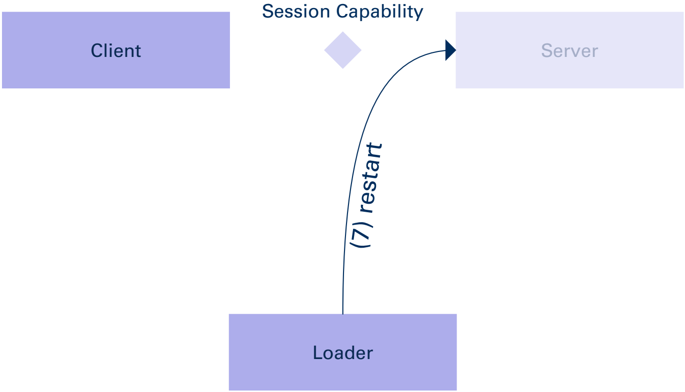
L4Re: Session Creation



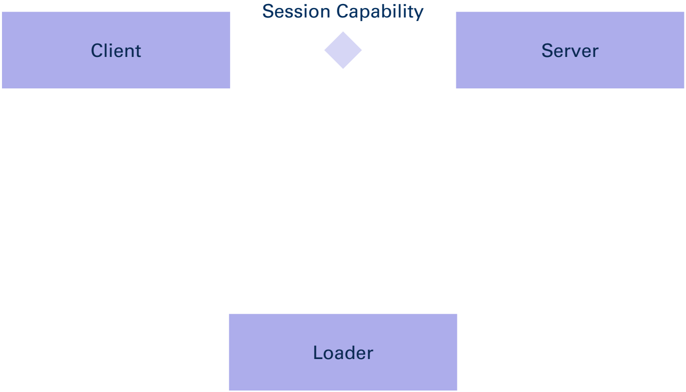
L4Re: Server Crash



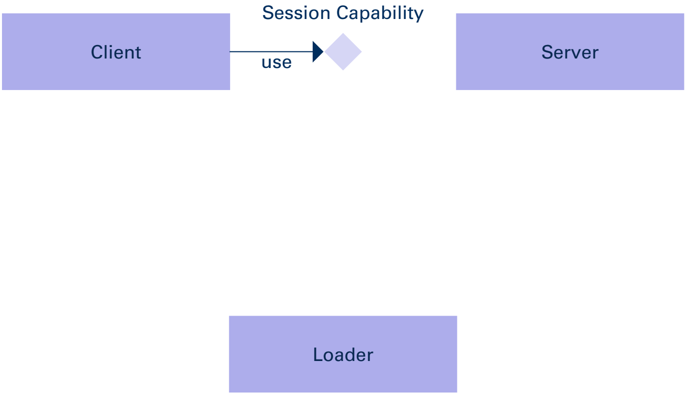
L4Re: Server Crash



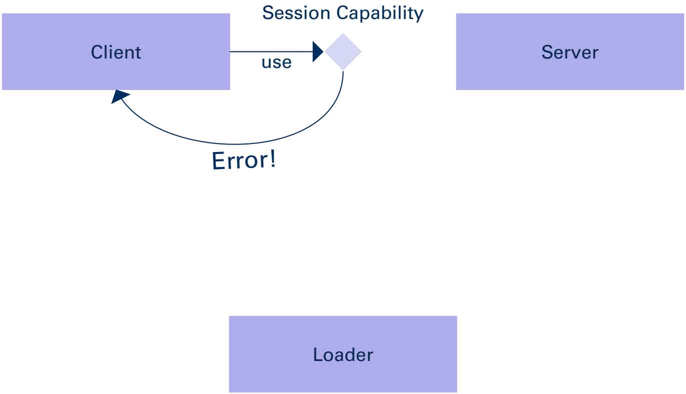
L4Re: Restarted Server



L4Re: Restarted Server



L4Re: Restarted Server



L4ReAnimator

- 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*

L4ReAnimator: Cleanup

- Some channels have resources attached (e.g., frame buffer for graphical console)
- Resource may come from a different resource (e.g., frame buffer from memory manager)
- Resources remain intact (stale) upon crash
- Client ends up using old version of the resource
- Requires additional app-specific knowledge
- *Unmap handler*

Summary

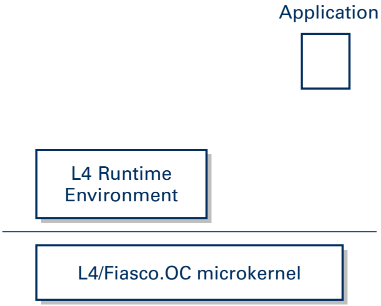
- L4ReAnimator
  - Capfault: Clients detect server restarts lazily
  - Capfault Handler: application-specific knowledge on how to regain access to the server
  - Unmap handler: clean up old resources after restart

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

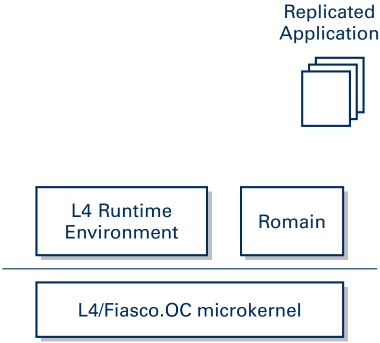
Transient Hardware Faults

- Radiation-induced soft errors
  - Mainly an issue in avionics+space?
- DRAM errors in large data centers
  - Google study: $>$2% failing DRAM DIMMs per year
  - ECC insufficient
- Decreasing transistor sizes $\rightarrow$ higher rate of errors in CPU functional units

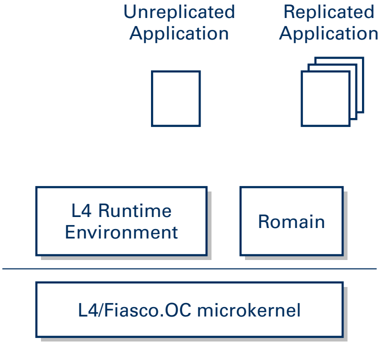
Transparent Replication as OS Service



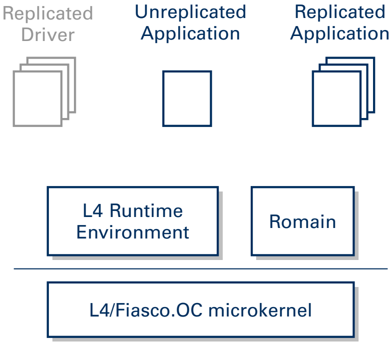
Transparent Replication as OS Service



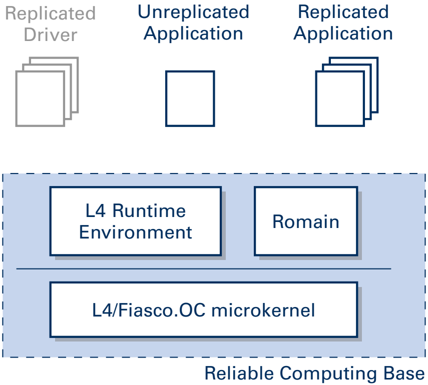
Transparent Replication as OS Service



Transparent Replication as OS Service

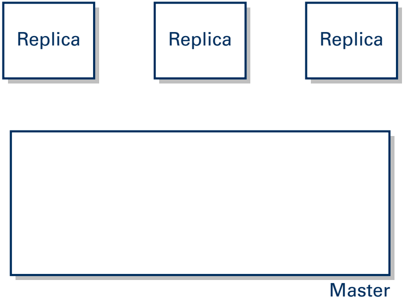


Transparent Replication as OS Service

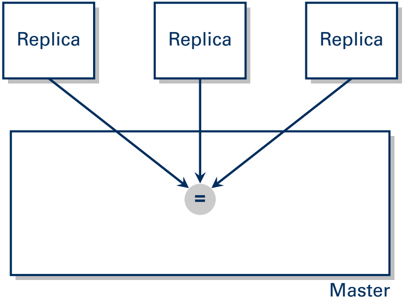


Romain: Structure

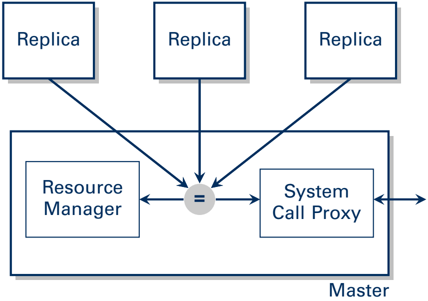
Romain: Structure



Romain: Structure



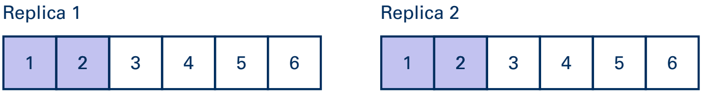
Romain: Structure



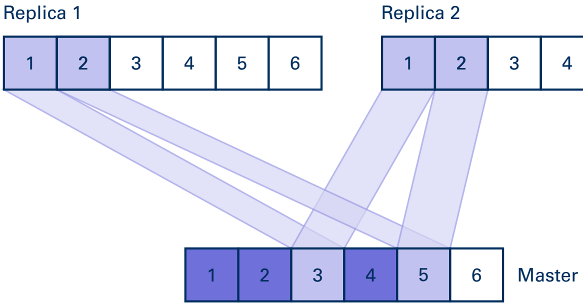
Resource Management: Capabilities

Replica 1

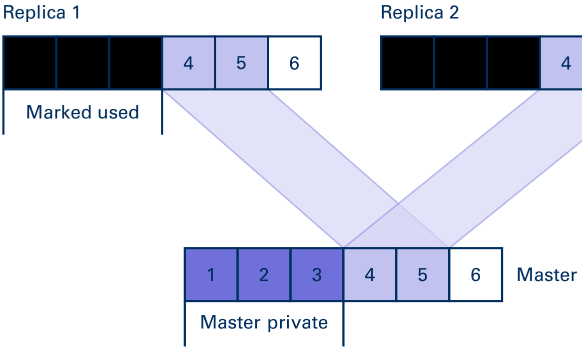
Resource Management: Capabilities



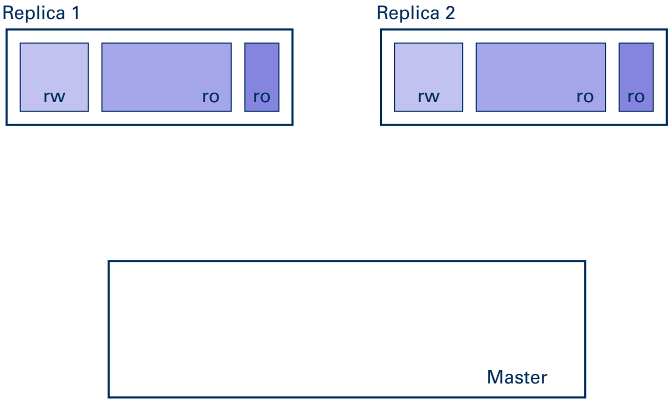
Resource Management: Capabilities



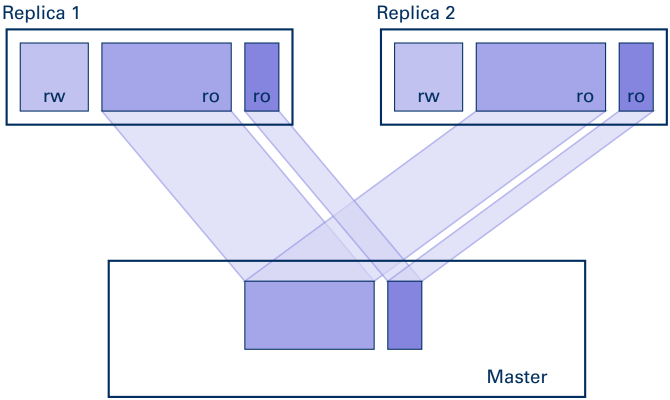
Partitioned Capability Tables



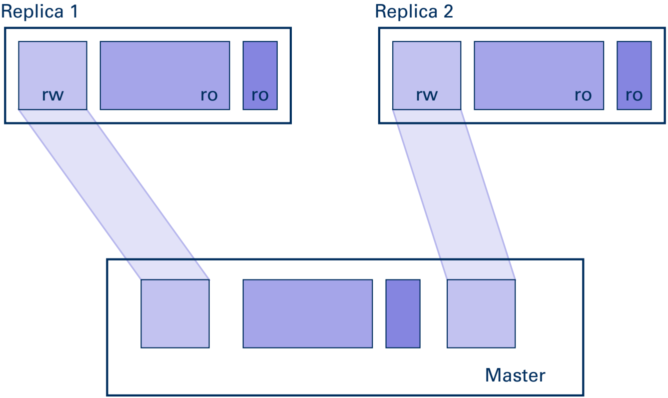
Replica Memory Management



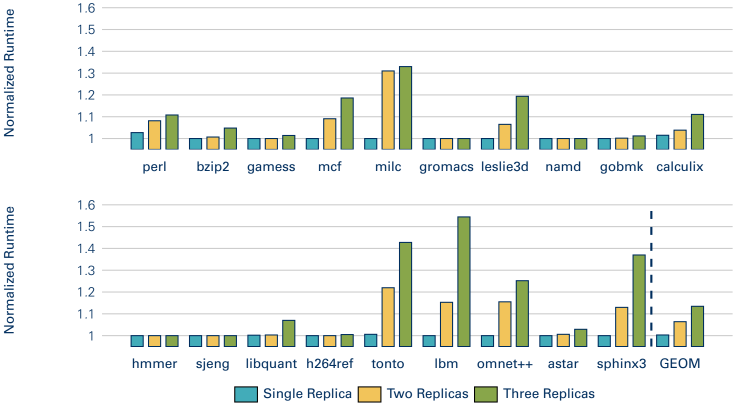
Replica Memory Management



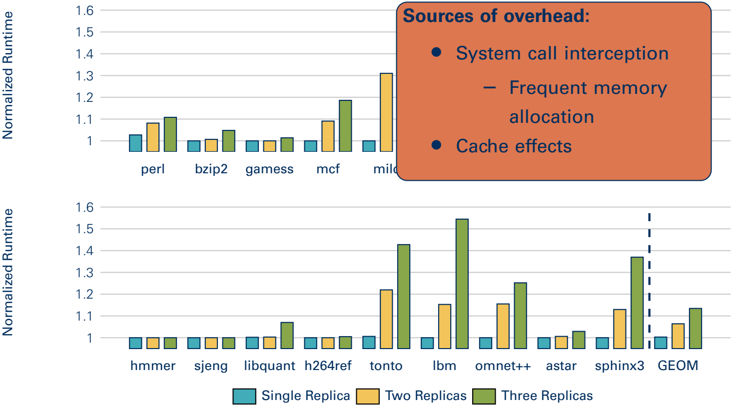
Replica Memory Management



Replicating SPEC CPU 2006



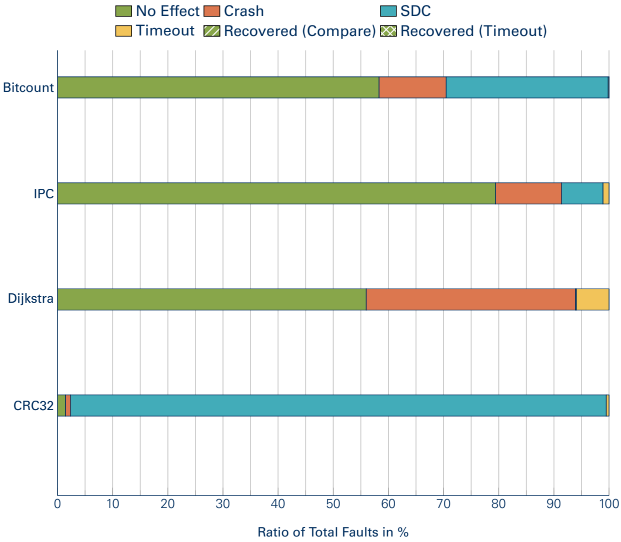
Replicating SPEC CPU 2006



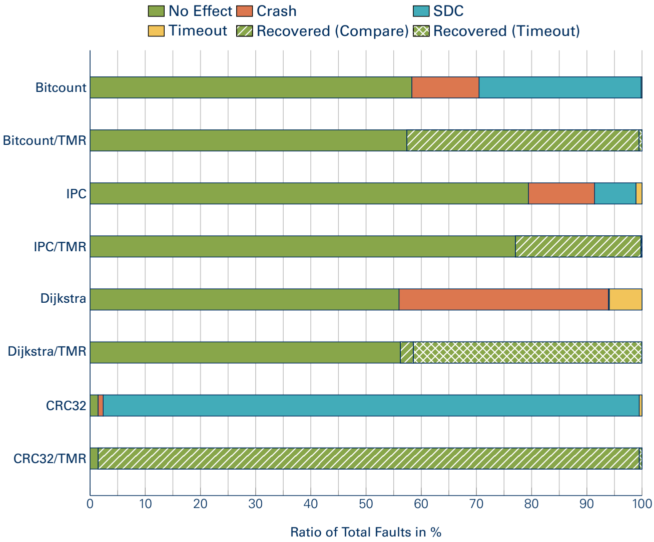
Sources of overhead:

- System call interception
  - Frequent memory allocation
- Cache effects

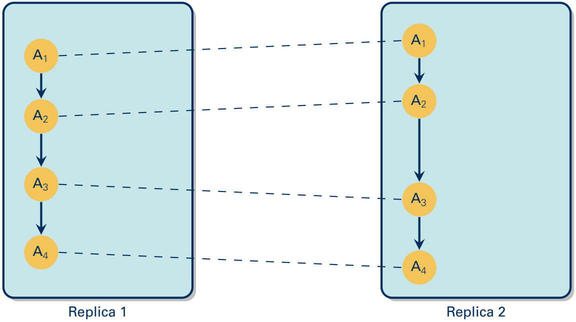
Error Coverage



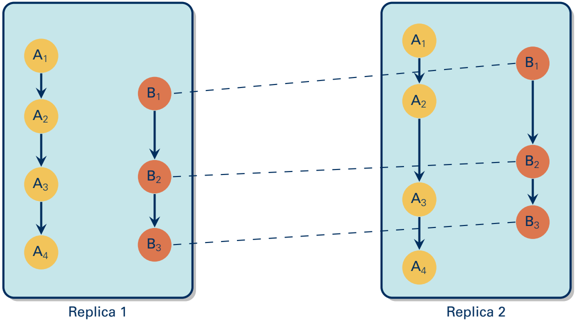
Error Coverage



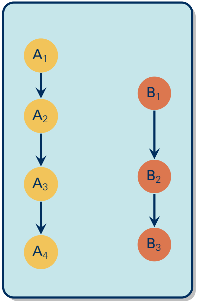
How About Multithreading?



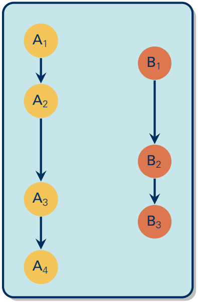
How About Multithreading?



How About Multithreading?

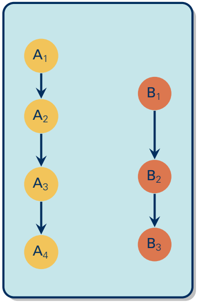


Replica 1

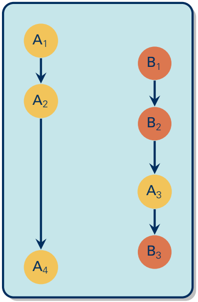


Replica 2

Problem: Nondeterminism



Replica 1



Replica 2

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions:
  No support for binary-only software

Solution: Deterministic Multithreading

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

Solution: Deterministic Multithreading

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

Solution: Deterministic Multithreading

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

Enforced Determinism

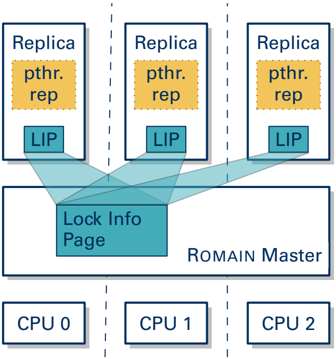
- Adapt `libpthread`: place `INT3` into four functions
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `__pthread_lock`
  - `__pthread_unlock`
- Lock operations reflected to Romain master
- Master enforces lock ordering

Enforced Determinism

- Adapt `libpthread`: place `INT3` into four functions
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `__pthread_lock`
  - `__pthread_unlock`
- Lock operations reflected to Romain master
- Master enforces lock ordering
- 300x overhead for worst-case microbenchmark in TMR!

Cooperative Determinism

- Replication-aware `libpthread`
- Replicas agree on acquisition order w/o master invocation
- Trade-off: `libpthread` becomes single point of failure



Cooperation: Lock Acquisition

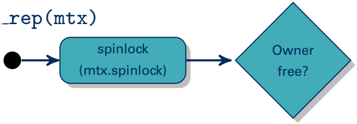
`lock_rep(mtx)`

Cooperation: Lock Acquisition

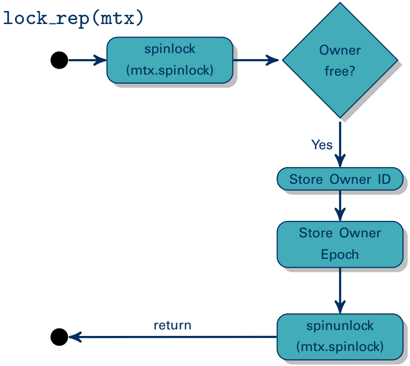
`lock_rep(mtx)`

Cooperation: Lock Acquisition

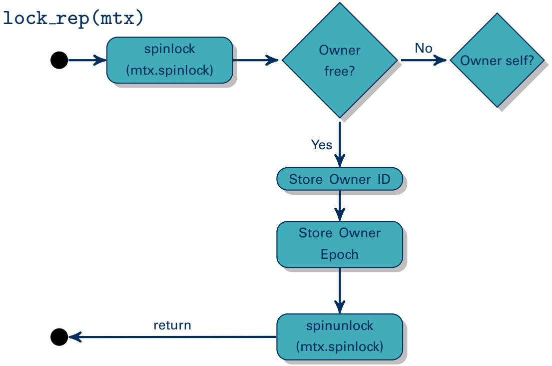
`lock_rep(mtx)`



Cooperation: Lock Acquisition



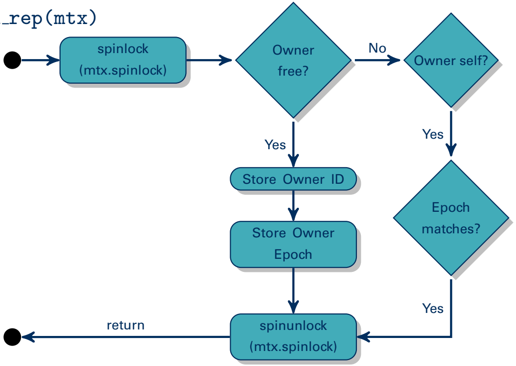
Cooperation: Lock Acquisition



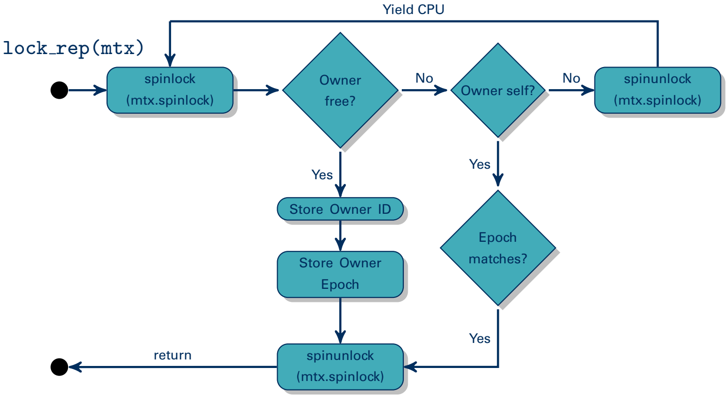
TECHNISCHE UNIVERSITÄT DRESDEN

Cooperation: Lock Acquisition

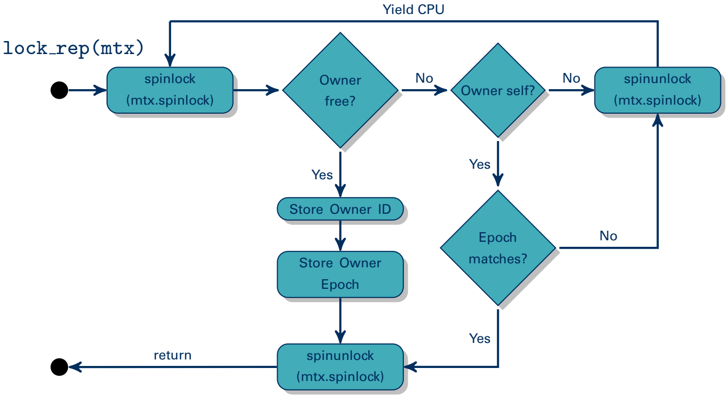
`lock_rep(mtx)`



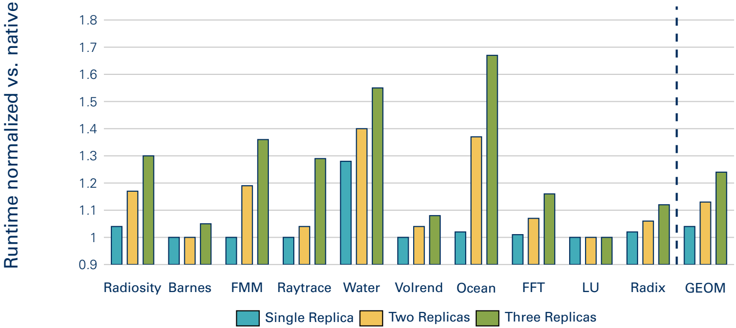
Cooperation: Lock Acquisition



Cooperation: Lock Acquisition

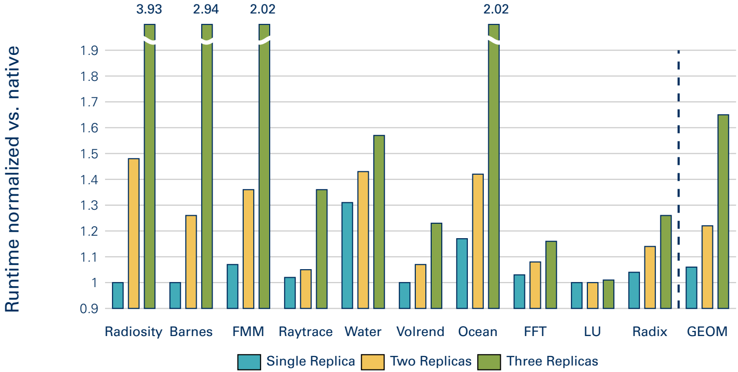


Overhead: SPLASH2, 2 workers

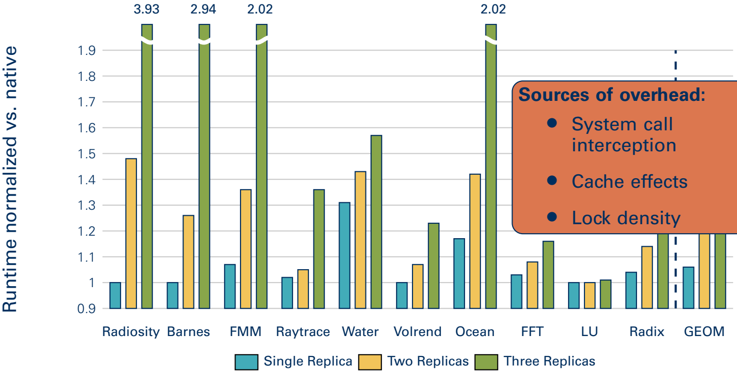


B. Döbel, H. Härtig: **Can we put Concurrency Back Into Redundant Multithreading?**, EMSOFT 2014

Overhead: SPLASH2, 4 workers



Overhead: SPLASH2, 4 workers



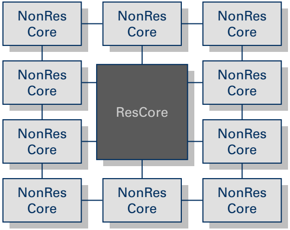
Sources of overhead:

- System call interception
- Cache effects
- Lock density

Hardening the RCB

- **We need:** Dedicated mechanisms to protect the RCB (HW or SW)
- **We have:** Full control over software
- Use FT-encoding compiler?
  - Has not been done for kernel code yet
- RAD-hardened hardware?
  - Too expensive

Why not split cores into resilient and non-resilient ones?



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
- **L4ReAnimator**: Dirk Vogt, Björn Döbel, Adam Lackorzynski: *Stay strong, stay safe: Enhancing Reliability of a Secure Operating System*, IIDS 2010
- **PLR**: Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseh Blomsted, Ramesh Peri: *Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance*, DSN 2007
- **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