

Maksym Planeta Björn Döbel

Operating Systems Meet Fault Tolerance

Microkernel-Based Operating Systems // Dresden, 17.01.2023

'If there is more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.'

Edward Murphy jr.

Goal of the Lecture

Dependable Systems

- Problems
- Operating system-related techniques

Dependability¹

- Availability
Average fraction of time that a component has been up and running
- Reliability
Probability that a component has been up and running continuously
- Maintainability
Time required to repair a faulty component

¹Algirdas Aviz, Jean-Claude Laprie, and Brian Randell. *Fundamental Concepts of Dependability*. 2001, p. 21.

Textbook terminology

Dependability threats:

- Failure
- Error
- Fault

Dependability means

- Prevention
- Removal
- Forecasting
- Tolerance

Dependable Operating Systems

Faults:

- Software (bugs)
- Hardware

Measures:

- Software Engineering
- Software Architectures

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?
 - What error types do exist?
 - 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

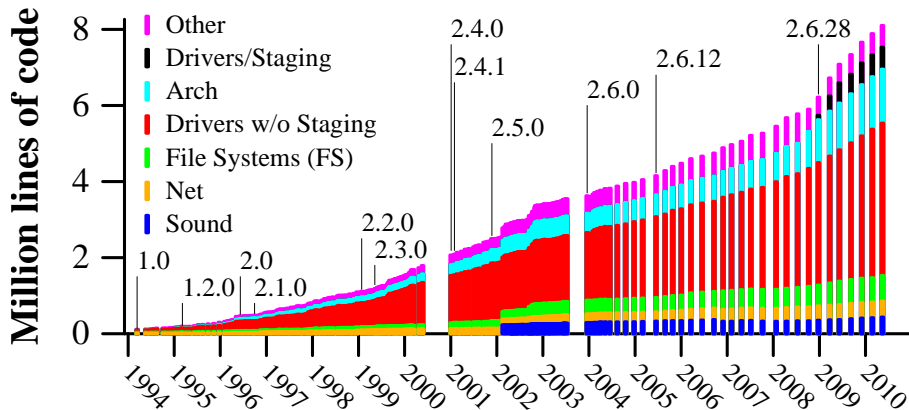


Figure: Linux directory sizes (in MLOC) [19]

Faults per Subdirectory (2001)

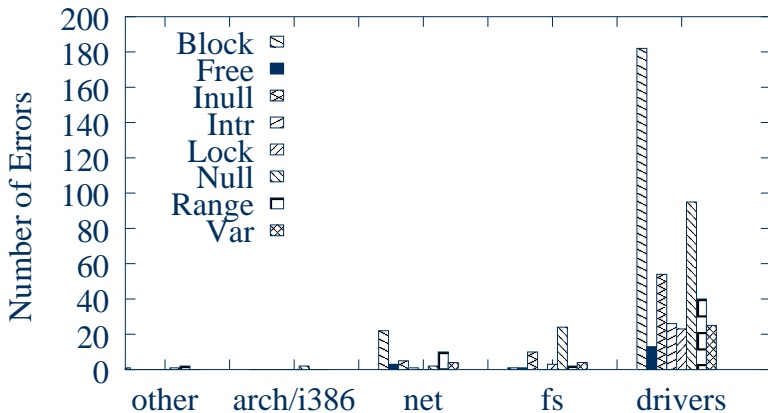


Figure: Number of errors per directory in Linux [4]

Fault Rate per Subdirectory (2001)

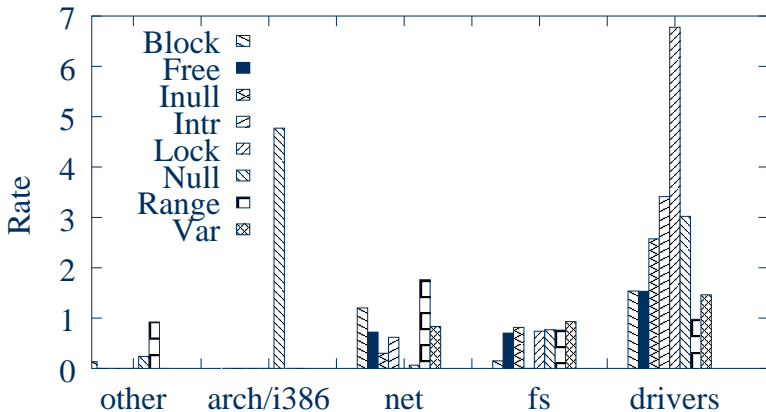


Figure: Rate of errors compared to other directories [4]

Fault Rate per Subdirectory (2011)

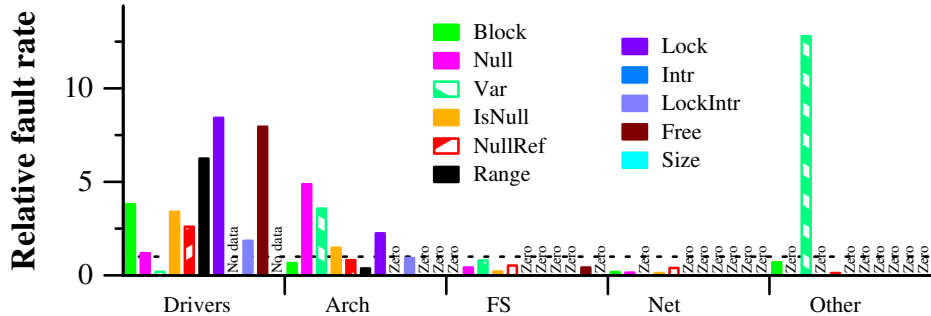


Figure: Rate of errors compared to other directories [19]

Bug Lifetimes (2011) [19]

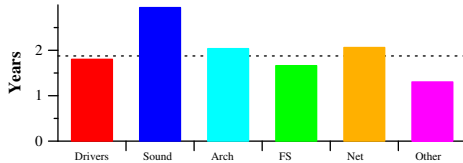


Figure: Per directory

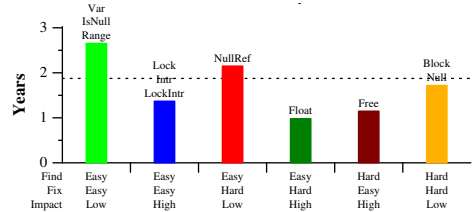


Figure: Per finding and fixing difficulty, and impact likelihood

Software Engineering Measures

- QA
Examples: Manual testing, automated testing, fuzzing
- Continuous Integration
- Static analysis
- Using safer languages
- Guidelines, best practices, etc.
Examples: MISRA C++, C++ Guideline Support Library

Example: MISRA C++ 2008

Rule 0-1-7

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Example: MISRA C++ 2008

Rule 0-1-7

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Rule 3-9-3

The underlying bit representations of floating-point values shall not be used.

Example: MISRA C++ 2008

Rule 0-1-7

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Rule 3-9-3

The underlying bit representations of floating-point values shall not be used.

Rule 6-4-6

The final clause of a switch statement shall be the default-clause.

Rule 3-4-1

(Required) An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Rationale

Defining variables in the minimum block scope possible reduces the visibility of those variables and therefore reduces the possibility that these identifiers will be used accidentally. A corollary of this is that global objects (including singleton function objects) shall be used in more than one function.

Rule 3-4-1: Example

```
void f(int32_t k)
{
    int32_t j = k * k; // Non-compliant
    {
        int32_t i = j; // Compliant
        std::cout << i << j << std::endl;
    }
}
```

In the above example, the definition of `j` could be moved into the same block as `i`, reducing the possibility that `j` will be incorrectly used later in `f`.

Safer languages

- Garbage collection (Go)
- Memory safety (Rust)
- No unused variables (Go, Rust)
- Check error return codes (Go, Rust)
- No uninitialised memory (Go, Rust)
- etc.

Writing a kernel in a high-level language²

- Biscuit: a monolithic kernel implemented in Go

²Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL: <https://www.usenix.org/conference/osdi18/presentation/cutler>.

Writing a kernel in a high-level language²

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection

²Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL: <https://www.usenix.org/conference/osdi18/presentation/cutler>.

Writing a kernel in a high-level language²

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection
- Development effort: 28k lines in Go and 1.5k lines in Assembly

²Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL: <https://www.usenix.org/conference/osdi18/presentation/cutler>.

Writing a kernel in a high-level language²

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection
- Development effort: 28k lines in Go and 1.5k lines in Assembly
- Implemented drivers: AHCI SATA disk controllers and Intel 82599-based Ethernet controllers

²Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL: <https://www.usenix.org/conference/osdi18/presentation/cutler>.

Writing a kernel in a high-level language²

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection
- Development effort: 28k lines in Go and 1.5k lines in Assembly
- Implemented drivers: AHCI SATA disk controllers and Intel 82599-based Ethernet controllers
- Out of 64 CVE-listed Linux kernel bugs, ≈ 40 would be fully or partially alleviated by Go

²Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL: <https://www.usenix.org/conference/osdi18/presentation/cutler>.

Writing a kernel in a high-level language²

- Biscuit: a monolithic kernel implemented in Go
- High-level features: closures, channels, garbage collection
- Development effort: 28k lines in Go and 1.5k lines in Assembly
- Implemented drivers: AHCI SATA disk controllers and Intel 82599-based Ethernet controllers
- Out of 64 CVE-listed Linux kernel bugs, ≈ 40 would be fully or partially alleviated by Go
- 5% to 15% slower, up to 600 μ s latencies for GC

²Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL: <https://www.usenix.org/conference/osdi18/presentation/cutler>.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers
 - No reading undefined memory

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers
 - No reading undefined memory
 - etc.

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers
 - No reading undefined memory
 - etc.
- Unsafe code is annotated

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers
 - No reading undefined memory
 - etc.
- Unsafe code is annotated
- Memory or synchronization problems are impossible in *safe* code

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers
 - No reading undefined memory
 - etc.
- Unsafe code is annotated
- Memory or synchronization problems are impossible in *safe* code
- Performance like in C or C++ code

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Writing a kernel in a safe language³

- Tock: an embedded OS implemented in Rust
- Compiler enforced rules:
 - Several immutable references or one mutable one
 - No null pointers
 - No reading undefined memory
 - etc.
- Unsafe code is annotated
- Memory or synchronization problems are impossible in *safe* code
- Performance like in C or C++ code
- Some software patterns don't work with (safe) Rust well

³Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.

Safe Monoculture Operating Systems

- Safe language for the safe OS
- Maintaining safety guarantees requires using the same language for the subcomponents
- Examples: Theseus⁴ (Rust), RedLeaf⁵ (Rust), Singularity⁶ (C#)

⁴Kevin Boos et al. 'Theseus: an Experiment in Operating System Structure and State Management.' In: OSDI. 2020, pp. 1–19. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/boos> (visited on 01/24/2021).

⁵Vikram Narayanan et al. 'RedLeaf: Isolation and Communication in a Safe Operating System.' In: OSDI. 2020, pp. 21–39. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram> (visited on 01/24/2021).

⁶Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 'Singularity: Scientific containers for mobility of compute.' In: *PLOS ONE* 12.5 (May 11, 2017), e0177459. ISSN: 1932-6203. DOI: 10/f969fz.

Software architectures addressing faults

- Means:
 - Compartmentalisation
 - Redundancy
 - Hardening

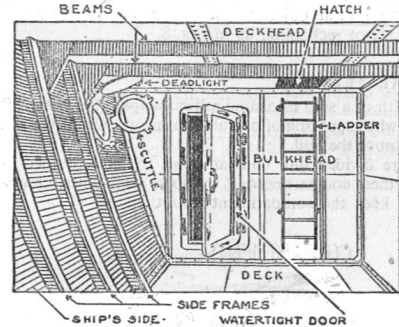


Figure: Ship building

Software architectures addressing faults

- Means:
 - Compartmentalisation
 - Redundancy
 - Hardening
- Address hardware faults

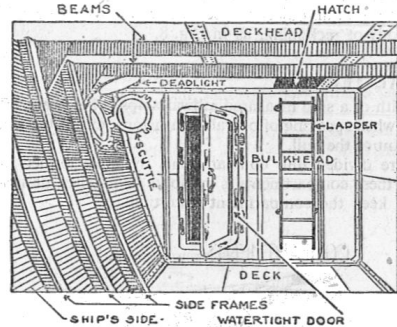


Figure: Ship building

Software architectures addressing faults

- Means:
 - Compartmentalisation
 - Redundancy
 - Hardening
- Address hardware faults
- Recovery
 - Rollback: return to a previous state
 - Transactions
 - Checkpoint/Restart
 - Roll-forward: everything else
 - Error correcting codes
 - Triple modular redundancy + majority voting

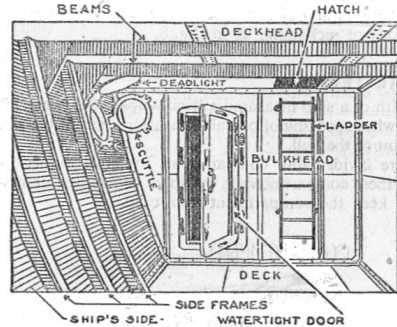
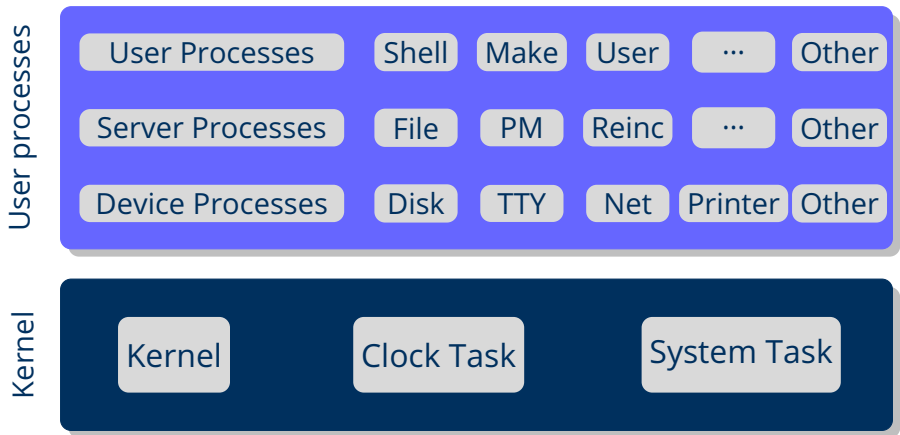


Figure: Ship building

Minix3: A Fault-tolerant OS



Minix3: Fault Tolerance⁷

- 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)

⁷Jorrit 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

L4ReAnimator: Restart on L4Re⁸

- 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

⁸Dirk 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*

Distributed snapshots⁹

- Localized checkpoints
- Problem: Unlimited rollbacks
- Solution: Create global snapshot
- No synchronized clock
- No shared memory
- Only point-to-point messages

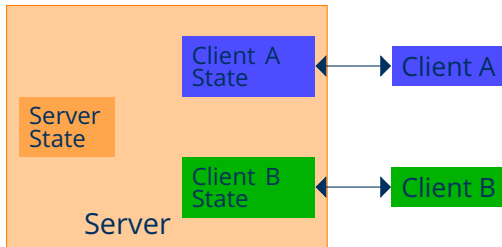
⁹K Mani Chandy and Leslie Lamport. 'Distributed snapshots: Determining global states of distributed systems.' In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

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: Servers and Sessions¹⁰

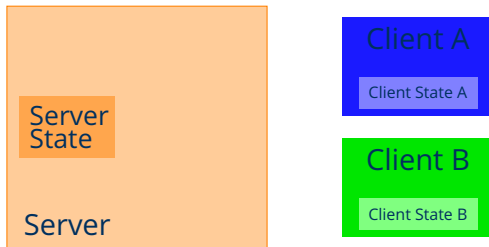
- State recovery is tricky
 - Minix3: Data Store for application data
 - But: applications interact
 - Servers store *session-specific* state
 - Server restart requires potential rollback for every participant



¹⁰Francis M David et al. 'CuriOS: Improving Reliability through Operating System Structure..' In: *OSDI*. 2008, pp. 59–72.

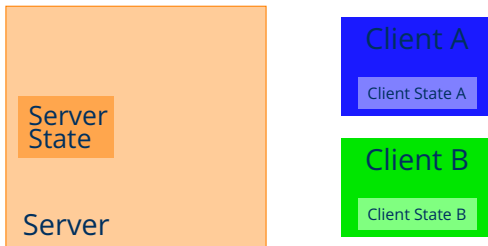
CuriOS: Server State Regions

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



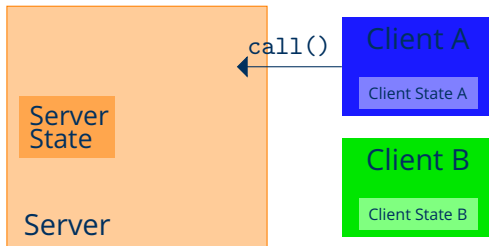
CuriOS: Protecting Sessions

- 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



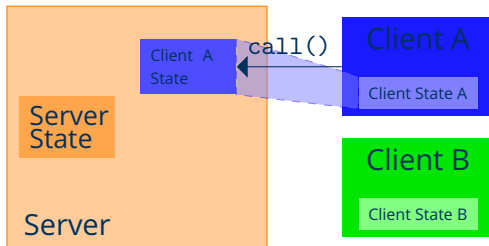
CuriOS: Protecting Sessions

- 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



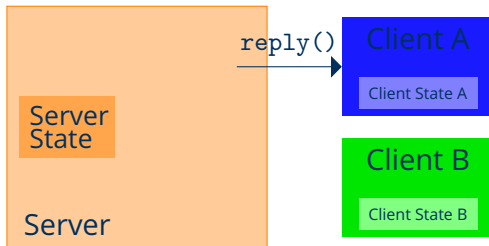
CuriOS: Protecting Sessions

- 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



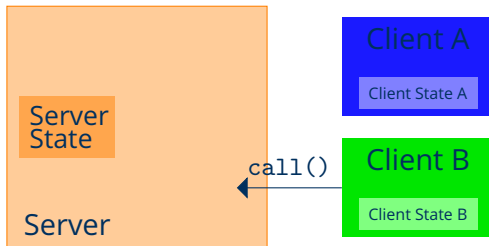
CuriOS: Protecting Sessions

- 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



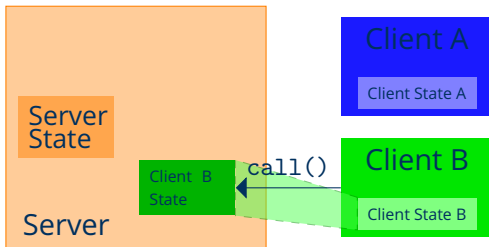
CuriOS: Protecting Sessions

- 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



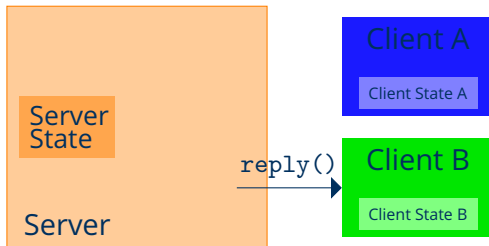
CuriOS: Protecting Sessions

- 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)



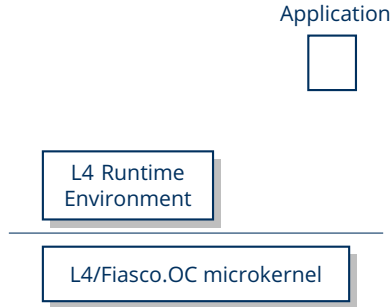
Transparent Restart

- 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

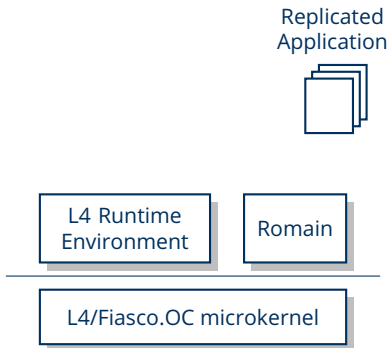
Transient Hardware Faults

- Radiation-induced soft errors
 - Mainly an issue in avionics+space?
- DRAM errors in large data centers
 - Google study: >2% failing DRAM DIMMs per year [20]
 - ECC insufficient [12]
- Decreasing transistor sizes → higher rate of errors in CPU functional units [7]

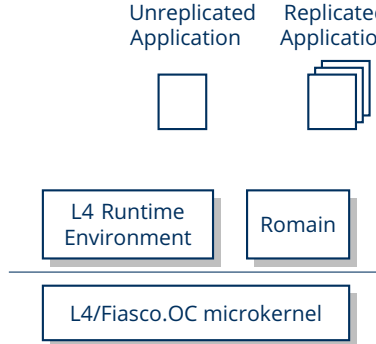
Transparent Replication as OS Service [9, 8]



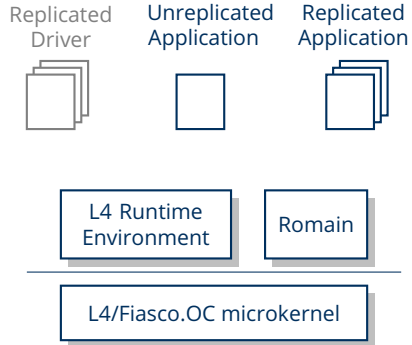
Transparent Replication as OS Service [9, 8]



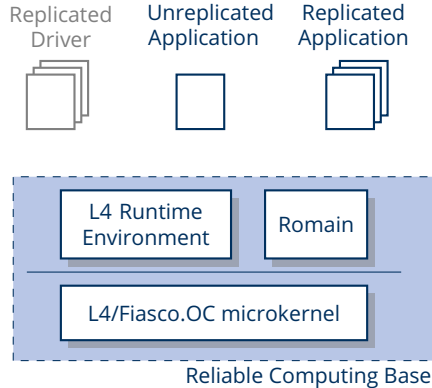
Transparent Replication as OS Service [9, 8]



Transparent Replication as OS Service [9, 8]



Transparent Replication as OS Service [9, 8]

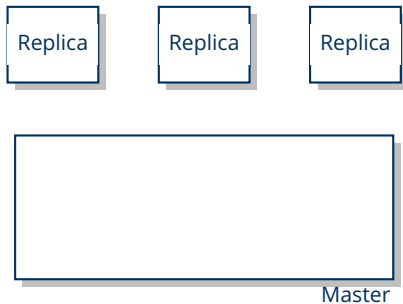


Romain: Structure

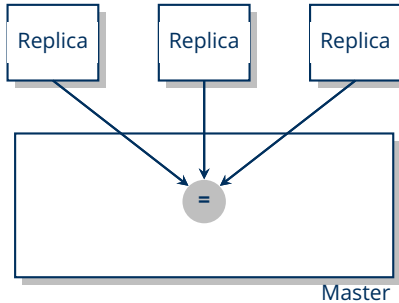


Master

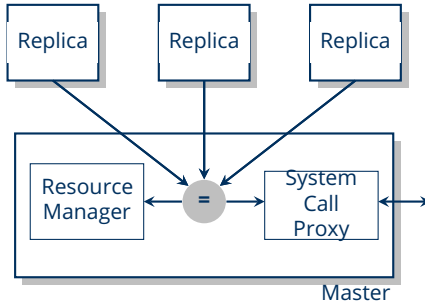
Romain: Structure



Romain: Structure



Romain: Structure

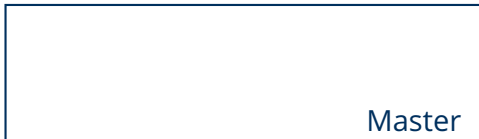


Replica Memory Management

Replica 1



Replica 2

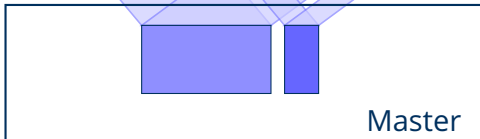


Replica Memory Management

Replica 1



Replica 2

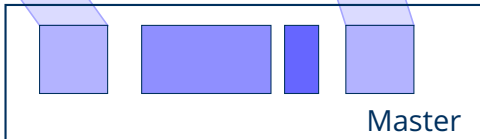


Replica Memory Management

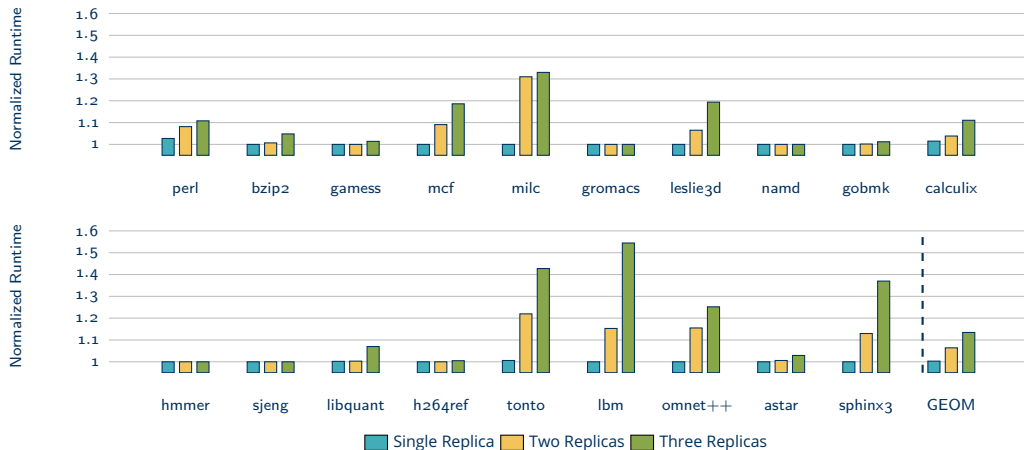
Replica 1



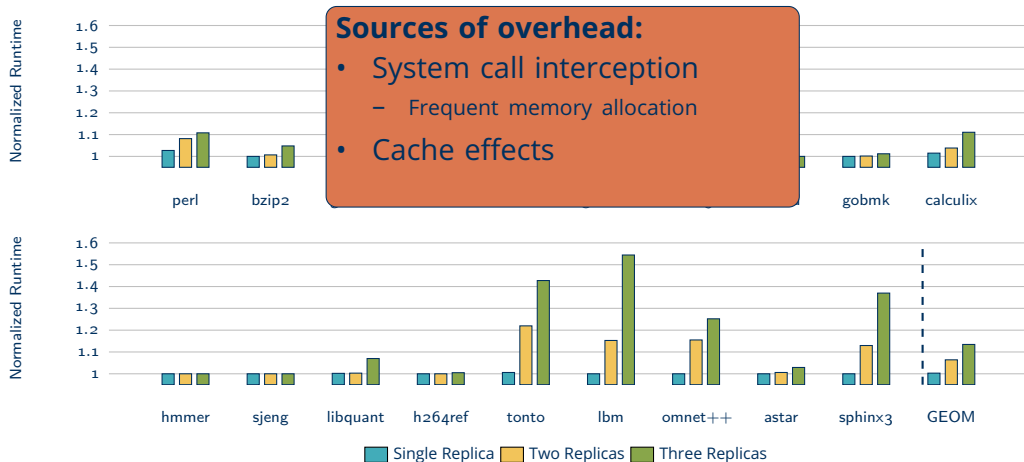
Replica 2



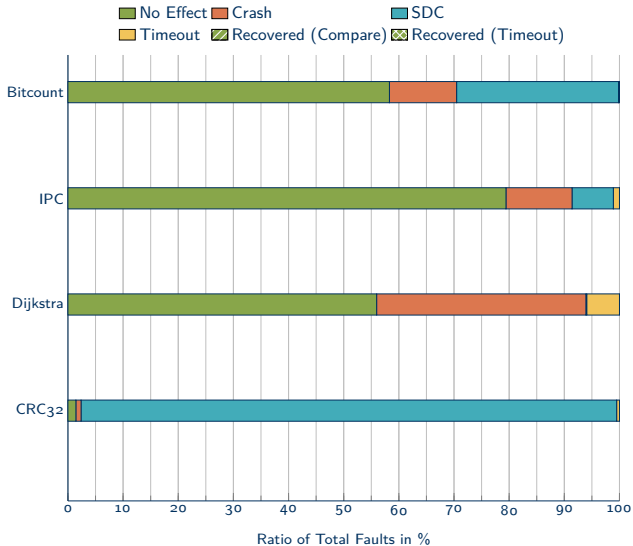
Replicating SPEC CPU 2006 [10]



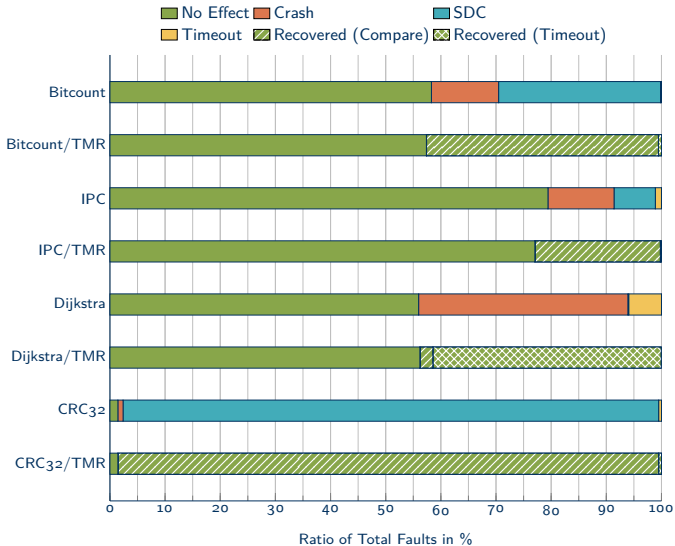
Replicating SPEC CPU 2006 [10]



Error Coverage [10]



Error Coverage [10]



Romain: Summary

- Faults: CPU and memory bit-flips
- Best-effort resilience
- Triple modular redundancy with small increase in makespan
- Multithreading support with deterministic multithreading¹¹

¹¹Björn Döbel and Hermann Härtig. 'Can we put concurrency back into redundant multithreading?' In: *EMSOFT*. 2014, pp. 1–10.

HAFT: Hardware-Assisted Fault Tolerance¹²

- CPU single-event upsets (SEU)
- Instruction-level redundancy for fault detection
- Hardware transaction memory for fault recovery
- *Best-effort* fault tolerance
- Improve efficiency through instruction-level parallelism (ILP) and compiler optimisations

¹²Dmitrii Kuvaiskii et al. 'HAFT: hardware-assisted fault tolerance.' In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16: Eleventh EuroSys Conference 2016. London United Kingdom: ACM, Apr. 18, 2016, pp. 1–17. ISBN: 978-1-4503-4240-7. DOI: 10/ghvf8p.

Instruction-level redundancy

(a) Native

1

2 **z = add** x, y

3

4

5

6

7 **ret** z

Instruction-level redundancy

(a) Native

1
2 **z = add** x, y
3
4
5
6
7 **ret** z

(b) ILR

z = **add** x, y
z2 = **add** x2, y2
d = **cmp neq** z, z2
br d, **crash**

ret z

DMR

Instruction-level redundancy

(a) Native

```
1  
2 z = add x, y  
3  
4  
5  
6  
7 ret z
```

(b) ILR

```
z = add x, y  
z2 = add x2, y2  
d = cmp neq z, z2  
br d, crash
```

```
ret z
```

DMR

(b) ILR

```
loop:  
r1 = add r1, r2  
r1' = add r1', r2'  
r1'' = add r1'', r2''  
majority(r1, r1', r1'')  
majority(r3, r3', r3'')  
cmp r1, r3
```

```
jne loop
```

TMR [15]

Instruction-level redundancy

(a) Native

```
1  
2 z = add x, y  
3  
4  
5  
6  
7 ret z
```

(b) ILR

```
z = add x, y  
z2 = add x2, y2  
d = cmp neq z, z2  
br d, crash
```

```
ret z
```

DMR

(b) ILR

```
loop:  
r1 = add r1, r2  
r1' = add r1', r2'  
r1'' = add r1'', r2''  
majority(r1, r1', r1'')  
majority(r3, r3', r3'')  
cmp r1, r3
```

```
jne loop
```

TMR [15]

(c) HAFT

```
xbegin  
z = add x, y  
z2 = add x2, y2  
d = cmp neq z, z2  
br d, xabort  
xend  
ret z
```

HAFT: Performance

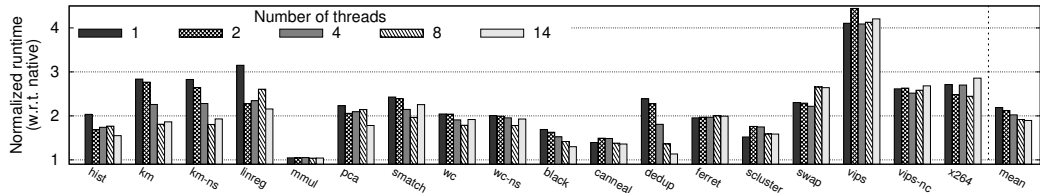


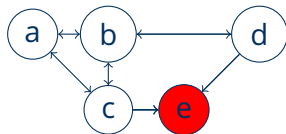
Figure 6: Performance overhead over native execution with the increasing number of threads (on a machine with 14 cores).

Romain vs. HAFT

	Romain	HAFT
Granularity	Syscall	Instruction
Parallelism	Thread-level	Instruction-level
Runtime overhead	$\approx 10\%$	$\approx 100\%$
Resource overhead	$\approx 210\%$	$\approx 100\%$
Faults	CPU & (some) Memory	CPU
Implementation	OS	Compiler & CPU features

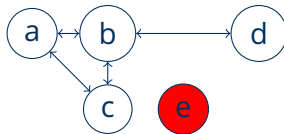
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages



Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages



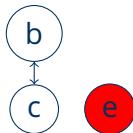
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages



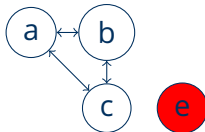
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages



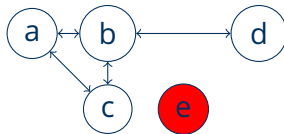
Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages



Software verification

- Combines software engineering and software architectures
- Define good and bad states
- Define axioms (i.e. initial state is good)
- Prove bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages



seL4: Formal verification of an OS kernel¹³

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

¹³Gerwin Klein et al. 'seL4: Formal verification of an OS kernel.' In: *SOSP*. 2009, pp. 207–220.

Verification of a microkernel

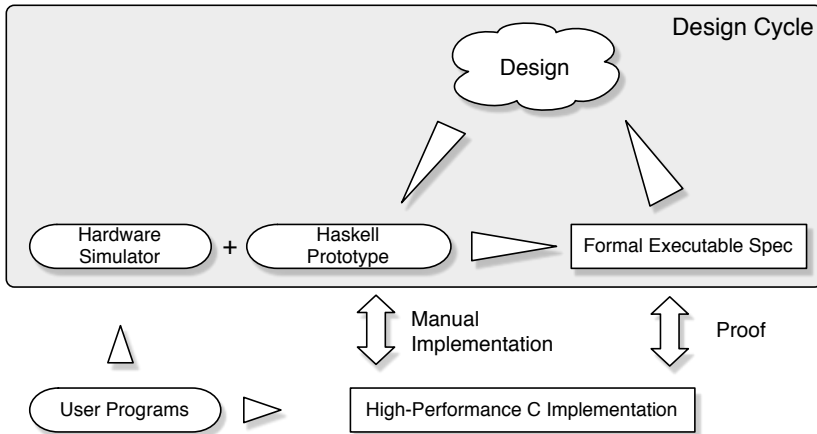


Figure: The seL4 design process [13]

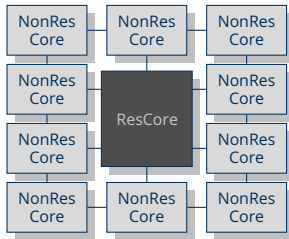
SeL4: Conclusion

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

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

- Dependability is robust development practices + reliability techniques
- Do not let failures propagate
- Silent data corruptions are the worst

Summary

- Dependability is robust development practices + reliability techniques
- Do not let failures propagate
- Silent data corruptions are the worst
- **Fail fast!**

Summary

- Dependability is robust development practices + reliability techniques
- Do not let failures propagate
- Silent data corruptions are the worst
- **Fail fast!**
- Further reading: D. Bernstein: *Some thoughts on security after ten years of qmail 1.0*

Summary

- Dependability is robust development practices + reliability techniques
- Do not let failures propagate
- Silent data corruptions are the worst
- **Fail fast!**
- Further reading: D. Bernstein: *Some thoughts on security after ten years of qmail 1.0*

Summary

- Dependability is robust development practices + reliability techniques
- Do not let failures propagate
- Silent data corruptions are the worst
- **Fail fast!**
- Further reading: D. Bernstein: *Some thoughts on security after ten years of gmail 1.0*

Next week (in previous life): Practical exercise starts at 14:50

Bibliography I

- [1] Algirdas Aviz, Jean-Claude Laprie, and Brian Randell. *Fundamental Concepts of Dependability*. 2001, p. 21.
- [2] Kevin Boos et al. 'Theseus: an Experiment in Operating System Structure and State Management.' In: OSDI. 2020, pp. 1–19. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/boos> (visited on 01/24/2021).
- [3] K Mani Chandy and Leslie Lamport. 'Distributed snapshots: Determining global states of distributed systems.' In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

Bibliography II

- [4] Andy Chou et al. 'An empirical study of operating systems errors.' In: *SOSP*. 2001, pp. 73–88.
- [5] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 'The benefits and costs of writing a POSIX kernel in a high-level language.' In: *OSDI*. Oct. 2018. URL:
<https://www.usenix.org/conference/osdi18/presentation/cutler>.
- [6] Francis M David et al. 'CuriOS: Improving Reliability through Operating System Structure..' In: *OSDI*. 2008, pp. 59–72.
- [7] 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

- [8] Björn Döbel and Hermann Härtig. 'Can we put concurrency back into redundant multithreading?' In: *EMSOFT*. 2014, pp. 1–10.
- [9] Björn Döbel, Hermann Härtig, and Michael Engel. 'Operating system support for redundant multithreading.' In: *EMSOFT*. 2012, pp. 83–92.
- [10] Björn Döbel. 'Operating System Support for Redundant Multithreading.' Dissertation. TU Dresden, 2014.
- [11] Jorrit N Herder et al. 'Fault isolation for device drivers.' In: *DSN*. 2009, pp. 33–42.
- [12] Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. 'Cosmic rays don't strike twice.' In: *ASPLOS*. 2012, pp. 111–122.

Bibliography IV

- [13] Gerwin Klein et al. 'seL4: Formal verification of an OS kernel.' In: *SOSP*. 2009, pp. 207–220.
- [14] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 'Singularity: Scientific containers for mobility of compute.' In: *PLOS ONE* 12.5 (May 11, 2017), e0177459. ISSN: 1932-6203. DOI: 10/f969fz.
- [15] D. Kuvaiskii et al. 'ELZAR: Triple Modular Redundancy Using Intel AVX (Practical Experience Report).' In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). June 2016, pp. 646–653. DOI: 10/ghvjrb.

Bibliography V

- [16] Dmitrii Kuvaiskii et al. 'HAFT: hardware-assisted fault tolerance.' In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16: Eleventh EuroSys Conference 2016. London United Kingdom: ACM, Apr. 18, 2016, pp. 1–17. ISBN: 978-1-4503-4240-7. DOI: 10/ghvf8p.
- [17] Amit Levy et al. 'Multiprogramming a 64kb computer safely and efficiently.' In: *SOSP*. 2017.
- [18] Vikram Narayanan et al. 'RedLeaf: Isolation and Communication in a Safe Operating System.' In: *OSDI*. 2020, pp. 21–39. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram> (visited on 01/24/2021).

Bibliography VI

- [19] Nicolas Palix et al. 'Faults in Linux: Ten years later.' In: *ASPLOS*. 2011, pp. 305–318.
- [20] 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.
- [21] 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.