Maksym Planeta  Björn Döbel

Operating Systems Meet Fault Tolerance
Microkernel-Based Operating Systems // Dresden, 21.01.2020
‘If there is 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.
Goal of the Lecture

Dependable Systems

• Problems
• Operating system-related techniques
Dependability

- 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
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!*

• 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!
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
To give an overview of the software we are studying, we first consider the evolution in code size of the Linux kernel between version 1.0, released in March 1994, and version 2.6.29, released in March 2009. As Linux grew increasingly complex, and was subject to a growing demand from end users, the code size increased. Figure 1 shows that the number of lines of code roughly linearly increased until version 2.6.29, when staging grew substantially. All in all, these changes amounted to 1.5 million lines of code.

Once the fault reports are correlated and assessed when available, as it is in these versions that new code is added, and this added code is then maintained in the next version of Linux. When a new version of Linux is released, it is only necessary to extend the results to a new version of Linux. When our experimental protocol is that it makes it quite easy to run the checkers on the new code, and then repeat the correlation process. As our collected data contains information not only about the faults that we have identified, but also contains information about Linux releases such as drivers with staging and drivers/staging directories.

For most directories, the code growth has been roughly linear since Linux 1.0. Some exceptions are sound drivers and drivers without staging, which grew more slowly. This is in contrast to the drivers/staging directory.

Finally, if both of a pair of reports occur unrelated, and receive some large header files from arch. Finally, staging grew substantially in 2.6.29. All in all, these changes amounted to 1.5 million lines of code.
Faults per Subdirectory (2001)

Figure: Number of errors per directory in Linux [2]
It is unknown whether this set of bugs is representative of all errors. We attempt to compensate for this problem by examining bugs across time, but misses a check on one. On the other hand, consider the rest of the kernel using the formula:

\[ \text{Rate} = \frac{\text{Number of Errors}}{\text{Total Lines of Code}} \]

We try to account for most of the bugs? Can we identify certain types of functions that have higher error rates?

Figure 4: This graph shows drivers have an error rate up to 7 times higher than the rest of the kernel. The number of bugs in a different part of the kernel is the range [2].

Is the actual ratio, though, far greater than the rest of the arch/i386 net drivers. Most errors are not blocked. They are not likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering code is almost three times greater than the rest of the arch/i386 net drivers.

Potential future work could use more sophisticated ranking algorithms (as with Intrinsa [11]) to contain the errors for which we check. We try to correct for this problem by examining bugs across time, but misses a check on one. On the other hand, consider the rest of the kernel using the formula:

\[ \text{Rate} = \frac{\text{Number of Errors}}{\text{Total Lines of Code}} \]

We try to account for most of the bugs? Can we identify certain types of functions that have higher error rates?

Since drivers account for the majority of the code, the number of bugs in a different part of the kernel is the range [2].

The second caveat is that we treat bugs equally. The third caveat is that we only check along very specific paths of the code. This intuition is that it only performs two potentially failing allocations and misses a check on one. On the other hand, consider the rest of the kernel using the formula:

\[ \text{Rate} = \frac{\text{Number of Errors}}{\text{Total Lines of Code}} \]

We try to account for most of the bugs? Can we identify certain types of functions that have higher error rates?

The clustering code is almost three times greater than the rest of the arch/i386 net drivers. Most errors are not blocked. They are not likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering code is almost three times greater than the rest of the arch/i386 net drivers.

Potential future work could use more sophisticated ranking algorithms (as with Intrinsa [11]) to contain the errors for which we check. We try to correct for this problem by examining bugs across time, but misses a check on one. On the other hand, consider the rest of the kernel using the formula:

\[ \text{Rate} = \frac{\text{Number of Errors}}{\text{Total Lines of Code}} \]

We try to account for most of the bugs? Can we identify certain types of functions that have higher error rates?

The clustering code is almost three times greater than the rest of the arch/i386 net drivers. Most errors are not blocked. They are not likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering code is almost three times greater than the rest of the arch/i386 net drivers.

Potential future work could use more sophisticated ranking algorithms (as with Intrinsa [11]) to contain the errors for which we check. We try to correct for this problem by examining bugs across time, but misses a check on one. On the other hand, consider the rest of the kernel using the formula:

\[ \text{Rate} = \frac{\text{Number of Errors}}{\text{Total Lines of Code}} \]

We try to account for most of the bugs? Can we identify certain types of functions that have higher error rates?

The clustering code is almost three times greater than the rest of the arch/i386 net drivers. Most errors are not blocked. They are not likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering code is almost three times greater than the rest of the arch/i386 net drivers.
In this section, we assess the extent to which the trends found a fault rate of only around 1.75 in Linux 2.4.1 code. et al. is rather small. Like Chou et al. is that the distributions they considered. We can thus confidently consider that our faults follow a logarithmic distribution, regardless of any differences in the checkers. Nevertheless, the distribution of these faults among the kinds of faults (a) Number of faults per directory and category

- Block
- Null
- Var
- IsNull
- NullRef
- Range

(b) Relative fault rate

- Lock
- Intr
- LockIntr
- Free
- Size

Figure: Rate of errors compared to other directories [13]
Bug Lifetimes (2011) [13]

Figure: Per directory

Figure: Per finding and fixing difficulty, and impact likelihood
Software Engineering addressing faults

• 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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 0-1-7</td>
<td>The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.</td>
</tr>
<tr>
<td>Rule 3-9-3</td>
<td>The underlying bit representations of floating-point values shall not be used.</td>
</tr>
<tr>
<td>Rule 6-4-6</td>
<td>The final clause of a switch statement shall be the default-clause.</td>
</tr>
</tbody>
</table>
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

```cpp
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`. 
Writing a kernel in a high-level language\(^1\)

- Biscuit: a monolithic kernel implemented in Go

---

Writing a kernel in a high-level language\(^1\)

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

Writing a kernel in a high-level language\textsuperscript{1}

- 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

Writing a kernel in a high-level language

• 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

---

Writing a kernel in a high-level language

• 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, \( \approx 40 \) would be fully or partially alleviated by Go

Writing a kernel in a high-level language

• 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 600µs latencies for GC

---

Writing a kernel in a safe language

- Tock: an embedded OS implemented in Rust

---

2Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language\textsuperscript{2}

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

\textsuperscript{2}Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language\textsuperscript{2}

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

\textsuperscript{2}Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language

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

\[2\] Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language

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

---

\(^2\) Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language

• 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.

\[2\] Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language

- 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

---

2 Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language

• 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

---

2 Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language

- 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

---

Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Writing a kernel in a safe language\(^2\)

- 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

\(^2\)Amit Levy et al. ‘Multiprogramming a 64kb computer safely and efficiently’. In: SOSP. 2017.
Break

• Faults are an issue.

• Hardware-related stuff is the worst.

• Now what can the OS do about it?
Minix3 – A Fault-tolerant OS

User processes
- User Processes
- Server Processes
- Device Processes
  - Disk
  - TTY
  - Net
  - Printer

Kernel
- Kernel
- Clock Task
- System Task
Minix3: Fault Tolerance\(^3\)

- **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
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

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 snapshots\textsuperscript{5}

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

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 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

- 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

Server

Client A

Client State A

Client B

Client State B

call()
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
Software verification

- Define good and bad states
- Define axioms (i.e. initial state is good)
- Proof bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Define good and bad states
- Define axioms (i.e. initial state is good)
- Proof bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Define good and bad states
- Define axioms (i.e. initial state is good)
- Proof bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Define good and bad states
- Define axioms (i.e. initial state is good)
- Proof bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Define good and bad states
- Define axioms (i.e. initial state is good)
- Proof bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
Software verification

- Define good and bad states
- Define axioms (i.e. initial state is good)
- Proof bad states (i.e. null pointer dereference) are unreachable
- Special theorem prover languages
seL4: Formal verification of an OS kernel

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

---

Verification of a microkernel

Design Cycle
Haskell Prototype
Design
Formal Executable Spec
High-Performance C Implementation
User Programs
Hardware Simulator
Haskell Prototype
Formal Executable Spec

Figure: The seL4 design process [11]
• 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 [14]
  - ECC insufficient [10]

- Decreasing transistor sizes $\rightarrow$ higher rate of errors in CPU functional units [5]
Transparent Replication as OS Service [7, 6]
Transparent Replication as OS Service \([7, 6]\)

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel

Dresden, 21.01.2020
Transparent Replication as OS Service [7, 6]

Unreplicated Application

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [7, 6]

Replicated Driver  Unreplicated Application  Replicated Application

L4 Runtime Environment  Romain

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [7, 6]

Replicated Driver

Unreplicated Application

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel

Reliable Computing Base

Maksym Planeta, Björn Döbel
Dresden, 21.01.2020
Romain: Structure

Master
Romain: Structure
Romain: Structure
Romain: Structure

- Replica
  - Resource Manager
  - System Call Proxy
- Master

Replica = OS Resilience

Maksym Planeta, Björn Döbel
Dresden, 21.01.2020

Slide 40 of 51
Replica Memory Management

Replica 1

Replica 2

Master
Replica Memory Management

Replica 1

Replica 2

Master
Replica Memory Management

Replica 1

Replica 2

Master
Replicating SPEC CPU 2006 [8]

### Single Replica

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normalized Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>1.0</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.0</td>
</tr>
<tr>
<td>gamess</td>
<td>1.0</td>
</tr>
<tr>
<td>mcf</td>
<td>1.0</td>
</tr>
<tr>
<td>milc</td>
<td>1.0</td>
</tr>
<tr>
<td>gromacs</td>
<td>1.0</td>
</tr>
<tr>
<td>leslie3d</td>
<td>1.0</td>
</tr>
<tr>
<td>namd</td>
<td>1.0</td>
</tr>
<tr>
<td>gobmk</td>
<td>1.0</td>
</tr>
<tr>
<td>calculix</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Two Replicas

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normalized Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>hmmer</td>
<td>1.5</td>
</tr>
<tr>
<td>sjeng</td>
<td>1.5</td>
</tr>
<tr>
<td>libquant</td>
<td>1.5</td>
</tr>
<tr>
<td>h264ref</td>
<td>1.5</td>
</tr>
<tr>
<td>tonto</td>
<td>1.5</td>
</tr>
<tr>
<td>lbm</td>
<td>1.5</td>
</tr>
<tr>
<td>omnet++</td>
<td>1.5</td>
</tr>
<tr>
<td>asar</td>
<td>1.5</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.5</td>
</tr>
<tr>
<td>GEOM</td>
<td>1.5</td>
</tr>
</tbody>
</table>

### Three Replicas

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normalized Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**OS Resilience**

Maksym Planeta, Björn Döbel
Dresden, 21.01.2020
Sources of overhead:

- System call interception
  - Frequent memory allocation
- Cache effects

Normalized Runtime

<table>
<thead>
<tr>
<th>Source</th>
<th>Single Replica</th>
<th>Two Replicas</th>
<th>Three Replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>gobmk</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>calculix</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>hmmer</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>sjeng</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>libquant</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>h264ref</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>tonto</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>lbm</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>omnet++</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>astar</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>GEOM</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

OS Resilience
Maksym Planeta, Björn Döbel
Dresden, 21.01.2020
Error Coverage [8]

<table>
<thead>
<tr>
<th>Bitcount</th>
<th>IPC</th>
<th>Dijkstra</th>
<th>CRC32</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Effect</td>
<td>Crash</td>
<td>SDC</td>
<td>No Effect</td>
</tr>
<tr>
<td>Ratio of Total Faults in %</td>
<td>Timeout</td>
<td>Recovered (Compare)</td>
<td>Recovered (Timeout)</td>
</tr>
</tbody>
</table>

- Bitcount: 60% No Effect, 20% Crash, 20% SDC
- IPC: 70% No Effect, 20% Crash, 10% SDC, 10% Timeout
- Dijkstra: 50% No Effect, 50% Crash, 20% SDC, 80% Timeout, 10% Recovered (Compare)
- CRC32: 100% Recovered (Timeout)
Error Coverage [8]

Ratio of Total Faults in %

- Bitcount
- Bitcount/TMR
- IPC
- IPC/TMR
- Dijkstra
- Dijkstra/TMR
- CRC32
- CRC32/TMR

Legend:
- No Effect
- Crash
- SDC
- Timeout
- Recovered (Compare)
- Recovered (Timeout)
Romain: Summary

- Faults: CPU and memory bit-flips
- Best-effort resilience
- Tripple modular redundancy with small increase in makespan
- Multithreading support with determinenistic multithreading

---

8Björn Döbel and Hermann Härting. ‘Can we put concurrency back into redundant multithreading?’ In: EMSOFT. 2014, pp. 1–10.
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?

![Diagram showing ResCore and NonResCore cores](image)
Summary

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

Next week: Practical exercise starts at 14:50
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**: D. 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: *Can We Put Concurrency Back Into Redundant Multithreading?*, EMSOFT 2014
Bibliography 1


Bibliography II


Bibliography III


