Predictable, System-Level Fault Tolerance in $C^3$

Jiguos Song, John Wittrock, Gabriel Parmer

jiguos@gwu.edu  gparmer@gwu.edu
Consequences of embedded system faults
Uncontrolled acceleration in Toyota Camrys

- Electronic Throttle Control System (ETCS)
- OSEK OS, 24 tasks, 280K LOC of C
- bit flip in scheduler data-structures  
  \[\rightarrow\] reproducible 30-sec uncontrolled acceleration

"a bit-flip there, will have the effect of killing one of the tasks"

– Bookout v Toyota
Decreasing process sizes $\rightarrow$ 5nm

+ faster
+ less power
+ smaller

- increased vulnerability to HW transient faults
- 65% of HW faults corrupt OS state
Fault tolerance: key technique in dependable system design
- detection
- recovery
- isolation

Goal: predictably recovery from system-level faults
Application fault tolerance

- example recovery techniques
  - recovery blocks
  - checkpointing
  - re-fork
- temporal redundancy
  - detect fault by job completion
  - replay execution from saved state
- re-execution impacts only lower-priority tasks
Application fault tolerance
- example recovery techniques
  - recovery blocks
  - checkpointing
  - re-fork
- temporal redundancy
  - detect fault by job completion
  - replay execution from saved state
- re-execution impacts only lower-priority tasks
Application fault tolerance

- example recovery techniques
  - recovery blocks
  - checkpointing
  - re-fork

- temporal redundancy
  - detect fault by job completion
  - replay execution from saved state

- re-execution impacts only lower-priority tasks
System-level fault tolerance

- failures in
  - scheduler
  - memory mapping manager
  - file-systems
  - ...

System components contain state for all tasks

- failure impacts memory of all tasks

Recovery requires resources

- processing time...to recovery scheduler
- memory...to recover memory mapper
System-level fault tolerance

- failures in
  - scheduler
  - memory mapping manager
  - file-systems
  - ...

System components contain state for all tasks

- failure impacts memory of all tasks

Recovery requires resources

- processing time...to recovery scheduler
- memory...to recover memory mapper
System-level fault tolerance

- failures in
  - scheduler
  - memory mapping manager
  - file-systems
  - ...

System components contain state for all tasks

- failure impacts memory of *all* tasks

Recovery requires resources

- processing time...to recovery scheduler
- memory...to recover memory mapper
System-level fault tolerance

- failures in
  - scheduler
  - memory mapping manager
  - file-systems
  - ...

System components contain state for all tasks

- failure impacts memory of all tasks

Recovery requires resources

- processing time...to recovery scheduler
- memory...to recover memory mapper
Recovery from system-level faults

- failure impacts timing of *all* tasks
- variable time recovery
  - # of threads, memory mappings
  - best-effort tasks impact recovery latency

Recovery inversion

- priority inversion due to system recovery
- recovery at what priority?
- recovery of what state at what priority?
Fault Model Assumptions

Fail-stop

- immediate fault detection
- assertions, hardware exceptions

Transient faults

- processor single-event upsets
- modeled as random register bit-flips
- known minimal inter-arrival

Trusted Computing Base for Fault-Tolerance (TCB-FT)

- no kernel faults
- minimal kernel (complexity + execution)
C³: The Computational Crash Cart

C³: Computational Crash Cart

- resuscitate system
- from system-level faults
- predictably

Main ideas

- pervasive fault isolation → restrict propagation
- efficient $\mu$-reboot of individual components
- interface-driven, application-oblivious recovery
- on-demand recovery → bound recovery inversion
**COMPOSITE: A Component-Based Foundation for C³**

System functionality as *components*
- user-level, protection domains
- fine-grained fault isolation

Low-level services are components
- scheduling, memory mapping, FS
- small, policy-less kernel

Component interaction: *invocation* of exported function
- contractually specified interfaces
- function call semantics

Threads orthogonal to components
- thread migration – predictable e2e IPC
- concurrent/parallel components

*Example component graph*
Recovery sequence in $C^3$

1. fault detection
2. safe-state recovery
   - $\mu$-reboot
3. consistent-state recovery
   - object state recovery

Diagram:

- App
- Lock
- Scheduler
- Kernel

Legend:
- component
- lock data-structure
- thd data-structure
- comp dependency
Recovery sequence in $C^3$

1. fault detection
2. safe-state recovery
   - $\mu$-reboot
3. consistent-state recovery
   - object state recovery

component types:
- Component
- lock data-structure
- thd data-structure
- comp dependency
Recovery sequence in $C^3$

1. fault detection
2. safe-state recovery
   - $\mu$-reboot
3. consistent-state recovery
   - object state recovery

- component
- lock data-structure
- thd data-structure
- comp dependency
Recovery sequence in $C^3$

1. fault detection
2. safe-state recovery
   - $\mu$-reboot
3. consistent-state recovery
   - object state recovery

- component
- lock data-structure
- thd data-structure
- comp dependency
μ-Reboot: Recovery of a Safe State

- faults vectored to loader
- redirect component invocations
- restore to initial (safe) state
- properly prioritized μ-reboot


\(\mu\text{-Reboot: Recovery of a Safe State}\)

- faults vectored to loader
- redirect component invocations
- restore to initial (safe) state
- properly prioritized \(\mu\text{-reboot}\)
Toward Recovering a Consistent State

$\mu$-reboot makes data-structures inconsistent

- objects in data-structures have states
- operations/functions mutually alter state
  $\rightarrow$ consensus on state changes
- examples:
  - lock taken/contested/released
  - thread blocked/runnable
  - thread active/preempted
μ-reboot makes data-structures inconsistent

- objects in data-structures have states
- operations/functions mutually alter state
  → consensus on state changes

- examples:
  - lock taken/contested/released
  - thread blocked/runnable
  - thread active/preempted
Toward Recovering a Consistent State

µ-reboot makes data-structures inconsistent. Objects in data-structures have states. Operations/functions mutually alter state. → consensus on state changes.

Examples:
- Lock taken/contested/released
- Thread blocked/runnable
- Thread active/preempted

Idea: use component operations to rebuild a consistent state in µ-rebooted component.
Avoid surrounding component's involvement
- interface-replication of object state
Interface-Driven Recovery: Consistency

Avoid surrounding component's involvement
- interface-replication of object state
- rebuild objects with operations from the interface
Avoid surrounding component’s involvement
- *interface*-replication of object state
- *rebuild* objects with operations from the interface
- *reflect* on server interfaces

Interface-Driven Recovery: Consistency
Avoid surrounding component's involvement

- **interface**-replication of object state
- *rebuild* objects with operations from the interface
- *reflect* on server interfaces
Avoid surrounding component's involvement
- **interface**-replication of object state
- *rebuild* objects with operations from the interface
- *reflect* on server interfaces

*Not* a log
- interface stores current state
  - not past information
- restores meta-data (i.e. data-structures)
  - not all communicated data
Where should interface object-tracking code be implemented?

- increase size of TCB-FT?
- single point of failure?
**Where** should interface object-tracking code be implemented?

- increase size of TCB-FT?
- single point of failure?

Interface code compiled and linked into both client and server
Where should interface object-tracking code be implemented?

- increase size of TCB-FT?
- single point of failure?

Interface code compiled and linked into both client and server
**When** should interface object-recovery code be invoked?

**Eager recovery**
- recover all objects *at μ-reboot*
- *recovery interference* = $\sum_{\forall \text{objects}}$ recovery overhead

![Diagram showing eager recovery with priority inversion and thread scheduling](image-url)
When should interface object-recovery code be invoked?

Eager recovery

- recover all objects \textit{at \textmu-reboot}
- \textit{recovery interference} = \sum_{\forall \text{objects}} \text{recovery overhead}

On-Demand recovery

- recover object \textit{when operation performed} on it
- \textit{recovery interference} = \sum_{\forall \text{HP objects}} \text{recovery overhead}
Experiments Setup

- $C^3$ implemented in *Composite* component-based OS
- Intel i7-2760QM, running at 2.4 Ghz
- 3 services evaluated:
  - the system scheduler (SCHED)
    - objects: threads
    - operation: created, blocked, woken and destroyed
  - the system physical memory manager and mapper (MM)
    - objects: pages
    - operation: granted, aliased and revoked
  - RAM-based file system (FS)
    - objects: files
    - operation: open, read, write and close
C³ efficiency (all values in \(\mu\)-seconds):

<table>
<thead>
<tr>
<th>Component</th>
<th>(\mu)-reboot</th>
<th>object recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sched</td>
<td>17.67</td>
<td>0.76</td>
</tr>
<tr>
<td>MM</td>
<td>20.06</td>
<td>0</td>
</tr>
<tr>
<td>FS</td>
<td>9.03</td>
<td>5.00</td>
</tr>
</tbody>
</table>

Fault injection experiments

- 100 detected register bit flips per component
- system *recovers from all detected faults*
- system never deviates from expected behavior
Schedulability Analysis

Extends Response-Time Analysis (RTA)

- adds eager and on-demand latencies
- given a minimum fault inter-arrival time

Compare against system-wide checkpointing

- Composite implementation
- memory bandwidth limited
- 0.1ms checkpoint/restore, 2MB system
Fault-Tolerant Systems Schedulability: Checkpointing and C³, 50 tasks, 100ms period

Fault-Aware Schedulability Success Ratio

Utilization

C³ "on-demand" recovery
C³ "eager" recovery
checkpointing 0.1ms/chkpt
checkpointing 1ms/chkpt
checkpointing 10ms/chkpt
Fault-Tolerant Systems Schedulability: Checkpointing and \( C^3 \), 100 tasks, 100ms period

- \( C^3 \) "on-demand" recovery
- \( C^3 \) "eager" recovery
- Checkpointing 0.1ms/chkpt
- Checkpointing 1ms/chkpt
- Checkpointing 10ms/chkpt
Computational Crash Cart

- pervasive **fault isolation** → restrict propagation
- efficient, predictable **μ-reboot** of individual components
- interface-driven, application-oblivious recovery
- on-demand recovery → bound **recovery inversion**
- timing analysis for predictable recovery
System level checkpoint overhead

<table>
<thead>
<tr>
<th>Memory Size (MB)</th>
<th>Checkpoint Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>512</td>
<td>100</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
</tr>
</tbody>
</table>

- COS Restore
- COS Checkpoint
- CRIU Checkpoint
- CRIU Restore
- Xen Restore
- Xen Checkpoint
- memcpy()
Overhead in $C^3$: no faults

<table>
<thead>
<tr>
<th>Component</th>
<th>without tracking</th>
<th>tracking overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sched</td>
<td>1.45</td>
<td>0.34</td>
</tr>
<tr>
<td>MM</td>
<td>0.52</td>
<td>0.09</td>
</tr>
<tr>
<td>FS</td>
<td>1.65</td>
<td>1.12</td>
</tr>
</tbody>
</table>

**Table:** infrastructure overhead ($\mu$-seconds)

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
<th>Stub LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sched</td>
<td>162(4.8%)</td>
<td>245</td>
</tr>
<tr>
<td>MM</td>
<td>93(19.2%)</td>
<td>358</td>
</tr>
<tr>
<td>FS</td>
<td>46(15.1%)</td>
<td>602</td>
</tr>
</tbody>
</table>

**Table:** lines of code added
FASSR vs Recovery Object Number

![Graph showing FASSR vs Recovery Object Number]

- FASSR Objects per task (util 70, $e_{ur:p_t}=0.02:100ms$)
- Lines represent different recovery methods:
  - od 20
  - od 50
  - od 100
  - eg 20
  - eg 50
  - eg 100

Objects per task range from 1 to 512.
### Table: recovery cost ($\mu$-seconds)

<table>
<thead>
<tr>
<th>Component</th>
<th>$\mu$-reboot – mem init.</th>
<th>$\mu$-reboot – exe init.</th>
<th>object recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sched</td>
<td>7.52</td>
<td>10.15</td>
<td>0.76</td>
</tr>
<tr>
<td>MM</td>
<td>16.06</td>
<td>4.00</td>
<td>0</td>
</tr>
<tr>
<td>FS</td>
<td>6.37</td>
<td>2.66</td>
<td>5.00</td>
</tr>
</tbody>
</table>

$\rightarrow$ Both $\mu$-reboot and object recovery are efficient
System-level Fault Tolerance Challenges

Recovering a consistent state

- assumed state of system objects is comparable in all parts of system
  - thread at specific priority is blocked?
  - shared page mapped at specific address?
  - file open at offset, opened with specific flags?

→ rebooting subsystem is insufficient