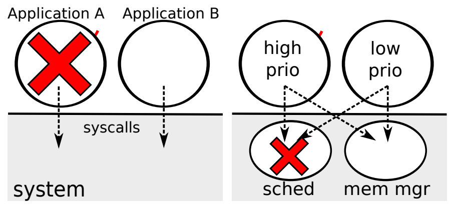
Predictable, System-Level Fault Tolerance in Composite

Jiguo Song, Gabriel Parmer
The George Washington University
Washington, DC
{jiguos,gparmer}@gwu.edu

Intermittent faults are an increasingly challenging difficulty in embedded and real-time systems. As process technologies shrink circuitry, it becomes increasingly susceptible to transient faults from radiation sources such as cosmic rays. Additionally, as software complexity increases, intermittent faults such as race conditions challenge software reliability. Given these motivations, research has approached the paired problems of recovering from a fault, and doing so predictably. However, most past research has been limited in focus to the predictable recovery of faults at the *application-level*. Examples include systems infrastructures [2] enabling application fault recovery, and scheduling theory [3] that considers periodic faults, and the impact on schedulability for recovery and re-execution of failed applications.



(a) Traditional Systems (b) C^3

Figure 1: (a) Traditional application-level, predictable fault tolerance using a technique such as check-pointing or recovery blocks for recovery. (b) C^3: for the recovery of failed system services, within bounded time, and at the priority of any application that requires service.

In this paper, we discuss a system we're researching called C^3, the Computational Crash Cart. Table 1(a) depicts a traditional system focused on application-level recovery, and (b) C^3 for predictable recovery of system-level services such as the scheduler and memory manager. We implementing C^3 in the COMPOSITE component-based operating system [1].

Challenges and techniques for predictable, system-level fault recovery:

- **Fault propagation.** Service implementation in the kernel in common monolithic systems makes fault propagation more likely. A fault in one logical service cannot be prevented from corrupting memory in other, unrelated system services. C^3 uses COMPOSITE's pervasive use of hardware protection domains to constrain propagation.
- **State recovery.** When a system service fails, it is not straightforward to recover its state. Such services are highly concurrent, and contain data-structures describing many task's resources. To illustrate the difficulties in state recovery, we observe a traditional technique: checkpointing. A checkpoint of the service has periods of inconsistency with the state of a task – a scheduler that dispatches a task after a checkpoint will lose the accounting for that time if it rolled back. Instead, C^3 takes advantage of communication protocols between tasks and services, and records the state of all resources manipulated via that communication. This communication is *replayed* when a service fails to reestablish a consistent service state.

Operation	Memory Manager	Scheduler
Reboot: End to End	35.24(0.61)	46.34(1.65)
Reboot: Memory Ops	31.72(0.33)	36.86(1.72)
COMPOSITE: Invocations	2.10(0.23)	1.29(0.13)
C^3: Invocations	2.31(0.08)	1.33(0.02)

Table 1: Recovery Costs (avg(stddev) in μsecs)

- **Prioritized recovery.** System services might be utilized by both hard real-time tasks of different priorities, and best-effort tasks. The recovery of a system service should execute at the proper system priority. C^3 does this via 1) explicit priority inheritance of the recovery process, and 2) per-task recovery of their own state in the service via replay. These techniques bound interference in the recovery process, and minimize inter-task interference.
- **Schedulable service failure.** Traditional techniques [3] for scheduling recovery do not apply to system service recovery as the timing of possibly all tasks in the system are effected by recovery. We are currently defining schedulability analysis for system recovery in C^3.

We are not focusing on related problems such as fault detection, and instead rely on complementary techniques [4].

Table 1 shows preliminary C^3 experiments for the recovery of the system scheduler and physical memory mapper, measured on an Intel i7 at 2.4 Ghz. Each service executes in a private memory protection domain (via page-tables), and invocations are via RPC. For the memory manager, a task maps a page, aliases it, and then removes both mappings (3 RPCs). For the scheduler, two threads switch back and forth by blocking and waking up (2 RPCs and a thread switch). End to end recovery costs measure the cost of these invocations when a fault occurs in the service. This necessitates rebooting the service, and replaying the communication from the tasks. We note that the costs of recovery are small (35 and 46 μsec) with small variation, and are dominated (over 80% of the cost) by `mem_cpy` and `mem_set` to reset the service's memory to an initial state. The overhead of tracking communication between components differs for the services: 12% overhead for C^3 invocations over native COMPOSITE for the memory manager, and 2% for the scheduler.

Continued and future work. We believe these results show the promise of C^3, and system-level fault recovery. Research remains to address the issues of schedulability, generality to other system services, ease of programming, and a validation of the timing properties.

References

[1] The COMPOSITE component-based system: http://composite.seas.gwu.edu.
[2] A. Egan, D. Kutz, D. Mikulin, R. Melhem, and D. Mosse. Fault-tolerant rt-mach and an application to real-time train control. *Software Practice and Experience*, 1999.
[3] P. Mejia-Alvarez and H. Aydin. Scheduling optional computations in fault-tolerant real-time systems. In *RTCSA*, 2000.
[4] K. Pattabiraman, V. Grover, and B. Zorn. Protecting critical data in unsafe languages. In *Eurosys*, 2008.