Toward Predictable, Efficient, System-Level Tolerance of Transient Faults*

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

***Abstract*—As embedded and real-time systems increase in complexity, and as chip process technologies continually decrease feature size, transient faults increasingly threaten system failure. This paper introduces C^3, an system to tolerate *system-level faults* (*e.g.* in the scheduler). When considering *predictable* recovery of system-level components, we introduce *recovery interference*, a side-effect of system-level recovery that causes possibly unbounded priority inversion. We discuss an interface-driven recovery technique that is effective, efficient, and uses *on-demand* recovery to avoid recovery interference.**

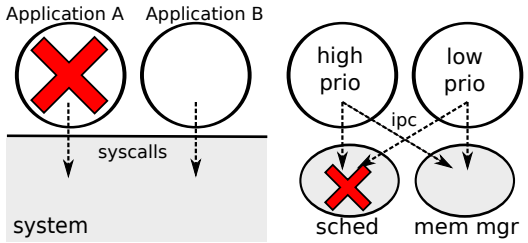
I. Introduction

The ability of embedded and real-time systems to tolerate unexpected changes in their environment is of increasing importance. An important class of environmental influences that can manifest in faulty software behavior are those induced by micro-architectural effects that deviate from the specified behavior. As chips continue toward smaller processes (*e.g.* down to a 22nm feature size and beyond), the likelihood of incorrect results increases due to manufacturing error, heat damage, and other physical effects. Additionally, Single-Event Upsets (SEUs) due to environmental radiation can cause corruption of transistor state leading to bit-flips in chip structures. Adapting to and tolerating these effects at the software level is a contributing factor to continue process-driven progress toward smaller, denser, and faster systems.

Tolerating faults in *system-level* components – that define system scheduling, memory management, and I/O processing – is important: nearly 65% of these hardware errors corrupt OS state [1] before they're detected. However, tolerating these faults is particularly difficult: such components provide services that are used by multiple applications (and other system components), and when they fail, their internal state must be rebuilt such that it is *consistent* with the service previously provided to the rest of the system. For example, if an application has previously opened a file, and written data to it, then a file system that experiences a failure must be reconstituted in a manner that includes the open, modified file; a faulted physical memory manager/mapper must re-create state the describes all memory mappings created in the system so far.

Figure 1 depicts traditional fault tolerance of applications, and our system for system-level service failure. The impact of recovery for a system-level component affects all applications that harness its functionality, at all priorities.

A complicating factor in reconstructing a consistent state in a failed system component is that their consistency is defined by complicated interactions with many other components. For example, a service request from a real-time task might require memory allocation to hold file data in the file-system, and recovery will be different before that file is updated, and after. Concurrency in system components complicates consistency as well, as the component state is defined by a specific interleaving of the threads. In contrast, many real-time tasks that accept input at job activation, and provide actuator commands at the end have very well-structured interactions with the system, thus enabling simple recovery techniques such as the use of recovery blocks [2] that essentially enable task state to be saved at activation, and restored if it fails.



(a) Traditional Systems (b) C^3

Fig. 1. (a) Traditional application-level, 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.

Importantly, re-building the state of a system component must be integrated into a schedulability analysis so that system designers can have increased confidence in the temporal behaviors of the system. This is complicated by the very nature of systems components: they provide service to many other system components, all of which can suffer delay during their recovery. In this paper, we focus on an introduction to the design and implementation of our Computational Crash Cart system, C^3, and discuss the decisions that impact schedulability analysis, though we leave that analysis for future work.

Contributions. (1) We introduce *recovery interference* that is similar to priority inversion in the sense that lower priority or best-effort tasks cause increases in the execution time of high-priority tasks during recovery. This effect is due to the recovery of objects (*e.g.* component-specific abstractions such as files, pages, threads) in a faulted system component that are not required by the high-priority task. The time taken to recover these objects represents interference in high-priority task execution. This recovery interference is *unbounded* when we cannot place an a-priori bound on the number of objects accessed only by low-priority/best effort tasks that must be recovered. (2) We introduce an *interface-driven* approach to recovering from system-level component faults. All communication between components is interposed on, and the state of each object operated on by that interface is tracked during

This material is based upon work supported by the National Science Foundation under Grants No. CNS 1137973, CNS 1149675, and CNS 1117243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

normal execution. When a component fails, all components that depend on it for service, transparently bring it into a consistent state by using the functions in the interface themselves to recreate the expected object state. Components additionally *introspect* on the interfaces of components they depend on to recreate the provided objects. (3) Using the interface-driven component recovery support, we introduce two system fault recovery techniques: *eager* recovery that at fault time rebuilds a fully consistent state for the component, and *on-demand* recovery. Eager recovery suffers from significant recovery interference as recovery of even low-priority and best-effort thread's objects is done at component reboot time. On-demand recovery, on the other hand focuses on recovering objects *at the priority of the thread operating on the object.* In fact the entire recovery process is conducted at the priority of the highest-priority thread accessing the faulted component.

This paper is organized as follows: Section II discusses related work, Section III describes the design and implementation of interface-driven, on-demand recovery in C^3, Section IV evaluates the system, and Section V concludes and discusses future work.

II. Related Work

Fault tolerance mechanisms. Many fault tolerance mechanisms have been studied for user-level tasks including N-version programming [3], and redundant multi-threading [4]. These techniques require N times the amount of resources to perform the redundant computation, and their focus has not been on low-level systems components. In contrast, recovery blocks [2] rely on re-execution of a task when a fault occurs. These techniques are less applicable to highly concurrent system components with complex dependencies on other system services.

System-level fault tolerance. A number of systems tolerate system faults using external trusted stores [5], or separating client data [6]. C^3 provide an efficient, predictable recovery solution that focuses on considering and minimizing recovery costs on the timing properties of the system.

Predictable fault tolerance. Past schedulability analysis with fault tolerance focus on *application* recovery (often with temporal replication or job re-execution). A sample of this research includes [7], [8]. In this paper, we investigate *system support* for the schedulability analysis of low-level component failures by providing low bounds on recovery, and avoiding recovery interference. We believe this will motivate future work on the supporting schedulability analysis.

III. Fault Tolerance in C^3

C^3 focuses on interface-driven, on-demand recovery of failed system components, motivated by the following goals.

G1 *Minimizing fault propagation.* In traditional monolithic operating systems, a failure in system services can propagate broadly to other system services. Boundaries to prevent errant memory writes from corrupting arbitrary memory are required. Even with memory protection boundaries, any channel for communication between components can propagate faults. Though this channel for propagation cannot be completely removed, it should be minimized by only enabling communication when required, and statically defining the correct ranges of passed data.

G2 *Component μ-reboot.* When a system component fails, it must be μ-rebooted [9]. The overhead of this operation is unavoidable, and no other component that requires the component's service can proceed until μ-reboot is completed. This operation must then be both efficient and predictable.

G3 *Rebuild a consistent state.* System services are unique in that their internal data-structures often represent side-effects in other components and applications. These include open files, and file contents, pages that have been mapped into other components, and runqueues of threads that execute in other components. These data-structures must be reconstructed after μ-reboot and placed into a consistent state with the rest of the system.

G4 *Predictable recovery.* The system must be able to bound the extent of recovery for real-time tasks, and should prevent recovery interference to higher-priority tasks. Notably, to the largest extent possible, the recovery procedure should be performed at the priority of the highest-priority task that requires the failed service. Figure 2 emphasizes the difficulty in providing predictable recovery.

G5 *Fault detection.* An erroneous condition is dealt with as a fault after it is detected. The sequence in Figure 2 begins *after* successful detection. This paper does not focus on fault detection, instead relying on programmer annotation (`asserts`) and hardware faults (*e.g.* page-faults).

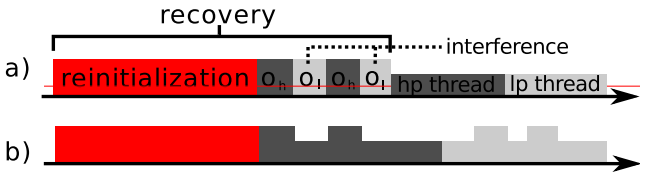


Fig. 2. A timeline of recovery for a system that includes the time for reinitializing the component, and the data-structures in the component for objects provided to a high-priority and low-priority thread. (a) does traditional *eager* recovery with recovery inversion for time spent recovering low-priority objects, and (b) *on-demand*, priority-driven recovery that executes recovery of objects as they are operated on by threads. Thus these objects are recovered for lower-priority threads at the correct priority.

A. Composite *Component-based OS*

We implement the C^3 prototype on top of the Composite component-based OS (composite.seas.gwu.edu). Abstractions, mechanisms, and policies for resource management and usage are defined in separate, fine-grained components. Each component exports a set of interfaces including functions that other components can invoke, and has a set of dependencies on components with functions it can invoke. By default, protection domains (provided by hardware page-tables) isolate each separate component, requiring invocations between components to be mediated by the kernel to switch page-tables. Additionally, a capability-based access control mechanism limits communication between components. This fine-grained isolation, and controlled communication helps provide **G1**.

Fault detection. Composite converts hardware exceptions such as page-faults into component invocations of special

handler functions. In this way, a fault produces execution in a recovery component that implements the recovery procedure.

Support for μ-reboot. We observe that the initial phase of recover where the memory of the failed component and execution in it is re-initialized must be completed before any additional recovery of state is performed. It therefore represents the bare-minimum cost of recovery (designated as "reinitialization" in Figure 2). In C^3, we optimize this cost by avoiding to clean up the previous component (e.g. analogous to killing the process), and also avoid recreating a new one (fork a new process to replace the previous). Instead, we reuse the existing component by reinitializing its memory. This avoids the costs memory management operations for page-table, and component meta-data recreation. Thus the costs of μ-reboot are bounded by a function of the size of the component image, and the cost of execution in `cos_init` – the component equivalent of the `main` function. This optimized and predictable support for μ-reboot satisfies **G2** and aids in **G4**.

B. Interface-driven State Replication and Reconstitution

Component's interfaces are explicit, and COMPOSITE has mechanisms to enable the stubs that marshal and demarshal invocation arguments to be customized. C^3 uses this support to provide light-weight stubs that: 1) uses a parametrized state machine to track the *current* state of individual objects (memory pages, threads, locks, files descriptors) that are operated on by the interface – for example, tracking the file path, current offset into the file, and data being written, 2) includes a number of *introspection* functions that can be used to ascertain the current state of an object provided by an interface, 3) include recovery functions – triggered when a depended-on component fails – that recreate a consistent state for each object (toward **G3**) through use of the normal functions in the interface (*e.g.* by re-opening and seeking to the current position), 4) and check for out-of-specification arguments to aid in the detection of faulty communication (toward **G5**).

Interface introspection. When a component is μ-reboot, it will introspect on the interfaces it depends on to recreate its own state for any objects previously provided to it – for example, open network connections, or to notify that component that it no longer needs those objects – for example, any locks that were only pertinent to pre-μ-reboot execution.

C. Demand-driven Recovery

A key question we address in C^3 is *when* should the state for the objects previously provided by a failed component be rebuilt using the interface-driven technique? We have evaluated two designs: (1) Eager recovery in which part of the μ-reboot process is to upcall into each component that depends on the failed component, and have them recreate all objects via interface functions. (2) Demand-driven recovery in which no object recovery via interfaces are recreated immediately upon failure. Instead, this recreation is performed on-demand: when a client component performs an operation on a component (*e.g.* a file read, blocking a thread, or mapping a page), the stub is notified that a failure has happened in the past, and it recreates the current state of the object. Figure 2(b) depicts the difference between these two approaches in at timeline starting with fault detection.

The significance of these two options is substantial: eager recovery results in execution *at the time of μ-reboot* to recovery *all* object state that existed in the component when it failed. This imposes recovery interference as lower-priority objects are recovered while higher-priority tasks must wait. If best-effort tasks utilize a shared service with real-time tasks (*e.g.* the scheduler), and they have no bound on the number of objects they can create (threads), recovery interference is unbounded. C^3 is therefore focused – by **G4** – on on-demand recovery to enforce timing bounds on recovery parameterized only by component re-initialization time, and the number and type of objects each thread uses.

IV. EVALUATION

We evaluate three important system components: 1) the system scheduler, 2) the system physical memory manager and mapper, and 3) a RAM-based file system. These components export the following objects, respectively: threads – that are created, blocked, woken, and destroyed; memory pages – that are granted, aliased (for shared memory), and revoked [10]; and files – that are opened, closed, read, and written.

Experiments are run on an Intel i7-2760QM running at 2.4 Ghz. Though these numbers might not be representative of constrained system performance, many non-deeply embedded domains aren't as constrained by size and power. Future work will involve evaluating on more constrained systems.

Workload. The results are generated for each system component while running the following workload:

- *Scheduler (Sched):* Two threads essentially perform a ping-pong, blocking and waking each other in turn.
- *Memory Manager (MM):* Real-time threads are granted memory pages and use those as a statically-allocated region; no further interactions are made with the memory manager. Best-effort subsystems are granted a number of pages, and these pages are aliased three times, and then revoked, which removes all aliases. This process is completed to synthesize memory strain in the system.
- *RAM File System (FS):* Files are opened, written to, read from, and closed.

A. Recovery Performance and Predictability

In this section, we evaluate (1) The cost of the re-initialization phase of component μ-reboot including a) the amount of time spent reinitializing memory, and b) the amount of time spent re-initializing execution in `cos_init`. (2) The cost of interface-driven re-creation of a consistent state for each object. This cost, combined with the number of objects, bounds the recovery interference cost of eager recovery, and the per-task cost of on-demand recovery.

Table I shows the costs of different phases of μ-reboot (memory reinitialization includes `memcpy` and `memset`, and execution reinitialization is execution of `cos_init`), and the cost to recreate, using interface functions, each object provided by each component. Objects used by best effort threads can be more expensive to recover (see the workload

Component	μ-reboot – memory init.	μ-reboot – execution init.	RT task object recovery	BE task object recovery
Sched	7.52 (0.10)	10.15 (1.00)	0.76 (0.06)*	← same
MM	16.06 (0.13)	4.00 (0.19)	0 (0)	5.23 (0.14)
FS	6.37 (0.06)	2.66 (0.08)	26.30 (2.20) / 5.00 (1.21)*	← same

TABLE I
THE AVERAGE (STDDEV) COSTS IN μ-SECONDS OF KEY RECOVERY OPERATIONS. * SEE IN-TEXT QUALIFICATION.

description) than for real-time threads. We observe a large deviation between the first file re-created and the rest due to stack initialization [11], thus "first/rest" for file object recovery. The per-thread recovery cost in the scheduler is incurred at recovery time, not on-demand as recovery is performed using introspection on the kernel interface. In on-demand recovery, the best-effort threads are recovered during idle time, thus avoiding interfere with real-time threads.

On-demand recovery. The per-object overheads for the objects used by real-time threads will impact only the execution of those individual threads when a recovery happens. This cost will factor into the schedulability test for each thread. The best effort (and lower-priority thread) overheads will *not* effect real-time threads.

Eager recovery. The per-object overheads for *all* threads would need to be factored into the response-time analysis for *each* thread, even those of the highest priority. Given a sufficiently large number of objects used by lower-priority (including best-effort) threads, this can have a highly-detrimental effect on recovery timing guarantees for real-time tasks.

Perspective: Linux forking overheads. To put these numbers into perspective, we compare the cost of μ-reboot in C^3 to the cost of forking a child process in Linux. Specifically, a child process faults, and a notification is delivered via `wait` to the parent, that re-forks the child. This process takes 42.64 + N(1.61) μ-seconds, where N is the number of pages in a static array in the child (when the child starts execution, it touches these pages to make sure they are mapped in). As the binary becomes larger, it comparably takes longer to fork it. Note that a direct comparison to C^3 cannot be made here, as Linux does not implement system services as processes, thus Linux cannot be used for system-level recovery. We include these numbers only to put the relative performance of recovery in C^3 into perspective.

B. Recovery Infrastructure Overhead

The interface-based tracking of state has an overhead on all component-communication. For example, when a file is opened, a data-structure on the client side is created to describe that file's path, mode, and offset; when a thread blocks using the scheduler interface, a data-structure saves this state. The cost, in μ-seconds, of non-faulty invocations for our three studied components for the each iteration of the workloads described above are 1.45 + 0.34 (0.04) – formatted as "performance without tracking + overhead of tracking (standard deviation of overhead)", 0.52 + 0.09 (0.03), and 1.65 + 1.8 (0.17), for the scheduler, memory manager, and file system, respectively. Though this overhead is not insignificant, there is room for optimization and we believe it is acceptable for systems that require pervasive fault tolerance.

C. Success-rate of Recovery

Fault Model. We are targeting transient faults introduced by in on-chip structures. We mimic these faults using bit-flips within registers. We inject these faults every timer-tick (100 times a second) by iterating through all threads and flipping register's bits only if they are executing within the target component. Undetected faults are ignored when measuring recovery success. Successful recoveries are defined by the continued execution of the workload post-recovery.

In this manner we inject 80 faults into the scheduler, and 300 each into the memory manager and file system. For these injected faults, we observed a *100% recovery rate* as the system is rebuilt via interfaces.

V. CONCLUSIONS AND FUTURE WORK

This paper has introduced the Computational Crash Cart system for the efficient and predictable tolerance of system-level faults. We've introduced *recovery interference* in which lower-priority tasks cause possibly unbounded interference in the run-times of higher-priority tasks due to the mechanisms of service recovery. To address this issue, we've also discussed the *on-demand recovery* of system components to ensure that all phases of recovery are performed at the proper task priority. We've shown the interface-driven recovery in C^3 is also efficient. In future work, we will address the issue of creating a scheduling analysis for systems with such support for system-level fault recovery. Source code and more information about the system is found at the COMPOSITE webpage: composite.seas.gwu.edu.

REFERENCES

[1] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008.

[2] B. Randell and J. Xu, "The evolution of the recovery block concept," in *in software fault tolerance*. John Wiley and Sons Ltd, 1994.

[3] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, December 1985.

[4] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the tenth ACM international conference on Embedded software*, 2012.

[5] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *Proceedings of the Sixth European Dependable Computing Conference*, 2006.

[6] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: Improving reliability through operating system structure," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[7] G. M. de A. Lima and A. Burns, "An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems," *IEEE Transactions on Computers*, vol. 52, pp. 1332–1346, 2003.

[8] P. Mejia-Alvarez, H. Aydin, D. Mosse, and R. Melhem, "Scheduling optional computations in fault-tolerant real-time systems," in *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, ser. RTCSA '00, 2000.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot–a technique for cheap recovery," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004, pp. 31–44.

[10] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[11] Q. Wang, J. Song, G. Parmer, G. Venkataramani, and A. Sweeney, "Increasing memory utilization with transient memory scheduling," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, 2012.