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

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

***Abstract*—Predictable reliability is an increasingly important aspect of embedded and real-time systems. This includes the ability to recover from unknown faults in a manner that maintains system timing guarantees, even when these faults occur within system components. This paper presents the C^3 system, which is the first system implementation we know of for predictable, system-level fault tolerance that doesn't require physical redundancy. We introduce both the system design, and two timing analyses that enable the predictable recovery from faults in operating system components, and identify *recovery inversion* as a main impediment to schedulable recovery. C^3 provides fault-tolerance for low-level system components using a combination of efficient μ-reboots, and an interface-driven mechanism to recreate component state. C^3 introduces *on-demand* recovery that properly prioritizes aspects of the recovery process to avoid this inversion and not inhibit system timeliness. We compare this system to both *eager* recovery, and to checkpointing of a paravirtualized real-time OS.**

I. Introduction

Real-time and embedded systems must often meet conflicting demands including predictability, efficiency, and reliability. As these systems control more of the physical world, system reliability is an increasingly important dimension of a system's correctness. Environmental effects such as Single-Event Upsets (SEUs) caused by radiation can cause corruption of transistor state leading to bit-flips in chip structures [1]. Additionally, as chips continue toward smaller processes (*e.g.* down to a 22nm feature size and beyond), micro-architectural effects will increasingly deviate from their specified behavior due to manufacturing error, heat damage, and other physical effects. In fact to surmount these challenges, a call for a "concerted effort on the part of all the players in a system design" was made [2].

Dependable systems are those that are able to avoid service failures that are more frequent or severe than is acceptable, in spite of erroneous hardware and software [3]. Toward dependability, fault tolerance is an important component of system design. Upon detection of an error, the system is recovered to a state without the erroneous behavior. Software infrastructures for fault tolerance show promise to aid continued process-driven progress, and the ability to maintain dependable service in the presence of adverse environmental stimulus. This paper focuses on the tolerance of intermittent faults (including transient faults) [3] that are due to environmental, process-related, or other non-deterministic sources. Due to the very nature of a fault happening at unpredictable locations within system execution, it is important for the tolerance of faults even in system-level code such as the operating system. Additionally, as real-time system's correctness considers temporal properties, the faults, and the recovery mechanisms must be integrated into system schedulability analysis.

*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.

Fault tolerance infrastructures often trade resource consumption for the ability to recover from faults. Notably, redundancy is a common technique for tolerating faults and can rely on hardware or software, using active or passive replication. However, the existing techniques usually have a negative impact on system Size, Weight, and Power (SWaP). Embedded systems with stringent SWaP or cost constraints benefit from lighter-weight approaches for uni-processors that attempt to avoid redundant work and are efficient and predictable. Thus, we focus on node-level fault tolerance with the goal of increasing dependability by decreasing the probability that a fault will cause node failure.

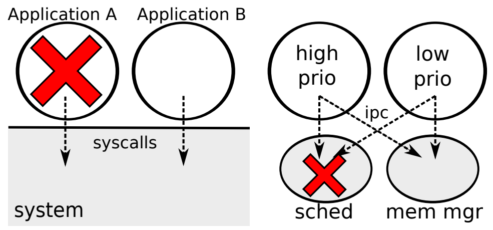


Fig. 1: Application-level faults are recoverable by restarting the application, or restoring a previous state. System-level faults effect the timing of multiple applications, and are difficult to recover from even when system services are in separate protection domains communicated with via Inter-Process Communication (IPC).

Much real-time, fault tolerance research assumes that faults cause failures at the application (user) level [4][5][6] as depicted in Figure 1. User-level applications have desirable properties that enable fault tolerance: First, *temporal redundancy* can be harnessed to re-execute a job that experienced a fault. Recovery of the application by re-execution can be done using techniques such as recovery blocks [7], process checkpointing [8], or application-specific management of state reconstruction that can be used to recover from a fault. Second, a single fault will only cause the recovery of a single application. The impact of re-execution is restricted to the timing of same and lower-priority applications.

Tolerating faults in *system-level* software with an emphasis on predictable recovery has received significantly less attention. Faults in system software (e.g, system scheduling, memory management, and I/O processing) are significant. Not only do they possibly compromise the *entire* system, but they are common: nearly 65% of hardware errors corrupt OS state [9] before they're detected. Figure 1 depicts system-level faults affecting multiple applications. A number of factors make the tolerance of system-level faults particularly difficult.

(1) System services that must be recovered have inconsistent state with the rest of the system. A failed scheduler must reach a consistent state which includes all threads in the system in the proper state. (2) Predictable recovery is difficult as the time to recover a failed service affects *all* tasks dependent on it. Additionally, recovery might be expensive due to the consistent state reconstruction for low-criticality tasks. This leads to *recovery inversion* where recovery of "low-criticality data-structures" delays high-criticality task execution. Recovery inversion is intrinsic in system-level recovery where state is recovered that corresponds to tasks of different priorities. (3) Multiple application requests concurrently execute in separate threads in system services. So recovery must work in a concurrent environment. (4) Fault propagation is significant as monolithic systems enable the trivial spread of an errant behavior throughout the system.

In this paper we introduce C^3, the Computational Crash Cart, implemented on the COMPOSITE component-based OS [10] that focuses on *predictable recovery from system-level faults*. C^3 uses a combination of fine-grained fault isolation, μ-reboot, and interface-guided consistent state recovery to tolerate faults even in some of the lowest-level system services. As a comparison case, we also present a low-overhead implementation of component checkpoint and restore. For both techniques, we derive their timing properties, and study how much impact they have on system schedulability in hard real-time systems.

Contributions.

- We detail the design and implementation of C^3 which is the first system, as far as we know, to provide full, predictable recovery from system-level faults without the resource costs of replication. We introduce a novel and effective interface-driven recovery method that is used to reestablish the state of a failed system component.
- We introduce the concept of *recovery inversion* in which the recovery process unduly interferes with the timing properties of real-time tasks. We detail a system recovery mechanism based on *on-demand* recovery to minimize and bound this inversion.
- As a comparison case for the C^3 recovery mechanisms, we present the implementation of an efficient and predictable checkpointing facility for embedded systems. While this mechanism doesn't necessarily scale beyond small (memory) systems, or systems with more than one component, we find it is effective within these assumptions.
- We focus on the timeliness of system recovery. We introduce a schedulability analysis that accounts for faults that occur at a minimum, specified rate.
- We evaluate the ability of C^3 and checkpointing to maintain schedulability of (schedulable) random task sets in the presence of faults.

II. RELATED WORK

System reliability and fault tolerance have been widely studied for decades [11] [12] [13] [14] [15], with the aim of delivering predictable service in the presence of faults.

Fault models. This work assumes a periodic fault model where faults have a minimum inter-arrival time. SEUs are simulated by this model. Though it is not realistic to expect periodic faults, modeling them as periodic enables a system designer to understand the system timing and recovery properties with a certain minimum fault inter-arrival. Thus, given expected fault distributions, a practitioner can make an engineering decision if the system is tolerant enough. Additionally, we pessimistically assume that each of these faults results in an error (i.e. no faults are harmless) – if faults are harmless, fault tolerance facilities are not required. In contrast to the periodic fault model, the *fault-burst* [16] model represents continuous faults occurring within a window. Future work will consider this model.

Replication. Existing fault tolerance relies on either *active* or *passive* replication [17], [18], [19] in distributed contexts with computation on multiple nodes. Process level [20], [21] replication can be made using a set of redundant processes per original application process and compares their output to ensure correct execution. Similarly, [22] replicates execution of event-driven state-machine computations to detect and prevent the propagation of faults. *N*-version programming [23] is based on replicated computation, *e.g. N* independent versions of the same program written by the different groups. Recovery block [7] rolls the system state back and another computation can be tried based on the result check. These existing techniques are either expensive, or designed to address persistent hardware faults or deterministic faults [3] (where a module will always fail for a given input). C^3 focuses on predictable, efficient system-level recovery from transient faults without physical machine, or process-level replication.

OS reliability. Nooks OS [24] achieves reliability by isolating the OS from device driver failures. Notably, we do not focus specifically on device drivers, and instead on general system services. CuriOS [25] achieves OS service reliability by saving client-specific state information in protected memory. Minix3 [26] used the *Reincarnation Server* to restart the faulty services. However, none of the these consider the temporal properties, or schedulability of the system.

μ-reboots [27] are used to provide reliability to web-services, and rely on an application being decomposed into separate modules, each of which can be restarted individually. The difficulty is rebuilding appropriate state upon reboot. For web services, this is simple as a data-store – often implemented as a database – is used to rebuild an up-to-date state. Without a mechanism for re-establishing a component's state, μ-rebooting components alone does not perform system recovery. A contribution of this paper is not only detailing an efficient method of μ-rebooting a low-level, failed component, but also a predictable interface-driven recovery mechanism.

Real-time application-level fault tolerance. Schedulability with temporal redundancy at the application-level has been investigated in the past [4], [5], [6], [8]. That research has shown that the schedulability of applications can be guaranteed in the presence of faults with proper design. For example, the feasibility analysis of fault tolerant real-time task sets with the fixed priorities was proposed in [11]. However, none of these address predictable fault tolerance of system level components.

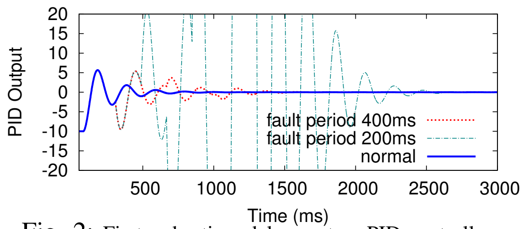


Fig. 2: First-order time delay system PID controller

Checkpointing. Different checkpoint schemes have been investigated [28], [29], [30] including off-line, on-line, probabilistic and deterministic checkpointing. However, if a fault occurs in a system level service, checkpointing can not always guarantee a stable system. Figure 2 shows an unoptimized PID controller output for a first order time delay system under normal conditions and under faults tolerated with checkpoints. The solid line shows the expected output of a PID controller converging to a targeted value of 0 around 800ms. The dotted line represents the output of a PID controller when the fault occurs in the scheduler every 400ms and the output slowly converges around 1500ms with some visible disturbance due to recovery. If the fault occurs every 200ms, a large variance in the output can be seen as the dashed line that converges slowly beyond 2700ms. In a *cyber-physical system*, checkpointing requires careful consideration due to the *inconsistency* between the current physical system state (*e.g.* a mobile system's physical location) and the state expected by the control system from checkpointed information. For checkpointing to be applicable to an embedded system, either this inconsistency must be acceptable for a specific control system, or the sensors of the system must report absolute, not relative system description. For example, this would mean using GPS rather than an accelerometer to track location.

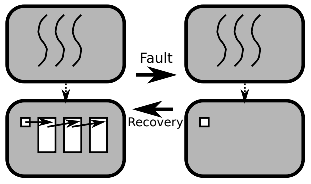
III. System Design

A. C^3: The Computational Crash Cart

C^3 [31] is a system designed around the goals of tolerating faults in system-level components that *all* applications are dependent on, doing so predictably, and with minimal (CPU and memory) overhead. C^3 is built on Composite.

Composite background. Composite is a component-based OS in which system policies and most abstractions are defined in fine-grained components. Components in Composite are code and data that implement some functionality that exports an interface of functions through which other components can harness that functionality. Components have a set of functional dependencies on interfaces that must be satisfied by other components. Components in Composite execute at user-level in separate hardware-provided protection domains, and access to resources and communication channels is restricted by a capability system. Even low-level services such as scheduling [32], physical memory management and mapping, synchronization, and I/O management are implemented as possibly hierarchically-arranged [33], user-level components. Invoking a function in the interface of a depended-on component transparently triggers thread-migration-based [34], [35] synchronous inter-component communication (called "component invocation"). In this way, the same schedulable thread executes through many components, and can be preempted at any time. By default, components are *passive*. A component becomes active only when invoked by threads from other components, or when a thread is explicitly created in it. Multiple threads can concurrently execute within a component and predictable resource sharing protocols are required just as they are in system services of more traditional OSes.

Limiting Fault propagation via fine-grained component isolation. As fine-grained components are memory-isolated via hardware page-tables, fault propagation via errant memory modifications cannot cross component boundaries. Even a simple web-server consists of 25 components, thus constraining fault propagation significantly. Components interact through the functions in their interface. Though faults can propagate through function invocations between components, in fault injection experiments we have not observed this to happen. If this propagation became significant, well specified interfaces with pervasive error checking and validation of inputs could aid in its prevention (as in [36], [37]). Pervasive fault isolation is not too heavyweight. A Composite webserver performs as well or better than Apache on Linux [38]. Though Apache is not the fastest web-server, this demonstrates that Composite can achieve useful levels of performance even with the overheads of pervasive fault isolation.



```
thdid_t sched_thd_crt(char *params);
int sched_exit(thdid_t current);
int sched_blk(thdid_t current,
              thdid_t dependency);
int sched_wakeup(thdid_t target);
int sched_set_params(thdid_t target,
                     char *params);
```

(a) (b)

Fig. 3: (a) Consistency between component data-structures tracking threads in a client and a server scheduler. When the scheduler fails, it loses data-structures holding the existing threads. Recovery restores a consistent state for those data-structures. (b) Example interface for the scheduler.

Component consistency. Consistency between the state of multiple components can now be more concretely defined. The "client" of an interface has a set of active objects that have been provided by the "server" component. For example, a real-time component might have used a scheduler component to create three threads. These objects have had operations performed on them via interface functions (*e.g.* threads have blocked, and woken up) that have expected (implicit) semantics (Figure 3(b)). A consistent state between a client and server simply dictates that the independent data-structures of each component contain the same set of objects, and that the same operations have been performed on the respective, component-local data-structures. Figure 3(a) depicts this inter-component state consistency. On the left, three threads in the client are consistent with the run-queue in the scheduler with the three threads. Upon failure of the scheduler, the component is recreated with an initial state that notably lacks a consistent data-structure for the three threads as seen by the empty run-queue. The goal of C^3 is to predictably rebuild a consistent state upon system component failure.

Component consistency example. The lock component provides locks to other components, and uses the scheduler interface to block and wakeup threads (it also creates dependencies with the `sched_blk` function for priority inheritance which we will ignore here). The lock component tracks threads blocked on a lock, and the lock holder in its data-structures. It uses the `sched_blk` call to block the contending threads. When the lock holder releases the lock, it will call `sched_wakeup` on all contending threads to wake them up. In the scheduler, threads are moved from the run-queue to the blocked queues, and back in response to these calls. If the scheduler and lock data-structures are inconsistent, then threads might be executed by the scheduler when they should be blocked waiting for the release of a lock. Alternatively, an inconsistency could result in a thread not being woken upon lock release, thus threatening the lock implementation's progress guarantees. If the data-structures aren't consistent, then the intended semantics of the lock component cannot be implemented, thus impacting whole system correctness. Components that are side-effect free (purely functional) are better implemented as libraries. Thus, when a component fails, it is, without further action, inconsistent with the rest of the system as its state must be assumed corrupted.

B. *C^3: Interface-Driven Fault Recovery*

Interfaces between components include a set of interrelated functions that commonly operate on a small number of different object types. Due to the separation of concerns promoted by component-based systems, the number of object types manipulated in an interface is typically one. These object types are threads for the scheduler interface, files for a file-system interface, or virtual memory pages for a memory management and mapping interface. C^3 tracks the semantic information about the state of these objects within the stub code interposed on the interface to rebuild a consistent state between components. Figure 4(a) depicts an invocation between components. Specifically, from a client component to the `sched_blk` function in the scheduler. The kernel must mediate communication between separate hardware protection domains, but interface- and function-specific stubs interpose on invocations. Traditional stubs are generated by an Interface Definition Language (IDL) that marshall arguments – pointers cannot be shared between protection domains. Stubs in C^3 also track the *current state* of each object. C^3 interfaces define each object as a state machine where the semantic transformations that different functions in the interface have on the object represent transitions between states. This is in fact a simple specification of the protocol to organize function invocations in each interface [36], [37]. C^3 expands this idea, by using the tracked object's state in interface stubs as a means to restore consistency.

An example of an object's manipulation in an interface is seen in Figure 3(b). Thread objects are manipulated by functions in the scheduler interface, and transfer between an initial state of runnable (`THD_RUNNABLE`) and blocked (`THD_BLOCKED`) via invocations to `sched_blk`, and back with `sched_wakeup`. Thus thread objects in this interface are described with two states. When an object is destroyed (*e.g.* with `sched_exit`), it is no longer tracked – a termination state. When a server fails, the client stubs recreate objects and transition them into the consistent state via the functions in the interface themselves. Server stubs provide the ability to query, or "reflect" an object's state by the client, enabling the client to rebuild its state upon failure. The stub code executes in both client and server, thus providing recovery guidance if the other side of the protection domain fails.

Object state tracking via interface interposition. To understand how stubs are involved when a component invokes a function provided by another, we use `sched_blk` – the function that blocks a thread – as an example. In Figure 4(a), $\textcircled{1}$ function invocation is redirected at link-time to the stub, $\textcircled{2}$ the current state of the operated-on object is found and updated by transitioning it to another state where appropriate according to the function called, $\textcircled{3}$ the kernel switches to the destination component's protection domain, $\textcircled{4}$ the state is redundantly tracked in the server stubs to aid client recovery, and $\textcircled{5}$ finally the destination function is called.

Figure 5 depicts the client stubs for the scheduler interface in Figure 3(b) that track a simple state-machine for each thread in the client stubs. Here we assume that the `cos_invoke_server` macro understands how to utilize the kernel ABI to actually invoke the corresponding function via kernel-mediated component communication.

As mentioned before, C^3 assumes that faults don't propagate via invocations. As pointers are not passed between components, and argument validation must be performed for invocations, it is difficult for faults to propagate in this manner. Our fault injection experiments confirm this assumption by producing only local faults. If invocation-based propagation becomes significant, future work must consider interface-level [36], [37] fault detection and/or recovery via recursive reboots [39].

Stub-driven recovery of failed servers. Note the logic in the client stubs in Figure 5 includes checks on a local variable, `fault`, and triggers recovery logic if this variable is non-zero. As part of the component invocation, this variable is set if the kernel detects that a fault occurs while the server is processing this request, or *if a fault occurred in the server since the last time the server was invoked.* The kernel maintains a version for each component that represents the number of times it has been recovered, and a comparable server version number for each invocation path (communication capability) from each client. If a version mismatch is detected on an invocation, the depicted fault handling path is executed in the client stub, and the client's version number is updated to avoid reporting further faults[1]. Once a fault is detected, the client stub attempts to use the interface's functions themselves to rebuild the object in a consistent state, and ensures that any existing objects tracked in the stubs will trigger the recovery

[1]Importantly, while the μ-reboot process is recovering a safe state, client versions are left un-updated. This prevents *any* execution within the component between when a fault is detected, and when the component's initial, safe state is recovered. Only when the component is brought to a safe state, thus can receive further invocations, are client's snapshots of the version updated. This is necessary to prevent further fault propagation, and enables recovery even in the presence of concurrent component execution.

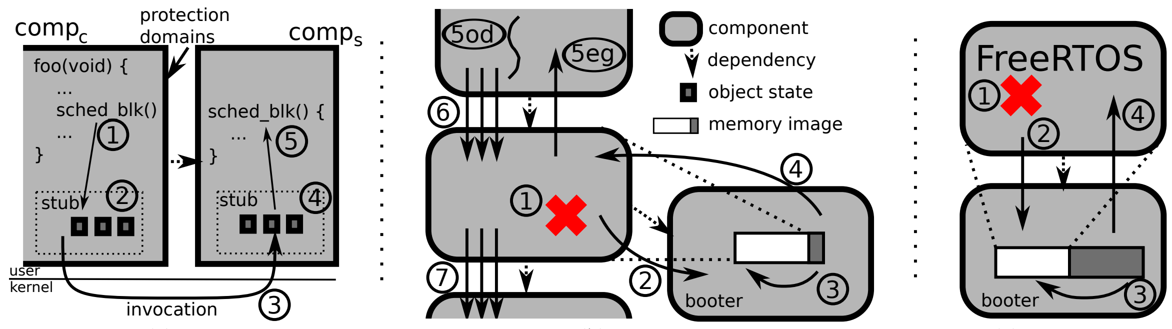


Fig. 4: Recovery mechanisms and processes in C^3. (a) Component invocation between components mediated by per-object, state-tracking stubs, and the kernel. (b) The timeline between a fault in a component, and its restoration via the booter-provided μ-reboot, and the recovery-aware interfaces. (c) Checkpointing and restore of the para-virtualized FreeRTOS system.

```
enum thd_state_t { THD_BLOCKED, THD_RUNNABLE };
struct thd_metadata {
  enum thd_state_t state; // current state
  thdid_t thdid, dependency;
  char *param; // describes priority, period, etc...
};

thdid_t client_stub_sched_crt(char *param) {
  int fault = 0;  // set if a fault occurs in the server
  thdid_t thdid;
  struct thd_metadata *obj;
retry:
  thdid = cos_invoke_server(sched_crt, param); //call server
  if (fault) goto retry;
  obj = obj_alloc();            // allocate tracked object
  obj->state = THD_RUNNABLE; // initial state
  obj->thdid = thdid;           // saved object data
  obj->param = alloc_copy(param);
  return thdid;
}

int client_stub_sched_blk(thdid_t curr, thdid_t dependency){
  int fault = 0, r;
  struct thd_metadata *obj = obj_lookup(curr); // find obj
  assert(obj && obj->state == THD_RUNNABLE);
  obj->state        = THD_BLOCKED; // state transition
  obj->dependency = dependency;    // save relevant obj data
  r = cos_invoke_server(sched_blk, curr, dependency);
  if (fault) return client_stub_recover(obj);
  return r;
}

// generic client recovery function
int client_stub_recover(struct thd_metadata *obj) {
  cos_invoke_server(sched_set_params, obj->thdid,
                    obj->param); // reestablish parameters
  switch(obj->state) {
  case THD_BLOCKED:
    // transition thread into consistent THD_BLOCKED
    // state using the sched_blk function itself
    return cos_invoke_server(sched_blk, obj->thdid,
                                        obj->dependency);
  case THD_RUNNABLE:
    // runnable threads are trivially recovered: if
    // they are executing this code, they are not blocked
    return 0;
  }
}
```

Fig. 5: Simplified and incomplete client stub code for the scheduler interface, and recovery upon detection of a fault in the server by recreating the thread in the server. This code corresponds to that in Figure 4(a), step 2.

code for that object. This is seen in Figure 5 by resetting the thread's initial parameters in the (newly initialized) scheduler, and transitioning the thread's state in the server into a blocked (consistent) state by invoking `sched_blk`. Note that this entire process is *completely transparent to the clients.*

Client stub code is guided by the goal of finding a path through the state machine from object initialization to the current client's object state. This path is a sequence of transitions encoded by the sequence of interface functions to be invoked in the recovery routine. Our experience so far indicates that the code for these state machines is straightforward to generate manually. However, for a large number of interfaces, or for objects with complex state machines, the automatic generation of these stubs via the IDL would be valuable to reduce programming error in them.

Note that the code in Figure 5 implements *on-demand* recovery. Only when an object is passed as an argument to a function in the interface, is the recovery on that object conducted. It is performed within the context of the thread that is accessing the object, thus avoiding recovery inversion during object recovery. Eager recovery would simply iterate through all objects in the client and recover them all when a fault is detected.

Stub-assisted recovery of failed clients. When a client component fails, it is often important for it to rebuild objects in a consistent state with those provided by a server. To facilitate this process, stubs provide a set of functions to reflect on the state of objects provided by the server to each client. This reflection (analogous to language-based reflection) enables the client to obtain the state of the object in the server, so that the client can recreate it with a consistent state. The server stubs in (4) in Figure 4(a) also track each thread as in Figure 5, but we omit them here.

The system call interface is treated as a component interface, and the equivalent of server stubs are coded into the kernel. Thus, the kernel provides reflection capabilities on the kernel-provided objects, such as virtual pages and threads. A simplified function in the system call API is **thdid_t** cos_thd_reflect (**int** iter), which enables a scheduler to iterate through and retrieve all threads it has previously created before it failed. This enables low-level components that use the kernel directly to reconstruct a consistent state for kernel-provided objects. For example, the scheduler directly creates and dispatches between threads. A failed scheduler uses the above function to determine which threads it had previously created. At that point, it can context switch to them, and when a thread invokes the scheduler using client-driven recovery as in Figure 5, it will reestablish a consistent state (*i.e.* put the thread back into the proper state, with consistent parameters).

C. C^3: Fault Recovery Procedure

Given the interface-driven recovery detailed in the previous section, Figure 4(b) shows the steps taken for recovery in C^3.

We assume a system component that is depended on by others, and might use the services of other components as well.

(1) A fault occurs and is detected (as discussed later, we currently assume fail-stop semantics). This work does not focus on fault detection, and our fault injection experiments indicate that hardware exceptions and `assertion` statements have detected many faults. Research into advanced detection methods (*e.g.* using compilers [40]) is complementary to fault tolerance in C^3.

(2) COMPOSITE converts exceptions into component invocations. The `booter` component receives this exception and immediately makes it so that any non-recovery thread executing in the component will cause a comparable exception. This avoids concurrency issues in that component. At this point, a lock is also taken for the component. This is an important step, as locks in COMPOSITE support priority inheritance. Given this step, recovery will occur exactly at the priority of the highest-priority thread attempting to use the component (keeping in mind that any thread invoking the component will cause an exception into the booter until recovery is complete).

(3) The `booter` component is in charge of initially booting the component, and stores an *image* of the *initial state* of the component. This state is *safe* in that C^3 knows it is unaffected by the fault that occurred. The overhead of this step is dominated by `memcpy` and `memset` to reset the component. This step is similar to μ-reboot [27] in that it restores a safe, initial state. It is optimized in C^3 to avoid creating a new component with separate page-tables (analogous to killing the previous process, and recreating it). Instead, this overhead is avoided by using the pre-existing structures and simply reinitializing memory. For this reason, μ-reboot in C^3 is at least a factor of two faster than process creation in Linux. This work assumes that neither the `booter` nor the kernel suffer faults as they are instrumental in the recovery process. We discuss this assumption later in this section.

(4) An upcall is made to the component for re-initialization.

(5eg) C^3 supports two mechanisms to recreate consistent states of the component with those that depend on it. The first is *eager* ("eg" here). All objects from all components that depend on the failed component are rebuilt at this time, eagerly. Upcalls are made into each component, and the recovery procedures in the stubs are activated. Though straightforward, this has a negative effect on the timing properties of the system. Some components might contain only low-priority threads, but their objects are recovered at this time, regardless. Thus, high-priority/criticality threads must wait until all of these objects are recovered. This creates significant (and possibly unbounded) *recovery inversion*.

(5od) Instead of eagerly recovering a consistent state for all objects, C^3 can resume normal execution of threads at this time. Later, when the system switches to a thread (at the proper priority), and that thread makes an invocation of the previously faulted component. The stubs know that a fault previously happened (via the `fault` parameter in Figure 5), and will rebuild a consistent state for the objects being operated on at that time. This is done by the actual thread requesting service, therefore at the proper priority. We call this *on-demand* ("od") recovery of a consistent state as it is performed at the time of an operation on an object, not at the time of the fault.

(6) This step depicts the client invocations of the server attempting to recreate a consistent state for each object as seen in the `client_stub_recover` function in Figure 5.

(7) The failed component, receives these invocations from the client, and if they operate on an object not yet represented in the component's data-structures, it reflects on the server interface below it to recreate appropriate objects. For example, when a μ-rebooted scheduler receives a request for a thread it does not have a thread control block for, it will reflect on the kernel to see if the thread exists. If so, it will then create the thread control block as appropriate.

Putting it all together: recovering the scheduler. Schedulers provide the policy for multiplexing the CPU amongst threads. The COMPOSITE kernel provides a very low-level thread abstraction – including register storage and management, and the scheduler simply dispatches between these [32]. Each thread has a set of initial parameters that define its priority, and determine system behavior (see `param` in Figure 3(b)). When the scheduler fails, and it is brought back to an initial state, it does not contain the requisite thread control blocks. First, it uses reflection on the kernel interface via `cos_thd_reflect` to retrieve the threads it previously created. The register and execution state of each thread are stored in the kernel, so when the component data-structures are recreated, the scheduler can directly dispatch these threads. Threads are now consistent with the kernel, but not in the state assumed by existing clients in the system (*e.g.* lock components). When switched to, the thread will eventually execute the client stub code as in Figure 5. This code will bring the scheduler and the client into consistency: if the thread was previously blocked, it will again be blocked; if it had a dependency, it will again regain that precedence constraint (*e.g.* for another thread holding a lock to implement priority inheritance); and it will now have the correct scheduling parameters – all directed by `client_stub_recover`. There is a subtlety with handling the `dependency` if the client blocks the thread. Threads are recovered sequentially, and if a thread in the process of recovery has a dependency on a thread that hasn't been recovered, this must trigger the recovery of that depended-on thread.

This gives rise to a common *design-pattern for recovery* within C^3 components: recovery-aware logic is added to *error-paths* within components. When a client calls a function operating on an object that is not represented in a μ-rebooted component, an error path will typically detect this, and return an error implying an invalid function argument. Instead, C^3 modifies these paths (for example, for thread, virtual page, and file lookup) by introspecting on a component's server's interfaces to determine the validity of the object, and any relevant underlying resources (for example, the kernel thread, physical frame corresponding a virtual page, and memory buffers corresponding to a file, respectively). Though C^3 does require recovery-aware changes to components, most of the recovery complexity is automated by the client stubs.

C^3 recovery complexity. For components to perform recovery, we find that 46, 162, and 93 lines of code[2] are added to the RAM file-system, scheduler, and memory manager, respectively, to support recovery. This constitutes 15.1%, 4.8%, and 19.2% of the total length of these components. Components dependent on these services required *no changes* beyond being relinked with the interface stubs. The stubs themselves are 602, 245, and 358 lines of code, respectively. A single interface with multiple implementations (interface-centric polymorphism is a goal of component-based systems) shares interface recovery code for all of them. Ideally, a small set of interfaces can provide recovery for a wide variety of components. Though these are currently large, and written manually, we are in the process of automatically generating them from terse declarative specifications as part of the Interface Definition Language (IDL).

C^3 Trusted Computing Base for Fault Tolerance (TCB-FT). Systems that provide fault tolerance via software mechanisms will often fail to tolerate faults given a fault in those mechanisms themselves. This motivates a discussion of what the TCB-FT (analogous to the Reliable Computing Base [41]) of the system is. The TCB-FT is the core of software that is assumed to not suffer faults to provide fault tolerance to the rest of the system. A goal, then, is to minimize the size and fault exposure of this TCB-FT.

The TCB-FT in C^3 includes the `booter` that orchestrates the μ-reboot process, and the kernel itself. The booter and kernel are less than 500 and 7000 lines of code, respectively. Additionally, the stub code and data-structures that track the state of objects across interfaces are integral in C^3 fault tolerance support. By many measures, this is a small amount of code (FreeRTOS is considered very small and is less than 4000 lines of code). We do not analyze the fault exposure in on-chip and memory structures of the TCB-FT, and leave that as future work. However, it is important that any increase in the number of in-kernel data-structures including threads and components could increase the fault vulnerability of the TCB-FT. Regardless, shrinking the size of this code-base is ongoing and future work, as is measuring and shrinking the fault exposure of this code.

Compiler techniques for increasing the resiliency of code by adding redundant computation come with significant execution and memory overhead [40]. However, as the `booter` is only executed in the event of a fault, these techniques might avoid incurring significant cost on task execution while still increasing fault tolerance. Though the `booter` would still be part of the TCB-FT, it would be hardened to transient faults itself.

D. System-Level Checkpoint/Restore

System-level checkpointing is an option for fault tolerance that relies on the roll-back (restoration) of the entire system state to a previous safe point (before the fault). Consistency between different services in a system is trivially maintained, as all software is rolled back to a safe state. *Application* checkpointing (e.g [42]), though useful, will not provide a consistent recovery point if a system component such as the scheduler fails; indeed, it often relies on the *system* to provide checkpoint and restore functionality. As noted in Section II, checkpoints do not recover a consistent state with the *physical environment*, so they might not be applicable to some systems.

We implement simple and predictable, yet effective checkpoint/restore functionality in Composite and apply it to a single component: a popular hard real-time, high-confidence operating system: FreeRTOS [43] that we paravirtualized to run on Composite. For simplicity, and to investigate the lower-bounds on the cost of checkpointing and restore (C/R) functionality, we focus on C/R support for a *single component*. Though not as powerful as C/R support for a graph of components, this enables a comparison of a very low-cost C/R infrastructure with C^3 while still providing C/R for a popular paravirtualized RTOS and its applications. Consistent with our assumptions in Section III-A, we assume that faults happen in user-level components that constitute the vast majority of the functionality and runtime on the system. We implement C/R functionality in the `booter` component. At boot time, the `booter` creates the FreeRTOS component, and later creates Composite-level threads on demand for FreeRTOS. FreeRTOS is all allowed to install its own timer interrupt handler, and directly switch between its own threads using Composite support for hierarchical resource management [33]. Consistent with the typical behaviors of a hard real-time system, we assume that all threads required by FreeRTOS are created at system bootup, thus simplifying C/R of thread state. Similarly, the `booter` provides an initial image of memory for FreeRTOS to use at boot time, and FreeRTOS can not allocate additional memory. This physical memory is mapped using shared memory via page-tables into both the `booter`, and the FreeRTOS component.

Figure 4(c) shows the FreeRTOS component executing on the system, and recovery via the `booter`. To checkpoint FreeRTOS, the `booter` periodically copies the active memory for FreeRTOS into an equal-sized extent of memory (the darker region next to the memory image), and saves all thread's state. This operation must occur at the highest priority, as memory cannot be modified while copying. This operation is bounded by the cost of `memcpy`. We avoid *live* checkpointing here – where checkpoints are taken concurrently with component execution – as such an approach induces page faults on memory writes to detect page modifications [44], and would have significant impact on system predictability. Restoring upon a fault is conducted as follows: (1) The fault is detected as above. (2) The hardware exception for the fault is converted into component invocation to the `booter`. (3) The saved checkpoint is copied into FreeRTOS's active memory, and thread state is restored. (4) The faulted thread in FreeRTOS is returned to, now with the register state from when the checkpoint occurred.

FreeRTOS time management. Checkpoint and restore

[2]All line counts derived using David A. Wheeler's 'SLOCCount'.

necessarily creates an inconsistency between FREERTOS's clock, and the real system time. Restoring a checkpoint explicitly rolls FREERTOS back to a previous time. We do not currently implement any intelligent means of updating FREERTOS's clock on restore. However, such mechanisms exist. Paravirtualized support to synchronize virtual time (in FREERTOS) with real time [45], and explicit support in hierarchical scheduling systems [33] for temporal consistency could be applied to this checkpointing system.

Checkpointing trusted computing base for fault tolerance. The implementation of checkpoint/restore in COMPOSITE has a TCB-FT consisting of the checkpoint-aware `booter`, and the kernel. These two bodies of software constitute less than 800 and 7000 lines of code each. As in C^3, compiler techniques for hardening this code could be used to decrease the chance of them being corrupted.

Faults that corrupt the state of memory would require additional action on the part of the `booter`. Specifically the integrity of the system checkpoint must be validated before it is restored. This could be done by using memory replication and voting (*i.e.* keeping 2 or 3 copies of the checkpoint, and validating them before restoring).

E. Recovery Inversion

Each of the approaches we discuss – C^3 eager recovery, C^3 on-demand recovery, and checkpointing – have implications on the system timing properties. Threads must wait for the objects they require to reach a consistent state before continuing execution. Thus recovery is guaranteed to have an impact on system schedulability. However, the different techniques present different amounts of *recovery inversion* in which the recovery of state for low-priority/criticality tasks prevent the execution of high-priority/criticality tasks. C^3 *eager recovery inversion:* all objects are recovered as part of the process initiated by the fault. If best-effort tasks exist in the system, and an a-priori bound cannot be found on their number, then this process is unbounded. Even if a bound can be found, the interference is significant. C^3 *on-demand recovery inversion:* Recovery of a consistent state for objects is performed by the thread that is accessing that object, at its priority. *Checkpointing:* The cost of recovery is mainly a function of the size of physical memory. If this size is large due to the best-effort, or low-criticality tasks of the system, this also entails a fair amount of interference. The effects of the recovery inversion are studied in Section VI.

IV. SYSTEM MODEL

A. Base System

In a fault-free single processor system, let Γ be the set of n periodic tasks, $\{\tau_1, \tau_2, ..., \tau_n\}$ scheduled under preemptive, fixed priority scheduling. τ_1 has the highest priority and τ_n has the lowest priority. Thus, task τ_i has higher priority than task τ_j *iff* $i < j$. The task τ_i can be described by the parameters (e_i, p_i).

- e_i is the *worst-case execution time* required by task τ_i for each of its jobs.
- p_i is the job inter-arrival time (period) of task τ_i. Without loss of generality, we assume that a task's deadline is equal to its period (implicit deadlines).
- $u_\Gamma = \sum_{i=1}^{n} \frac{e_i}{p_i}$ is the utilization of task set Γ
- p_t is the average period of all tasks in Γ
- S is a set of such task sets, $S = \{\Gamma_1, \Gamma_2, ...\}$

Though not necessary for the analysis, for simplicity, we assign priorities according to a rate-monotonic policy.

B. Fault Model

This work focuses on tolerating transient faults at the system level and we assume that overheads for fault detection are integrated into either task worst-case execution times (*e.g.* execution of `asserts`), or into the recovery time of a component (*e.g.* page-fault execution). We also assume that the fault will be detected immediately after corrupting system state – we assume fail-stop behavior. Though this is a significant assumption, past work characterizing faults finds that 65% [24], 80.6% [46], and 93% [47] of injected faults with detectable failures resulted in fail-stop behavior. In future work, we will consider latent faults whose detection is delayed arbitrarily after state corruption.

For the purpose of the timing analysis, the relevant fault properties are defined as:

- p_{ft} is the period of transient fault occurances. It specifies the minimum inter-arrival time between consecutive faults. Unless stated otherwise, we choose 200ms as it is consistent with a realistic fault period stated in [48]. Note that faults correspond to the corruption of state in the system due, in our case, to Single-Event Upsets (SEUs). Errors resulting from the fault cause undesired behavior, which is what is actually detected. p_{ft} denotes the maximum number of detected errors over a window of time, thus pessimistically assumes that each fault causes an error.
- $r_j^x(m_j)$ is the cost to recover m_j objects for task τ_j if component c^x failed. In COMPOSITE, the services are implemented as components, denoted as $c^1, c^2, ...$
- $r_j(m_j) \equiv \max_{\forall x}\{r_j^x(m_j)\}$
- $e_{\mu r}^x$ is the execution time of μ-reboot of component c^x.
- $e_{\mu r} \equiv \max_{\forall x}\{e_{\mu r}^x\}$
- p_{cp} is the period of system level checkpointing. We assume that check-pointing process is atomic and that no fault can occur during checkpointing.
- e_{cp} is the cost of system level checkpointing where $e_{cp} < p_{cp}$.
- r_{cp} is the cost of restoring the system back to the checkpointed state where $r_{cp} < p_{cp}$.

V. TIMING MODEL

A. Response Time Analysis

When there is no fault presented in the system, the schedulability of a task τ_i can be evaluated using the traditional Response Time Analysis (RTA) [49]

$$R_i^{n+1} = e_i + \sum_{j<i} \left\lceil \frac{R_i^n}{p_j} \right\rceil e_j \quad (1)$$

where R_i is the response time of the task τ_i. The second term is the interference due to higher priority tasks.

Task τ_i is said to be schedulable *iff* all its jobs do not miss their deadlines. The task set Γ' is said to be schedulable *iff* all tasks in Γ' are schedulable. Let S' be a set of such fault-free schedulable tast sets, $S' = \{\Gamma'_1, \Gamma'_2, ...\} \subseteq S$.

Past research augmented traditional response time analysis with the fault tolerance overheads of task re-execution, exception handling, recovery blocks, and checkpointing, starting with [11]. Here we provide an analogous expansion of the RTA to include *system-level* overheads and to include the recovery inversion due to system-level recovery.

Fault-aware schedulability analysis. The schedulability analysis needs to consider the effect of fault tolerance overheads on task timing. It becomes more challenging to validate the timing constraints of the system with faults since the faulty service can affect every client's temporal behavior and the recovery adds complexity into the feasibility analysis.

To investigate the feasibility of a system in the presence of faults, we need define the fault-aware schedulability analysis model. S'' is the set of fault tolerance overhead-aware schedulable tast sets, $S'' = \{\Gamma'_1, \Gamma'_2, ...\} \subseteq S'$. Thus, we define the Fault-Aware Schedulability Success Ratio, FASSR, as $|S''|/|S'| \in [0, 1]$. FASSR represents the percentage of fault-aware schedulable task sets, of fault-free schedulable task sets. It represents the ability of fault tolerance systems to maintain system schedulability.

C^3 analysis. C^3 provides two interface-driven system level fault recovery strategies: (1) *On-demand* recovery is aware of the priority during the recovery process. Objects will be only recovered at the priority of the thread that is accessing them. The response time for task τ_i should suffer interference only from the recovery of the objects for higher priority tasks. (2) *Eager* recovery will recover all objects for all tasks at the time of the fault. The response time for task τ_i will not only be affected by the recovery of objects for higher priority tasks, but also by the recovery for lower priority tasks. The response time analysis equation for *on-demand* and *eager* is generalized as

$$R_i^{n+1} = e_i + \sum_{j<i} \left\lceil \frac{R_i^n}{p_j} \right\rceil e_j + \left\lceil \frac{R_i^n}{p_{ft}} \right\rceil (e_{\mu r} + \sum_{j \leq K} r_j(m_j)) \quad (2)$$

where p_{ft} is the fault period and $e_{\mu r}$ is the cost of μ-reboot for the faulty component. $r_j(m_j)$ is the recovery cost of m_j objects for the task τ_j. Notice that $e_{\mu r}$ causes inevitable recovery interference in C^3, as no computation in recovering components is possible until they are brought into a safe state by the μ-reboot.

K is defined separately for each recovery scheme: (1) For *on-demand*, $K = i$ for τ_i. Only the recovery of the objects for higher priority tasks cause interference on task τ_i. (2) For *eager*, $K = n$, where n is the total number of tasks in task set. This implies that the response time of task τ_i is interfered with by object recovery for even lower-priority tasks resulting in recovery inversion.

We assume that real-time tasks use a predictable number of objects (that m_j is known a-priori). We believe this is reasonable as it is common to pre-allocate a fixed amount of memory, use a bounded number of threads, and only access files and mailboxes that are opened at initialization time. In the presence of best-effort tasks where m_j cannot be known, *on-demand* recovery becomes essential for recovery predictability.

Checkpointing analysis. The system state is periodically saved, which is used to restore the system to a safe point if a fault occurs. Both checkpointing and restore overheads must be considered in the RTA.

Therefore the response time analysis equation for system level *checkpoint* is given as

$$R_i^{n+1} = e_i + \sum_{j<i} \left\lceil \frac{R_i^n}{p_j} \right\rceil e_j + \left\lceil \frac{R_i^n}{p_{cp}} \right\rceil e_{cp} + \left\lceil \frac{R_i^n}{p_{ft}} \right\rceil r_{cp} \quad (3)$$

For simplicity, and motivated by system measurements (Section VI), we assume that $e_{cp} = r_{cp}$. We assume that the checkpoint operation itself is atomic (*i.e.* performed at the highest priority).

VI. Evaluation

A. C^3 *System Evaluation*

We evaluate three important components in the system: 1) the system scheduler, 2) the system physical memory manager and mapper, and 3) a RAM-based file system. Unless otherwise specified, experiments are run on an Intel i7-2760QM running at 2.4 Ghz.

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 using `sched_blk` and `sched_wakeup`.
- *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 once, and then revoked, which removes all aliases. This process is completed to synthesize memory strain in the system.
- *RAM File System (FS):* A file is opened, a byte is written to it, read from it, and then it is closed.

Fault tolerance effectiveness: fault injection. We mimic transient faults using bit-flips within registers and 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 a target component. Previous research [50] has shown that using this fault injection technique accurately models actual pipeline transient faults. Previous research [51], [52], [53], [54], [50], [55] has shown that error rates in pipeline logic caused by transient faults are currently higher than error rates in memory. Additionally, the ever-decreasing physical footprint of on-chip transistors increases the impact of transient faults in pipelines[2]. These factors, and a desire for simplicity motivate our fault injection methodology. Undetected faults (*i.e.* those that cause no detectable errors or changes in system behavior) are ignored when measuring recovery success. Successful recoveries are defined by the continued execution of the workload post-recovery.

Component	μ-reboot – mem init.	μ-reboot – exe init.	RT object recovery	BE object recovery	nominal workload overhead
Sched	7.52 (0.10)	10.15 (1.00)	0.76 (0.06)	← same	1.45+0.34 (0.04)
MM	16.06 (0.13)	4.00 (0.19)	0 (0)	5.23 (0.14)	0.52+0.09 (0.03)
FS	6.37 (0.06)	2.66 (0.08)	5.00 (1.21)	← same	1.65+1.12 (0.11)

TABLE I: The average (stddev) costs in μ-seconds of key recovery operations and the infrastructure overhead

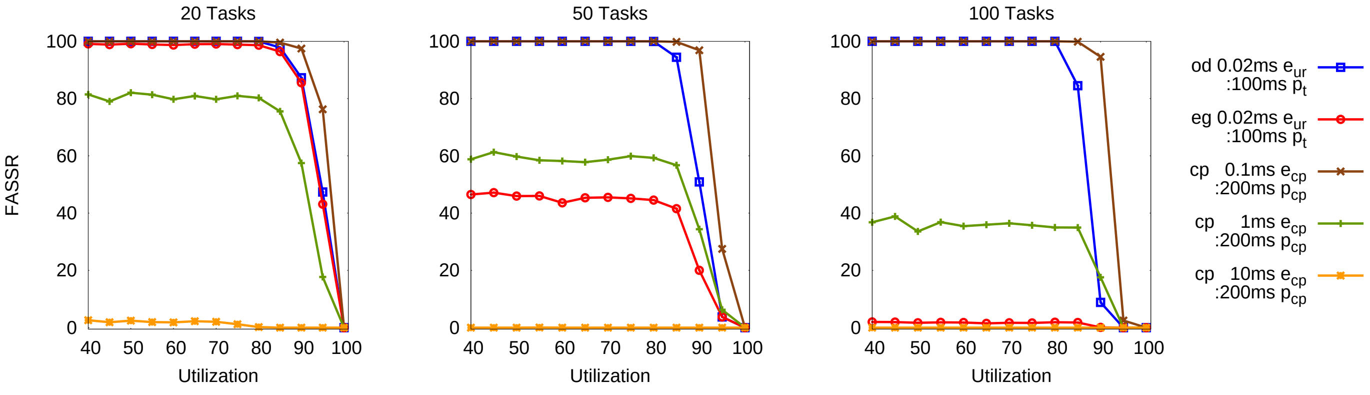


Fig. 6: FASSR vs Utilization. (*od*:*on-demand*, *eg*:*eager*, *cp*:*checkpoint*)

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.

C^3 microbenchmarks. First 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* and *on-demand*. We also evaluate the overhead of interface-based tracking of object state on all component-communication (due to the code in component stubs). This overhead varies depending on the type of interface, and tracked objects.

Table I shows the costs of different phases of the C^3 recovery process outlined in Figure 4(b) (memory reinitialization includes `memcpy` and `memset`, and execution reinitialization is execution of `cos_init`), and the cost to recreate, using interface functions, the objects provided by each component are shown. Objects used by best effort threads can be more expensive to recover (see the workload description) than for real-time threads.

The table also shows the overhead of object tracking in stubs for the system scheduler, system memory manager and the RAM-based file system. Although the overhead from interface tracking of objects is inevitable, it only adds relatively small overhead in the studied workloads. The costs of the workloads, and the additional overhead for interface-driven tracking are labeled as "nominal workload overhead" in Table I, and should be interpreted as "overhead without tracking + overhead with tracking (standard deviation of overhead)".

B. Checkpointing System Evaluation

We evaluate checkpointing/restore (C/R) overheads in *a*) COMPOSITE, *b*) checkpoint/restore in user space (CRIU) in Linux [42], and *c*) Xen virtual machine(VM). Evaluations are run on an Intel i7-3700S running at 2.8Ghz for COMPOSITE and CRIU. `memcpy` overhead is measured on the same machine as a reference. This machine is configured with less than 900MB of RAM, so we only plot `memcpy` costs up to 256MB of RAM. The Xen VM checkpoint/restore is performed on an Intel Xeon E5-2420 at 1.90GHz. We could not configure Xen to run on the i7 machine, nor could we get CRIU to work on the Xeon system due to driver conflicts. Though neither CRIU nor Xen are designed for real-time checkpointing, we don't know of any systems that are (aside from checkpointing in C^3). C/R overheads of these systems should be interpreted only as context for what overheads to practically expect. Our results investigate various C/R costs.

Workload. The following workload is used to obtain the overhead of C/R operations:

- COMPOSITE*:* For a paravirtualized hard real-time OS, FREERTOS, of different static heap sizes (512k, 1M and 2M), COMPOSITE periodically checkpoints the FREERTOS component, and restores it on fault.
- *CRIU:* In Linux 3.6 configured with CRIU (ver 0.2), a FREERTOS process is periodically checkpointed and restored onto a 200MB ramdisk. FREERTOS is configured with heap sizes of 512KB, 1MB and 2MB.
- *Xen VM:* Xen version 4.1.2 with Dom0 (Linux 3.5) periodically checkpoints and restores a VM (with 512MB, and 1GB of memory).

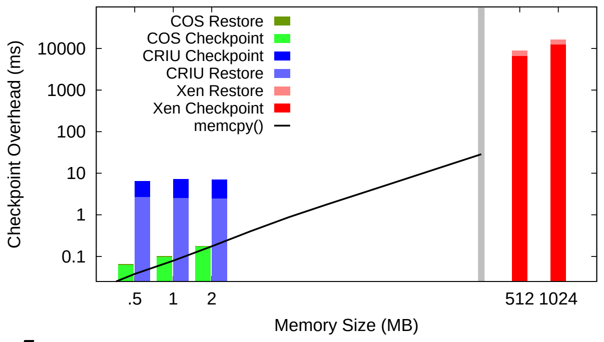


Fig. 7: Checkpoint Operation Overhead. Measurements on the two sides of the vertical bar are from different hardware.

Figure 7 compares the overheads of the different C/R systems. Standard deviation bars are small enough that they aren't visible, and each value represents the average of 200 measurements. The difference between checkpoint and restore is small because most overhead is due to memory operations. COMPOSITE has the least overhead (around 0.1ms), as its dominant cost (99%) is `memcpy` for a heap-size of 2MB. CRIU has around one magnitude higher overhead (1∼10ms) than COMPOSITE while the overhead of C/R in Xen (5∼15s) is at least several magnitude higher than CRIU and COMPOSITE. Note that for systems larger than FREERTOS, `memcpy` costs increase largely linearly, and it bounds the speed of C/R operations.

C. Fault-aware Schedulability Evaluation

The experiments in this section compare three system level fault tolerance techniques, *on-demand*, *eager* and *checkpoint*. To evaluate how the system schedulability is affected by the fault tolerance mechanisms, we use fault-aware schedulability success ratio, FASSR as defined in Section V-A. Intuitively, the FASSR shows the fraction of task sets that are schedulable given a fault-oblivious analysis (Equation 1) that are *still schedulable* given the fault-aware analysis introduced in Section V-A. Low FASSR values correspond directly to a fault-tolerance infrastructure inhibiting system schedulability. From the base system (no fault), we generate 50 *schedulable* task sets. In each set there are 20 tasks, 50 tasks and 100 tasks. We generate each task τ_i, by selecting p_i with average of 100ms from an exponential distribution. A utilization is selected from an exponential distribution with average $u_i = u_\Gamma/n$, and $e_i = p_i u_i$.

Default parameters. Unless otherwise specified, systems are generated with $e_{\mu r} = 0.02\ ms$ and $r_i(1) = 0.005\ ms$ (both from μbenchmarks), $m = 10$ (number of objects to be recovered per task for a component), $p_{cp} = 200ms$ (2x average period), $p_{ft} = 200ms$ (from [48]), $e_{cp} \in [0.1, 10]\ ms$ (from reasonable μbenchmarks), and $u_\Gamma = 0.7$.

Figure 6 shows FASSR as the function of utilization for different sized task sets. Changes in task set size here affect results as they change the ratio of task execution time to recovery costs (the latter of which stay constant). FASSR decreases for higher utilization. *On-demand* maintains high FASSR, and declines only toward 85% while *eager* performs poorly for >20 tasks. Checkpointing with 0.1ms checkpoint costs is very competitive across all dimensions, but checkpoints with more costly operations perform significantly poorer.

Discussion. *On-demand* has better schedulability behavior than *eager* due to its avoidance of *recovery inversion*. Note that in our workload, we generate no best-effort tasks. If we do, *eager* recovery can be unbounded as the number of objects cannot be bounded. Additionally, checkpointing with low checkpoint costs avoids most interference and is efficient. Thus checkpointing might be appropriate for very small systems. However, for systems with significant amounts of memory (8MB or more), checkpoint-driven recovery does not provide favorable schedulability. We believe these results show that *on-demand* C^3-based recovery is viable and effective at predictable, system-level fault recovery for embedded systems.

Figure 8 illustrates the relation between FASSR and the number of recovered objects (m_i) for *on-demand* and *eager*.

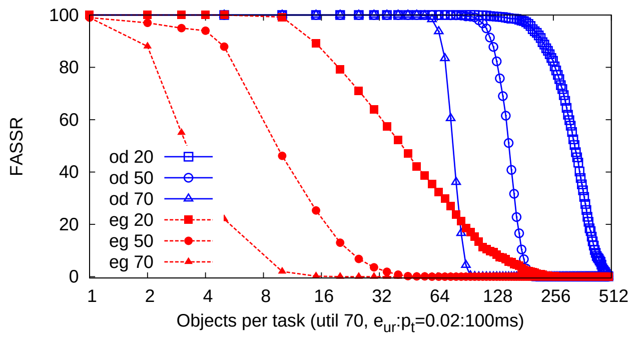


Fig. 8: FASSR vs Objects for eager, on-demand, and different task set sizes.

Eager degrades quickly with increasing m_i, especially for large task sets due to the interference between tasks. *On-demand* maintains a high FASSR, and quickly drops at different m_i thresholds.

Discussion. Recovery inversion causes significant and quick drops in FASSR for *eager*, while *on-demand* maintains significantly better schedulability behavior, scaling to a large m_i. As it is rare that a real-time process will require 100s of threads or files, we believe the system is able to provide predictable recovery for many practically-sized systems.

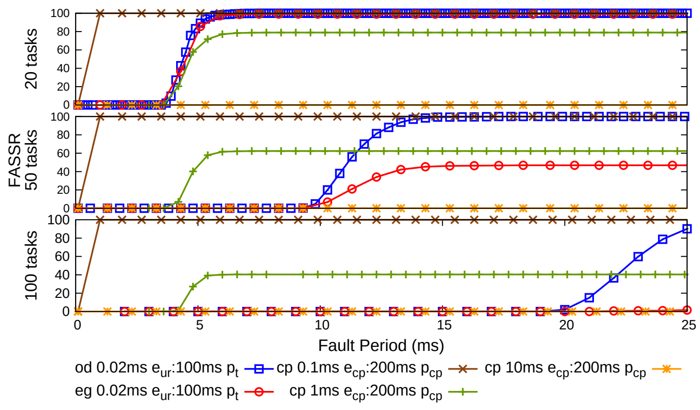


Fig. 9: Effect of fault periodicity on FASSR.

Figure 9 shows the effect of fault period on the FASSR for a task set using the different fault tolerance mechanisms. The x-axis plots an increasing fault periodicity. The system is often completely unschedulable for all task sets until some fault period is reached, after which a steady FASSR is maintained regardless of how large the fault period is. The parameters for the fault tolerance methods use the default values. Figure 9 plots on-demand, eager, and checkpointing with checkpoint costs increasing from 0.1 to 10ms.

Discussion. As expected, *on-demand* achieves higher FASSR than *eager* due to less recovery inversion, and checkpointing effectiveness degrades quickly for expensive checkpoint costs. Interestingly, since this function forms a step with a maximum less than or equal to 100%, certain recovery techniques will simply be unacceptable for some task sets even for very large fault periods.

VII. CONCLUSION

This paper presents the C^3 system, which is the first system we know of for predictable, system-level fault tolerance that doesn't require replication (*e.g.* physical or process-based). We also provide an implementation of component checkpointing to use as a significant comparison point. In an evaluation of both systems, we find that recovery can occur, given proper system design, within 10s to 100s of μ-seconds. We identify *recovery inversion* as a significant impediment to scheduling system fault tolerance, and present *on-demand* recovery as a solution. The overheads of recovery are integrated into a schedulability test for both C^3 and checkpointing, and evaluated. The results show that on-demand recovery in C^3 is effective at maintaining system schedulability even in the presence of system-level faults, and also that checkpointing can be useful in specific, low-memory situations.

Please find the source for C^3, and schedulability simulations at http://composite.seas.gwu.edu/.

Acknowledgments. We'd like to thank the anonymous reviewers for their helpful feedback, and our shepherd for helping to significantly improve the clarity of this paper.

References

[1] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[2] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, 2005.

[3] J.-C. Laprie and B. Randell, "Basic concepts and taxonomy of dependable and secure computing," *TDSC*, 2004.

[4] P. Mejía-Alvarez, H. Aydin, D. Mosse, and R. Melhem, "Scheduling optional computations in fault-tolerant real-time systems," in *RTCSA*, 2000.

[5] P. Mejía-Alvarez and D. Mossé, "A responsiveness approach for scheduling fault recovery in real-time systems," in *RTAS*, 1999.

[6] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright, "Probabilistic scheduling guarantees for fault-tolerant real-time systems," in *DCCA*, 1999.

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

[8] S. Punnekkat and A. Burns, "Analysis of checkpointing for schedulability of real-time systems," in *RTCSA Workshop*, 1997.

[9] M.-L. Li, P. Ramachandran, S. K. Sahoo, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *ASPLOS*, 2008.

[10] G. Parmer, "Composite: A component-based operating system for predictable and dependable computing," Ph.D. dissertation, Boston University, 2009, adviser-Richard West.

[11] A. Burns, R. Davis, and S. Punnekkat, "Feasibility analysis of fault-tolerant real-time task sets," in *ECRTS Workshop*, 1996.

[12] 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*, 2003.

[13] G. Lima and A. Burns, "An effective schedulability analysis for fault-tolerant hard real-time systems," *ECRTS*, 2001.

[14] H. Aydin, "Exact fault-sensitive feasibility analysis of real-time tasks," *IEEE Transactions on Computers*, 2007.

[15] M. Pandya and M. Malek, "Minimum achievable utilization for fault-tolerant processing of periodic tasks," *IEEE Transactions on Computers*, 1998.

[16] F. Many and D. Doose, "Scheduling analysis under fault bursts," in *RTAS*, 2011.

[17] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive failover for real-time middleware with passive replication," in *RTAS*, 2009.

[18] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, 1990.

[19] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim, "Safer: System-level architecture for failure evasion in real-time applications," in *RTSS*, 2012.

[20] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *TDSC*, 2009.

[21] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *EMSOFT*, 2012.

[22] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems," in *USENIX ATC*, 2012.

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

[24] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP*, 2003.

[25] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: Improving reliability through operating system structure," in *OSDI*, 2008.

[26] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Reorganizing unix for reliability," in *ACSAC*, 2006.

[27] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot–a technique for cheap recovery," in *OSDI*, 2004.

[28] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," in *ISSRE*, 1996.

[29] S.-W. Kwak, B.-J. Choi, and B.-K. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults," *Reliability, IEEE Transactions on*, 2001.

[30] H. Lee, H. Shin, and S.-L. Min, "Worst case timing requirement of real-time tasks with time redundancy," in *RTCSA*, 1999.

[31] J. Song and G. Parmer, "Toward predictable, efficient, system-level tolerance of transient faults," in *APRES workshop*, 2013.

[32] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS*, 2008.

[33] ——, "HiRes: A system for predictable hierarchical resource management," in *RTAS*, 2011.

[34] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *OSPERT*, 2010.

[35] B. Ford and J. Lepreau, "Evolving mach 3.0 to a migrating thread model," in *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, 1994.

[36] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: synthesizing dynamic bug detectors for foreign language interfaces," in *PLDI*, 2010.

[37] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi, "Language support for fast and reliable message-based communication in Singularity OS," in *EuroSys*, 2006.

[38] G. Parmer and R. West, "Mutable protection domains: Adapting system fault isolation for reliability and efficiency," in *TSE*, 2012.

[39] G. Candea and A. Fox, "Recursive restartability: Turning the reboot sledgehammer into a scalpel," in *HotOS*, 2001.

[40] U. Schiffel, A. Schmitt, M. SuBkraut, and C. Fetzer, "Software-implemented hardware error detection: Costs and gains," in *DEPEND*, 2010.

[41] M. Engel and B. Döbel, "The reliable computing base a paradigm for software-based reliability," in *SOBRES*, 2012.

[42] "CRIU: Checkpoint-Restore in User-space: http://www.criu.org, retrieved 5/1/13."

[43] "FreeRTOS: http://www.freertos.org, retrieved 5/1/13."

[44] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI*, 2005.

[45] V. Whitepaper, "Timekeeping in vmware virtual machines, retrieved 7/2013." [Online]. Available: http://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf

[46] P. Chevochot, I. Puaut, and P. Solidor, "Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from cots components," in *DSN*, 2000.

[47] S. Chandra and P. M. Chen, "How fail-stop are faulty programs?" in *FTCS*, 1998.

[48] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Soft error and energy consumption interactions: a data cache perspective," in *ISLPED*, 2004.

[49] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.

[50] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *VLSI Test Symposium*, 1999.

[51] A. W. A. Dixit, R. Heald, "Trends from ten years of soft error experimentation," in *SELSE*, 2009.

[52] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery methods," in *DSN*, 2006.

[53] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *DSN*, 2004.

[54] M. Rebaudengo, M. Reorda, and M. Violante, "An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor," in *DATE*, 2003.

[55] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of error rates in combinational and sequential logic," in *Nuclear Science*, 1997.