SuperGlue: IDL-Based, System-Level Fault Tolerance for Embedded Systems

Jiguo Song*, Gedare Bloom†, Gabriel Parmer*
*The George Washington University
{jiguos, gparmer}@gwu.edu
†Howard University
gedare.bloom@howard.edu

***Abstract*—As the processor feature sizes shrink, mitigating faults in low level system services has become a critical aspect of dependable system design. In this paper we introduce SuperGlue, an interface description language (IDL) and compiler for recovery from transient faults in a component-based operating system. SuperGlue generates code for interface-driven recovery that uses commodity hardware isolation, micro-rebooting, and interface-directed fault recovery to provide predictable and efficient recovery from faults that impact low-level system services.**

SuperGlue decreases the amount of recovery code system designers need to implement by an order of magnitude, and replaces it with declarative specifications. We evaluate SuperGlue with a fault injection campaign in low-level system components (*e.g.,* memory mapping manager and scheduler). Additionally, we evaluate the performance of SuperGlue in a web-server application. Results show that SuperGlue improves system reliability with only a small performance degradation of 11.84%**.**

I. Introduction

Recent advances in silicon technology enable processors with billions of on-chip transistors, however these advances increasingly cause processors to be susceptible to transient faults such as single event upsets (SEUs) or other soft errors. The risk of a soft error induced system failure has become more prominent and of great concern in systems that require high dependability such as safety-critical embedded systems. Fault tolerance and recovery from transient faults in such embedded systems historically uses triple modular redundancy (TMR), but the cost of TMR in terms of size, weight, and power (SWaP) is large due to software and hardware replication. The challenge for an embedded system is to minimize SWaP while maximizing fault tolerance.

Efficient fault tolerance approaches aim to protect and recover specific modules of a system, such as the device drivers [1] or application modules [2], [3], [4]. These approaches are problematic as a fault in an unprotected module can bring down the system. Consider a fault that crashes a thread scheduler, which would invalidate fault tolerance mechanisms operating at an application or user level. Nearly 65% of hardware errors corrupt operating system (OS) state [5] before detection. Once OS state is corrupted, the fault can propagate to any part of physical memory and affect user-level software. OS architecture plays an important role in system reliability. Microkernels, for example, improve system reliability by decomposing system services into partitioned modules, or components, with well-defined, isolated boundaries between them. These components do not interfere with each other in unpredictable ways, and this isolation naturally enhances fault tolerance by preventing blanket access to all of memory and limiting fault propagation: faults can propagate between two components only through the data shared over the interface between them [6]. Isolation helps to limit the impact of faults by constraining the effects of system-level faults, yet it is not sufficient for fault tolerance. For example, a failed scheduler cannot simply be rebooted with the expectation the resultant system behaves correctly.

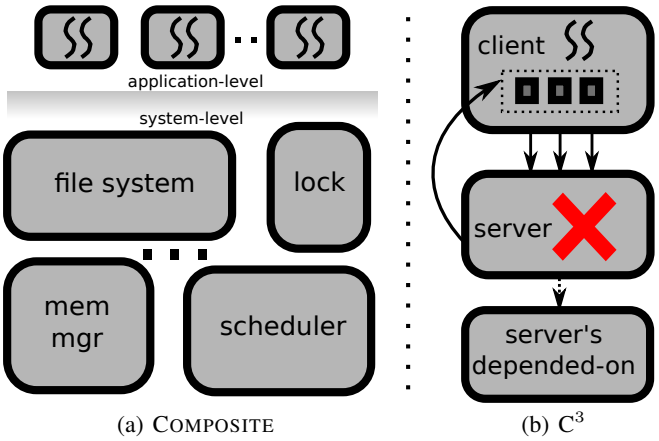
Prior work on the Computational Crash-Cart, or C^3 [7], leverages fine-grained isolation between system components to enable *interface-driven recovery* of a failed system component by leveraging logic within the interface stub code between communicating components. Stub code encodes and translates state about the components and uses component interface functions to rebuild and recover the state of a failed system component such as a scheduler. Interface-driven recovery avoids the overheads of check-pointing and replication, thus lowering SWaP. Of note for real-time embedded systems, the C^3 interface-driven recovery is *predictable* and has a demonstrated schedulability analysis for hard real-time systems. Unfortunately, C^3 uses recovery mechanisms in an ad-hoc manner, and the hand-written stub code that interposes on invocations between components is complex and error-prone.

In this paper we introduce SuperGlue, which is a model for interface-driven recovery and a declarative specification of component behavior implemented as an interface definition language (IDL). SuperGlue aims to provide predictable recovery from the failure of low-level system services without extensive changes to those services' code. Instead, SuperGlue utilizes an IDL specification of each service's interface and then tracks how a service uses resource descriptors and resources with a state machine. The net result for SuperGlue is a system with predictable, efficient recovery from system faults using an order of magnitude fewer lines of recovery code, that is written in a declarative manner, compared to a system using error-prone, hand-written recovery code.

The contributions in this work include:

- a model for inter-component interactions and component semantics that differentiates between necessary recovery mechanisms;
- a compiler that synthesizes interface-driven recovery code

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675 and ONR Award No. N00014-14-1-0386. 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 or ONR.



(a) COMPOSITE (b) C^3

Fig. 1: (a) A set of hardware-isolated components in COMPOSITE (b) Interface-driven recovery with C^3 for a system-level components. Dotted rectangle is an interface stub, small bold black squares in the stub represent recovery metadata, and squiggly vertical lines are threads.

based on an IDL specification;

- and an evaluation of SuperGlue overhead and fault tolerance in a component-based operating system subjected to fault injection in system-level components.

Experimental results show that SuperGlue can produce a dependable OS for embedded systems with minor performance degradation compared to a similar system without fault tolerance. We demonstrate our approach using COMPOSITE [8], an existing component-based OS that supports predictable real-time embedded systems, and C^3, which provides fault recovery mechanisms for COMPOSITE.

II. BACKGROUND: COMPOSITE AND C^3

SuperGlue uses C^3 mechanisms implemented in the COMPOSITE μ-kernel. This section briefly introduces these systems after explaining our fault model.

A. Fault Model

We assume transient faults that affect processor pipelines and registers with a *fail-stop* model. Thus, faults are detected immediately after corrupting system state. Past work in fault characterization finds that 65% [1], 80.6% [9], and 93% [10] of injected faults with detectable failures result in fail-stop behavior. Our own results confirm these numbers. Given the pressure on decreasing chip processes sizes, we focus on transient faults impacting the chip's pipeline and on-chip structures [11], rather than persistent memory corruption where ECC can prevent most faults [12].

B. COMPOSITE *Component-based OS*

COMPOSITE is an open-source OS that consists of a small kernel [13] (less than 7000 lines of code) and a collection of user-level components that implement the bulk of system-level services such as scheduling, memory management, time management, and synchronization. Components have well-defined interfaces consisting of sets of function calls provided by the *server* component to a *client* component that invokes them. Invoking a function in a component's interface triggers a *component invocation*–COMPOSITE's inter-process communication primitive–that is mediated by capability-based access control in the kernel [13]. COMPOSITE focuses on the fine-grained decomposition of system functionality into separate, isolated components. For example, a componentized web-server consists of over 20 separate components [8].

Unlike most modern μ-kernels, COMPOSITE uses synchronous thread migration [14], [15] instead of synchronous rendezvous between threads as in L4 variants [16]. Components are *active* when a thread invokes them or if they request the creation of a thread. They are also concurrent in that multiple threads can be runnable within a component at a time. Figure 1(a) depicts a set of application- and system-level components, shown in black bold rectangles, with threads (squiggly lines) executing in some components. The difference between system-level and application-level components is that system components manage (hardware) resources, often provided as a service to many clients. Hardware (page-table) protection mechanisms isolate each component's memory and prevent sharing data structures or passing addresses directly.

C. C^3 *Interface-Driven Recovery*

C^3 is a fault tolerance infrastructure built on top of COMPOSITE that makes use of the fine-grained isolation between components. Such isolation naturally provides a base-level of fault tolerance by constraining the scope of fault propagation: Barbosa et al. [6] reported a decrease in fault propagation for μC/OS-II from about 60%–70% down to only 22% of transient faults by making each process's address space private and hardware-protected. C^3 leverages the low incidence of fault propagation between components (no propagation was observed in experiments) to enable non-faulty components to aid in the recovery of a failed system-level component.

Interface-driven recovery in C^3 begins by μ-rebooting the failed component using a "booter" component to re-initialize the faulty component at the cost of `memcpy`. This reboot places the component into a *safe* (fault-free) state. Then the state of the component is made consistent with what its clients and servers expect by referring to a summary of component invocations prior to the crash. Each thread that was executing within a faulty component diverts back to the invoking client and executes stub code in the interface that recreates the server's state. Figure 1(b) depicts the process of interface-driven recovery. Descriptor state, in small bold black squares, is tracked on the client-side (dotted rectangle) of a component invocation. A fault in a component causes recovery of its clients by micro-rebooting the faulty component, resetting it to an initial state, and rebuilding server state using the descriptor tracking information stored in the client. When the failed server depends on another component, it must, in some cases, recover descriptors it had before failure. In such cases, a component can *reflect* on its depended-on servers to explicitly request the state of descriptors from each of those servers.

As an example of interface-driven recovery, consider a fault in a component that provides locks for mutual exclusion. When the component is μ-rebooted, multiple clients might have already created multiple locks. Furthermore, threads

might have acquired or be contending locks. The recovered component's internal state must be made consistent with the expectations of other components. Thus, the client's stub code in this example will regenerate the component's state by recreating, acquiring, or contending locks.

C^3 stub code executes at user-level in the client and server. This code tracks the state of each *object* acted on by the functions in the interface – *e.g.,* locks in the previous example, file descriptors for a file system (FS), or page virtual addresses for a memory mapping manager. With SuperGlue we call these objects *descriptors*. The straight-forward way to track the modifications made to the descriptors (*e.g.,* lock taken, lock contended, file offset advanced) maintains a log of operations. However, as C^3 targets embedded systems, unbounded memory consumption for the log is unacceptable. Instead, C^3 encodes the *state* of a descriptor with a state machine that contains a bounded amount of data. This data is derived from the arguments and return values of each interface function and is specific to the interface. For example, the path in the FS namespace is tracked for an open file descriptor, along with the offset of the descriptor into the file, which is updated based on the return values from read and write. Recreating a descriptor is equivalent to transiting a path through the state machine that brings the descriptor into the expected state (*e.g.,* an acquired lock) and sets the descriptor's data to consistent values (*e.g.,* `open` and `lseek`). A client restores a faulty server's state associated with a descriptor by making component invocations from the stub code, as above. A faulty client restores its descriptors by reflecting on the server.

Interface-driven recovery is insufficient for some components. For example, for an in memory FS (RamFS) to recover the data within a file, the state machine technique would need to be augmented with the data in the file. Instead such components include invocations within the RamFS component to a third *storage* component that stores an association including $< id, offset, length, *data >$ where *id* is an identifier to uniquely identify the file (*e.g.,* a hash on its path), and *offset*, *length*, and $*data$ track a slice of the file. When the RamFS is recovering and receives client invocations to recreate file state, it manually invokes the storage component to retrieve the file contents. In RamFS, the file information is shared using a zero-copy buffer mapping subsystem [17] in which all but the producing component (client) has read-only access to the buffer. This access restriction prevents fault propagation through the buffer. This buffer is what the storage component maintains for the RamFS server (as *data*).

C^3 recovery may be conducted either on-demand, or eagerly. *Eager* recovery iterates through *all* descriptors in a client interface to recover the entire state of a component for each of its clients immediately. In contrast, *on-demand* recovery delays recovery of a descriptor until it is accessed by a thread. Thus, each descriptor is recovered lazily. Importantly, this means that the descriptor is recovered *at the priority of the thread accessing the descriptor*, which has the effect of lessening the amount of interference due to priority inversion that recovery has on high-priority processes. On-demand has the effect of properly prioritizing the recovery process, which has a significant impact on system schedulability [7].

D. Example: Recovering the Memory Manager

The memory manager (MM) component in COMPOSITE provides and maintains virtual-to-physical memory mappings and provides an API close to that of the recursive address space model [16]. A thread invokes `mman_get_page` to request access to a page and creates the root mapping between that virtual page and some physical frame. Memory is shared between components with `mman_alias_page`, which creates a child mapping from a parent in a tree rooted with the physical frame. `mman_release_page` revokes a mapping and the subtree rooted at it (all transitive aliases). The MM descriptors are the virtual addresses in the component in which they are mapped.

The state of a mapping can be described by its virtual address and client component, and this state is tracked on the interface between the MM and the client that created the mapping. If a fault occurs in the MM, the mapping trees may be corrupted. μ-rebooting resets all trees. Eager recovery rebuilds all of the root mappings. On-demand recovery occurs when a thread makes a component invocation to the MM. A mapping can only be recovered after its aliased-from parent mapping is recovered. However, since memory can be shared between different components (two virtual pages may refer to the same physical frame), *upcalls* are made into client components in order to rebuild correct state between dependent mappings. This process is *transparent to client execution* and happens on-demand.

E. Assumptions and Scope of This Work

C^3, and comparably SuperGlue, do not protect several classes of system software. These include applications, the kernel itself, and the zero-copy shared buffer management component. Significant previous work has been done on embedded system application reliability, including that on checkpointing, recovery blocks, and redundant computation. SuperGlue does not target application-level faults. Instead, it focuses on system-level components (e.g., schedulers and file-systems) whose failure impacts the rest of the system that depends on them and is complementary to application-level fault tolerance techniques. The COMPOSITE kernel itself is small with little state (mainly just page tables, capability tables, and threads) and optimized to not consume much execution time. On our hardware (detailed in Section V), all kernel execution paths are bounded (for predictability), non-preemptible, and short (the longest taking around 1/2 μ-second). Unless occurring at an extremely high frequency, a pipeline fault is unlikely to impact the kernel.

If faults in the kernel and the shared buffer component prohibit system reliability, other techniques such as compiler-based redundant operation or memory encodings can be used. In such a case, C^3 and SuperGlue still provide protection to the rest of the system components, that do not need to suffer the overheads from such encodings.

F. C^3 recovery mechanisms: summary and limitations.

COMPOSITE provides hardware-based isolation between components, and system-level recovery with C^3. Although

C^3 uses interface-driven recovery, the procedures for recovering different kinds of components do vary. On one hand, Song et al. [7] gave an example for recovering the thread scheduler that requires reflecting on kernel data structures. On the other hand, recovering the MM component requires upcalls into client components, in addition to reflecting on the component-kernel interface. Indeed, C^3 offers no guidance to system developers in *how to apply* the interface-driven recovery mechanisms, nor is it clear when recovery should use reflection, upcalls, be done on-demand or eagerly, or if any ordering must be imposed on descriptor recovery. Importantly, C^3 stubs are manually written, and are complex and error prone. Some interface stubs are more than 398 (*e.g.,* the file system component stubs) lines of code (LOC), while the components that implement those interfaces are often around 500 LOC.

The need for SuperGlue. These limitations motivate SuperGlue which (1) creates a model that abstracts the system recovery mechanisms and properties of C^3 into an interface that (2) is integrated into an IDL supporting terse, declarative interface descriptions to automatically generate recovery code.

III. SuperGlue System Model

SuperGlue defines a model of interface and component semantics to better use the mechanisms of C^3. This model addresses the following questions about how to use C^3 (1) what is the shape of the state machine, and what is the recovery path through it? (2) what should client stub code do when it is activated by an inter-component exception? (3) when should upcalls be used to trigger stubs, and into which components? (4) when is reflection needed to recover descriptor state from a server? (5) and when must the storage component and zero-copy buffer be used, and what are the means to access storage component services? In this section we present a system model with the specifications for the resources and components that manage these resources, and the descriptors and the component interfaces that manipulate descriptors' states. We then build a set of rules for recovering a given faulty component based on the dependency relationships of that component. These specifications and rules enable interface-driven fault recovery for system-level services in a principled way that both addresses the questions outlined above and enables IDL compilation to produce interface-driven recovery code.

A. Descriptor-Resource Model

Operating systems provide high-level abstractions to hardware resources to their clients. These abstractions are often named using an opaque descriptor (e.g., a file descriptor). Whereas in C^3 the concepts of a resource and descriptor were conflated as "object", SuperGlue decouples and distinguishes between them. Interfaces are parameterized using the following terminology:

- r is a specific type of resource. Most system resources are abstract, and provided by a component's implementation of an interface. These include threads, memory mappings, locks, and event channels.
- $C = \{c^r, ...\}$ is the system's set of components. c^r is the component that manages resources of type r (for presentation, we simplify such that each component manages only a single type of resource).
- d_r is a class of descriptors used by client(s) which is associated with a resource r through c^r's interface.
- B_r is true if and only if a thread can *block* while accessing r in component c^r. Recall that COMPOSITE invocations are synchronous, so blocking also delays execution in the client when a thread blocks in the server.
- D_r is meta-data associated with the resource r.
- G_{d_r} is true if and only if a specific descriptor is *globally* addressable between components. Otherwise, each descriptor is *locally* addressable only from within each client component.
- $P_{d_r} \in \{Parent, XCParent, Solo\}$, describes if descriptors of type d_r can have dependencies on each other. $Parent$ expresses that when a dependency is created, the creation function takes another descriptor as an argument. Thus, upon fault and recover, the same *parent* must be passed in as an argument. The `accept` POSIX function is an example of this where new descriptors are created from existing ones. $XCParent$ states that the parent/child relationship can span components. $Solo$ denotes that no inter-descriptor dependencies exist.
- C_{d_r} is true if and only if $P_{d_r} \neq Solo$, and when a descriptor is *closed*, its entire tree of children is also deleted. This behavior is common in modern capability systems that support recursive revocation [18].
- D_{d_r} is meta-data necessary for recovery that is associated with descriptors of type d_r. For example, for files, this includes the *offset* and *file path*.
- Y_{d_r} is true if and only if $P_{d_r} \neq Solo \wedge \neg C_{d_r}$, and when a descriptor is closed, the stub's data tracking the descriptor is also deleted. Otherwise, the descriptor meta-data remains and can used by the children.

All of these variables are composed into the descriptor-resource model,

$$DR = (B_r, D_r, G_{d_r}, P_{d_r}, C_{d_r}, Y_{d_r}, D_{d_r}) \tag{1}$$

B. Descriptor State Machines

Integral in the recovery of server components is the state machine that is implicit in how descriptors are manipulated by interface functions. SuperGlue makes these state machines explicit. The motivation is two-fold. First, formalizing valid transitions enables fault detection if invalid branches are attempted. Second, the state machine is used to track each descriptor's state, thus succinctly summarizing the "current" state of a descriptor (along with D_{d_r}) without logging all interface operations. A state machine that describes the state of descriptor d_r includes:

$$SM_{d_r} = (I_{d_r}, S_{d_r}, \sigma, s_0, s_f) \tag{2}$$

where

- $I_{d_r} = \{f_i, ...\}$ is the set of functions in the interface exported by c^r.
- $S_{d_r} = \{s_i, ...\}$ is the set of states of the descriptor. As we will see, these are implicit and inferred by the compiler.
- $\sigma : S_{d_r} \times I_{d_r} \rightarrow S_{d_r}$ is the state transition function. Given an input state, and an interface function, σ determines the next state.
- $s_0 \in S_{d_r}$ is the initial state of a descriptor when created.
- s_f is a special type of state: the faulty state. There are implicit transitions to it from each other state, triggered by an failure in the server.

Functions in I_{d_r} trigger transitions between states. These functions are further characterized:

- $I_{d_r}^{create} \subseteq I_{d_r}$ is the set of functions that return a new descriptor in state s_0.
- $I_{d_r}^{terminate} \subseteq I_{d_r}$ is the set of functions that take a descriptor as an argument and signify its destruction.
- $I_{d_r}^{block} \subseteq I_{d_r}$ is the set of functions that can block the invoking thread.
- $I_{d_r}^{wakeup} \subseteq I_{d_r}$ is the set of functions that correspondingly wake up a thread.

It should be noted that $I_{d_r}^{block} \neq \varnothing \leftrightarrow B_{d_r}$. The blocking semantics of components should be part of the interface specification as a main factor in determining whether to use eager or on-demand recovery.

Lock component example. A descriptor to a lock, d_{lock}, is put into the initial state "available" when the *lock_alloc* $\in I_{d_{lock}}^{create}$ function is called which creates a descriptor tracking structure in the client. The descriptor can transit into "taken" state or under contention, into "block" state, by calling *lock_take* $\in I_{d_{lock}}^{block}$. Calling *lock_release* $\in I_{d_{lock}}^{wakeup}$ function transits the descriptor into "available" state again and allows other thread to take the lock. Calling *lock_free* $\in I_{d_{lock}}^{terminate}$ function will deallocate the tracking data structure when the function returns.

Basic component recovery (R0). When a fault occurs in the server component c^r and it is μ-rebooted, the descriptor d_r will first transit to the faulty state s_f and the client stubs will be notified of the fault via an exception. At this point, a precomputed, shortest path through the state machine is taken, invoking the corresponding functions $f_0, \ldots, f_n \in I_{d_r}$ such that $\sigma(\sigma(\sigma(s_f, f_0), ...), f_n) = s_{expected}$ where $s_{expected}$ is the state of the descriptor *before* the fault, and each function f_0 through f_n constitutes the $walk$ from the faulty state to the destination. This basic mechanism is shared between all model configurations.

The bottom diagram in Figure 2 shows how the descriptor state is manipulated from the fault state s_f to its "expected" state during the recovery for the scheduler, file system (RamFS) and event manager. For example, if a thread is in the "blocked" state when the scheduler fails, client stub code i) ensures that the scheduler recreates the thread in its own structures, and ii) then re-blocks the thread to match the client's expectations.

C. Recovery Mapping from Model to Mechanism

In addition to the base recovery **R0**, we classify the interface-driven recovery mechanisms into a number of categories. Each of these is depicted in the top of Figure 2.

Timing of recovery. Song et al. [7] provides the timing analysis for eager versus on-demand recovery. However, we must first determine when to use eager or on-demand recovery.

- **T0: Eager Recovery.** If B_r, then some eager recovery is required at a high-priority at fault-time. Eager recovery must be conducted within the faulted component as part of the initialization using specialized support similar to `__attribute__((constructor))` for execution before the equivalent of `main`. The function in I^{wakeup} provided by the recovering server's server is invoked to wake up each thread that was previous blocked by the faulty component.

- **T1: On-Demand Recovery.** Aside from the initial μ-reboot and re-initialization, all recovery can be conducted on demand if $\neg B_r$. Even if B_r is true and the eager recovery **T0** is required for resuming execution of all threads that were blocked previously by a faulty component, a descriptor with corrupted state in the faulty component can be recovered from the component's interface at the time when the descriptor is being accessed by a thread. Therefore all client stub-directed state machine recovery should be on-demand.

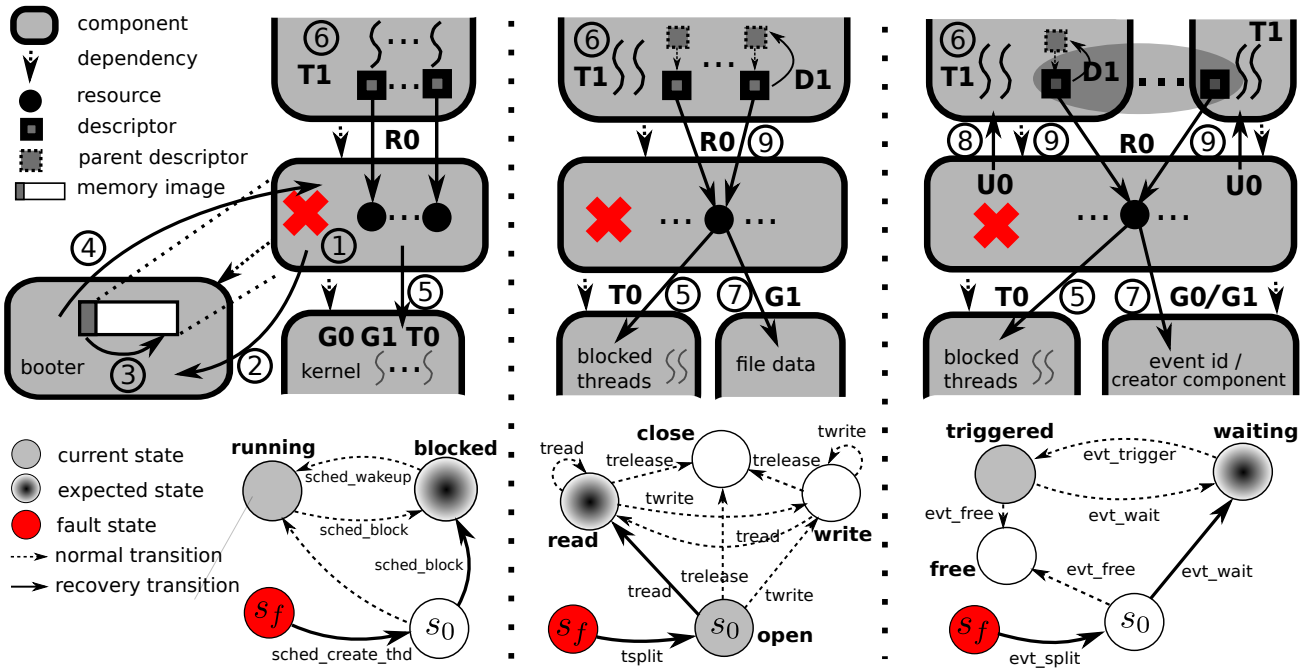
Recovery with Dependencies. Parent/child descriptor dependencies necessitate ordering descriptor recovery.

- **D0: Recovery with Children Dependency.** If C_{d_r} (and by implication $P_{d_r} \neq Solo$), then calling $f \in I_{d_r}^{terminate}$ to terminate a descriptor d_r requires the reconstruction of all its children descriptors. The semantics of recursive revocation [16] rely on children termination along with a parent as the child dependency often implies some side-effect that must also be terminated (*e.g.,* a virtual page mapping for the memory manager).

- **D1: Recovery with Parent Dependency.** If parent dependencies exist, $P_{d_r} = Parent \wedge P_{d_r} = XCParent$, then descriptors are processed from the root of the dependency tree to the descriptor being recovered. When $P_{d_r} = Parent$, this ordering is by simple `parent` links within the descriptor data-structure. For example, when the memory manager recovers an aliased page (descriptor), its parent mapping previously aliased from the root mapping must be reconstructed recursively along the path of mappings from the page to the physical frame (root).

Recovery with the Storage Component

- **G0: Recovery with Global Descriptor.** When the descriptors are globally addressable (G_{d_r} is true), multiple clients share the same descriptor namespace. In such cases, **R0** is not sufficient as a *single* client component does not have full context to recover the descriptor. Instead, a storage component keeps the mapping between each descriptor and their creator component (*i.e.,* executed the function $\in I_{d_r}^{create}$). When the descriptor is used and the recovered component does not find the descriptor id it returns an error (`EINVAL`). The server-side stub catches this error and queries the storage component that, if it holds a record of the descriptor, makes an upcall into the



(a) Scheduler (b) RamFS (c) Event

Fig. 2: Top diagram shows recovery mechanisms and the timeline between a fault and recovery via micro-reboot in the (a) Scheduler component. (b) RamFS component. (c) Event component. The oval shadowed area in (c) indicates that the same descriptor can be accessed from multiple client components (a.k.a, the global descriptor). Bottom diagram shows the transition from the fault state to a descriptor's "expected" state in each service during the recovery. The edge is the interface function and the node is the descriptor state. The dashed line is the normal state transition and the solid line is the recovery transition.

creating component to recreate the descriptor via **R0**. After recreation, the server stub replays the previous invocation using the recovered descriptor.

This mechanism is a case where SuperGlue not only orchestrates recovery through interfaces, but also uses additional logic in those interfaces that is cognizant of erroneous return values to interact with the storage component. In C^3, explicit code to interact with storage components was required. With SuperGlue, such code is not required.

• **G1: Recovery with Resource Data.** When there is resource data ($D_r \neq \varnothing$) it is redundantly stored within a storage component that needs to be introspected to restore the data for the resource. For example, a file's data (shared buffer references) can be stored in a storage component, and retrieved when the file is accessed and not found.

Though the same trick as for **G0** can be used to recreate the file using an erroneous return value-aware server stub, thus completely automating the interactions with the storage component, a race condition exists: when writing to a file it is non-atomically added to the data-structures in the RamFS and the storage component. Another thread could thus see the RamFS's file data, and the system could crash *before* the data is saved in the storage component. Though that thread saw the file data, upon recovery, it would be gone. Thus, we manually add storage component interactions within the critical region that modifies the RamFS data-structures. Future work may integrate locking into the stubs to thus make storage component interactions automated.

Recovery with Upcalls

• **U0: Recovery using Upcalls.** When a descriptor is global (G_{d_r}), recovery uses upcalls into client components to set up the initial state of descriptors as detailed in **G0**.

D. Server Recovery

Given the above taxonomy of interfaces, component model, and the mapping of the model to the underlying system mechanisms, the steps for SuperGlue-assisted recovery of a server component c^r follow. They are also depicted in the top of Figure 2 for the scheduler, file system (RamFS) and event manager components.

(1) A transient fault corrupts and crashes the component c^r.

(2) The hardware exception handler is vectored to the `booter` component.

(3) The `booter` micro-reboots [19] the faulty component to `memcpy` a good image and bring c^r into a safe state.

(4) An upcall is made into the newly rebooted c^r for component re-initialization.

(5) Blocked threads are woken up in the component c^r eagerly **(T0)** by calling the interface function $I_{d_r}^{wakeup}$ for each thread while inheriting the highest priority of those threads.

(6) When the system switches to a thread (according to its priority), and it attempts to execute in c^r, or utilize a de-

scriptor, the client stub is activated. Here **R0** conducts the state machine-directed recovery at the executing thread's priority using on-demand recovery **(T1)**. Any depended-on descriptors are recovered **(D1)** first. Dependent descriptors are recovered **(D0)** when the current descriptor is terminated through a function $\in I_{d_r}^{terminate}$.

(7) If the executing thread finds that the descriptor it is accessing is not yet available in the server, the thread queries the storage component to retrieve the resource data **(G1)**. Furthermore, if the server relies on global descriptors that are missing, they will also be recovered **(G0)** from storage components.

(8) An upcall is made into the client component that originally created the descriptor **(G0)** to rebuild the descriptor in the expected state **(U0)**.

(9) The recovering component c^r receives component invocations from the client **(R0)** on-demand at correct thread priorities calling $I_{d_r}^{create}$ to transit the descriptor into its initial state s_0 and then calling other interface functions to transit to the expected state.

IV. SuperGlue IDL and Compiler

In this section, we discuss SuperGlue IDL and compiler for producing interface-driven recovery code from the declarative interface specifications to bridge the gap between the high-level model of a component-based OS and the low-level interface recovery code.

A. SuperGlue IDL

The model-based syntax for the SuperGlue IDL is introduced in Table I. Syntax derived from the descriptor-resource model uses a Boolean expression for the corresponding specification. For the specifications based on the descriptor state machine, syntax is defined using an interface function for transitioning the descriptor state.

Syntax for the descriptor state tracking is applied directly to the function prototypes in the header file for each server component to enable SuperGlue to derive the tracking data structure. This syntax indicates how to track descriptors including what information to track, which descriptor is the parent descriptor, and how to look up the descriptor.

Figure 3 depicts a complete example of a SuperGlue IDL file that describes the event notification component's interface. A few important notes about the language: The state machines in the current language are implicit. Pairs of functions are used to describe the different possible routes execution can take. Though states could be made explicit, we leaned toward the side of simplicity in the current SuperGlue implementation. The descriptor and resource data are not specified in a single location. Instead arguments and return values from interface functions are annotated as being tracked either in the descriptor, or for the resource. In each case, the compiler internally constructs the states, and the tracking structures.

B. SuperGlue compiler

The SuperGlue IDL compiler is factored into a pipeline. To leverage existing, well-tested code-bases, the first stage uses the C preprocessor. First, a normal C header (`.h`) file

```
service_global_info = {
        desc_has_parent   = parent,
        desc_close_remove = true,
        desc_is_global    = true,
        desc_block        = true,
        desc_has_data     = true
};

sm_transition(evt_split,   evt_wait);
sm_transition(evt_wait,    evt_trigger);
sm_transition(evt_trigger, evt_wait);
sm_transition(evt_trigger, evt_free);
sm_transition(evt_split,   evt_free);
sm_creation(evt_split);
sm_terminal(evt_free);
sm_block(evt_wait);
sm_wakeup(evt_trigger);

desc_data_retval(long, evtid)
evt_split(desc_data(componentid_t compid),
          desc_data(parent_desc(long parent_evtid)),
          desc_data(int grp));
long evt_wait(componentid_t compid, desc(long evtid));
int evt_trigger(componentid_t compid, desc(long evtid));
int evt_free(componentid_t compid, desc(long evtid));
```

Fig. 3: Example SuperGlue interface specification for the event notification component.

```
/* predicate: true */
CSTUB_FN(IDL_fntype, IDL_fname) (IDL_parsdecl) {
  long fault = 0;
  int  ret   = 0;
redo:
  cli_if_desc_update_IDL_fname(IDL_params);

  ret = cli_if_invoke_IDL_fname(IDL_params);
  if (fault){
    CSTUB_FAULT_UPDATE();
    if (cli_if_desc_update_post_fault_IDL_fname()) goto
        redo;
  }
  ret = cli_if_track_IDL_fname(ret, IDL_params);
  return ret;
}
```

Fig. 4: Code generation template example for each invocation stub.

is generated by defining many of the SuperGlue language syntactic features as nil. Second, the preprocessor is used to tokenize each aspect of the SuperGlue file, adding attributes to variables and functions. A front end parser[1] parses the resulting file, then extracts the specifications from the abstract syntax tree into an intermediate representation that encodes the resource-descriptor and state machine models. With this representation, the shortest path through the state machine is found to each state. The back end is implemented as a network of templates associated with predicates. The templates implement the logic of the recovery mechanisms, and include calls to other templates. The predicates encode those aspects of the model that map to the recovery mechanisms as discussed in Section III-C. Templates are only included in the generated code if the predicate evaluates to true given the intermediate representation of the models. The SuperGlue compiler uses code templates to keep the back end easy to understand and maintain. The back-end is executed twice with two different sets of template inputs, once to generate the client stub, and one to generate the server. In total, the SuperGlue compiler includes 72 template-predicate pairs.

[1]We use `pycparser`, a C parser at https://github.com/eliben/pycparser.

	Model	Note	Syntax
descriptor-resource model	B_r	if the thread gets blocked when accessing c^r	`desc_block = true|false`
	D_r	if the resource has data	`resc_has_data = true|false`
	G_{d_r}	if the descriptor is global	`desc_is_global = true|false`
	P_{d_r}	when there are dependencies, if dependencies can span components	`desc_has_parent =` $Solo \mid Parent \mid XCParent$
	C_{d_r}	when close descriptor, if closes children	`desc_close_children = true|false`
	Y_{d_r}	when close descriptor, if removes dependency	`desc_close_remove = true|false`
	D_{d_r}	if the descriptor has data	`desc_has_data = true|false`
descriptor state machine	$I_{d_r}^{create}$	fn creates a new descriptor	`sm_creation(fn)`
	$I_{d_r}^{terminate}$	fn terminates the descriptor	`sm_terminal(fn)`
	$I_{d_r}^{block}$	thread can get blocked when call fn	`sm_block(fn)`
	$I_{d_r}^{wakeup}$	fn unblocks the thread	`sm_wakeup(fn)`
descriptor state tracking	track returned value	usually this tracks new descriptor	`desc_data_retval(type, value)fn`
	update descriptor state	track a parameter	`desc_data(type, value)`
	look up descriptor	look up the descriptor using its id	`desc(descriptor id)`
	track parent descriptor	track the descriptor's parent descriptor	`parent_desc(parent descriptor id)`

TABLE I: Syntax Definition in SuperGlue

```
/* predicate: f ∈ I_{d_r}^{create} ∧ ¬G_{d_r} */
static inline int cli_if_track_IDL_fname(int ret,
                                         IDL_parsdecl) {
  if (ret == -EINVAL) return ret;

  struct desc_track *desc = call_desc_alloc();
  if (!desc) return -ENOMEM;
  call_desc_track(desc, ret, IDL_params);

  return desc->IDL_id;
}
```

Fig. 5: Code generation template example for descriptor state tracking.

Two predicate-template pairs are depicted in Figures 4 and 5. Fig 4 shows an example of a code template for generating the stub invocation code. The code `cli_if_desc_update_IDL_fname(IDL_params)` performs the invocation to the server component. SuperGlue updates `IDL_fname` and `IDL_params` with the name of the interface function being called and its parameters. Before and after this function call, `cli_if_desc_update_IDL_fname (IDL_params)` corresponds to the template for checking descriptor state, and `ret = cli_if_track_IDL_fname (ret, IDL_params)` corresponds to the template for tracking descriptor state, respectively. Each template is only used if their predicates evaluate to true, thus the resulting generated code is the composition only of the relevant templates for a given SuperGlue specification.

V. Evaluation

We evaluate the fault tolerance properties provided by SuperGlue in a component-based embedded system, using a bit-flip based Software Implemented Fault Inject (SWIFI) approach. We first describe the fault model and SWIFI technique we used to inject faults into system components. We then proceed to describe the experiments performed and analyze the results.

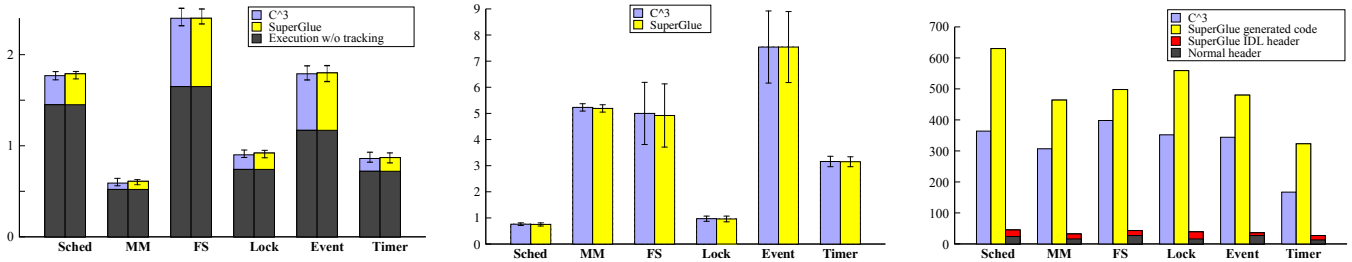
A. Fault Model and SWIFI

The ever-decreasing physical footprint of on-chip transistors increases the impact of transient faults in pipelines [11], and therefore leads to errors that corrupt OS state [5] [20]. This work focuses on tolerating transient faults (assuming a Single Event Upset (SEU) fault model) and assumes a *fail-stop* model. We ignore SEUs at the memory level, assuming solutions such as error-correcting codes (ECC) [12] implemented in hardware to be available, and instead we focus on SEUs in functional units of the CPU (registers) and their effects in system-level components. Nicolaidis [21] showed that SWFI-based single-bit flip in registers can accurately model transient faults in pipeline logic, which have error rates that are currently higher than memory [22], [23], [24], [25].

We used a runtime SWIFI technique that injects faults into registers to mimic transient faults by flipping bits within the chosen registers. Instructions are encoded as single-word (32bits) opcode and registers are also single-word sized in the platform, so the fault type can be defined by a 32-bit fault mask in which the bits to be affected are set to "1" and the bits that should be left untouched set to "0". During the evaluation, a fault mask of *0xFFFFFFFF* is chosen and the faults are injected by iterating through all threads and flipping register's bits only if they are executing within one of the target server components that provides system-level service: scheduler, memory manager, file system, lock, event manager, and timer manager. We mimic the fault distribution by randomly selecting a register from eight 32-bit registers (6 general purpose registers and 2 special registers ESP and EBP) periodically and flipping a random bit in the selected register. Notice that in practice the fault distribution is not uniform, but it is a first order approximation used by previous fault injection approaches [26] [27]. In [28], the calculation shows that at most one fault occurs over a window of 509.15 seconds with probability 99.999999% (assuming the distribution of the transient faults in any fixed time interval follows a Poisson distribution).

B. Benchmark Workloads

Using above-mentioned SWIFI technique, faults are injected into system components that provide system-level services: scheduler, memory manager, file system, lock, event manager, and timer manager. We first describe for each system



(a) Infrastructure Overhead (μs) (b) Descriptor Recovery Overhead (μs) (c) Recovery Code in LOC

Fig. 6: SuperGlue micro-benchmarks for each system component (compared with C^3): (a) infrastructure overhead with descriptor state tracking (μs). (b) per-descriptor recovery overhead (μs). (c) LOC for stub-based recovery code added to each system component.

component the workload used in fault injection campaign:

- *Scheduler* (**Sched**): Two threads perform a ping-pong, blocking and waking each other in turn using `sched_blk` and `sched_wakeup`.
- *Memory Manager* (**MM**): A thread is granted memory pages, and these pages are aliased into a different component, and then revoked, which removes all aliases.
- *RAM File System* (**FS**): A file is opened, a byte is written to it, read from it, and then it is closed.
- *Lock* (**Lock**): A thread holds a lock and another thread contends the same lock. After the owner thread releases, the other thread acquires the lock.
- *Event* (**Event**): A thread is blocked waiting for an event and the other thread triggers the event from a different component.
- *Timer Manager* (**Timer**): A thread wakes up, then blocks for a certain amount of time periodically.

C. Micro-Benchmark Results

The experiments reported in this section have been performed in COMPOSITE component-based OS on an Intel i7-2760QM running at 2.4 Ghz with only one core enabled. SuperGlue is evaluated with six system-level components: scheduler, memory manager, file system, lock, event manager, and timer manager. In Fig 6, we compare SuperGlue with C^3 [7] and present the micro-benchmarks results for each system component.

Fig 6(a) shows the average infrastructure overhead (in microseconds, with standard deviation) for tracking descriptor state in SuperGlue and in C^3, where the infrastructure overhead is defined as the base execution time of the micro-benchmark plus the average (stdev) time needed to track descriptor state. The result shows that SuperGlue has the similar amount of overhead as C^3. Fig 6(b) for each system component compares the per-descriptor recovery overhead in SuperGlue and C^3, which is the average (stdev) time to recover a descriptor to its "expected" state from the fault state. The recovery overhead correlates with the number of steps determined from Section III. For example, the cost of recovering an event descriptor is higher than the cost for a lock descriptor because the event server relies on all mentioned recovery mechanisms, except (**D0**) , whereas a lock descriptor only needs eager recovery (**T0**), base recovery (**R0**), and on-demand recovery (**T1**).

Fig 6(c) shows the lines of code (LOC) in SuperGlue IDL header file, which is written in SuperGlue IDL, for each system component and compares the LOC of recovery code added by SuperGlue and by C^3. The result shows with only a small amount of declarative code written in SuperGlue IDL, SuperGlue compiler is able to generate interface-driven fault recovery code for many system-level services in a component-based OS. For example, with only 32 LOC written in SuperGlue IDL, the compiler generates 464 LOC that recovers the memory manager from faults. These results show that SuperGlue can be both efficient and effective in recovering low-level system services. SuperGlue seeks to make the process of constructing dependable embedded system "correct-by-construction,", instead of "construct-by-correction" which is time-consuming and error-prone.

D. Fault Injection Campaign Results

We evaluate the effectiveness of SuperGlue through a fault injection campaign. Depending on the effects of a fault, and if they are detected, we define injected faults in the target system component as follows:

- F_a is the set of injected faults that cause the target component to deviate from its expected behavior and are detected (e.g., as the fault generates a hardware exception, triggers an assertion, causes a system hang or even crashes the system). This type of fault is defined as an *activated fault* in [6].
- F_r is the set of *activated faults* in the target component that are recovered by SuperGlue successfully. A successful recovery is defined by the continued execution that abides by the target component and workload specifications post-recovery.
- $\frac{|F_r|}{|F_a|}$ is fault recovery *success rate*, which denotes how many activated faults in the target component have been recovered successfully.
- F_u is the set of injected faults that are not detected (*undetected faults*).

System Component	Injected	Recovered Faults	Not recovered (segfault)	Not recovered (propagated)	Not recovered (other reason)	Undetected	Fault Activation Ratio	Recovery Success Rate
Sched	500	436	54	0	2	9	98.36%	88.58%
MM	500	431	35	1	4	30	94.26%	91.48%
FS	500	455	18	0	0	29	94.7%	96.14%
Lock	500	433	33	2	0	31	93.82%	92.35%
Event	500	450	16	2	0	33	93.83%	96%
Timer	500	460	26	0	0	18	97.23%	94.62%

TABLE II: SWIFI-based Fault Injection Campaign with SuperGlue

The total number of injected faults is given by $|F_a \cup F_u|$ and the fault activation ratio is given by $\frac{|F_a|}{|F_a \cup F_u|}$. During each campaign, a maximum number of faults (i.e., $|F_a \cup F_u| = 500$) were injected into each low level system component while the workload is running. After each injection, the executing thread resumes and the program is run to completion (unless the system crashes and we need reboot the machine). After each workload execution, the system is rebooted to clear any residual errors before the next run. We record the number of recovered faults F_r, the number of non recovered activated faults and report both the fault *activation ratio* and the fault recovery *success rate*. For each system component, we evaluate SuperGlue by executing that component's workload repeatedly while a SWIFI thread in a separate component is responsible for injecting faults into the target component periodically, for example randomly flipping bits in chosen registers every 1 second. Note that register bit-flips do not always lead to errors (e.g., a flipped register can be overwritten before it is read) and those are undetected faults. The system, without being rebooted, continues execution with the next fault being injected when one of the following conditions is observed: if the injected component is recovered successfully from the activated fault, or the injected fault is not activated at all (*undetected fault*). Otherwise, the system needs to be rebooted and resumes the fault injection campaign until the maximum number of faults have been reached.

Table II shows the result of fault injection campaign for each system component with SuperGlue and it can be seen that most of activated faults in the server component can be effectively recovered. For example 96.14% of activated faults in **FS** component have been successfully recovered. This work focuses on recovering faults rather than detecting faults, however, we also report our observations on the effect of injected faults in Table II. Very few activated faults propagate to non system-level client components and cause an unrecoverable fault, due to hardware-based isolation between components. For example, Table II shows that only 1 out of 470 detected faults (about 0.2%) became unrecoverable due to propagation when the faults are injected into the **MM** component. Although this situation can be improved with well specified interfaces with pervasive error checking and validation of inputs (as in [29], [30]), SuperGlue focuses on the recovery of system-level components in this work.

Among all unrecoverable faults, we observed that some activated faults lead to segfault crashes (i.e., the system exits with segmentation fault). For example, **Sched** component has the most segfault crashes (10% of injected faults lead to segfault crash), and for **Event** component this is around 3%. This is the main impact on the fault recovery *success rate*; further investigation might be necessary. There are also some activated faults that cause the system to hang, rather than crash, and we label these as "Not recovered (other reason)" in Table II. Injected faults might cause infinite loops in the target component. This type of fault is defined as *latent fault* and has been discussed in C'MON [28]. The result of fault activation ratio, which is defined as the percent of activated faults in all injected faults, shows that our SWIFI can effectively inject and activate faults in the target component. For example, 98.36% of injected faults were activated in **Sched** component when using SWIFI.

E. A Web Server Benchmark Workload

To thoroughly evaluate the degradation and overheads, that is, the holistic cost of recovery, we use an application web server that is system and I/O intensive in which average case performance is a priority. This web server, which makes use of all system-level components, enables us to determine the impact of the recovery infrastructure on best-effort tasks, whereas prior work [7] has demonstrated the real-time properties of interface-driven recovery. We evaluate a custom web server implemented in COMPOSITE, COMPOSITE with SuperGlue and COMPOSITE with C^3 under normal condition and under the presence of injected faults. We also compare performance with the Apache HTTP server of version 2.2.14, which is running on Linux 3.2.6 on a machine with Intel i7-2760QM at 2.4 Ghz. Performance is measured by how many requests per second are handled by each web server with *ab*, the Apache HTTP server benchmarking tool of version 2.3. During each test, *ab* sends 50000 requests with a maximum of 10 requests concurrently to the server for benchmarking.

Figure 7 shows the performance in throughput (HTTP requests per second) of all four variations when running for one minute, which is repeated 20 times with measurements taken for each, and we report the average and standard deviation. Apache achieves around 17600 requests per second, and the COMPOSITE base web server achieves about 16200 requests per second. COMPOSITE with SuperGlue achieves average 14281 requests per second (11.84% slowdown), while COMPOSITE with C^3 achieves average 14500 requests per second (10.5% slowdown).

We evaluated recovery using COMPOSITE with SuperGlue and COMPOSITE with C^3 by injecting faults into one system-level component every 10 seconds. We observe that recovery proceeds in parallel to continued web server operations, and after recovery the web server achieves similar throughput as

before the fault occurred. For example, when the fault occurs in the scheduler, the web server throughput is only disturbed temporarily for less than 2 seconds and continues serving clients, without dropping the network throughput down to zero. These results indicate that SuperGlue can improve system reliability with minor performance degradation.

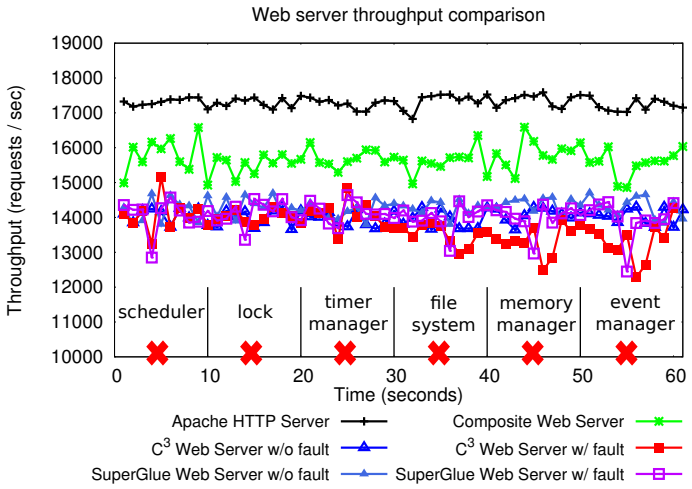


Fig. 7: Web Server Throughput. Requests/Second for Apache, COMPOSITE, COMPOSITE with C^3 without faults, COMPOSITE with C^3 COMPOSITE with SuperGlue without faults, and COMPOSITE with SuperGlue with one crash injected every 10 seconds, as indicated by red crosses, into a different system-level component.

VI. RELATED WORK

There has been considerable work on improving OS reliability. Two well-known approaches are TMR and checkpoint-restore, both of which incur heavy overhead. TMR achieves reliability through redundancy, which comes at the cost of more than tripling SWaP costs. Checkpoint-restore, for example in EROS [31], Otherworld [32] and Tardigrade [33], takes a snapshot of the OS services and rolls back to a previous state when a fault occurs. In addition to the storage and time spent creating checkpoints, checkpointing client (user-space) state incurs extra overhead for consistency. Another drawback of restoring a checkpoint, aside from the overhead, is that computation and communication since the last saved checkpoint can be lost and the current physical system state and the state expected by the control system from checkpointed information could mismatch [7]. Additionally, process level [34] replication can be made to improve system reliability by using a set of redundant processes per original application process and compares their output to ensure correct execution.

A. Dependability in OS

Nooks [1] improves Linux reliability by moving device drivers into a light-weight protection domain with limited access to the kernel's memory space. A shadow driver, designed with the same interface as the original driver, monitors the transfer of data during normal execution. Device drivers and their shadow drivers run in privileged mode. When a fault happens, the shadow driver becomes active instead of the original driver so it can be recovered with previously saved state.

Minix [35] is a microkernel OS, in which a special isolated component, the *reincarnation server*, can restart faulty components by recreating a fresh copy. Minix does not track client state however, and therefore the recovery mechanisms only work well for stateless servers such as device drivers. In other words, the *reincarnation server* is not applicable for OS services that use client state. Additionally, system-level services in Minix such as memory management are still implemented in the kernel space, which makes building a reliable OS tolerant of system-level faults challenging.

CuriOS [36] is a microkernel OS in which a Server State Region (SSR) stores each server's client-related information that is protected from both the server and client. All SSRs are managed by a single separate component (SSRManager) and requests that cause crashes are isolated by restricting write from clients. A SSR can be created and deleted when a client accesses a server and returns. When a fault occurs in a server, a recovery routine in the restarted server is invoked to enumerate all associated SSRs for recreating the internal state of the restarted server. However, SSRManager is a single point of failure and not desirable for a reliable OS when faults can occur in system-level services. In contrast, COMPOSITE with C^3 [7] as we have discussed in SectionII, focuses on the fault tolerance mechanism for system-level services and C'MON [28] allows predictable detection of latent faults in system-level services in component-based OS.

Instead, SuperGlue aims at building reliable OS while reducing programming effort. SuperGlue makes constructing reliable OS a much less error-prone process. It achieves this by mapping a high-level abstract system model (see SectionIII) to the low-level interface-driven recovery mechanisms, and improves the reliability for system-level services in a component-based OS with only a small performance degradation. Although the system model and fault recovery mechanisms could be further integrated with formal specification techniques [37] [38] to achieve greater system assurance, we have focused on using a model that enables a concise definition of system behavior to evaluate SuperGlue's effectiveness.

B. Interface Synthesis

To support software design, automatic code generation with interface description language (IDL) have been studied extensively in the past for different purposes. Flick [39] aims to build a highly flexible IDL compiler which can be used with various IDL types as well as generate code for different communication platforms. For example, Flick supports IDLs such as CORBA IDL at the frontend, and L4 at the backend. Flick achieves this modularity through a series of programmer-visible intermediate languages which can be operated on independently.

Jinn [30] is a dynamic analysis framework for Java Native Interface (JNI) and can be used to synthesize runtime checks to detect language interface violations. Jinn defines three categories of rules that can be expressed using eleven finite state machines. Based on these FSMs, Jinn enforces these

rules by dynamically injecting checks into user code with language interposition at the JNI interfaces for uncovering software bugs.

In contrast, SuperGlue IDL aiming at system reliability, allows declarative high-level description about a component-based OS based on a resource-descriptor relation model and a descriptor state machine, and enables SuperGlue compiler to generate the fault recovery code for system-level services and enhance component-based embedded system dependability.

VII. Conclusion

Faults in system-level services usually necessitate rebooting the system, thus disrupting all applications. In real-time and embedded systems, this means violating the temporal guarantees of applications, thus violating system correctness. This paper presents SuperGlue, an infrastructure built on top of the predictable recovery mechanisms of C^3 to improve the programmability of those mechanisms. We introduce a model of component and interface semantics that enables the IDL-based, declarative specification of the salient properties that the SuperGlue compiler uses to generate recovery code. The average SuperGlue IDL file replaces C header files and is 37 lines of code, an order of magnitude improvement over C^3 which required manually written, error-prone recovery code. We demonstrate that the SuperGlue infrastructure causes a non-prohibitive slowdown of 11.84% in a throughput-oriented application (a web-server). Even with injected faults, the slowdown is only 13.6%.

References

[1] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, 2003.

[2] Pedro Mejía-Alvarez and Daniel Mossé. A responsiveness approach for scheduling fault recovery in real-time systems. In *RTAS*, 1999.

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

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

[5] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *ASPLOS*, 2008.

[6] R. Barbosa, J. Karlsson, Qiu Yu, and Xiaozhen Mao. Toward dependability benchmarking of partitioning operating systems. In *DSN*, 2011.

[7] Jiguo Song, John Wittrock, and Gabriel Parmer. Predictable, efficient system-level fault tolerance in C^3. In *RTSS*, 2013.

[8] Gabriel Parmer and Richard West. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. In *ACM Transactions on Software Engineering (TSE)*, July/August 2012.

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

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

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

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

[13] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. Speck: A kernel for scalable predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[14] Gabriel Parmer. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *OSPERT*, 2010.

[15] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, 1994.

[16] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[17] Yuxin Ren, Gabriel Parmer, Gedare Bloom, and Teo Georgiev. Cbufs: Efficient, system-wide memory management and sharing. In *Proceedings of the 2016 International Symposium on Memory Management*, 2016.

[18] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *SOSP*, 2013.

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

[20] Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 2005.

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

[22] A. Wood A. Dixit, R. Heald. Trends from ten years of soft error experimentation. In *SELSE*, 2009.

[23] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery methods. In *DSN*, 2006.

[24] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN*, 2004.

[25] M. Rebaudengo, M.S. 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.

[26] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO*, 2005.

[27] N.J. Wang and S.J. Patel. Restore: Symptom based soft error detection in microprocessors. In *DSN*, 2005.

[28] Jiguo Song and Gabriel Parmer. C'Mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *RTSS*, 2013.

[29] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.

[30] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *PLDI*, 2010.

[31] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *SOSP*, 1999.

[32] Alex Depoutovitch and Michael Stumm. Otherworld: giving applications a chance to survive OS kernel crashes. In *Proceedings of the 5th European conference on Computer systems*, pages 181–194, New York, NY, USA, 2010. ACM.

[33] Jacob R. Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In NSDI, 2015.

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

[35] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Reorganizing unix for reliability. In *ACSAC*, 2006.

[36] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Curios: Improving reliability through operating system structure. In *OSDI'08*.

[37] M. Rodriguez, J. C. Fabre, and J. Arlat. Formal specification for building robust real-time microkernels. In *RTSS*, 2000.

[38] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of cots microkernel-based systems. *IEEE Transactions on Computers*, 2002.

[39] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: a flexible, optimizing idl compiler. In *PLDI*, 1997.