# C'MON: a Predictable Monitoring Infrastructure for System-Level Latent Fault Detection and Recovery

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

***Abstract*—Embedded and real-time systems must balance between many often conflicting goals including predictability, high utilization, efficiency, reliability, and SWaP (size, weight, and power). Reliability is particularly difficult to achieve without significantly impacting the other factors. Though reliability solutions exist for application-level, they are invalidated by system-level faults that are particularly difficult to detect and recover from.**

**This paper presents the C'MON system for predictably and efficiently monitoring system-level execution, and validating that it conforms with the high-level analytical models that underlie the timing guarantees of the system. Latent faults such as timing errors, incorrect scheduler decisions, unbounded priority inversions, or deadlocks are detected, the faulty component is identified, and using previous work in system recovery, the system is brought back to a stable state – all without missing deadlines.**

## 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. System faults can corrupt execution state, create erroneous computation that deviates from the intended behavior, and violate the timing models that underlie the schedulability analysis of the system.

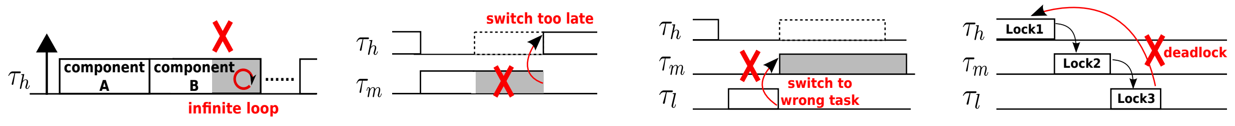
System faults can originate from software bugs or from environmental effects such as Single-Event Upsets (SEUs). Though rigorous software engineering processes (*e.g.,* code verification) are used to mitigate software bugs, the impact of SEUs is particularly difficult to avoid. SEUs are caused by high energy particle strikes such as neutrons from cosmic rays [1] that corrupt transistor state leading to bit-flips in chip structures [2]. As chip technologies trend toward smaller processes (*i.e.,* 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.

Most previous approaches for real-time fault tolerance on a single node focus on application-level fault tolerance [3], [4], [5]. However, ignoring faults in system-level components is dangerous: a fault in the scheduler would defeat all of these user-level techniques. Such system-level faults are frequent: nearly 65% of hardware errors corrupt OS state [6] before they're detected. Due to fault propagation, and the consequent corruption of system-global state, recovering from system-level faults in monolithic systems is particularly difficult. Fault propagation can corrupt any memory in the system as system components can access all of physical memory. For example, a fault in a scheduler manifesting as argument corruption to `memset` (*e.g.,* passing an accidentally large range argument) can easily overwrite all of the system's memory. Only after all data is corrupted and it is too late to recover, will the error be detected. For real-time systems, a pernicious dimension of fault propagation exists: a fault that causes changes in logical code can result in unexpected variability in run-time, which can cause *missed deadlines for all system tasks*. This motivates our previous work on the COMPUTATIONAL CRASH CART, $C^3$ [7], a combination of fine-grained isolation via hardware-provided protection domains, interface-driven recovery, and OS component micro-rebooting yielding *predictable* and efficient fault tolerance at *system level*.

A common assumption of past real-time research is that errors are detected immediately by the system without any fault propagation (*i.e.,* the *fail-stop model*), or that they don't propagate to a validation function that determines if a fault has occurred [5], [7], [8], [9]. This assumption underlies *temporal redundancy* in which a fault is detected at the end of a job's execution, enabling the re-execution of the job. However, other research [10], [11], [12] has shown that violations of the fail-stop model are significant and that faults propagate and corrupt memory. In this work, we will assume the following fault lifetime (similar to [13]): (1) an SEU causes a *fault* in hardware state, (2) some faults manifest as *errors* in which the fault negatively impacts software execution state and causes deviations from expected behavior, (3) if an error is *detected* (*e.g.,* as it generates a hardware exception, or triggers an `assertion`), then the temporal offset of detection from the fault is the *error detection latency*, and (4) a system *failure* occurs when the system cannot meet its objectives (*i.e.,* misses a deadline). Fault tolerance infrastructures attempt to avoid a failure *in spite of* faults. The fail-stop model assumes that error detection latency is negligible or zero. In contrast, *latent faults* have a possibly unbounded error detection latency. They are dangerous: before detection and after the error, execution is possibly compromised. This research provides a system infrastructure and timing analysis for the *predictable detection of latent faults* for hard real-time systems. To avoid system failure (*e.g.,* avoid missing deadlines) and recover from the fault, we rely on our previous work $C^3$ [7].

**C'MON: predictable fault tolerance for latent fault.**

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) component execution overrun (b) potentially unbounded priority inversion (c) improper scheduling (d) deadlock

Fig. 1: Detected latent faults. (a) A task gets stuck in an infinite loop in a component. (b) The scheduler delays switching to the higher priority task. (c) The scheduler switches to an incorrect task. (d) The circular dependency is formed due to the improper sequences of lock acquisitions and releases. Dotted-line rectangular area represents task's normal behavior when no fault occurs. $\tau_l$, $\tau_m$ and $\tau_h$ are tasks with lower, medium and higher priority respectively.

Though C$^3$ [7] provides the facilities to recover from detected failures predictably, it assumes fail-stop faults with immediate detection (*e.g.,* via `assert` or hardware exception). However, "*knowing that a fault has occurred is more important than the actual failure*" [14] – and the fail-stop assumption rules out the detection of, hence recovery from, significant classes of latent faults [10], [11], [12]. C'MON, or the C$^3$ MONITORING SYSTEM, is a system-wide monitoring infrastructure that tracks all communication between each component and the rest of the system (such as the *event* of component invocation, context switch or interrupt), and dynamically validates that execution behavior and timing conform to the models specified offline to analyze the system. When a deviation from the expected behavior is detected, C$^3$ as a complementary system-level recovery mechanism can be used to recover the system in a timely manner. C'MON relies on the COMPOSITE [15] component-based operating system that breaks system level software into relatively small, isolated, user-level components, one for each logical policy or mechanism in the system. This isolation restricts the propagation of faults due to (1) hardware-provided memory protection via page-tables, and (2) controlled communication between components. The goal then, is to determine if a component demonstrates erroneous behavior.

Though the C'MON infrastructure could detect broader behavioral faults based on communication arguments, such as unexpected patterns of communication between components, we scope this work to focus on *temporal overruns* and *behavioral faults* that affect the system timing as this is intrinsic to the correctness of real-time systems. Figure 1 shows examples of such faults. **The temporal overrun fault** occurs when a parameter of the system model such as the worst-case execution time of a thread, component, or critical section is violated at run-time. For example, *component execution overrun* can occur when a task gets stuck in an infinite loop in a component as shown in Figure 1(a). **The behavioral fault** occurs when the decisions made by specific components that control system timing demonstrate a faulty outcome. For example, priority inheritance policies might result in gratuitous priority inversion such that the higher priority task suffers *unbounded priority inversion* as shown in Figure 1(b); or the scheduler might choose to run a lower priority task when a higher priority task is runnable, which implies the *improper scheduling* as shown in Figure 1(c); or the presence of circular wait for system resources such as locks that results in *deadlock* as shown in Figure 1(d). The detection of faults using the high-level behavior is significant: instead of relying on programmer-inserted assertions, domain-specific knowledge to assess correct execution, or other logic-centric representations, C'MON instead relies on *observed execution behavior and compares it to the high-level model of the system*, thus making a direct correlation between the mathematical analysis that determines correctness, and the system's execution.

C'MON is designed with the following goals: (1) *Simplicity* – minimize the complexity and footprint of the monitoring infrastructure that must be trusted for proper fault detection (the C'MON is less than 1K Lines of Code (LoC), and the kernel is less than 7K LoC). (2) *Predictability* – execution overhead for both logging component operations with synchronization, and the processing of those buffers in the C'MON must be bounded. (3) *Efficiency* – especially the common-case logging of component operations must be efficient and have minimal impact on the average-case execution of system-level components that service both real-time and best-effort tasks.

**Contributions.** The contributions of this research include:

- *Design and implementation of system-level latent fault detection.* This paper presents the design of C'MON in the COMPOSITE system. This includes the design of a wait-free multi-producer, single consumer buffer for event monitoring, and a SYSTEM MONITOR COMPONENT (SMC) for predictable and efficient event processing and constraint validation. We additionally provide algorithms for practical optimizations: minimizing memory consumption, and finding a monitor periodicity that results in a schedulable system.
- *Schedulability analysis of* C'MON *for predictable reliability.* We present a system overhead aware schedulability analysis for a system with C'MON. This analysis includes time for the system to *detect and recover from faults without missing any deadlines.*
- *Evaluation of* C'MON. This paper evaluates the C'MON implementation not only in terms of execution time, but also in terms of latent fault detection effectiveness. System overheads are utilized in the schedulability analysis, and the overall schedulability of the system is assessed.

**Organization.** This paper is organized as follows: Section II defines the fault model, Section III presents the system model used for analysis, Section IV discusses C'MON design and implementation, Section V introduces the schedulability analysis, and Section VI presents our evaluation. Additionally Sections VII and VIII present the related works and conclusions, respectively.

## II. FAULT MODEL

SEU-induced transient faults can affect the control or data flow in a component, through the corruption of either registers or memory. This can lead to timing irregularities due to greater than expected numbers of iterations, and even infinite loops as shown in Figure 1(a), or incorrect scheduling decisions as shown in Figure 1(c). Research [16] indicates that about 70% of transient faults manifest as control flow errors and the rest as data flow errors. Although pervasive software-based data and control flow checking can be achieved by replicating instructions or by using signature comparison [17][18], these techniques usually require source code modification and introduce noticeable memory or performance overheads that heavily depend on applications. In contrast, C'MON leverages the interfaces between components and focuses on detecting, and therefore recovering from temporal and behavioral faults in system-level services efficiently and predictably without modifying the component source code.

**Fault propagation.** Where the fault can occur within a component is unpredictable, however, with proper spatial and temporal partitioning [19], fault propagation can be greatly limited or mitigated. Barbosa et al. in [20] reported that in $\mu$C/OS-II only $22\%$ of transient faults can propagate if the private address space of each process is protected by memory management hardware while $60\% - 70\%$ can propagate to other process without such partitioning. The results from [21][22] showed that only $17\% - 21\%$ of transient faults can propagate from the kernel level to the application level, a boundary without an isolation barrier. As COMPOSITE provides pervasive protection boundaries to constrain the scope of error propagation, and stub-based validation of function arguments passed between components, we make the following assumption underlying the use of isolation boundaries in all fault tolerant systems:

**Assumption 1.** *Faults in a component can corrupt all state within the component, which can impact all that harness that component's functionality, but these faults do not propagate outside of the component.*

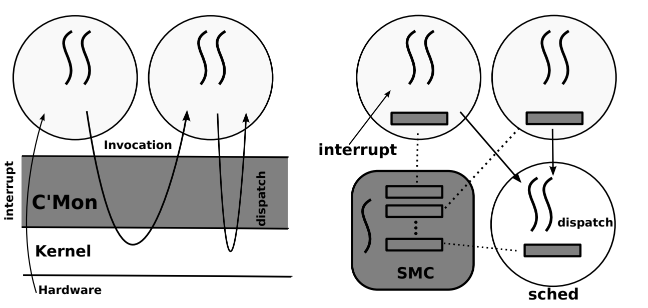
**Fault frequency.** Different fault frequency models have been proposed in the past, including bounded fault models (e.g., two faults must be separated by a minimum distance [5][8][9], or each fault has a bounded interval during which a burst of unknown number of errors can occur [23], or at most $n$ faults can occur within a certain period [24]), and unbounded fault models (e.g., fault occurrence characteristics are modeled by random parameters[4][25]). We adopt the bounded model for latent faults and make the following assumption:

**Assumption 2.** *Two consecutive latent faults are separated by the minimum time interval, denoted by $p_{ft}$.*

In Appendix A we discuss how to compute this value.

## III. SYSTEM OVERVIEW AND MODEL

**COMPOSITE background.** C'MON is built on top of the COMPOSITE [15] component-based OS in which system policies and most abstractions are defined in fine-grained user-level 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



(a) Event monitoring (b) SMC

Fig. 2: C'MON system architecture overview

by a capability system. Even low-level services such as scheduling [15], physical memory management and mapping, synchronization, and I/O management are implemented as possibly hierarchically-arranged [26], user-level components. Invoking a function in the interface of a depended-on component transparently triggers thread-migration-based [27], [28] 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. An important implication of executing system code as user-level components for this work is that the kernel itself is minimal and does not include scheduling policy, and instead only a dispatching mechanism. This means that all requests for timing-related functions are made to the synchronization, timing, and scheduling components, thus subjecting them to the C'MON monitoring and latent fault detection.

**C'MON architecture overview.** C'MON is implemented as monitoring code that is interposed on the interfaces between components in COMPOSITE and publishes events to a system monitor component that treats the events as sensor information to detect latent faults:

• *Event monitoring.* In COMPOSITE, interface stub code is generated for component and system call invocations: it simply transforms normal C calling conventions (`cdecl` in our case) into the ABI of the kernel (*e.g.,* register layout). This means that system components and applications require no code modifications to use monitoring-aware interface stubs. C'MON harnesses this stub code by *monitoring all interactions between each component and the rest of the system beyond its protection domain,* as shown in Figure 2(a). In C'MON we make the assumption that faults will propagate within a component via memory operations trivially, but that such memory errors will not span protection domain boundaries (see Assumption 1), therefore monitoring component interactions with the rest of the system makes sense: if a fault is detected, the system needs only to attribute it to that component and reconstruct a uncorrupted state.

• *System Monitor Component (*SMC*).* Events are published to the buffers that are shared between components and the SMC, as shown in Figure 2(b), when components communicate.

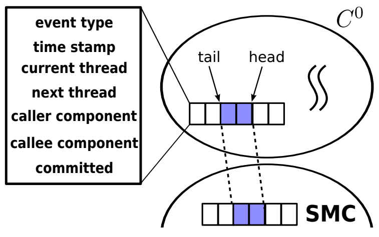
The SMC validates that dynamic execution conforms to the system model, and if it does not (*fault detection*), determines which component requires recovery (*fault localization*), therefore enabling the faulty service to be recovered. The SMC logic is essentially a large state machine which processes events and updates its own state by tracking thread associated activities (e.g., thread execution in components, prioritization, and preemptions). Importantly, each event includes a cycle-accurate time stamp counter, enabling fine-grained timing and execution tracking which is essential for detecting latent faults.

**System model and terminology.** The system in which the SMC examines the run-time constraints is modeled as follows:

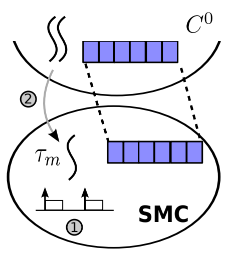
- $C = \{c^x, ...\}$ is the set of components in the system. Of these components, $c^s$ and $c^l$ are the scheduler and lock components, respectively.
- $T = \{\tau_i, ...\}$ is the set of sporadic tasks. Each task has an infinite number of jobs, with implicit deadlines.
- $p_i$ is the minimal job inter-arrival time (period) of task $\tau_i$.
- $r_{i,j}$ is the release time of the $j^{th}$ job of task $\tau_i$, thus its deadline is at $r_{i,j} + p_i$.
- $e_i$ is the worst case execution time required by task $\tau_i$ for each of its jobs.
- $u_T = \sum_{\forall \tau_i \in T} \frac{e_i}{p_i}$ is the utilization of the task set $T$.
- For simplicity, we assume fixed priority scheduling. $hp_i$ and $lp_i$ are the sets of higher and lower priority tasks than $\tau_i$, respectively.
- $e_i^x$ is the total cumulative worst case execution time of task $\tau_i$ in component $c^x$ within a job of $\tau_i$.
- $n_i^x$ is the maximum number of events that occur in $c^x$ within a job of $\tau_i$. This value bounds the rate of events generated over windows of time and is:
$$n_i^x = n_{i,ipc}^x + n_{i,ints}^x \tag{1}$$
- $n_{i,ipc}^x$ is the maximum number of IPC events that occur in $c^x$ within a job of $\tau_i$
$$n_{i,ipc}^x = \sum_{\forall c^y \in C, c^y \neq c^x} 2(n_{i,ipc}^{x \to y} + n_{i,ipc}^{y \to x}) \tag{2}$$
where $n_{i,ipc}^{x \to y}$ is the maximum number of invocation events within a job of $\tau_i$ from component $c^x$ to $c^y$. Such a value can be derived from a static analysis of system code, or empirical measurement.
- $n_{i,ints}^x$ is the maximum number of interrupts that can happen within a job of $\tau_i$ within component $c^x$. We harness the user-level interrupt vectoring in COMPOSITE to activate a job of $\tau_i$ in response to an interrupt [15].
- $e^x$ is the worst case execution time in component $c^x$ due to single invocation of component $c^x$, not including the time in any subsequently invoked components.
- The worst case execution time $e_i$ of task $\tau_i$ is
$$e_i = \sum_{\forall c^x \in C} e_i^x = \sum_{\forall c^x \in C} \sum_{\substack{\forall c^y \in C \\ c^y \neq c^x}} n_{i,ipc}^{y \to x} e^x \tag{3}$$
- $\tau_m$ is the monitor task that executes in the SMC. We assume now for simplicity that $\tau_m$ has the highest priority of all tasks in task set $T$.
- $p_m$ is the period of the monitor task $\tau_m$. When the SMC is activated periodically, we call this a *time-triggered* SMC *activation*.
- $B^x$ is the size of the event buffer holding all events in component $c^x$, and $B$ is the total buffer size of all components ($B = \sum_{\forall c^x \in C} B^x$).
- When an event buffer is full, an invocation must be made to the SMC. We call this a *buffer-triggered* SMC activation. The worst-case cost of this activation is $INV^m$.
- $e_m$ is the worst-case execution time of the monitoring task $\tau_m$ for processing events which is proportional to the per-event processing cost to detect latent faults, times the number of events generated per $p_m$.
- To detect appropriate bounds on system priority inversion we, for simplicity, assume a single resource per component, and define the resource hold times to be $rht^x$ for the resource in $c^x$. The set of components with resources that can be held while $c^x$'s resource is held are $rds^x = \{c^y, \ldots\}$ – the resource dependency set. Note that $rht^x$ includes the execution of nested resources. The maximum block time for a thread executing in $c^x$ using the priority inheritance protocol (PIP) is $b^x = rht^x + \sum_{\forall c^y \in rds^x} b^y$ where the latter term considers the nested contention for resources held by lower priority threads. Under the pessimistic assumption that each thread uses each resource, the worst case block time for any thread is $b = \max_{\forall c^x \in C} b^x$.
- We assume that faults will occur sporadically, but with a minimal inter-arrival time defined by $p_{ft}$. In an aggressive environment, a $p_{ft} = 500ms$ gives "six nines" assurance that at maximum a single fault occurs within this period. Please see Appendix A for the calculation.

## IV. SYSTEM DESIGN

C'MON is implemented in the COMPOSITE component-based system as specialized stub code that interposes on all communication between a component and the rest of the system, and a SMC component. We cover these in turn.



(a) shared event buffer



(b) monitor activation

Fig. 3: (a) An event buffer shared between SMC and the component $c^0$. (b) The monitor task $\tau_m$ is ① time-triggered (periodically), or ② buffer-triggered (the event buffer is full).

**Interface stub-based monitor implementation.** The system monitoring code in C'MON is implemented within the stub code (linked into components transparently at build time) that interposes on and tracks function calls between components, thread dispatches in schedulers and interrupt thread activations. These *events* are added to per-component buffers and each buffer is shared between a component and the SMC.

Figure 3(a) depicts such a buffer, and the contents of its entries.

*Restartable wait-free ring buffers.* As the event buffer is shared memory, both the SMC and the component being monitored must synchronize with each other. Also multiple threads (*multi-producer*) in a component can concurrently add events into the buffer from which the monitor task $\tau_m$ (*single-consumer*) in the SMC processes all entries. Using locks for synchronization would not only bind the SMC to the component (i.e., if the component suffers a failure while holding a buffer lock, the SMC would comparably be impacted), but also result in a "chicken or the egg" problem: to synchronize using a lock, we need to invoke a lock component, which requires adding an event to the buffer, which requires this synchronization. To address these issues, in C'MON the event buffer is implemented as a *multi-producer, single consumer restartable wait-free ring buffer*.

```
void event_add(evt_t type, tid_t thd, cid_t comp) {
  int tail;
  // when processing buffer, SMC will rollback execution
  // for threads with uncommitted events to here
  smc_restart_addr:
  while (1) {
    tail = rb->tail;
    struct event *e = rb->ring[tail % rb->size];
    /* buffer-triggered activation */
    if (ringbuf_full(rb)) smc_process();
    /* claim this event frame as our own? */
    if (cas(&e->owner, getthdid(), 0)) {
      /* ``help'' the holding thread increment the tail */
      cas(&rb->tail, tail+1, tail);
      continue;
    } else {
      /* we have the event entry! */
      break;
    }
  }
  /* either we, or the contending thread will tail++ */
  cas(&rb->tail, tail+1, tail);
  evt_populate(type, getthdid(), thd, getcompid(), comp);
  e->tsc = rdtsc();
  e->commit = 1;
  /* at this point, the event is successfully added */
}
```

Fig. 4: Basic "publish" operation for our restartable wait-free ring buffer.

Figure 4 summarizes the logic for a thread to publish an event to the component's buffer. The problem with traditional ring buffers is that the `tail` index into the buffer which is incremented by the producer is susceptible to the race between two producers (i.e., when one increments the tail, but then is blocked until the ring overflows, thus creating a "bubble" with an inconsistent event in the buffer, which might be reused by another thread concurrently). Our restartable wait-free ring buffer enables such events that are not yet, or partially published, to (1) be ignored by the SMC, (2) and for other threads to continue using the buffer. Note that the number of such wasted entries is bounded by the maximum number of preemptions of threads over the monitor task's period $p_m$. The core of the algorithm in Figure 4 is to (1) identify each entry with the "owner" that is attempting to populate it, (2) have that owner explicitly commit the update of an entry so that the SMC knows it is valid, and (3) enable the SMC to "restart" or roll back the execution of any thread publishing an incomplete entry so that they start the entire publishing process again. The latter functionality is trivial to accomplish in COMPOSITE as the SMC can simply make the system call to modify the register contents of threads. If an owned, but not yet committed entry is detected, when buffers are being processed, the SMC will roll the instruction pointer back to a 512 byte boundary, and we ensure that the `smc_restart_addr` address is aligned on this boundary. When this rollback happens, the event entry does not have "commit" bit set, so the SMC ignores it. The number of rollbacks is limited by the number of preemptions. Compared to [29], our *multi-producer, single consumer restartable wait-free ring buffer* is guaranteed to record every event (i.e., roll back and log again) and bound the number of wasted event entries as well.

**The SMC implementation.** To check the run-time constraints, the SMC needs to be activated and process all published events, through ① *time-triggered activation*, or ② *buffer-triggered activation*, as shown in Figure 3(b).

*Time-triggered activation* asynchronously occurs when the monitor task $\tau_m$ is periodically activated in the SMC to check run-time constraints at the periodicity of $p_m$. Importantly $p_m$ puts an upper bound on the amount of possibly erroneous computation, enabling it to be integrated into the schedulability analysis. To provide both spatial *and* temporal isolation, the SMC executes in a separate protection domain, and is run in a lower-level environment than the rest of the system such that timer-ticks are vectored to it, and the SMC can propagate them to the rest of the system (*e.g.,* to the system's scheduler). The techniques for this are detailed in HIRES [26]. In HIRES, a protocol between hierarchically arranged schedulers is used to enable virtualized subsystems to coordinate by making timing delegations between parents and children. In C'MON, the SMC acts as the parent scheduler for the system, and delegates almost all time (aside from $\tau_m$ execution every $p_m$) to the rest of the system. This effectively guarantees that the SMC's time-triggered execution is at the highest priority. This also indicates that $p_m$ is the smallest period of all tasks, which is essential for avoiding any deadline miss in C'MON.

*Buffer-triggered activation* synchronously occurs when any component's shared event buffer is full when a component invocation attempts to log its communication. In such a case, the SMC has to be activated via component invocation by the running thread. These activations could raise a synchronization problem: the threads making *buffer-triggered activation* race on accesses to SMC data-structures with the monitor task $\tau_m$ for *time-triggered activation*. Though traditional synchronization could be used here, we decided to avoid the complexity of implementing locks and the appropriate synchronization protocols, and instead make all event processing in the SMC be conducted solely by $\tau_m$. Thus upon a *buffer-triggered activation*, the intended event is stored, the invocation is made to the SMC, at which point the SMC immediately switches to the monitor task $\tau_m$ which processes published events from all components and examines the run-time constraints at the highest priority.

*Time-space trade-off.* There is a significant time-space trade-off in C'MON design. On the one hand, if the event buffers are small (*e.g.,* holding zero or one entry), then a thread wishing to add the event must do synchronous *buffer-triggered activation* more frequently by invoking the SMC and switching to $\tau_m$ to process all events. On the other hand, if the system is willing to devote more memory to the event buffers, the costs of these synchronous activations can be decreased, possibly increasing system schedulability. Therefore, one of design objectives in C'MON is to use the least possible amount of memory for event buffers and still guarantee that the system timing constraints are satisfied. In Section V, we introduce an algorithm for assigning memory, while for optimizing schedulability.

**Constraint checks implementation.** System execution must adhere to the system model specified in Section III. Here we briefly discuss how the SMC validates that actual execution conforms to the system model. For a thorough treatment of the system events, and how the constraints we discuss here map to the system model, please see Appendix B.

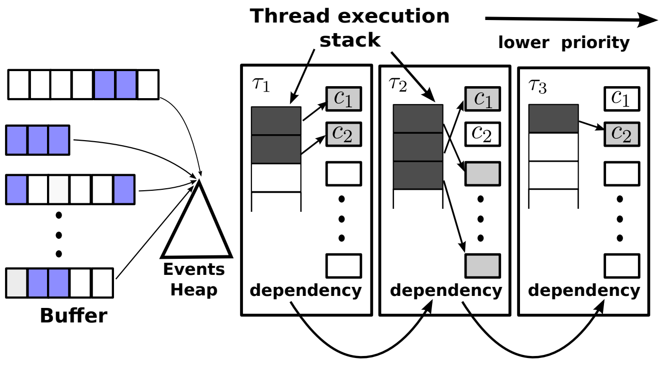


Fig. 5: The main SMC data-structures for event processing.

The data-structures used to track thread execution are depicted in Figure 5. To validate that the run-time execution conforms to or differs from the system model (thus the faulty component is detected), the SMC examines each thread execution in components as it processes all published events in the order of occurrence. Each of the system components has a possibly different sized buffer shared with the SMC. A heap is used to sort the head entry of each of the buffers to ensure ordered event traversal. Though for a $N$-component system this naively involves a $O(log\,N)$ heap operation for each event, we avoid this in most cases by tracking where the next event should come from *unless there is a preemption*. (i.e., an event of invocation from component $c^x$ to $c^y$ should expect the next event in $c^y$ if no preemption occurs). Thus, the number of actual heap operations is proportional not to the number of events, but the number of preemptions.

Figure 5 also shows the per-thread data-structure maintained by SMC, including a stack that tracks which component the thread is executing in and a small structure for each component. When a thread invokes a component, an item is pushed on the stack. This item includes a pointer to the component structure in the thread being invoked, and the timestamp for that invocation. A thread's cumulative execution time is incremented for each event it published. A thread's cumulative component execution time is tracked in the per-thread component structures, and the worst-case component invocation time is updated in the stack entry. Proper scheduling behavior involving always choosing the highest priority thread is validated by maintaining a shadow run-queue sorted by priority. Blocking times (*i.e.,* when the system breaks the "highest-priority thread should aways run") are tracked for PIP. Instead of using a stack as a common implementation technique, each thread structure maintains a dependency field. When an invocation due to shared resource contention is made, this dependency field is made to point to the thread holding that resource which will create a dependency tree, as shown in Figure 5. The dependencies are followed for the highest-priority thread until a thread is found with no dependencies to determine which thread should be chosen for execution by the scheduler. If the cycle is found during the traversal in the dependencies tree, the deadlock is detected. All executions following dependencies are tracked and checked against the maximum blocking time in the system model (see the blocking time discussion in Section III).

**Recovery in C'MON.** Once detected and localized, the faulty system service must be recovered. As a complementary fault recovery function, the previous work $C^3$ [7] is used in this paper to recover the faulty component, even though other recovery mechanisms (*e.g.,* checkpointing as also studied in [7]) could also be used. Here we only give a brief summary on how $C^3$ works. For more details, please refer to [7]. $C^3$ is a fault tolerance system built on COMPOSITE. It assumes fine-grained protection domains, and explicit interfaces between components. $C^3$ recovers the faulty service by (1) first providing efficient and predictable micro-reboot to bring the component to a safe (initial) state, then (2) using the intelligent recovery code and functions in the interfaces of the components themselves to reconstruct the state that the rest of the system expects. For example, if a scheduler is identified as a faulty service (e.g., C'MON has detected a *improper scheduling* decision made in the scheduler), $C^3$ will be activated to reboot the scheduler to an initial state, then the rest of the components in the system make invocations on the functions exported by the scheduler to make it aware of all of the threads in the system, of their priorities, and of the thread state (*e.g.,* if they are blocked). And eventually the scheduler will be brought back to a consistent state. Though $C^3$ can provide predictable system-level recovery, the focus of C'MON in this work is on system-level latent fault detection with an emphasis on timing faults.

**SMC and shared buffer vulnerability.** It is important in C'MON that the SMC itself not be erroneous. On the one hand, as with all components in COMPOSITE, the SMC has its own protection domain and has no dependencies on other components. The amount of memory required for the SMC is small: 920 bytes for each system thread, and 56 bytes for each component in our system. The SMC is simple (under 1K LoC), and other components have very limited access to it: the SMC only exports two function entry points for (1) processing events (upon the *buffer-triggered activation*), and (2) mapping in a shared buffer. Also compiler techniques (*e.g.,* [30]) for hardening the code could be used to further decrease the chance of SMC being corrupted. Therefore, even though it is still possible that faults occur in SMC, with a small footprint, simple interfaces and possible compiler hardening techniques, the chance that SMC could be impacted by an SEU is minimized while a wide range of temporal constraint violations it enables to be detected.

To correctly detect the run-time constraints violation, the events logged in the shared buffers must be either uncorrupted, or able to assist faulty component detection. To facilitate this, the shared buffer contains no pointers. Regardless, a faulty component can corrupt the events in its shared buffer. In C'MON, all events are logged *redundantly* in all components involved in the interaction (i.e., when a component invokes another, both components publish events on their exit and entry for a total of *four* events (see Figure 9)). Thus multiple components buffers are referenced to validate the integrity of any event entry. For example, if a component $c^x$ is invoking $c^y$, $c^x$ will attempt to generate an event declaring it is calling $c^y$. If an argument of that event is corrupted in the shared buffer between $c^x$ and the SMC, such that it is logged as an invocation to $c^z$ instead, the corruption will be detected by a mismatch between the logs in $c^x$ and $c^z$. Details about how this general class of log corruption bugs are detected are beyond the scope of this paper and are discussed in an online extended version of this paper at www.seas.gwu.edu/~gparmer/. As an alternative design, C'MON could implement logging in the kernel. However, such a design (1) increases the footprint, thus fault surface, of the kernel significantly, (2) doesn't have redundant data spanning multiple protection domains, thus an event buffer corruption could be hard to recover from, and (3) it would significantly increase the complexity of the kernel.

## V. System Timing Analysis

A primary goal of C'MON is to provide predictable monitoring, hence enabling the recovery for hard real-time systems. This section integrates the system overheads from our C'MON system into a schedulability analysis. This analysis will enable a system, even in the presence of latent timing faults, to avoid missing deadlines. We extend a response-time analysis [31] (RTA) that is broadly applied to fixed-priority systems. The overall structure of the RTA is based on a recurrence that finds a fixed point less than the task's deadline, otherwise determines the task is not schedulable:

$$R_i^{n+1} = R_i^n(RTA) + R_i^n(C^3) + R_i^n(MON) + R_i^n(F) \quad (4)$$

In the following we will discuss how each term on the right hand side of above equation contributes in the RTA (please refer to Section III for the terminology we use here):

**i)** $R_i^n(RTA)$ is the traditional response-time analysis [31]:

$$R_i^n(RTA) = e_i + b_i + \sum_{\forall \tau_j \in hp_i} \left\lceil \frac{R_i^n}{p_j} \right\rceil e_j \quad (5)$$

**ii)** $R_i^n(C^3)$ is the contribution from C$^3$ [7] using "on-demand" recovery. It contains the cost of micro-rebooting ($e_{ur}$) the faulty component, and the cost of rebuilding the state ($r_j(m_j)$) for *m* objects in the failed component for higher-priority task $\tau_j$ during the recovery:

$$R_i^n(C^3) = \left\lceil \frac{R_i^n}{p_{ft}} \right\rceil (e_{ur} + \sum_{\forall \tau_j \in hp_i} r_j(m_j)) \quad (6)$$

**iii)** $R_i^n(MON)$ is the contribution from the monitoring overhead in C'MON, which includes (1) synchronous *buffer-triggered activation* in SMC, and (2) the execution time ($e_m$) of the monitor task $\tau_m$ that is proportional to the maximum number of events generated per monitor task period $p_m$, and their per-event processing cost. To obtain $R_i^n(MON)$, we must first determine how many *buffer-triggered activations* can occur within $p_m$. This is a function of the amount of memory in each buffer, and the number of events generated between *time-triggered activations*. First, we define the buffer size $B^x_{nosyn}$ for a component $c^x$ for which no synchronous *buffer-triggered activation* could arise:

$$B^x_{nosyn} = \sum_{\forall \tau_i \in T} \left\lceil \frac{p_m}{p_i} \right\rceil n_i^x + \sum_{\forall \tau_h \in hp_i} \left\lceil \frac{p_m}{p_h} \right\rceil = \sum_{\forall \tau_i \in T} n_i^x + |hp_i|$$

The first term determines the maximum number of events generated within $p_m$ (where $n_i^x$ is the maximum number of events that occur in $c^x$ within a job of $\tau_i$), and the second considers the entries required in the buffer due to the buffer entries being skipped because of the restart-based synchronization around the wait-free buffers (see Figure 4). As $\forall \tau_i, p_m < p_i$, the simplification on the right holds. Recall that $B^x$ is the size of the event buffer holding all events in component $c^x$, the maximum number of *buffer-triggered activations* in the SMC is derived as:

$$M_m = \max_{\forall c^x \in C} \left\lceil \frac{B^x_{nosyn}}{B^x} \right\rceil - 1$$

Given that $INV^m$ is the cost of the invocation to the SMC due to each *buffer-triggered activation*, $R_i^n(MON)$ is finally derived as:

$$R_i^n(MON) = \left\lceil \frac{R_i^n}{p_m} \right\rceil (M_m \times INV^m + e_m) \quad (7)$$

**iv)** $R_i^n(F)$ is the contribution from the fault localization overhead and wasted computation. When a fault occurs and is detected, the faulty component must be localized to enable the recovery of the affected service and we denote such overhead as $e_m(localization)$ (see detailed analysis in Appendix B). Note the overhead for actually recovering the faulty component is already considered in $R_i^n(C^3)$. Additionally, there is a period of computation within the faulty component which cannot be trusted. All of that execution could have simply been iterations through an infinite loop. This *wasted computation* is the period of the monitor task $p_m$, plus the worst-case execution time of a system-level component, which we denote $w^f = max_{\forall c^x \in C}\{e^x\}$. This considers the worst-case where a fault is only detected on a component execution overrun. So $R_i^n(F)$ is given as:

$$R_i^n(F) = \left\lceil \frac{R_i^n}{p_{ft}} \right\rceil (e_m(localization) + p_m + w^f) \quad (8)$$

### A. Memory Allocation for Schedulability

The values for $B^x$ are fixed in the RTA. However, they can have a large impact on the schedulability of the system which manifests as a time-space trade-off between using little memory for event buffers, and having lower computational overheads for synchronous *buffer-triggered activation* of the

SMC. Thus, C'MON integrates a greedy heuristic that attempts to *achieve system schedulability, while minimizing the amount of memory required for buffers*. Assume $B_{nosyn}$ is the total buffer size for which no synchronous *buffer-triggered activation* could occur from any component (i.e., $B_{nosyn} = \sum_{\forall c^x \in C} B^x_{nosyn}$) and recall that $B$ is the total buffer size of all components (i.e., $B = \sum_{\forall c^x \in C} B^x$), we first initialize $\forall c^x, B^x = 1$ so that every event after the first causes a synchronous activation of $\tau_m$. Then: (1) If a RTA of the system is successful, a solution is found for the event buffer sizes. (2) Otherwise, double the amount of memory allocated ($B = B \times 2$). (3) If $B \geq B_{nosyn}$, then the system is unschedulable for any value of $B$. (4) Otherwise binary search between the current value of $B$ and $B/2$ to find the minimum value of $B$ that can still make the system schedulable.

When a value of $B$ is evaluated for producing a schedulable system, C'MON must determine how to allocate this memory between components. Again, we use a simple heuristic that allocates the minimum amount of memory to the component that will decrease the number of the *buffer-triggered activation* the most. This is done repeatedly until all memory is allocated. This algorithm is run *offline* during a timing analysis, and returns answers immediately, so we believe it is practical.

### B. Monitor Periodicity of Time-triggered Activation

The periodicity of the monitor task ($p_m$) has an impact on system schedulability as well, and it's value must be determined by the system designer. A large value will not detect faults soon enough for the system to recover, while a small value will induce more *time-triggered activation* overheads. Section VI investigates $p_m$'s impact on system schedulability. We use a trivial algorithm for computing a schedulable $p_m$ (if one is found): Iterate the $p_m$ starting from the length of a single timer tick, up through the minimum periodicity of any real-time task, recomputing the RTA each time.

## VI. EXPERIMENTAL EVALUATION AND RESULTS

### A. C'MON *micro-benchmarks*

We evaluate four types of overhead in C'MON infrastructure: 1) per-event buffering overhead – the fast path of publishing an event, 2) per-event processing overhead (including constraint check and faulty component localization), 3) SMC *buffer-triggered activation* overhead, and 4) periodically SMC *time-triggered activation* overhead, Unless otherwise specified, experiments are run on an Intel i7-2760QM running at 2.4 Ghz (only one core enabled) and Table I shows these C'MON infrastructure overhead (in units of $\mu$-seconds):

| | |
|---|---|
| per-event buffering | 0.052 (0.008), 0.08 |
| per-event processing | 0.23 (0.048), 0.35 |
| SMC buffer-triggered activation | 0.51 (0.014), 0.71 |
| SMC time-triggered activation | 0.54 (0.023), 0.67 |

TABLE I: The "average (stddev), maximum" infrastructure overhead

Note that above "per-event buffering" overhead could have been reduced by nearly 50% if the events were logged in the kernel. However, the kernel footprint and complexity, thus the fault surface, would increase.

### B. C'MON *latent fault detection and localization*

**Workload.** We evaluate the fault tolerance effectiveness of C'MON for three system level components: 1) the system scheduler, 2) the system physical memory manager and mapper, and 3) the lock manager. Each system service is evaluated in C'MON while running the following workloads:

- *Scheduler (Sched):* Three threads at high, medium and low priority contend over a number of locks, resulting in priority inversion.
- *Memory Manager (MM):* Systems are granted a number of pages, these pages are aliased once, and then revoked, which removes all aliases.
- *Lock:* Three threads at high, medium and low priority acquire and release locks in the properly nested manner.

**Fault injection.** Four types of erroneous timing behaviors are randomly created in the scheduler, memory manager and lock to mimic the occurrence of the latent fault:

- component execution overrun – a long-running or infinite loop is occasionally introduced into *Sched* and *MM*.
- unbounded priority inversion (PI) – a delay is occasionally introduced into the scheduler as the scheduling decision is made whenever a low priority thread switches to a high priority thread during PI.
- improper scheduling – occasionally the scheduler chooses a lower priority thread to run when a higher priority thread is runnable during PI.
- deadlock – occasionally a task is made to contend the lock that one of its dependents is holding.

Each type of faults listed above is injected 1000 times. We measure how many times the erroneous behaviors are detected and how many times the faulty services are localized by C'MON successfully, hence enabling the recovery (e.g., using $C^3$ to recover the faulty component).

| | fault type | detected | localized |
|---|---|---|---|
| MM execution overrun | Fig 1(a) | 1000 | 1000 |
| Sched execution overrun | Fig 1(a) | 1000 | 1000 |
| Unbounded PI | Fig 1(b) | 1000 | 1000 |
| Improper scheduling | Fig 1(c) | 1000 | 1000 |
| Deadlock | Fig 1(d) | 1000 | 1000 |

TABLE II: Fault detection and localization success rate

The results shows that C'MON can effectively detect the abnormal timing behavior of tasks and localize the faulty services for all injected faults at the system-level.

### C. *Schedulability Evaluation in* C'MON

To evaluate the schedulability, each task set has 50 tasks with an average period of 100ms where the task's period follows a uniform random distribution and the number of events that occur in each component follows a random exponential distribution. The fault period $p_{ft}$ is fixed at 500 ms and the number of objects recovered by $C^3$ is fixed at 5 (i.e., $m_j = 5$ in Eq. (6)). In order to evaluate the impact imposed by the monitoring infrastructure on the system schedulability, we define the *event rate* (evts/ms) as the total number of events that occur in one millisecond.

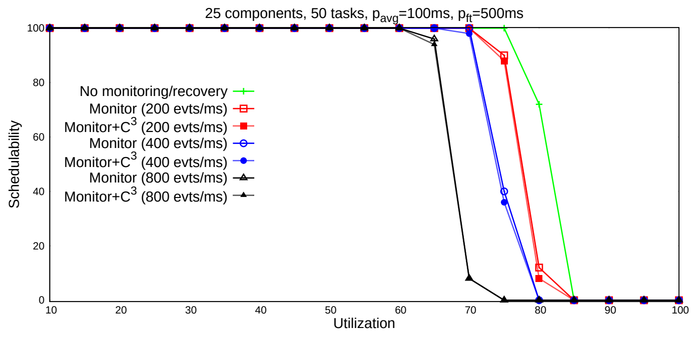


Fig. 6: Schedulability vs Utilization

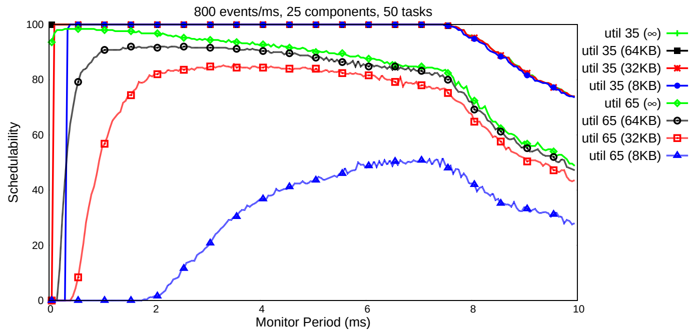


Fig. 8: Schedulability vs Monitoring Period

**Figure 6** illustrates the system schedulability as the function of task set utilization at different event rates. For comparison, we choose the system with event rates of 200, 400 and 800 events/ms, respectively. The system with lower event rate can attain a higher schedulability than the system with higher event rate at high utilization (*e.g.,* the system with 200 events/ms starts decreasing its schedulability until after 70% utilization). This is due to the fact that the system with a higher event rate will render more events within $p_m$, thus increasing $e_m$, and the interference on system tasks. Also we observe that incorporating C$^3$ in the system with C'MON has negligible impact on the schedulability.

**Discussion.** The C'MON infrastructure does have an impact on the schedulability of systems, and depends significantly on the event rate of the system in question. In our system, the decrease due to C'MON was limited to between 5% and 15% of available utilization. C'MON pairs well with C$^3$ to predictably provide not only fault detection, but also recoverability.

**Figure 7** presents the lower bound of total amount of memory required by the system with C'MON to be schedulable, at different event rates. For a given event rate, we show the memory requirements for the system to be schedulable, when it consists of 10, 25 and 50 components. The result indicates that the systems with more components and higher event rates need more memory to achieve schedulability at higher utilization. For example, at 800 events/ms and 60% utilization, the system with 50 components needs 60KB and the one with 10 components only needs less than 10KB.

**Discussion.** The number of the *buffer-triggered activation* depends on both event rate and allocated memory size. The buffer can be more quickly filled up in a system at higher event rate which results in more synchronous SMC invocations, thus hurting system schedulability. Higher utilizations leave less slack that can be filled with overhead from the *buffer-triggered activation*.

**Figure 8** depicts how the system schedulability is affected by the choice of monitoring period $p_m$. In the previous experiments, we search for a $p_m$ resulting in a schedulable system (how to find the schedulable $p_m$ is discussed in Section V) . Here we show its impact explicitly. For comparison, we have chosen task set of 35% and 65% utilization and different amount of memory (8KB, 32KB, 64KB and infinite) are allocated to the system. The result shows that all task sets have a very small schedulability, if not zero, when $p_m$ is very small and the schedulability increases when $p_m$ increases. Also all task sets have a noticeable decline in schedulability around $p_m$=$7.8ms$. Importantly, the tasks in our task set are generated with a minimum periodicity of 10ms. This is why the search space for $p_m$ is so small and narrow. A task set with larger periodicities would afford more leeway in $p_m$ selection.

**Discussion.** The monitoring period $p_m$ can affect the system schedulability in two ways, one is through the C'MON overhead as the denominator in Eq. (7) and the other one is within the wasted time in Eq. (8). For very small $p_m$, the C'MON overhead dominates the interference and for large $p_m$ (e.g., $7.8ms$), the wasted time dominates the interference. In between, the schedulability is decided by the ratio of these two, which causes small "sawtooths" in the graphs. These are smoothed out here as we report the average over 10 runs.

**Fault period.** The fault period $p_{ft}$ affects the system schedulability in the similar way as it does in C$^3$ [7]. Therefore we make the similar statement: *a system could be unschedulable until some $p_{ft}$ is reached, after which the schedulability is maintained regardless of how large $p_{ft}$ is.*

## VII. RELATED WORKS

**Run-time Timing Constraint Monitoring.** Run-time monitoring has been used to detect violations in design assumptions. Hardware-based runtime monitors such as [32][33] bring minimal intrusiveness to the target system, however, these techniques suffer from high cost and poor portability. Chodrow et al. [34] proposed synchronous and asynchronous timing violations monitoring based on a constraint-graph algorithm using RTL [35]. The monitor in [36] detected the event occurrence by inserting the detection code into applications and kernel. Mok et al. [37] proposed a Java runtime timing constraint monitor (JRTM) to detect temporal constraint violations at the earliest possible time. The runtime monitoring of event flows end-to-end timing in distributed real-time systems is investigated in [38]. Haitao Zhu et al. [39] investigated bounding detection latency in order to achieve predictable monitoring. As the tracing infrastructure in the traditional system, Feather-trace [29] has been used to track timing of execution within monolithic systems. Another issue related to the predictable event monitoring, such as the bounded event histories size, is studied in [40][41]. However, most of these works did not take the intrusiveness of monitoring, the detection latency and the event history size into consideration in the system schedulability, nor are they applied in a fault tolerance infrastructure for system level fault detection and

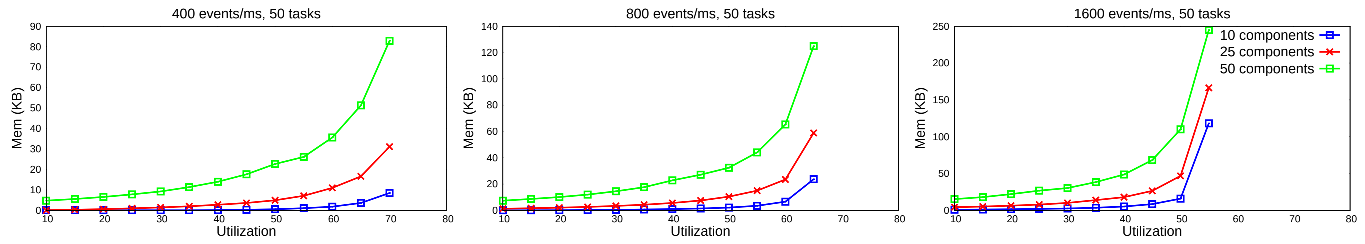


Fig. 7: Minimum memory required for system schedulability.

recovery. C'MON not only detects by monitoring with fine-grained fault isolation of system-level code, but also allows the system-level recovery to tolerate latent faults predictably.

**Reliable system design.** Many research works have been dedicated for ensuring that operating systems are resilient to faults. Swift et al. [42] present Nooks where device drivers are isolated within protection domains inside a monolithic kernel address space, where hardware and software prevent them from corrupting the rest kernel. Additionally Nooks [42] provides monitoring of driver interactions for recovery. *$\mu$-kernels* provide fault tolerance by isolating kernel extensions in user-mode where fault containment can be more easily achieved. CuriOS [43] saves client-specific state information in protected memory, Minix [44] tolerates the faulty service via a *Reincarnation Server*, and Singularity [45] achieves the isolation through software-isolated processes written in type-safe language. Our previous work $C^3$ [7] tolerates the transient faults in even more general system critical services (i.e. scheduler, memory manager and file system). Checkpointing is another popular way to ensure system reliability and has been studied extensively in the past. The works presented in [5][46], among others, study the effect of checkpointing on the schedulability of fault-tolerant task sets. We investigated the checkpointing and schedulability problem at the system level in [7] as well. Most of these research projects assume the *fail-stop model* and do not provide automatic runtime means for dealing with error detection latency at system level, which is undesirable for real-time embedded systems. Differentiating from previous works, C'MON detects, hence allows the complementary recovery techniques (e.g., $C^3$) to recover from *latent* faults at *system level predictably*.

## VIII. Conclusions

**Limitations.** Though C'MON has been proved effective and predictable in detecting latent faults, it has limitations: 1) The characteristics of the detected faults in C'MON are inherited from $C^3$, therefore C'MON does not detect deterministic faults (i.e., hardware faults or deterministic software bugs), 2) C'MON leverages and, therefore is tightly related to, the structural properties of COMPOSITE (i.e., the components communication through the interfaces), 3) C'MON assumes that two successive faults arrive with a minimum inter-arrival time (see Assumption 2), so it does not detect the *burst faults* [23] or *unbounded faults* [25].

C'MON is the first system we know of to provide predictable latent fault detection and enable the recovery for the lowest-level components of a system including the scheduler, physical memory manager and lock. It does so by combining an efficient and predictable monitoring infrastructure, with a system-overhead-aware timing analysis to produce systems that can detect latent timing faults, *and recover from them* without missing any deadlines. We present a low-level system architecture based on fine-grained fault isolation, and an event processing system based on a restartable wait-free ring buffer, and an isolated and simple SYSTEM MONITORING COMPONENT. Also C'MON provides facilities for optimizing buffer memory usage, and optimizing the monitor's periodicity. C'MON leverages all of this to detect, hence recover from latent faults with only a relatively small (5% - 15%) decrease in the utilization of schedulable tasks for the systems we studied. All code for C'MON and COMPOSITE is located at composite.seas.gwu.edu.

## References

[1] J. Ziegler, "Terrestrial cosmic rays," *IBM J. Res. Dev.*, 1996.
[2] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.
[3] P. Mejía-Alvarez and D. Mossé, "A responsiveness approach for scheduling fault recovery in real-time systems," in *RTAS*, 1999.
[4] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright, "Probabilistic scheduling guarantees for fault-tolerant real-time systems," in *DCCA*, 1999.
[5] S. Punnekkat and A. Burns, "Analysis of checkpointing for schedulability of real-time systems," in *RTCSA Workshop*, 1997.
[6] 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.
[7] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in $C^3$," in *RTSS*, 2013.
[8] S. Ghosh, R. Melhem, and D. Mosse, "Enhancing real-time schedules to tolerate transient faults," in *RTSS*, 1995.
[9] A. Burns, R. Davis, and S. Punnekkat, "Feasibility analysis of fault-tolerant real-time task sets," in *ECRTS Workshop*, 1996.
[10] H. Madeira and J. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking," in *FTCS*, 1994.
[11] P. Chevochot and I. Puaut, "Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from cots components," in *DSN*, 2001.
[12] S. Chandra and P. Chen, "How fail-stop are faulty programs?" in *FTCS*, 1998.
[13] K. S. Yim, Z. Kalbarczyk, and R. Iyer, "Quantitative analysis of long-latency failures in system software," in *PRDC*, 2009.
[14] M. W. Maier and E. Rechtin, *The Art of Systems Architecting*, 2000.
[15] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS*, 2008.
[16] R. Venkatasubramanian, J. Hayes, and B. Murray, "Low-cost on-line fault detection using control flow assertions," in *IOLTS*, 2003.
[17] R. Shafik, G. Rauwerda, J. Potman, K. Sunesen, D. Pradhan, J. Mathew, and I. Sourdis, "Software modification aided transient error tolerance for embedded systems," in *DSD*, 2013.
[18] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *ASPLOS*, 2010.
[19] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, "A comparison of partitioning operating systems for integrated systems," in *SAFECOMP*, 2007.
[20] R. Barbosa, J. Karlsson, Q. Yu, and X. Mao, "Toward dependability benchmarking of partitioning operating systems," in *DSN*, 2011.
[21] J. Strnadel and F. Slimarik, "On distribution and impact of fault effects at real-time kernel and application levels," in *DSD, 2012*.
[22] N. Ignat, B. Nicolescu, Y. Savaria, and G. Nicolescu, "Soft-error classification and impact analysis on real-time operating systems," in *DATE*, 2006.
[23] F. Many and D. Doose, "Scheduling analysis under fault bursts," in *RTAS*, 2011.

[24] C.-C. Han, K. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Trans. Comput.*, 2003.
[25] I. Broster, A. Burns, and G. Rodriguez-Navas, "Probabilistic analysis of can with faults," in *RTSS 2002*.
[26] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *RTAS*, 2011.
[27] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *OSPERT*, 2010.
[28] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *WTEC*, 1994.
[29] B. Brandenburg and J. Anderson, "Feather-trace: A light-weight event tracing toolkit," in *OSPERT*, 2007.
[30] U. Schiffel, A. Schmitt, M. SuBkraut, and C. Fetzer, "Software-implemented hardware error detection: Costs and gains," in *DEPEND*, 2010.
[31] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.
[32] B. Plattner, "Real-time execution monitoring," *IEEE Softw. Eng.*, 1984.
[33] J.-P. Tsai, K.-Y. Fang, and H.-Y. Chen, "A noninvasive architecture to monitor real-time distributed systems," *IEEE Computer*, 1990.
[34] S. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *RTSS*, 1991.
[35] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Trans. Software Engineering*, 1986.
[36] P. S. Dodd, C. V. Ravishankar, Chinya, and V. Ravishankar, "Monitoring and debugging distributed realtime programs," in *SPE*, 1992.
[37] A. Mok and G. Liu, "Early detection of timing constraint violation at runtime," in *RTSS*, 1997.
[38] H. Kim, S. Yi, W. Jung, and H. Cha, "A decentralized approach for monitoring timing constraints of event flows," in *RTSS*, 2010.
[39] H. Zhu, M. Dwyer, and S. Goddard, "Predictable runtime monitoring," in *ECRTS*, 2009.
[40] R. Pineiro, K. Ioannidou, S. Brandt, and C. Maltzahn, "Rad-flows: Buffering for predictable communication," in *RTAS*, 2011.
[41] A. Mok, P. Konana, G. Liu, C.-G. Lee, and H. Woo, "Specifying timing constraints and composite events: an application in the design of electronic brokerages," *IEEE Trans. Software Engineering*, 2004.
[42] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP*, 2003.
[43] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: Improving reliability through operating system structure," in *OSDI'08*.
[44] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Failure resilience for device drivers," in *DSN*, 2007.
[45] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, 2007.
[46] Y. Zhang and K. Chakrabarty, "Fault recovery based on checkpointing for hard real-time embedded systems," in *DFT*, 2003.
[47] "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices," *JEDEC Standard, JESD89A*, 2006.
[48] E. Normand, D. Oberg, J. Wert, J. Ness, P. Majewski, S. Wender, and A. Gavron, "Single event upset and charge collection measurements using high energy protons and neutrons," *Nuclear Science*, 1994.
[49] P. Hazucha and C. Svensson, "Impact of cmos technology scaling on the atmospheric neutron soft error rate," *Nuclear Science*, 2000.
[50] J. Ferreira, "An experiment to assess bit error rate in can," in *RTN*, 2004.
[51] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, 1990.

## APPENDIX A

**Computing the fault periodicity.** SEUs are inherently probabilistic and determined by environmental effects (*i.e.,* cosmic rays). However, models exist that can predict the distributions of faults. The goal of C'MON is not to obviate all faults with absolute certainty, but to decrease significantly the probability that a fault will lead to system failure. Specifically, in an aggressive environment with significant radiation, an engineer can place a "six nines" (99.999999%) probability that zero or one fault will happen within a 0.5 second period. If this is an acceptable risk, then $p_{ft} \leq 0.5s$ is appropriate for the system.

This work makes the assumption that the fault periodicity ($p_{ft}$) can be derived and used in the timing analysis. Realistically, system engineers must choose a value of $p_{ft}$ that implies an acceptable probability that at most one fault occurs within that window. This section describes a method to derive $p_{ft}$.

We assume that the distribution of the transient faults in any fixed time interval follows a Poisson distribution, as suggested in [4].

$$Pr_k(t) = \frac{e^{-\lambda t}(\lambda t)^k}{k!} \tag{9}$$

gives the probability of $k$ faults during an interval of duration $t$, where $\lambda$ is the expected number of faults in unit time. $\lambda$ is also called soft error rate or fault rate and is assumed to be constant. Therefore, the probability of at most *one fault* in the time interval $t$ ($p_{ft}$ in the rest of the paper), the assumption of the timing analysis in Section V, is given by

$$Pr_{k<2}(t) = e^{-\lambda t}(1 + \lambda t) \tag{10}$$

According to Equation (6.10) in [47], the fault rate $\lambda$ is a function of terrestrial neutron flux and the "SEU cross section", an intrinsic parameter of a chip or circuit. A typical value of differential flux at sea level in New York City is $4.58\text{x}10^{-5}$ cm$^{-2}$ MeV$^{-1}$ s$^{-1}$ when the neutron energy is around 10.51MeV. A typical value of the cross section is between $10^{-14}$ and $10^{-12}$cm$^2$/bit [48]. In [49], a fault rate of 12700 to 18200 FIT/chip (Failures In Time over $10^9$ hours per chip) is given for a 4Mbit SRAM chip at New York City, which is equivalent to $10^{-5}$ faults per hour. According to [49], this rate will increase at higher altitudes (*e.g.,* a flight endures $10^{-3}$ faults per hour at 40K ft). Indeed, the typical values for $\lambda$ range from hundreds of faults per hour in aggressive environments to $10^{-3}$ faults per hour in lab conditions, as shown by Ferreira et al. [50].

Assuming $\lambda = 10^{-3}$, with Equation (10) we obtain a six nines probability (*i.e.,* 99.999999%) that at most one fault occurs over a window of 509.15 seconds (*i.e.,* $p_{ft} \leq 509.15$ seconds). In a more aggressive environment with $\lambda = 1$, we achieve the same probability when $p_{ft} \leq 0.5$ second.

## APPENDIX B

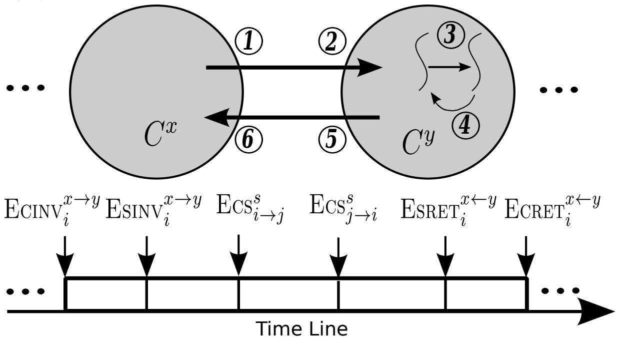
**Event Model.** A thread's computation is viewed as a sequence of events occurrence where *event* informally represents the state change that may occur [34]. The set of event types of note in the system that are observable by the logging infrastructure are defined as ELOG:

$$\begin{aligned}\text{ELOG} = \{&\text{ECINV}_i^{x\rightarrow y}, \text{ESINV}_i^{x\rightarrow y}, \text{ESRET}_i^{x\leftarrow y}, \\ &\text{ECRET}_i^{x\leftarrow y}, \text{ECS}_{i\rightarrow j}^{s}, \text{EINT}_{i\rightarrow h}^{x\rightarrow w}\}\end{aligned} \tag{11}$$

where $\tau_h$ is activated in $c^w$ by the interrupt and the scheduler component is $c^s$, $\forall c^x, c^y \in C$ and $\forall \tau_i, \tau_j \in T$.

- $\text{ECINV}_i^{x\rightarrow y}$ indicates an event in client component $c^x$ when task $\tau_i$ is about to make the invocation to server component $c^y$ from $c^x$ ($c^x \neq c^y$).
- $\text{ESINV}_i^{x\rightarrow y}$ indicates an event in server component $c^y$ when $\tau_i$ enters $c^y$ due to the invocation to $c^y$ from $c^x$ ($c^x \neq c^y$).
- $\text{ESRET}_i^{x\leftarrow y}$ indicates an event in $c^y$ when $\tau_i$ is about to return to $c^x$ from $c^y$ ($c^x \neq c^y$).

- $\text{ECRET}_i^{x \leftarrow y}$ indicates an event in $c^x$ when $\tau_i$ returns back to $c^x$ from $c^y$ ($c^x \neq c^y$).
- $\text{ECS}_{i \to j}^s$ indicates an event in $c^s$ when context switch from $\tau_i$ to $\tau_j$ occurs ($\tau_i \neq \tau_j$).
- $\text{EINT}_{i \to h}^{x \to w}$ indicates an event when interrupt task $\tau_h$ is activated in $c^w$ due to an interrupt while $\tau_i$ is executing in $c^x$.



1
2
3
4
5
6
$C^x$
$C^y$
$\text{ECINV}_i^{x \to y}$
$\text{ESINV}_i^{x \to y}$
$\text{ECS}_{i \to j}^s$
$\text{ECS}_{j \to i}^s$
$\text{ESRET}_i^{x \leftarrow y}$
$\text{ECRET}_i^{x \leftarrow y}$
Time Line

Fig. 9: Event Model

Figure 9 illustrates a few of these events involved in an invocation between components, and a context switch in a scheduler. The decomposition of events transforms the execution state of task $\tau_i$ into the trace of events occurred in components, which allows us to profile the temporal behavior of a task with finer granularity (i.e. at component level). This is important for a fault tolerant system to quickly detect any violation of specified constraints and take the appropriate corrective action in the early stage.

In a sequence of events occurred in the system, the $k^{th}$ event is denoted as $\text{E}_{i,k}^x$ if it occurs in component $c^x$ by task $\tau_i$. In the case that the component and task information are not relevant, event $\text{E}_{i,k}^x$ is simply denoted as $\text{E}_k$.

**Definition** $\Phi(\text{E}_k)$ *is the event type of* $\text{E}_k$. $\forall k, \Phi(\text{E}_k) \in \text{ELOG}$,

**Definition** $@(\text{E}_k)$ *is the occurrence time of event* $\text{E}_k$.

$$\forall k, @(\text{E}_k) < @(\text{E}_{k+1}) \tag{12}$$

**Definition** $\Delta_i(\text{E}_{k_1}, \text{E}_{k_2})$ *is task* $\tau_i$*'s total accumulated execution time between any two events* $\text{E}_{k_1}$ *and* $\text{E}_{k_2}$ *(*$k_2 > k_1$*)*

$$\Delta_i(\text{E}_{k_1}, \text{E}_{k_2}) = \sum_{k=k_1}^{k_2-1} \delta_i(\text{E}_{l,k}^x, \text{E}_{h,k+1}^y) \tag{13}$$

*where* $\delta_i(\text{E}_{l,k}^x, \text{E}_{h,k+1}^y)$ *is task* $\tau_i$*'s cumulative time between two consecutive events,* $\text{E}_{l,k}^x$ *and* $\text{E}_{h,k+1}^y$

$$\delta_i(\text{E}_{l,k}^x, \text{E}_{h,k+1}^y) = \begin{cases} @(\text{E}_{h,k+1}^y) - @(\text{E}_{l,k}^x) & \textit{if } \tau_l = \tau_i \\ 0 & \textit{otherwise} \end{cases} \tag{14}$$

**Definition** $\Delta_i^x(\text{E}_{k_1}, \text{E}_{k_2})$ *is task* $\tau_i$*'s total cumulative execution time in* $c^x$ *between any two events* $\text{E}_{k_1}$ *and* $\text{E}_{k_2}$ *(*$k_2 > k_1$*)*

$$\Delta_i^x(\text{E}_{k_1}, \text{E}_{k_2}) = \sum_{k=k_1}^{k_2-1} \delta_i^x(\text{E}_{l,k}^z, \text{E}_{h,k+1}^y) \tag{15}$$

*where* $\delta_i^x(\text{E}_{l,k}^z, \text{E}_{h,k+1}^y)$ *is task* $\tau_i$*'s cumulative execution time in* $c^x$ *between two consecutive events,* $\text{E}_{l,k}^z$ *and* $\text{E}_{h,k+1}^y$

$$\delta_i^x(\text{E}_{l,k}^z, \text{E}_{h,k+1}^y) = \begin{cases} \delta_i(\text{E}_{l,k}^z, \text{E}_{h,k+1}^y) & \textit{if } F^x(\text{E}_{l,k}^z) \\ 0 & \textit{otherwise} \end{cases} \tag{16}$$

where $F^x(\text{E}_{l,k}^z) = (\forall_{c^v \in C, \tau_j \in T}\ c^z = c^x \wedge \Phi(\text{E}_{l,k}^z) \in \{\text{ESINV}_l^{v \to z}, \text{ECRET}_l^{z \leftarrow v}, \text{EINT}_{j \to l}^{v \to z}, \text{ECS}_{j \to l}^z\})$

## A. Constraints for Timing Fault Detection

The following are a set of constraints that must hold at all points in time for the system's dynamic execution to adhere to the model that underlies the analysis of the system.

• *Bounded number of invocations.* $\forall c^x, c^y \in C, \tau_i \in T, j \geq 0, t = r_{i,j}, k > 0$,

$$S_{inv} = \{\text{ECINV}_{i,k}^{x \to y} | @(\text{ECINV}_{i,k}^{x \to y}) \geq t \wedge @(\text{ECINV}_{i,k}^{x \to y}) < t + p_i\}$$

$$|S_{inv}| \leq n_{i,ipc}^{x \to y} \tag{17}$$

• *Bounded number of interrupts that interfere with a task.* Assume $\tau_i$ is a thread that activates in response to an interrupt in $c^y$. $\forall c^x \in C, \tau_l, \tau_h \in T, j \geq 0, t = r_{l,j}, k > 0$,

$$S_{int} = \{\text{EINT}_{h \to i,k}^{x \to y} | @(\text{EINT}_{h \to i,k}^{x \to y}) \geq t \wedge @(\text{EINT}_{h \to i,k}^{x \to y}) < t + p_l\}$$

$$|S_{int}| \leq n_{i,int}^y \tag{18}$$

• *Task (and interrupt) execution bounds.* $\forall \tau_i \in T, k_1, k_2 > 0, j \geq 0$, where $@\text{E}_{k_1} = r_{i,j}$ and $@\text{E}_{k_2} = r_{i,j} + p_i$,

$$\Delta_i(\text{E}_{k_1}, \text{E}_{k_2}) \leq e_i \tag{19}$$

• *Component execution bounds.* $\forall k_1, k_2, c^x, c^y \in C$ where $\Phi(\text{E}_{k_1}) = \text{ESINV}_i^{x \to y}$ and $\text{E}_{k_2}$ is the next closest event such that $\Phi(\text{E}_{k_2}) = \text{ESRET}_i^{x \leftarrow y}$ and $\forall \tau_i \in T$,

$$\Delta_i^y(\text{E}_{k_1}, \text{E}_{k_2}) \leq w^y \tag{20}$$

**Discussion.** These constraints, in sum, ensure that threads and interrupts do not execute in a manner that would invalidate the system timing analysis based on the system model. To validate that the system has proper timing behavior, the behavior of the scheduler must also be validated.

**Validating scheduler behavior.** We assume a fixed-priority preemptive scheduling policy, and predictable resource sharing using the Priority Inheritance Policy (PIP) [51]. We choose PIP over priority ceiling as it is the more complex policy to validate. The scheduler and synchronization components exhibit correct behavior if the highest priority thread amongst those active is always chosen to execute at all points in time, *unless* resource sharing prevents high-priority thread from execution. In that case, the extent of the priority inversion due to this sharing must be bounded as defined by the system model. Here we introduce the constraints the policy must adhere to.

C'MON detects contention on shared resources in a component $c^x$ when it sees an event $\text{ECINV}_{i,k_1}^{x \to l}$ (recall that $c^l$ is the lock component). This interference finishes with the closest occurrence of $\text{ECRET}_{i,k_2}^{x \leftarrow l}$. The maximum allowed observed execution in $\tau_i$ is constrained by the resource block time. For all $k_1$ and $k_2$, as defined above,

$$b^x \geq \sum_{\forall \tau_j \in lp_i} \Delta_j(\text{ECINV}_{i,k_1}^{x \to l}, \text{ECRET}_{i,k_2}^{x \leftarrow l}) \tag{21}$$

Thus, C'MON tracks all scheduling decisions (context switches, and interrupt activations), and keeps a running total of this priority inversion, checking to make sure it is within this bound.