A Synchronous IPC Protocol for Predictable Access to Shared Resources in Mixed-Criticality Systems

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—In mixed-criticality systems, highly critical tasks must be temporally and logically isolated from faults in lower-criticality tasks. Such strict isolation, however, is difficult to ensure even for independent tasks, and has not yet been attained if low- and high-criticality tasks share resources subject to mutual exclusion constraints (*e.g.*, shared data structures, peripheral I/O devices, or OS services), as it is often the case in practical systems.**

Taking a pragmatic, systems-oriented point of view, this paper argues that traditional real-time locking approaches are unsuitable in a mixed-criticality context: locking is a cooperative activity and requires trust, which is inherently in conflict with the paramount isolation requirements. Instead, a solution based on resource servers (in the microkernel sense) is proposed, and MC-IPC, a novel synchronous multiprocessor IPC protocol for invoking such servers, is presented.

The MC-IPC protocol enables strict temporal and logical isolation among mutually untrusted tasks and thus can be used to share resources among tasks of different criticalities. It is shown to be practically viable with a prototype implementation in LITMUS$^{\mathrm{RT}}$ and validated with a case study involving several antagonistic failure modes. Finally, MC-IPC is shown to offer analytical benefits in the context of Vestal's mixed-criticality task model.

I. Introduction

In a mixed-criticality system [11, 30], real-time tasks of different *criticalities* share a common hardware platform. Intuitively, a task's criticality expresses its "importance" to the survival or correct operation of the system. For instance, in an unmanned aerial vehicle (UAV), tasks related to maintaining stable flight conditions are undoubtedly more critical than mission-related functionality such as long-term path finding. In a certification context, this intuition is reflected by the fact that lower-criticality tasks are subject to less stringent (and less costly) certification requirements than higher-criticality tasks.

A commonsense requirement for such mixed-criticality systems is that high-criticality tasks must be isolated from faults such as software defects or other anomalous behavior in tasks of lower criticality. Besides the intuitive observation that faults in "less important" subsystems should not cause crucial functionality to fail, economic incentives also favor strict isolation. For example, a typical certification requirement is that (otherwise) low-criticality tasks must be certified at the level of assurance of the highest-criticality tasks they can interfere with (*e.g.*, see the ISO 26262 and DO-178B standards). It is thus essential that the employed real-time operating system (RTOS) ensures *freedom from interference*—that is, strict *temporal* and *logical isolation*.

Unfortunately, ensuring such strict isolation can be challenging in practice. While solutions for isolating and analyzing *independent* tasks are readily available (*e.g.*, see [3, 11, 15, 24]), the problem of isolating tasks that (must) share resources (*e.g.*, such as I/O devices, drivers, or OS services) has remained largely open to date [11]. Prior work on real-time resource sharing has focused mainly on *lock-based* and *non-blocking synchronization* techniques, which we deem unsuitable for use in mixed-criticality systems, as is argued in more detail in Sec. II-C.

A much more promising approach is to encapsulate shared resources in *resource servers*, as it is commonly done in microkernel OSs. Such servers protect shared resources from direct access, and tasks seeking to use the encapsulated resource must instead invoke the server via a synchronous *inter-process communication* (IPC) protocol. Resource servers provide logical isolation, but by themselves are insufficient in a mixed-criticality context—in addition, a *predictable* IPC protocol *resilient* to task failures is needed to ensure strict temporal isolation even if tasks of different criticalities contend for the same server.

The main contribution of this paper is MC-IPC, the first such protocol. Crucially, the MC-IPC protocol enforces strict logical and temporal isolation among untrusted tasks even

- if tasks attempt to monopolize a resource,
- if resource-sharing tasks malfunction in arbitrary ways, and
- if the identity, number, and criticality of tasks sharing a resource is unknown or untrusted at design time.

Consequently, when certifying high-criticality tasks, only the resource access layer (*i.e.*, the IPC primitives and the resource servers) must be certified at the high assurance level, but *not* all other tasks that may use the same resources as high-criticality tasks. In fact, under the MC-IPC protocol, even non-critical, non-real-time, best-effort tasks may share resources with the highest-criticality real-time tasks, as cooperative and timely behavior is neither assumed nor required.

The MC-IPC protocol, defined in Sec. IV, is flexible: it can be applied on multiprocessors under clustered scheduling with non-uniform clusters (Sec. III) and supports both dynamic event-driven and static table-driven scheduling (Secs. III and V). Furthermore, it is practical: we have implemented a prototype (Sec. V) in LITMUS$^{\mathrm{RT}}$ [2], and have validated the prototype experimentally with a case study on an Intel multicore platform (Sec. VI). Finally, the MC-IPC protocol can also be integrated with Vestal's mixed-criticality task model [30] to allow for reduced pessimism when analyzing the worst-case IPC delays experienced by lower-criticality tasks (Sec. VII).

To the best of our knowledge, this work is the first to consider resource sharing in *multiprocessor* mixed-criticality systems (see Sec. VIII for related work), and the first to experimentally validate a mixed-criticality resource-sharing approach in a real OS. Additionally, no prior approach to multiprocessor real-time synchronization ensures temporal and logical isolation in the presence of an unknown number of untrusted tasks.

We begin with a discussion of how interference arises and how it may be mitigated, from which we derive the requirements for our solution, which we describe in Sec. IV.

II. Requirements, Risks, and Mitigation Techniques

Many competing definitions of "mixed-criticality" workloads have been offered in recent years (see [11] for a comprehensive survey). Originally formalized by Vestal [30] as a means to reclaim, at design time, system capacity lost to the widespread pessimism inherent at the highest criticality levels, sophisticated task models that explicitly incorporate a notion of criticality have seen a rapid proliferation and evolution [11].

In contrast, our focus in this work is of a more pragmatic, systems-oriented nature. Rather than targeting a specific analytical model, we seek to identify and mitigate the risks that arise when mutually untrusted tasks of varying criticalities share a common hardware platform, as such risks must be mitigated at runtime regardless of the analysis employed at design time.

A. The Need for Isolation

It stands to reason that, as a lowest common denominator, all reasonable definitions of a mixed-criticality environment mandate or assume *isolation* as a key requirement: if low-criticality tasks *could* potentially affect the logical or temporal correctness of higher-criticality tasks, then it must be carefully established that they in fact *do not*, at the level of assurance of the tasks at risk. That is, a lack of isolation at the OS level translates into an increased certification burden, which runs counter to the very idea of mixed-criticality systems—to avoid the need to uniformly apply the highest standards to all tasks.

In addition to shielding high- from low-criticality tasks, it can also be desirable to isolate tasks of equal criticality from each other. In fact, logically isolating all tasks from each other regardless of criticality already is a universally accepted best practice, as witnessed by the pervasive use of process-based isolation even in non-critical systems.

Similarly, temporally isolating tasks of equal criticality has significant benefits. For one, increased compartmentalization and isolation generally benefits resilience. Moreover, isolated subsystems are easier to reason about (and hence certify) than subsystems that are exposed to potential sources of interference, regardless of whether the interference stems from sources of higher, equal, or lower criticality. And finally, it is simply a sound engineering principle to *minimize trust* among safety-critical components whenever possible.

We therefore consider isolation, or freedom from interference, to be the central requirement of mixed-criticality systems (as also previously argued by Petters *et al.* [27], among others). In the following, we examine how interference may arise and how it can be prevented with existing techniques, both to extract a realistic system model and to identify where these techniques fail when sharing resources. We first consider independent tasks.

B. State of the Art: Isolating Independent Tasks

OS-based isolation techniques seek to prevent failures that arise from co-locating untrusted tasks on a shared platform, that is, failures that cannot arise if tasks execute in physical isolation on dedicated systems.

Logical isolation is violated if a task can cause other tasks to malfunction (*e.g.*, to crash or to hang) by corrupting their state. As already pointed out, logical isolation of independent tasks can be readily ensured by placing each task in a separate *protection domain* (*i.e.*, address space or segment). We thus assume that all tasks are encapsulated in this fashion.

Temporal isolation is violated if the execution of one task delays another task in ways that were not anticipated *a priori*, or if the expected magnitude of such delays is exceeded.

Delays fundamentally arise because of resource contention, which may arise at three principal levels. All tasks are affected by **(i)** implicit contention for micro-architectural resources (*e.g.*, processor caches, memory bandwidth, *etc.*) and by **(ii)** explicit contention for preemptable resources (*i.e.*, primarily processor time). Further, tasks that are not independent are also subject to **(iii)** explicit contention for non-preemptable, serially-reusable resources such as peripheral devices (I/O ports, sensors, *etc.*), OS services (*e.g.*, drivers, file systems, the network stack), or shared data structures in general.

Concerning (i), implicit contention for micro-architectural resources is a serious concern, and especially so on multicore platforms, but it is beyond the scope of this paper. Nonetheless, we note that the OS can lessen the impact of such contention on commodity hardware using techniques such as page coloring [23, 31], performance-counter-based rate-limiting [32, 33], and interference-aware scheduling approaches [13, 18, 26]. We assume that tasks are isolated from implicit contention, either by the hardware itself or by using such mitigation techniques.

Contention for processor time is the most-studied resource conflict in real-time systems; numerous suitable schedulers exist, including many specifically designed for mixed-criticality systems [11]. However, no matter the policy, tasks cannot be trusted to stay below their specified *worst-case execution time* (WCET), which must be policed instead. The standard mitigation technique is to employ *reservation-based scheduling* [24], where each task is encapsulated in a reservation with a pre-determined budget. Once the budget is exhausted, the task is prevented from executing until the reservation's budget is replenished according to the rules of a reservation algorithm (see Sec. III). We adopt reservation-based scheduling for our solution as it is effective and easy to implement, and assume familiarity with the concept on behalf of the reader.

To summarize, *independent* tasks can be adequately isolated using existing, well-established techniques. Tasks that explicitly share resources subject to mutual exclusion requirements, however, are substantially more difficult to isolate.

C. The Challenge: Isolating Resource-Sharing Tasks

Mutual exclusion is typically implemented using either spin locks or semaphores, and both types of locks have received significant attention in recent years (*e.g.*, see [6, 7] for recent surveys). Unfortunately, locks are fundamentally inappropriate in the context of mixed-criticality systems, as locks imply trust. Specifically, when tasks use locks to coordinate access to a shared resource, they (implicitly) trust that **(i)** no task accesses the shared resource without first acquiring the lock; **(ii)** every task ensures that the resource is in a consistent state when the lock is released; and **(iii)** all tasks release the lock in a timely manner (*e.g.*, tasks must not "forget" to unlock the resource).

If (i) and (ii) are not met, then logical isolation cannot be guaranteed. Temporal isolation is at risk if (iii) is violated. Non-blocking approaches such as wait-free or lock-free data structures do not depend on assumptions (i)–(iii), but still require trusting that the consistency of a shared data structure is respected and maintained by all sharing tasks. In short, when using locks or non-blocking data structures, no isolation guarantees can be made if individual tasks fail to cooperate.

However, cooperation cannot be assumed if resources are shared among untrusted tasks. Instead, as mentioned in Sec. I, access to shared resources must be fully mediated by trusted resource servers. Rather than accessing a shared resource's state directly (as with regular locks), tasks instead invoke the server with a synchronous IPC protocol (*i.e.*, tasks are blocked while awaiting the server's reply) to carry out operations on their behalf. Provided the server implements each supported operation correctly (*i.e.*, according to its specification, while rejecting ill-formed, non-sensical, or unauthorized requests), this approach immediately restores logical isolation: only the server must be trusted, but not other tasks accessing the same server.

This, however, leaves one pressing open question: how can *temporal* isolation be ensured when invoking a resource server? That is, how can it be ensured by the OS that a task will actually receive a reply within a predictable time frame if the maximum number of tasks invoking the same server is untrusted or simply unknown in advance? For example, central OS resources such as the network stack or device drivers are likely to be shared among many tasks of different criticalities—some resources may even be required by non-real-time background tasks. *Predictable* sharing of such resources requires a synchronous IPC protocol that ensures strict temporal isolation.

Prior work has focused on priority- and FIFO-ordered wait queues (*e.g.*, TU Dresden's Fiasco.OC microkernel [19] uses priority queues to order IPC requests, and the *Multiprocessor Bandwidth Inheritance Protocol* (MBWI) [12], which can be easily adapted into an IPC protocol, uses FIFO queues). Unfortunately, both queue types require strong trust assumptions. A FIFO queue is obviously problematic if the number of tasks is unknown, and while priority queues isolate a task against interference from best-effort and lower-priority tasks, they also allow for starvation: unpredictable delays arise if two or more higher-priority tasks saturate a server. In particular, note that "higher priority" does not necessarily imply "higher criticality" since low-criticality tasks may have tighter timing constraints than high-criticality tasks [5]. Thus, on multiprocessors, strict temporal isolation cannot be achieved with either approach.

To conclude, commonsense isolation requirements strongly suggest that mixed-criticality workloads should be deployed in reservation-based environments in which all resources are encapsulated in resource servers, but to date there does not exist a method to invoke such servers without trusting the runtime environment. In this paper, we remove these trust assumptions.

III. System Model and Assumptions

In accordance with the requirements analysis presented in the preceding section, we formalize the system model as follows. To avoid needlessly restricting generality, these definitions are chosen to be as broad as possible. A concrete implementation with specific choices is presented in Sec. V.

The system consists of m identical processors that are organized into K disjoint *clusters*, denoted $C_1, \ldots, C_K$, of sizes $m_1, \ldots, m_K$, respectively, where $m = \sum_k m_k$. We assume *clustered* scheduling in the model even though our implementation and many systems in practice are based on *partitioned* scheduling (where $K = m$ and $m_k = 1$ for each C_k) because the MC-IPC protocol easily generalizes to $m_k > 1$.

The main schedulable entities are n *reservations* $R_1, \ldots, R_n$. Associated with each reservation R_j is a *current budget* B_j and a *current priority* Y_j. We assume that priorities are unique (*i.e.*, any ties in priority are broken arbitrarily but consistently).

Each R_i is statically assigned to one of the clusters; we let $C(R_j)$ denote the cluster to which R_j is assigned. A reservation contains one or more *sporadic tasks*. We let e_i denote the WCET and p_i the period of task T_i; however, the task parameters and the number of tasks are not necessarily known in advance. A reservation is *active* if (one of) the contained tasks has a *pending* (*i.e.*, incomplete) job, and *inactive* otherwise. An incomplete job is considered pending regardless of whether it is ready to execute, waiting for IPC, or suspended for other reasons.

In each cluster C_k, there is a *top-level scheduler* that, at each point in time t, selects the (up to) m_k highest-priority reservations with non-zero budget (as determined by each reservation's Y_j and B_j) active at time t. A reservation R_j that is selected for scheduling is assigned to a processor by the top-level scheduler and must then dispatch (one of) its client(s).

There are further n_r serially-reusable shared resources $\ell_1, \ldots, \ell_{n_r}$. Each such resource ℓ_q is encapsulated in a corresponding *resource server* S_q that may be *invoked* by potentially any task in the system. (Access control is an orthogonal issue beyond the scope of this paper.) Each server supports a resource-specific set of operations, is sequential, and serves one request at a time. We let L_q denote the *maximum operation length* of server S_q, that is, L_q bounds the maximum amount of budget that S_q consumes to satisfy any single request, accounting both for its own execution and any self-suspensions.

Servers benefit from *bandwidth inheritance* [12, 21, 29]: while a task T_i in a reservation R_j is waiting for a server S_q to reply, S_q is dispatched using R_j's budget whenever T_i would have been dispatched (*i.e.*, when R_j is selected by the top-level scheduler and R_j in turn attempts to dispatch T_i), unless S_q is suspended or already executing on another processor.

In principle, a server could invoke other servers as part of handling a request (*i.e.*, requests could be nested), in which case bandwidth inheritance must take effect transitively. However, the analysis presented in Sec. IV does not yet extend to the nested case; we therefore require in this paper that servers do not invoke other servers and leave the nested case to future work.

In the interest of generality, the employed scheduling policy—how a reservation R_j's current priority Y_j is determined—and reservation rules—how and when R_j's current budget B_j is replenished—are intentionally left underspecified. The MC-IPC protocol's analytical guarantees (Sec. IV-B) depend on only the following two assumptions:

A1 an active reservation's current priority Y_j changes only when its budget is exhausted or replenished; and

A2 an active reservation's current budget B_j drains at unit speed whenever the reservation is selected for service by the top-level scheduler, regardless of whether it has a ready task or server to dispatch.

Note that Assumption A2 covers both the regular execution of contained tasks as well as bandwidth inheritance. Further, if an active reservation is selected by the top-level scheduler and none of its client tasks are ready (*i.e.*, there are pending jobs, but they are all waiting for IPC or are suspended) and no server is inheriting its budget, then the reservation *idles*: it consumes budget at unit speed without dispatching a task or server, and background tasks or tasks from other reservations may be dispatched instead as a form of slack reclamation.

We make no further assumptions about the specific type of reservations employed, but note that these assumptions are consistent with (for example) *constant-bandwidth servers*[1] [3] and table-driven scheduling. In particular, each Y_j may be either static, determined by EDF, or set by any other prioritization rule.

As a concrete example, our prototype (discussed in Sec. V) supports a combination of table-driven scheduling and EDF-based sporadic reservations. A *table-driven reservation*, also called a *time partition*, is simply a set of fixed-length intervals in a cyclicly repeating static schedule. Given a *cycle time* (or schedule length) H, a table-driven reservation R_t is defined by a set of r disjoint intervals (or *slots*) $\{[t_1, t_2), \ldots, [t_{2r-1}, t_{2r})\}$, where $0 \leq t_1 < t_2 < \ldots t_{2r-1} < t_{2r} \leq H$. Conceptually, R_t's budget is replenished to $B_t = t_{2x} - t_{2x-1}$ at each point in time $t = l \cdot H + t_{2x-1}$ for $l \in \{0, 1, 2, \ldots\}$ and $1 \leq x \leq r$. A table-driven reservation has a fixed priority Y_t higher than any sporadic reservations. The slots of different table-driven reservations assigned to the same processor must not overlap.

A *sporadic reservation* [24] is simply a sporadic task subject to *budget* and *period enforcement*. Suppose such a reservation R_j encapsulates a sporadic task T_i with an untrusted WCET e_i and period p_i, which are to be policed. If R_j is inactive at time t and T_i releases a job J_i, then B_j is set to e_j and, under EDF, Y_j is set to $t + p_i$. If J_i completes before B_j is exhausted, R_j discards the remaining budget. After t, B_j is not replenished again until time $t + p_i$, which prevents undue interference.

With these definitions in place, we next introduce the MC-IPC protocol and establish its analytical properties.

IV. The MC-IPC Protocol

The synchronous mixed-criticality IPC protocol described in this section is simple in structure, consisting mainly of a few queues (*i.e.*, linked lists), which is desirable as it is intended to be implemented in the OS kernel and because IPC performance is generally performance-critical in microkernels.

The MC-IPC protocol achieves resilience against fluctuations in the number of clients or attempts to monopolize a server by channeling all IPC requests through a three-level, multi-ended queue. First, contention is resolved within each cluster using a priority and a FIFO queue, and then contention among clusters is resolved using a final global FIFO queue.[2]

Additionally, requests of tasks that exhaust their reservation's budget while queued are immediately pruned (and must be reissued when the budget is replenished). To avoid interference from best-effort background tasks, such tasks are queued in a separate queue (of arbitrary order) that is served only if no real-time tasks are present. We provide a precise definition next.

A. Protocol Definition

A pseudo-code definition of the MC-IPC protocol is given in Fig. 1. The pseudo-code and the following discussion assume that there is a single task per reservation. If a reservation contains more than one task, then at most one of it may invoke S_q at any time (*i.e.*, reservation-internal contention should be resolved before it reaches S_q's IPC gate). However, this restriction can be omitted in the common case of single-core clusters (*i.e.*, if $m_k = 1$ as under partitioned scheduling).

For each resource server S_q, there exists an IPC gate that supports four principal operations: mc_invoke, called by clients to submit a request to the server, mc_wait, called by the server to receive a new request, mc_reply, called by the server to report the result of a finished request, and finally mc_abort, which is called by the top-level scheduler when the reservation of a waiting task exhausts its current budget. For the sake of simplicity, explicit synchronization has been omitted from Fig. 1; each of these operations is assumed to be atomic.

Each IPC gate consists of $2K + 2$ queues, $K + 1$ task pointers, and K flags. We explain their purpose in the context of the operations referencing them, beginning with mc_invoke.

1) mc_invoke: When a task T_i in a reservation R_j located in cluster C_k invokes the server S_q, the task is first enqueued into the appropriate queue (lines 8–18), then the server is allowed to consume R_j's budget (line 19), and finally the task is suspended to await the server's reply (line 20).

Into which queue T_i is enqueued depends on its type (*i.e.*, real-time or best-effort) and on the number of currently contending tasks. Best-effort tasks are simply enqueued into a global queue called $background\_queue$ (line 9), which can be of any order (*e.g.*, a FIFO queue is assumed here for simplicity).

Real-time tasks are inserted into one of three queues, depending on the current contention in C_k. By design, it is irrelevant how many tasks in clusters other than C_k are contending for S_q.

In the best case, T_i is the only task in C_k invoking S_q, in which case it immediately may become the $local\_head[k]$ in C_k (lines 11–12). The task referenced by $local\_head[k]$ is the only task in C_k that may contend on a global level. To do so, T_i is inserted into the $global\_queue$, unless the flag $local\_wait[k]$ has been set to block it (lines 13–14). The purpose of $local\_wait[k]$ is to ensure that at most one task from each cluster causes contention at the global level at any time; it is set when best-effort tasks are served (line 27) or when a reservation's budget is exhausted (line 56), as explained below.

[1]The term "server" is unfortunately overloaded in the real-time and OS literature, as it is used for both reservation algorithms and resource servers. To avoid ambiguity, we use the term "reservation" to refer to the scheduling abstraction, unless when referring to well-established algorithms.

[2]Readers familiar with suspensions-oblivious real-time locking protocols will recognize a similarity with the $O(m)$ *Independence-Preserving Locking Protocol* (OMIP) [7]; see Sec. VIII for a discussion of related work.

```
1   struct MC_IPC_GATE: // one for each server $S_q$
2       task_ref_t currently_served, local_head[$K$]
3       fifo_queue_t global_queue, background_queue, head_queue[$K$]
4       prio_queue_t tail_queue[$K$]
5       bool_t local_wait[$K$] // initially false

7   mc_invoke($T_i$, $R_j$, $C_k$): // called by the client
8       if is_background_task($T_i$):
9           add_tail($T_i$, background_queue)
10      else:
11          if local_head[$k$] = $null$:
12              local_head[$k$] $\leftarrow T_i$
13              if $\neg$local_wait[$k$]:
14                  add_tail($T_i$, global_queue)
15          else if length(head_queue[$k$]) $< m_k - 1$:
16              add_tail($T_i$, head_queue[$k$])
17          else:
18              add_sorted($T_i$, tail_queue[$k$], with_priority=$Y_j$)
19      setup_bandwidth_inheritance($S_q$, $R_j$)
20      notify_server_and_await_reply($S_q$)

22  mc_wait(request_buffer): // called by the server $S_q$
23      await $\neg$empty(background_queue) $\vee$ $\neg$empty(global_queue)
24      if empty(global_queue):
25          currently_served $\leftarrow$ dequeue_head(background_queue)
26          let $C_k$ = cluster_of(currently_served)
27          local_wait[$k$] $\leftarrow$ $true$
28      else:
29          currently_served $\leftarrow$ dequeue_head(global_queue)
30      receive_request_from(currently_served, request_buffer)

32  mc_reply(reply_buffer): // called by the server $S_q$
33      let $T_i$ = currently_served, $R_j$ = reservation_of($T_i$)
34      let $C_k$ = cluster_of($R_j$)
35      currently_served $\leftarrow$ $null$, local_wait[$k$] $\leftarrow$ $false$
36      if local_head[$k$] = $T_i$:
37          move_head_to_tail(tail_queue[$k$], head_queue[$k$])
38          local_head[$k$] $\leftarrow$ dequeue_head_or_null(head_queue[$k$])
39      if local_head[$k$] $\neq$ $null$:
40          add_tail(local_head[$k$], global_queue)
41      stop_bandwidth_inheritance($S_q$, $R_j$)
42      send_reply_and_notify_client($T_i$, reply_buffer)

44  mc_abort($T_i$, $R_j$, $C_k$): // called by the top−level scheduler
45      if $T_i \in$ tail_queue[$k$]:
46          remove($T_i$, tail_queue[$k$])
47          stop_bandwidth_inheritance($S_q$, $R_j$)
48      else if $T_i \in$ head_queue[$k$]:
49          remove($T_i$, head_queue[$k$])
50          move_head_to_tail(tail_queue[$k$], head_queue[$k$])
51          stop_bandwidth_inheritance($S_q$, $R_j$)
52      else:
53          if $T_i \in$ gobal_queue: // is $T_i$ still waiting?
54              remove($T_i$, global_queue)
55          else:
56              local_wait[$k$] $\leftarrow$ $true$
57          if local_head[$k$] = $T_i$:
58              local_head[$k$] $\leftarrow$ dequeue_head_or_null(head_queue[$k$])
59              move_head_to_tail(tail_queue[$k$], head_queue[$k$])
```

Fig. 1. Pseudo-code definition of the MC-IPC protocol

If at most m_k tasks from C_k are waiting (including T_i), then T_i may directly enter $head\_queue[k]$, the FIFO queue in C_k (lines 15–16). Due to the check in line 15, $head\_queue[k]$ contains at most $m_k - 1$ tasks. Once a task has entered $head\_queue[k]$, it cannot be delayed by later-issued requests in its cluster due to the strong progress properties of FIFO queues. This is exploited in Lem. 1 in Sec. IV-B.

Finally, if more than m_k tasks from C_k are waiting for the server, then T_i is enqueued into $tail\_queue[k]$, the priority queue in cluster C_k (lines 17–18). As the name implies, tasks in this queue are furthest removed from being served. Over time, tasks in $tail\_queue[k]$ are propagated to $head\_queue[k]$ in order of decreasing priority as mc_reply or mc_abort remove tasks from $head\_queue[k]$. This fact forms the basis of Lem. 2.

2) mc_wait: When the server is ready to service a request, it simply waits for either a background or real-time task to be enqueued (line 23). A background task will be served only if no real-time tasks are present (lines 24–27). Otherwise, S_q simply serves the first task in $global\_queue$ (line 29), which ensures that all clusters are served in a round-robin manner. Importantly, if a background task is selected for service, the flag $local\_wait[k]$ in the client's cluster is set (line 27) to prevent a later-arriving real-time task to directly enter the $global\_queue$ (recall line 13). The selected task is stored in $currently\_served$.

3) mc_reply: When a request has been carried out, the server clears $currently\_served$ and $local\_wait[k]$ (line 35). Unless $local\_head[k]$ has already been updated (by mc_abort, see below) in the mean time (line 36), S_q moves the task in $tail\_queue[k]$ stemming from the highest-priority reservation (if any) to the end of $head\_queue[k]$ (line 37) and promotes the head of $head\_queue[k]$ (if any) to $local\_head[k]$. The new $local\_head[k]$ (if any) is added to $global\_queue$ (line 40). Finally, before resuming the task T_i that issued the just-finished request (line 42), S_q becomes ineligible to consume the budget of T_i's reservation R_j (line 41).

4) mc_abort: Finally, the correct action to take when a request must be pruned depends on how far the task has already progressed in the queue. It can simply be removed from $tail\_queue[k]$ (lines 45–47). If it is removed from $head\_queue[k]$ instead, it may be necessary to propagate a task from $tail\_queue[k]$ (lines 48–51). This is also necessary if the task is removed from $global\_queue$, in which case it is also necessary to select a new $local\_head[k]$ (lines 53–59). If the to-be-pruned task is already being served, then it can no longer be aborted and pruned, and instead the flag $local\_wait[k]$ is set to block subsequent tasks in the same cluster from entering $global\_queue$. This added delay is accounted for in Lem. 3.

These operations are simple in structure and can be easily implemented in an OS kernel. Importantly, the queue sizes, their structure, and propagation rules are carefully chosen to enable the main analytical property of MC-IPC: the maximum amount of budget expended by R_j depends only on m_k, K, and L_q, and is independent of n or the rate of requests, as we establish next.

B. Bounding the Maximum Budget Consumption

In the following analysis, we denote the requesting task as T_i, T_i's reservation as R_j, and the cluster to which R_j is assigned as C_k. For simplicity, we further assume that T_i is the only client of R_j, and that R_j does *not* exhaust its budget during T_i's request—tasks that are pruned due to budget exhaustion lose all progress and must reissue their request, which resets the analysis.

We trace R_j's budget consumption as T_i progresses through the IPC gate's queue structure. To this end,

- let t_1 denote the time at which T_i invokes S_q,
- let t_2 denote the first point in time (on or after t_1) at which $local\_wait[k] = false$, where $t_2 \geq t_1$,

- let t_3 denote the time at which T_i enters $head\_queue[k]$, where $t_3 \geq t_1$,
- define t_3' as $t_3' \triangleq \max(t_2, t_3)$, and
- let t_4 denote the time that T_i's request is completed.

We initially make the simplifying assumption that no task contending for S_q exhausts its budget and begin with $[t_3', t_4)$.

Lemma 1. *If no task contending for S_q exhausts its budget (i.e., if $local\_wait[k] = false$) while T_i is waiting, then R_j drains at most $m_k \cdot K \cdot L_q$ budget during $[t_3', t_4)$.*

Proof: Since $local\_wait[k] = false$ during $[t_3', t_4)$, and due to the strong progress guarantee of the two FIFO queues $head\_queue[k]$ and $global\_queue$ that T_i traverses during $[t_3', t_4)$, at most $m_k \cdot K - 1$ requests can precede T_i's request after time t_3'. Since S_q makes progress when R_j expends budget (ensured by bandwidth inheritance [12]), T_i's request has been served after R_j has expended at most $m_k \cdot K \cdot L_q$ budget. ■

Next, we bound the budget drain while T_i is in $tail\_queue[k]$.

Lemma 2. *If no task contending for S_q exhausts its budget (i.e., if $local\_wait[k] = false$) while T_i is waiting, then R_j drains at most $m_k \cdot K \cdot L_q$ budget during $[t_2, t_3')$.*

Proof: By contradiction. If $t_2 = t_3'$ the claim follows trivially, so assume $t_2 < t_3'$. Suppose there exists a time $t_b \in [t_2, t_3')$ such that at least $m_k \cdot K \cdot L_q$ budget has been used by R_j during $[t_2, t_b)$ and R_j is draining budget at time t_b.

Since $m_k \cdot K \cdot L_q$ budget was consumed during $[t_2, t_b)$ and since S_q makes progress whenever R_j drains budget (due to bandwidth inheritance), at least $m_k \cdot K$ requests were completed during that time. Further, since $local\_wait[k] = false$ throughout the interval, this implies that none of the tasks enqueued at time t_2 still remains in $head\_queue[k]$. Since T_i still resides in $tail\_queue[k]$ at time t_b (by the definition of time t_3), this implies that the reservations of all tasks enqueued in $head\_queue[k]$ at time t_b, and the reservation of the task referenced by $local\_head[k]$, had a higher current priority than R_j when they entered $head\_queue[k]$. And by Assumption A1, their priority has not changed. Since T_i is still in $tail\_queue[k]$, there are m_k higher-priority active reservations in cluster C_k at time t_b. Hence the top-level scheduler did not select R_j for scheduling at time t_b, which thus also does not drain budget at time t_b (Assumption A2). Contradiction. ■

Finally, we consider the initial interval during which $local\_wait[k] = true$ (if at all), thereby blocking tasks in cluster C_k from entering $global\_queue$.

Lemma 3. *R_j drains at most L_q budget during $[t_1, t_2)$.*

Proof: Let $T_x = currently\_served$ denote the task being served at time t_1. If $local\_wait[k] = false$ at time t_1, the claim follows trivially. If $t_1 < t_2$, then T_x is either a background task or it exhausted its reservation's budget. Since S_q makes progress whenever T_i's reservation expends budget (due to bandwidth inheritance), T_x's request is completed after R_j consumed at most L_q, at which point $local\_wait[k]$ is reset. ■

Finally, we observe that the MC-IPC protocol shields T_i from other tasks that exhaust their reservation's budget.

Lemma 4. *R_j's total budget consumption does not increase if another task T_x exhausts its budget while T_i is waiting.*

Proof: If T_x is pruned from the IPC gate's queues, then T_i's relative position either remains unchanged (if T_x was logically behind T_i) or it advances towards becoming the currently served task, which obviously does not increase R_j's budget consumption. If T_x cannot be pruned because its request is already being served, then it preceded T_i's request and would have contributed towards T_i's budget consumption even if T_x had not exhausted its reservation's budget mid-request: due to idling (Assumption A2), R_j's reservation drains budget even while S_q is inheriting budget from T_x's reservation. ■

This allows us to state the main MC-IPC theorem.

Theorem 1. *R_j consumes at most $(1 + 2m_k \cdot K) \cdot L_q$ units of budget during $[t_1, t_4)$ (i.e., during a single invocation of S_q).*

Proof: By Lems. 1–3, R_j consumes at most L_q, $m_k \cdot K \cdot L_q$, and $m_k \cdot K \cdot L_q$ units of budget during $[t_1, t_2)$, $[t_2, t_3')$, and $[t_3', t_4)$, respectively, assuming no budget exhaustion occurs while T_i is waiting. By Lem. 4, R_j's budget consumption does not increase if other tasks exhaust their budget while T_i is waiting. Hence R_j drains at most $(1 + 2m_k \cdot K) \cdot L_q$ budget during T_i's invocation of S_q, regardless of the requests or other actions of any tasks in any other reservation. ■

C. Discussion

Theorem 1 provides a strong isolation property: it allows the budget of critical real-time tasks to be dimensioned such that no deadline will be missed, even when sharing resources with untrusted tasks. Suppose task T_i is isolated in a sporadic reservation R_j located in cluster C_k with a replenishment period p_i, T_i's true WCET is e_i, and further suppose that any of its jobs invokes each S_q at most $N_{i,q}$ times. Then, assuming the set of admitted reservations in C_k is schedulable (*i.e.*, R_j can always consume its entire budget by its deadline), no job of T_i will miss its deadline provided T_i's reservation R_j has a budget of at least $e_i + \sum_{q=1}^{n_r} N_{i,q} \cdot (1 + 2m_k \cdot K) \cdot L_q$. Note that the required budget is independent of the runtime behavior of any other task (*i.e.*, which resources it accesses, and how often), and also of the total number of tasks or reservations. MC-IPC thus enables strict temporal and logical isolation despite resource sharing.

An important point to clarify is that the MC-IPC protocol does not abort a request once it's being serviced (as aborting an in-process request may be impossible for certain resources such as I/O devices). The parameter L_q is hence *not* a budget, but rather a bound on the true maximum operation length (analogous to a task's WCET). We discuss in Sec. VII how different L_q estimates may be used in the analysis of high- and low-criticality tasks.

Conveniently, no additional scheduling rules are required for the MC-IPC to work correctly if there is more than one task in a reservation. While we have focused on the case of a single task per reservation, the MC-IPC analysis and the bound on budget consumption due to IPC still hold in this case. (Other tasks in the same reservation may of course consume additional budget, which can be bounded with regular schedulability analysis.)

Finally, we note that it is advantageous to choose a uniform cluster size c: if $m_1 = m_2 = \ldots = m_K = c$, and if m is an integer multiple of c, then $K = \frac{m}{c}$, in which case the required budget reduces to $e_i + \sum_{q=1}^{n_r} N_{i,q} \cdot (1 + 2m) \cdot L_q$.

Next, we discuss a concrete implementation of MC-IPC under partitioned scheduling ($c = 1$ and $K = m$). Whereas so far our goal was to define MC-IPC without loss of generality, the following discussion pertains to a specific configuration that we consider to be of particular practical relevance.

V. Prototype Implementation

We implemented the MC-IPC protocol and a reservation-based scheduler satisfying Assumptions A1 and A2 in LITMUS$^{\text{RT}}$ 2014.2 [2, 6], a real-time extension of the Linux kernel. We chose LITMUS$^{\text{RT}}$ as the basis for our prototype since it already provides much of the required multiprocessor scheduling support and due to our familiarity with it.

Since Linux (and hence LITMUS$^{\text{RT}}$) is not a microkernel, it did not yet provide suitable IPC system calls. We therefore added two system calls, *litmus_ipc_invoke* and *litmus_ipc_reply_wait*, to expose the corresponding MC-IPC operations to real-time and best-effort applications. Mimicking the L4 API, the latter system call implements both the *mc_wait* and *mc_reply* operations as they are called in immediate succession in the main server loop anyway. In our prototype, the maximum message size is limited to 4096 bytes.

We implemented a partitioned, reservation-based scheduler as a new scheduler plugin and added support for three tiers of service: table-driven reservations for high-criticality tasks, denoted as R_j^H, sporadic reservations for low-criticality tasks, denoted as R_j^L, and support for background best-effort tasks.

The sporadic reservations support configurations with either fixed or EDF-based priorities. (In the case-study reported in Sec. VI, we configured all low-criticality reservations to use EDF-based priorities.) The best-effort support shares the same implementation, as best-effort tasks are in fact supported by sporadic reservations with an effectively infinite deadline.

The high-criticality reservations are realized as typical cyclic time partitions, as defined in Sec. III. While the reservation budget is conceptually replenished at the start of each scheduling slot, the budget is not actually explicitly tracked in the implementation. Instead, the remaining budget is implicitly determined by the time remaining until the end of the current slot. While no high-criticality reservation is active, low-criticality reservations or best-effort tasks are scheduled instead.

We believe our choice—table-driven scheduling for high-criticality reservations and event-driven scheduling for low-criticality reservations—to reflect tradeoffs commonly made in practice, which is to favor predictability and ease of validation for high-criticality tasks, and efficiency and flexibility for low-criticality tasks. In particular, note that low-criticality reservations can consume any unused budget of higher-criticality reservations, as assumed by Vestal's model [30].

Inspired by Steinberg *et al.*'s proposal [29] for implementing bandwidth inheritance (on a uniprocessor), bandwidth inheritance was implemented as follows. When a task T_i in a reservation R_j blocks on an IPC gate serviced by S_q, R_j remains active and is kept in the ready-queue, and a bandwidth inheritance marker is added to R_j's client list to indicate that S_q is a potential client of R_j. Thereafter, whenever R_j is selected for service by the top-level scheduler, S_q is dispatched instead, if it is ready and not already executing elsewhere. Conveniently, LITMUS$^{\text{RT}}$ transparently supports process migrations in its layer interfacing with Linux, which frees the top-level scheduler from explicitly migrating servers across cluster boundaries.

If S_q is already executing elsewhere or suspended when R_j attempts to dispatch it, then another local ready task (if any) from a lower-priority reservation is dispatched instead. To comply with the budget idling requirement stipulated by Assumption A2, R_j's budget is drained at unit speed even in this case.

A challenging aspect of implementing bandwidth inheritance on multiprocessors is the check that is required when a server S_q is preempted. In the interest of minimizing overheads, the scheduler needs to quickly determine if S_q is currently eligible to be dispatched on another processor, which depends on whether there are reservations from which S_q is inheriting, and on whether any of those reservations are currently the highest-priority reservation on their assigned core. However, in our implementation, the server is not actually aware of which reservations it may currently inherit from. Instead, each core maintains a sequence number of scheduling decisions. Whenever a reservation R_j from which S_q may inherit finds S_q to be unavailable, it stores the current sequence number in S_q's IPC gate. When S_q is preempted, a quick comparison of the stored with the current sequence numbers for all cores will reveal possible scheduling candidates: S_q can be scheduled only on cores on which the sequence numbers still match. Servers that resume from self-suspensions are handled similarly.

A slight departure from the protocol specification in Sec. IV is an opportunistic improvement that takes effect when a request of a task T_i is aborted due to a budget overrun of its reservation R_j. Instead of canceling the request and returning an error code to T_i, the request is actually transferred to the best-effort *background_queue*, which may allow T_i's request to be processed earlier than otherwise possible (if the server runs out of other clients before R_j's budget is replenished). When R_j's budget is replenished, then its request, if it is still pending, is automatically reissued with its proper priority.

In total, the implementation of the MC-IPC primitive and the supporting reservation-based scheduler added ≈3,700 lines of code to the LITMUS$^{\text{RT}}$ kernel, and ≈340 lines of code to LITMUS$^{\text{RT}}$'s user-space library. Next, we report on a case study that we conducted to test whether temporal isolation is maintained as expected in the presence of malfunctioning tasks.

VI. Case Study

To assess the practical efficacy of the MC-IPC approach, we required a plausible workload that nonetheless is fully configurable (*i.e.*, a dummy application that allows to freely choose WCETs, periods, the number of tasks and requests, *etc.*).

For this purpose, suppose a system subject to high integrity requirements needs to collect data and forward it for further processing such that the data stream cannot be replayed or tampered with. For instance, such requirements may arise in cyber-physical systems in the context of real-time surveillance applications or in sensor feeds for safety-critical control systems.

Each sample must be time-stamped, given a unique sequence number, and cryptographically signed to ensure integrity and freshness. However, to prevent leaking sensitive key material in the case of vulnerabilities, the private keys should not be accessible to all tasks, and especially not to low-criticality tasks. Instead, only a simple, central *key server* S_{key} that is not exposed to the network may access the private keys and provides a service to sign packets. (Recent events surrounding the so-called "Heartbleed" OpenSSL vulnerability [1] suggest that it is good practice to retain key material in a separate process.)

The server S_{key} is a shared resource that must be accessed by all data-collecting tasks, regardless of their criticality. In our experiments, we measured the cost of invoking S_{key} when using the MC-IPC protocol and two IPC protocol variants representing the baseline in current systems: one using a FIFO queue like the MBWI [12], denoted FIFO-IPC, and one using a priority queue as in the Fiasco.OC microkernel [19], denoted PRIO-IPC.

A. Setup

We used an Intel x86-64 multicore platform with a Xeon E5-2665 processor clocked at 2.4 GHz, of which we used $m = 4$ cores for the experiment. Intel's "Turbo Boost" and power-management-related sources of unpredictability were disabled. We implemented S_{key} with OpenSSL and measured that creating a 2048-bit RSA signature requires roughly $L_{key} = 2ms$ of processor time on our platform.

With a cycle time of $H = 100ms$, we created a total of 14 reservations as follows:

- on core C_1, we created a single high-criticality reservation R_1^H in the window $[0ms, 50ms)$;
- on core C_2, we created two high-criticality reservations R_2^H and R_3^H in the respective windows $[0ms, 50ms)$ and $[50ms, 100ms)$;
- on core C_3, we created a high-criticality reservation R_4^H in $[0ms, 50ms)$ and five low-criticality reservations $R_5^L, \ldots, R_9^L$ with a budget of $20ms$ and periods $100ms$, $150ms$, $250ms$, $500ms$, and $1000ms$, respectively; and finally,
- on core C_4, another five low-criticality reservations $R_{10}^L, \ldots, R_{14}^L$ with parameters identical to $R_5^L, \ldots, R_9^L$.

Each reservation contained initially a single (simulated) data acquisition task. Each job of such a task first acquired new sensor data (simulated by reading from */dev/urandom*) and carried out some processing by repeatedly iterating over the array holding the "sample." The "sample" was then submitted—with the MC-IPC, FIFO-IPC, or PRIO-IPC protocol—to S_{key} to have it timestamped, assigned a sequence number, and signed. Finally, after receiving the server's reply containing the signature, the task assembled the final packet (consisting of the data itself, the timestamp, a sequence number, and the signature) and wrote it to a datagram socket (simulated with a write to */dev/null*).

In the PRIO-IPC and MC-IPC configurations, each reservation further requires a priority, which we assigned as follows: best-effort background tasks had the lowest priority, low-criticality reservations were EDF-ordered, and the four high-criticality reservations had fixed priorities in the order $R_1^H < R_2^H < R_3^H < R_4^H$, where R_4^H has the highest priority.

In the following, we focus on task T_1 in reservation R_1^H, which we use as the vantage point in this case study.

B. Anticipated Worst-Case Delays

Each of the three considered IPC protocols is *predictable*, in the sense that the maximum budget drain can be bounded in advance—however, with the FIFO-IPC and PRIO-IPC protocols, *only in the absence of failures*. Thus, based on the maximum operation length L_{key} we derived the following bounds on the maximum delay encountered by T_1:

- with the MC-IPC protocol, according to Theorem 1, T_1 should be delayed for at most $(1 + 2m) \cdot L_{key} = 18ms$;
- with the FIFO-IPC protocol, since there are $n = 14$ tasks in total, T_1 should be delayed for at most $n \cdot L_{key} = 28ms$;
- and with the PRIO-IPC protocol, since only the tasks in two higher-priority reservations can overlap with R_1^H, each of which issue only a single request per cycle, T_1 should be delayed for at most $(2 + 1 + 1) \cdot L_{key} = 8ms$ (including T_1's own request and one blocking lower-priority request).

Note that for a high-criticality reservation, the encountered delay corresponds exactly to the amount of budget drained since high-criticality reservations are never preempted. In the following, we compare the anticipated delays with the actual measured delays.

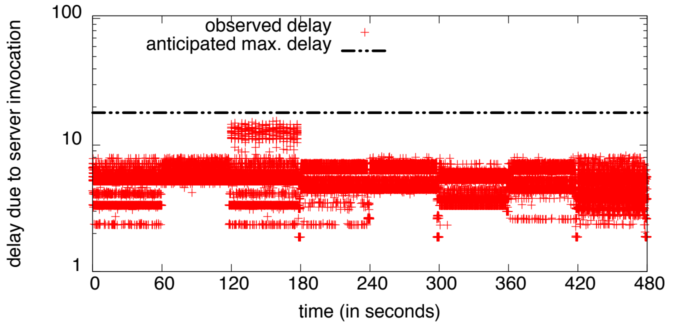
C. Observed Delay in the Presence of Failures

We executed the task set with each IPC primitive ten times and measured the delay encountered by T_1 when invoking S_{key}. The resulting scatterplots are shown in Fig. 2, where each data point corresponds to a job of T_1 (note the logarithmic scale). Each run lasted for eight minutes and transitioned through eight phases in which the environmental conditions changed, as described next.

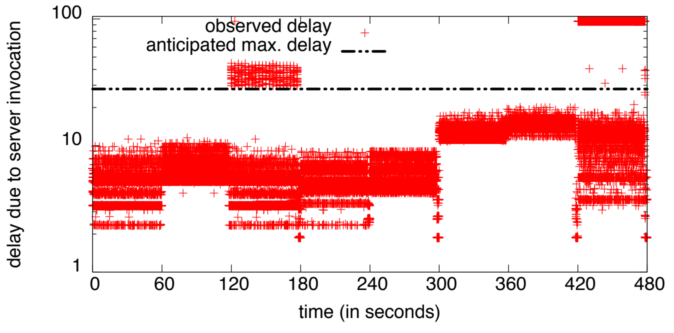
1) Phase 1 (0s-60s): In the first minute, the system operates normally. As can be seen in Fig. 2, the observed delay stays below the predicted maxima with each of the three IPC protocols.

2) Phase 2 (60s-120s): On core C_4, a task in a low-criticality reservation with period $100ms$ malfunctions by entering an infinite loop, sending requests to S_{key} as quickly as possible. The distribution of observed delays is affected visibly under each protocol due to the added contention; however, the predicted worst-case is not exceeded: the worst case with FIFO-IPC and MC-IPC is independent of request rates, and PRIO-IPC shields high-priority tasks against lower-priority tasks. The malfunctioning task is terminated after one minute.

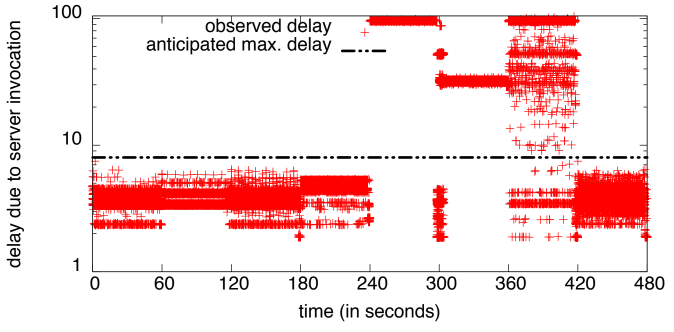
3) Phase 3 (120s-180s): In the third minute, the system is flooded with low-utilization, low-criticality reservations: on each of the four cores, 16 additional low-criticality reservations with a budget of $100ms$ and a period of $3000ms$ are launched. While this overloads the system at the level of low-criticality reservations, the high-criticality reservations are expected to be unaffected as they have statically higher priority than any of the low-criticality and best-effort reservations. However, even though the new tasks operate correctly (*i.e.*, they invoke S_{key} only once per job), the large number of unexpected tasks causes the predicted FIFO-IPC delay bound to be violated, as seen in Fig. 2(b): the *high*-criticality task T_1 is unexpectedly delayed by *low*-criticality tasks, owing to a lack of temporal isolation. Worse, such violations still occur (although less frequently) even



(a) Observed delay with MC-IPC vs. predicted delay bound ($18ms$)



(b) Observed delay with FIFO-IPC vs. predicted delay bound ($28ms$)



(c) Observed delay with PRIO-IPC vs. predicted delay bound ($8ms$)

Fig. 2. Delay incurred by task T_1 in reservation R_1^H when invoking S_{key}.

if the utilization of the newly arrived reservations is scaled down to avoid overload (*i.e.*, in the absence of an admission control failure). In contrast, the MC-IPC worst-delay is independent of n, and PRIO-IPC isolates T_1 against lower-priority activity.

4) Phase 4 (180s-240s): Next, the high-criticality, high-priority task in R_4^H malfunctions, effectively launching a denial-of-service (DoS) attack against S_{key}. However, this has no immediate ill effects: the worst case with FIFO-IPC and MC-IPC is independent of request rates, and with PRIO-IPC, a single malfunctioning higher-priority task does not cause starvation by itself because two consecutive requests are still separated by a small gap due to the system call overhead.

5) Phase 5 (240s-300s): However, in the fifth minute, a second high-criticality task in reservation R_2^H malfunctions and joins the task in R_4^H in flooding S_{key} with requests. The FIFO-IPC and MC-IPC protocols remain unaffected, but the PRIO-IPC protocol suffers catastrophic consequences, as evident in Fig. 2(c): the observed delays reach $100ms$, which shows that T_1 overran its budget due to being starved by the flood of requests.

6) Phase 6 (300s-360s): After killing and restarting both malfunctioning tasks, another failure occurs: "accidentally," R_4^H launches 15 additional (correctly operating) tasks. Since R_4^H has the highest priority, this drives R_1^H's budget consumption up to $\approx 32ms$, well in excess of the anticipated worst case of $8ms$. With 15 additional tasks, actually up to $(15 + 2 + 1 + 1) \cdot L_{key}$ = $38ms$ delay could be possible; however, the theoretical worst case is not encountered in this case.

As expected, delays under the FIFO-IPC protocol are also significantly elevated, but stay below the predicted worst case, again simply because the worst case is actually not encountered.

7) Phase 7 (360s-420s): Phase 7 is a combination of phases 4 and 6: the 15 unexpected tasks in R_4^H are still present, and the task in R_2^H malfunctions again. Delays under the MC-IPC protocol remain largely unaffected, and the FIFO-IPC protocol continues to be somewhat affected, but is isolated from changes in request rates. The PRIO-IPC protocol, however, fails: many jobs of T_1 do not manage to complete within R_1^H's budget. This shows that a single higher-priority task engaged in a DoS attack can violate temporal isolation if there is a sufficiently large number of correctly operating higher-priority tasks.

8) Phase 8 (420s-480s): Finally, in the last minute, the additional tasks from phase 6 have been terminated and all real-time tasks resume correct operations. However, the system is hit with a wave of best-effort tasks: on each core, 20 best-effort tasks start to operate correctly. Best-effort activity has no negative impact on T_1 when using the MC-IPC or the PRIO-IPC, but the unexpectedly large number of tasks causes massive budget overruns under the FIFO-IPC, as seen in Fig. 2(b).

To summarize the observed results, the FIFO-IPC protocol is resilient to unexpectedly high request rates, but fails if it encounters an unexpectedly large number of tasks. The PRIO-IPC protocol is resilient to anomalous behavior in lower-priority reservations, but requires trusting all higher-priority reservations. Only the MC-IPC protocol has ensured strict temporal isolation and remained below the predicted delay bound in all cases.

VII. Integration with Vestal's Model

While we have adopted a pragmatic, systems-oriented view of mixed-criticality systems in this paper, the MC-IPC protocol is in fact also applicable in the context of Vestal's mixed-criticality task model [30], as we briefly illustrate in the following.

The distinguishing feature in Vestal's model is that instead of a single WCET bound e_i, each task has a WCET bound specific to each criticality level. That is, assuming for simplicity that there are only two criticality levels *low* (L) and *high* (H), each task has a WCET bound at a lower level of assurance e_i^L and a WCET bound at a higher level of assurance e_i^H, where $e_i^L \leq e_i^H$, which reflects that WCET bounds tend to become more pessimistic at higher levels of assurance [11, 30].

Importantly, the temporal correctness condition for a task system modeled in this fashion requires that *all* tasks meet all their deadlines if all jobs of each task T_i (of either high or low criticality) execute for at most e_i^L time units, whereas only high-criticality tasks are required to meet their deadlines if a job of any T_i executes for more than e_i^L time units.

In the context of our system model, a concrete implementation of this concept can be realized along the following lines [4, 15, 25]: a low-criticality sporadic reservation encapsulating a task T_i is provisioned with a budget of e_i^L time units, whereas a high-criticality temporal partition encapsulating a high-criticality task T_h would be dimensioned to allow for up to e_h^H time units of execution, with the expectation that only up to e_h^L will be used. This leaves $e_h^H - e_h^L$ expected spare capacity to lower-criticality tasks, which allows to statically reclaim, at design time, some of the pessimism inherent in high levels of assurance [11, 30].

If all high-criticality reservations are table-driven (as in our case study), no action at runtime is required. Otherwise, if there exist high-criticality sporadic reservations, some runtime monitoring is required: if any high-criticality reservation actually consumes more than e_h^L time units of budget, which is trivial to detect (but highly unlikely to occur), then the system switches temporarily into a high-criticality mode. In this mode, some or all of the low-criticality reservations are demoted to (just above) best-effort status, until the overload (with respect to the low-criticality analysis assumptions) has dissipated (*e.g.*, until an idle instant is reached), at which point normal operations resume. No jobs are abandoned, and there is a good chance that most low-criticality jobs will still finish before their deadline.

The MC-IPC protocol seamlessly integrates with this interpretation of Vestal's model. Instead of considering only a single maximum operation length L_q for each server, we introduce criticality-dependent maximum operation lengths L_q^L and L_q^H, which, just as it is the case with criticality-dependent WCET bounds [11, 30], reflect different levels of uncertainty in the bound on the true maximum operation length (where $L_q^L \leq L_q^H$). Further, let K_q^L and K_q^H denote criticality-dependent upper bounds on the number of clusters from which a server will be invoked, where $K_q^L \leq K_q^H$. A safe high-assurance assumption is $K_q^H = K$ (*i.e.*, in the worst case, S_q is invoked from every cluster that exists), but for lower assurance levels, it may be reasonable to exclude clusters which are known to not host any clients of S_q under normal conditions such that $K_q^L < K$.

Recall from Sec. IV that $N_{i,q}$ denotes the maximum number of times that a job of T_i invokes a server S_q. Then the budget of a low-criticality reservation R_j^L in cluster C_k encapsulating task T_i should be $e_i^L + \sum_{q=1}^{n_r} N_{i,q} \cdot (1 + 2 \cdot K_q^L \cdot m_k) \cdot L_q^L$. In contrast, the budget of a high-criticality reservation R_j^H should be $e_i^H + \sum_{q=1}^{n_r} N_{i,q} \cdot (1 + 2 \cdot K_q^H \cdot m_k) \cdot L_q^H$.If the low-criticality estimates L_q^L and K_q^L are accurate, that is, if no S_q ever takes more than L_q^L time units to process any request and if requests are issued from at most K_q^L clusters, then both low- and high-criticality tasks will meet their deadlines. Otherwise, if either L_q^L or K_q^L are exceeded at runtime, then there exists a remote possibility that a low-criticality task is provisioned with insufficient budget to meet its deadline. However, even in this case, the high-criticality tasks are still guaranteed to meet their deadlines, which is in agreement with the goals and spirit of Vestal's model [30], and a majority of the work on mixed-criticality scheduling [11].

Finally, in the case of a switch to high-criticality mode, pending requests of demoted low-criticality reservations can simply be aborted as if the reservations had exhausted their budget. The fact that a resource server could be executing on the budget of a demoted reservation at the time of the switch to high-criticality mode does not cause further complications since the MC-IPC protocol tolerates potential budget overruns and requests by best-effort tasks anyway.

To conclude, with the strict temporal isolation provided by MC-IPC, it is possible both to share resources among tasks of different criticality without the loss of temporal or logical isolation, and to statically reclaim part of the system capacity lost to the pessimism inherent in the analysis of worst-case contention. We therefore believe MC-IPC to be well-suited for mixed-criticality workloads, from both a systems (Secs. V and VI) and an analytical point of view (Secs. IV-B and VII).

VIII. Related Work

This work resides at the intersection of mixed-criticality systems, real-time resource sharing, microkernels and IPC, and reservation-based scheduling, which are vast areas on their own that each have received more attention in prior work than what could be adequately surveyed in this limited space. We thus focus on the most closely related prior works.

For a comprehensive overview of the field of mixed-criticality real-time scheduling pioneered by Vestal's 2007 paper [30], we refer the reader to Burns and Davis' authoritative survey [11].

As argued in Sec. II, from a systems perspective, strict isolation is paramount in mixed-criticality systems, which makes it natural to decompose such systems into untrusted components running on a small, trusted executive, which, depending on context and minor differences in APIs, is called either a microkernel, hypervisor, or separation kernel. Prominent examples include the L4 family [14, 17, 19], Quest-V [22], as well as commercial products such as Sysgo's PikeOS. Our work is well aligned with these systems in both goals and assumptions, and the MC-IPC protocol could be supported in any of them.

This paper pertains to the explicit sharing of resources subject to mutual exclusion constraints at the software level, an area that, in the context of real-time systems, dates back to Sha *et al.*'s seminal work on priority inheritance [28]. In a hardware context, the term "shared resources" is also used to describe architectural elements such as the memory bus, bank controllers, or shared caches. As mentioned in Sec. II-B, such shared hardware resources also give rise to substantial interference, a problem that has seen much recent attention [13, 18, 23, 26, 31, 32].

Central to our solution are the concepts of reservation-based scheduling [24] and bandwidth inheritance [12, 21]. For simplicity, we implemented simple sporadic reservations [24] in our prototype; in a production system, more flexible reservation techniques such as CBS [3] or RBED [9, 27] may be desirable. The use of bandwidth inheritance in microkernel IPC primitives, also known as "helping" or "timeslice donation," dates back to at least 2001 [16] and is now a standard technique. Steinberg *et al.* [29] discuss an efficient implementation.

Our choice to realize high-criticality reservations with a table-driven approach and low-criticality reservations as sporadic servers was guided by UNC's earlier work on RTOS support for mixed-criticality systems [4, 15, 25], which is also based on LITMUS$^{\mathrm{RT}}$. However, these earlier efforts did not consider resource sharing and lacked bandwidth inheritance.

Some initial progress has been made towards the analysis of real-time locking protocols and priority inversions in the context of mixed-criticality systems (*e.g.*, see [10, 11, 20, 34]). However, unlike this work, the just-cited approaches apply only to uniprocessor systems, and as argued in Sec. II-C, we consider locks to be fundamentally limited in a mixed-criticality context. Nonetheless, there exists an obvious duality between synchronous IPC and real-time locking, and it will be interesting to explore if and how the analytical guarantees obtained in [10, 20, 34] can be transferred into the IPC setting.

Finally, the MC-IPC protocol extends the queue structure first developed for the OMIP [7], a locking protocol with an asymptotically optimal priority-inversion bound under suspension-oblivious analysis [6, 8]. However, the work in [7] does not support temporal isolation, requires mutual trust among resource-sharing tasks, does not support non-uniform clusters, and does not account for potential budget overruns. Nonetheless, the OMIP is conceptually a predecessor of the MC-IPC protocol, and a contribution of this paper is to bring to light a fundamental connection between the idling rule (Assumption A2) and suspension-oblivious blocking analysis [8].

IX. Conclusion

We have proposed the MC-IPC protocol, a synchronous IPC protocol that enables the temporally and logically isolated sharing of resources in mixed-criticality systems, and have shown it to be effective both in a practical system (Sec. VI), and in the context of Vestal's mixed-criticality task model (Sec. VII).

This paper breaks new ground in two major directions: it is the first work to consider explicit resource-sharing in multiprocessor mixed-criticality systems, and second, it is the first solution to eliminate trust assumptions concerning the number of tasks, the number of admitted reservations, maximum request frequencies, and the cooperation of other tasks, thus enabling stronger temporal isolation than any prior proposal for multiprocessor real-time synchronization in general.

In summary, with the MC-IPC protocol, a task's temporal and logical correctness depends only on **(i)** the kernel (for the scheduler, logical isolation, and IPC), **(ii)** proper admission control (to ensure that the set of admitted reservations is feasible), and **(iii)** the correctness of the invoked servers—*not*, however, on the correctness of other tasks invoking shared servers.

In future work, it would be interesting to evaluate MC-IPC in a true microkernel. Additionally, we seek to extend the protocol and its analysis to allow servers to invoke other servers.

References

[1] "OpenSSL TLS heartbeat read overrun," security advisory, https://www.openssl.org/news/secadv_20140407.txt, CVE-2014-0160.

[2] "The LITMUS^{RT} project," web site, http://www.litmus-rt.org.

[3] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS '98*, 1998.

[4] J. Anderson, S. Baruah, and B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[5] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *RTSS'11*, 2011.

[6] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, UNC Chapel Hill, 2011.

[7] ——, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *ECRTS'13*, 2013.

[8] B. Brandenburg and J. Anderson, "Optimality results for multiprocessor real-time locking," in *RTSS'10*, 2010.

[9] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes," in *RTSS'03*, 2003.

[10] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," in *ReTiMiCS'13*, 2013.

[11] A. Burns and R. Davis, "Mixed criticality systems: A review," University of York, Department of Computer Science, Tech. Rep.

[12] D. Faggioli, G. Lipari, and T. Cucinotta, "Analysis and implementation of the multiprocessor bandwidth inheritance protocol," *Real-Time Systems*, vol. 48, no. 6, pp. 789–825, 2012.

[13] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *EMSOFT'13*, 2013, pp. 1–15.

[14] G. Heiser, "Hypervisors for consumer electronics," in *CCNC'09*.

[15] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson, "RTOS support for multicore mixed-criticality systems," in *RTAS'12*.

[16] M. Hohmuth and H. Härtig, "Pragmatic nonblocking synchronization for real-time systems," in *USENIX ATC'01*, 2001.

[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *SOSP '09*, 2009.

[18] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, and M. Roy, "Run-time control to increase task parallelism in mixed-critical systems," in *ECRTS'14*, 2014.

[19] A. Lackorzynski and A. Warg, "Taming subsystems: capabilities as universal resource access control in L4," in *IIES '09*, 2009.

[20] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *RTAS'11*, 2011.

[21] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *RTSS'01*, 2001.

[22] Y. Li, M. Danish, and R. West, "Quest-V: A virtualized multikernel for high-confidence systems," 2011, arXiv:1112.5136.

[23] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *RTAS'13*, 2013.

[24] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: an abstraction for managing processor usage," in *Proc. Fourth Workshop on Workstation Operating Systems*, 1993.

[25] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *CIT'10*, 2010.

[26] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," *ECRTS'14*, 2014.

[27] S. Petters, M. Lawitzky, R. Heffernan, and K. Elphinstone, "Towards real multi-criticality scheduling," in *RTCSA'09*, 2009.

[28] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.

[29] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice donation in component-based systems," in *OSPERT'10*, 2010.

[30] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS'07*, 2007.

[31] B. Ward, J. Herman, C. Kenna, and J. Anderson, "Making shared caches more predictable on multicore platforms," in *ECRTS'13*.

[32] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *ECRTS'12*, 2012.

[33] ——, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS'13*, 2013.

[34] Q. Zhao, Z. Gu, and H. Zeng, "HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling," *Embedded Systems Letters*, vol. 6, no. 1, pp. 8–11, 2014.