G(IP)^{2}C: Temporally Isolated Multiprocessor Real-Time IPC with Server-to-Server Invocations

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

***Abstract*—Synchronous *inter-process communication* (IPC) is a central operation in microkernel-based operating systems, which are commonly employed in mixed-criticality real-time systems. A key desideratum in an IPC protocol for time-sensitive systems is *temporal isolation*: when invoking a *shared server*, the worst-case interference incurred by the waiting client (*i.e.*, the maximum amount of budget its reservation drains while waiting for the reply) should be bounded irrespective of the behavior of competing, untrusted clients. Additionally, an IPC protocol should support *server-to-server* (S2S) invocations, so that servers may invoke other servers when handling requests, which enables modern software engineering practices (*e.g.*, reuse of shared functionality, decomposition of complex services into cooperating servers, *etc.*).**

However, no prior synchronous multiprocessor IPC protocol achieves both. The main contribution of this paper is to remedy this limitation: the proposed G(IP)^{2}C protocol for partitioned, reservation-based multiprocessor scheduling ensures a strong notion of temporal isolation while permitting S2S invocations without placing any restrictions on which processors clients and servers reside on. The protocol is defined as a set of request-sequencing, bandwidth-delegation, and budget-exhaustion rules, analyzed in terms of maximum budget drain, extended to multi-occupancy reservations and background tasks, and shown to be practically realizable with a prototype implementation in LITMUS^{RT}.

I. Introduction

Many real-time operating systems for critical and mixed-criticality systems follow a microkernel design. Notable examples are QNX [32], the L4 familly [10, 18], Quest-V [22], and CompositeOS [29]. A key driver for this design preference is the high degree of isolation and fault containment achievable by microkernel-based systems, which realize most of the functionality provided by the operating system (device drivers, file systems, *etc.*) in user-space processes called *servers*.

In particular, microkernel-based systems offer an elegant and robust solution to the problem of resource sharing in the presence of untrusted tasks: access to shared resources (such as actual hardware devices or higher-level OS facilities) can be mediated by encapsulating them in *resource servers*. In this approach, instead of exercising direct, unchecked access to shared resources (which would require *trust*), clients invoke the corresponding resource servers using a *synchronous inter-process communication* (IPC) protocol to request operations to be carried out on their behalf according to a well-defined, access-controlled interface (which requires trusting the server, but not other clients). Unsurprisingly, the IPC infrastructure is a central part of any microkernel, influencing its overall design [23], and subject to much optimization and benchmarking.

When hosting time-sensitive applications, temporal predictability also becomes a key design objective. Consequently, a real-time IPC protocol should guarantee *temporal isolation* (*i.e.*, bounded delay) to clients invoking shared resource servers, which however is easier said than done. In the context of mixed-criticality systems in particular, such guarantees are difficult to offer since the number of competing tasks may be uncertain and since tasks cannot be trusted to be well-behaved [5].

The challenge of ensuring temporal isolation does not get any easier if resource servers may invoke other servers as part of handling requests. We refer to such requests as *server-to-server* (S2S) requests, as opposed to *client-to-server* (C2S) requests emitted by top-level applications. S2S requests introduce the possibility for a client to be delayed due to contention for servers it does not explicitly invoke. This problem is similar to the transitive blocking problem encountered in the context of nested locking protocols, which can result in exponential delays as shown by Takada and Sakamura [34].

While IPC protocols are typically designed to be extremely fast in the absence of contention [10, 23, 24], historically, much less attention has been given to the sequencing of concurrent requests, and even less so in the context of multiprocessor systems. Prior work in this space can be divided roughly into two categories: **(1)** flexible approaches permitting S2S invocations, which however do not ensure temporal isolation, and **(2)** approaches offering strong temporal isolation guarantees that alas support only C2S requests.

Related work in the first category (reviewed in Section IX) focuses mostly on FIFO and priority queues, which do not guarantee temporal isolation in the presence of untrusted tasks, especially if servers and clients reside on different processors. To our knowledge, the *mixed-criticality IPC* (MC-IPC) protocol [5] is the only protocol in the second category. The MC-IPC protocol offers strong temporal isolation properties in the context of mixed-criticality systems, regardless of the number and behavior of competing tasks. Unfortunately, the MC-IPC protocol lacks support for S2S requests.

Proper handling of S2S requests must reconcile two difficult problems: first, how to interleave concurrent S2S requests in a way that does not cause deadlock or exponential worst-case delays for clients; and second, how to do so while preserving some level of parallelism? Advances in real-time nested locking protocols, especially the *real-time nested locking protocol* (RNLP) [36, 37] and more recently the *group-independence-preserving protocol* (GIPP) [31], offer solutions that can help reconcile these aspects in systems using *job-level fixed priority* (JLFP) schedulers. However, these solutions do not directly apply in the IPC context since they assume fully trusted

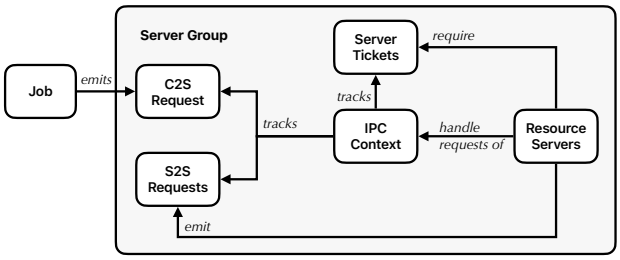


Fig. 1: G(IP)^{2}C protocol components overview.

worst-case execution times and critical-section lengths. In contrast, budgets are typically enforced with reservation-based scheduling [27] in a mixed-criticality setting. A real-time IPC protocol must hence account for the possibility of budget exhaustion during IPC, which is a major complication.

The main contribution of this paper is the first protocol that simultaneously solves all of these challenges: the *group-independence-preserving IPC* protocol (Section IV) for partitioned, reservation-based multiprocessor scheduling, henceforth abbreviated as the G(IP)^{2}C protocol. The G(IP)^{2}C protocol prevents deadlock and ensures temporal isolation (Section V), is suitable for use with untrusted tasks and in mixed-criticality systems, and allows for S2S requests while still ensuring a degree of parallelism in the processing of concurrent C2S requests. We additionally show how to adapt the G(IP)^{2}C protocol to allow for *multi-occupancy reservations* (*i.e.*, multiple tasks sharing a reservation) and best-effort background tasks (Section VI). Finally, we report on a prototype implementation of the G(IP)^{2}C protocol in LITMUS$^{\text{RT}}$ [3, 9] that shows the protocol to be realizable in practice (Section VII).

II. High-Level Overview

The G(IP)^{2}C protocol involves a fair number of interacting elements and techniques. For ease of understanding, we begin with a high-level overview of the G(IP)^{2}C protocol, with a focus on its structure (see Fig. 1), central elements, and the underlying intuition. A precise definition is given later in Section IV.

The G(IP)^{2}C protocol orchestrates the communication between possibly untrusted clients and trusted resource servers. Invocation requests are issued to resource servers using the `gip_invoke` system call. Conceptually, a resource server respectively selects a request to handle and completes it with the `gip_wait` and `gip_reply` operations. For efficiency, these two operations are actually combined into a single `gip_reply_wait` system call in which a server atomically completes a request and picks the next one (if any). The client of a request is suspended until it receives an answer from the invoked server. The G(IP)^{2}C protocol guarantees bounds on this waiting time in terms of the budget expended by the reservation hosting the client.

There are two distinct types of requests depending on the nature of the issuer: C2S requests are emitted by tasks of top-level applications, whereas S2S requests are emitted by resource servers as part of processing another request. It follows that the processing of a C2S request can result in many subsequent S2S requests, which must be tracked. The **IPC context** is the main structure used internally in the G(IP)^{2}C protocol to track the progress of C2S and S2S requests. It embeds the necessary information to let resource servers determine when a request can be processed safely without breaking temporal isolation.

The lifecycle of a C2S request consists of an *IPC context acquisition phase* (Phase 1) followed by a *request service phase* (Phase 2). The main purpose of Phase 1 is to resolve intra-processor contention. To this end, the G(IP)^{2}C provides only one *shared* IPC context for each processor and each "server group" (introduced shortly). Tasks must hence first acquire their local IPC context before they can issue a C2S request.

Phase 2 starts once a task has acquired the necessary IPC context. During this phase, the IPC context is enqueued in a global queue of requests. Resource servers process requests from this queue, potentially in parallel, based on the information contained in the IPC context. Phase 2 ends when the initial C2S request is complete, at which point the client is resumed.

To avoid deadlock and the risk of exponential delay, it is necessary to introduce non-work-conserving behavior into the request service phase (as pioneered by the RNLP [36] and also employed in the GIPP [31]). Specifically, servers must *not* always serve the next C2S or S2S request *immediately* when the prior request is finished, as that can easily lead to pathological scenarios. Rather, servers may start serving a request only if there is no risk of thereby delaying any earlier, still-unfinished C2S requests (directly or indirectly).

To track which resource server is or will be involved in the processing of a request, we introduce the notion of a **server ticket**. A server ticket is a single-use token required to invoke a resource server and consumed on reply. Each acquired IPC context is equipped with a finite multiset of server tickets, representing the resources needed to fulfill the client request in the worst case. As a result, the multiset of server tickets of an IPC context represents the current and future resource servers potentially involved in processing this request.

It follows that two C2S requests do not interfere if their ticket multisets do not intersect. Hence, during the request service phase, a request is served only if there is no conflicting C2S request emitted earlier. This approach, inspired by the *dynamic group locks* (DGL) variant of the RNLP [37], also prevents deadlock (as shown in Section V).

Clearly, IPC contexts with disjoint sets of server tickets do not directly affect each other through the just-sketched ticketing rules. It would thus be undesirable for such unrelated C2S requests to compete for the IPC context of the same processor. Rather, such disjoint C2S requests should be handled in separate scopes. We introduce the notion of a **server group** for that purpose. Two resource servers belong to the same server group if there is a dependency among them, *i.e.*, if at least one operation of one of the two servers requires invoking the other. Therefore, server groups partition the set of resource servers into disjoint sets that can safely operate in parallel. The G(IP)^{2}C protocol thus operates at the server group level: each server group has a separate global queue and set of IPC

contexts. Clients interact with only one server group at a time.

The notion of progress in the G(IP)^{2}C protocol is deeply tied to the employed scheduling model. The G(IP)^{2}C protocol is suited for systems using reservation-based scheduling [27]. Tasks are encapsulated in reservations with a finite budget of processor time that is occasionally replenished. By scheduling these reservations instead of their contained tasks directly, temporal isolation can be offered in the presence of untrusted or unknown worst-case execution times.

In the G(IP)^{2}C protocol, resource servers are *passive*: they are not assigned to a reservation and instead execute on behalf of their client using a form of *bandwidth inheritance* [11, 21]. Bandwidth inheritance is a mechanism allowing a task to make its budget available to another one. This is achieved by decoupling a task's scheduling context from the rest of its execution context: each task contained in a reservation is assigned a **service token** representing its scheduling context.

Service tokens are scheduled by reservations and can be transferred to other tasks, giving them the right to be dispatched in place of the task giving up its service token. We then say that a task *delegates its bandwidth* to another. Bandwidth delegation takes effect across reservation and processor boundaries: a task making use of a service token that it received via bandwidth delegation is transferred to the corresponding reservation, temporarily migrating to another processor if necessary.

Bandwidth delegation is necessarily transitive. A key challenge is to manage an effective bandwidth delegation flow from regular tasks to resource servers. To that end, we introduce the notion of a *bandwidth proxy*. A bandwidth proxy is a *non-executable* entity that can receive and delegate bandwidth. These proxies greatly simplify the delegation rules that ensure that resource servers always receive the service tokens of all clients waiting for them, either directly or indirectly.

With a high-level overview of the G(IP)^{2}C protocol and its key entities in place, we now offer a more formal definition.

III. System Model

We consider a partitioned system with m identical processors $p_1, \ldots, p_m$. The main schedulable entities are N reservations $R_1, \ldots, R_N$. Each reservation R_j has a current budget B_j and a current priority Y_j. We assume that priorities are unique (i.e., any ties are broken arbitrarily but consistently). Reservations are statically assigned to processors and do not migrate.

Each reservation hosts one or more *tasks* releasing *jobs* over time. We denote by J_i a job of a task T_i. The number of tasks within each reservation and their parameters are left unspecified. A reservation is *active* if at least one of the contained tasks has a *pending* (*i.e.*, incomplete) job and is *inactive* otherwise. An incomplete job is pending regardless of whether it is ready to execute, waiting for IPC, or suspended for other reasons.

The system contains an unspecified number of serially reusable resources $\ell_1, \ell_2, \ldots$ shared by the tasks. Each resource ℓ_q is encapsulated in a corresponding resource server S_q that can be invoked by regular jobs and other resource servers. Each server S_q has a set of resource-specific *operations*, is sequential, and belongs to exactly one *server group*. Resource servers can call other servers when handling requests as long as they belong to the same server group.

Access rights are managed by means of *server tickets*: a server ticket is a server-specific token a client must present before invoking a resource server. For each operation o, we assume there exists a finite multiset $\mathcal{L}_o$ containing enough server tickets to carry out o. The ticket multiset $\mathcal{L}_o$ can be a conservative superset of the server tickets actually needed to carry out o (*e.g.*, as determined by a static analysis). While we do not address security considerations in this paper, we note that our notion of server tickets is well suited for integration with a *capability system* as commonly found in microkernels.

Upon its release, a job J_i receives a *service token* τ_i representing its right to be scheduled in its reservation. This token is referred to as J_i's *assigned service token*. τ_i has an intra-reservation priority, which is determined on J_i's release.

For each processor, there is a *top-level scheduler* in charge of selecting, at any time t, the highest-priority active reservation with non-zero budget. The selected reservation, in turn, invokes its *reservation-level scheduler* to select the highest-priority service token held by an executable entity in the ready state (if any). Finally, the reservation-level scheduler dispatches the holder of this service token.

We employ the *passive server model*: resource servers are not contained in reservations and instead benefit from *bandwidth delegation*. Bandwidth delegation is a mechanism allowing the transfer of service tokens between jobs, resource servers, and non-executable entities called *bandwidth proxies*. The set of service tokens held by an entity at time t (either assigned or obtained by delegation) is called its *bandwidth*. An executable entity (job or resource server) is dispatched using one of the scheduled service tokens comprising its bandwidth. If several service tokens are available, the choice can be made arbitrarily.

Upon starting delegation, a *delegator* fully transfers its bandwidth to a *recipient*, *i.e.*, a delegator's bandwidth is always empty. Delegation is transitive: a delegator receiving bandwidth forwards it to its recipient. An entity retrieves all its bandwidth when it ceases delegation. Cyclic delegation is forbidden.

The employed scheduling and budgeting rules (*i.e.*, how a reservation R_j's current priority Y_j is determined and how and when R_j's current budget B_j is replenished) are left underspecified for generality. The G(IP)^{2}C protocol's analytical guarantees 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 delegation. 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 [8].

IV. The G(IP)^{2}C Protocol

In this section, we introduce the main rules of the G(IP)^{2}C protocol. For the sake of simplicity, we first assume that each reservation contains only one task and that no background job uses the IPC infrastructure. We will later lift these limitations with protocol extensions presented in Section VI.

The G(IP)^{2}C protocol rules fall into three categories: *sequencing*, *bandwidth delegation*, and *abortion* rules. We introduce these rules in the context of a job J_i invoking a resource server S_q in group g to carry out operation o. T_i is assigned to reservation R_j assigned to a processor p_k. Fig. 2 illustrates the structures and concepts introduced throughout this section.

A. Sequencing Rules

The G(IP)^{2}C protocol orchestrates the invocation of resource servers by jobs of applications (*i.e.*, tasks). Conceptually, there are three operations: jobs issue C2S requests to servers using the `gip_invoke` system call, servers wait for requests with `gip_wait` and complete them with `gip_reply`. Additionally, resource servers can invoke other servers as part of handling client requests. In this case, they emit S2S requests, also using `gip_invoke`. The `gip_invoke` and `gip_wait` operations are blocking; their callers (jobs or resource servers) are suspended until completion of the respective operation.

The lifecycle of a C2S request emitted by J_i consists of two consecutive phases. In the first phase, J_i must acquire an IPC context, which is used to track the progress of its request, and which also limits the number of concurrent C2S on each processor. Phase 1 begins with a job's `gip_invoke` invocation. Then, in the second phase, the acquired IPC context is enqueued in a *group queue* from which resource servers pick requests to handle. The request selection rules are designed such that a C2S request cannot be delayed after a resource server starts handling it. The request is then said to be *committed*. This notion is important for two reasons: a committed request cannot be aborted, and committing a request affects the progress in Phase 1. For convenience, we use the term committed also to refer to IPC contexts tracking committed C2S requests.

Phase 1. The *IPC context-acquisition phase* resolves intra-processor contention. During this phase, jobs of reservations on the same processor contend for an IPC context. There is one such IPC context per group and per processor. We denote by $\mathcal{C}_g^k$ the IPC context for the group g and the processor p_k. An IPC context is either *acquired* or *available* and, additionally, has an associated *slot* that can be occupied by a waiting reservation. Each IPC context $\mathcal{C}_g^k$ is guarded by a priority queue PQ_g^k of reservations, as illustrated in Fig. 2a. Finally, a reservation *requires* $\mathcal{C}_g^k$ if it has a job waiting to acquire or holding $\mathcal{C}_g^k$. The rules under which J_i acquires $\mathcal{C}_g^k$ are stated below.

- **C1** R_j is enqueued in PQ_g^k when it starts requiring $\mathcal{C}_g^k$.
- **C2** Whenever $\mathcal{C}_g^k$'s slot is free, the highest-priority reservation queued in PQ_g^k (if any) is dequeued from PQ_g^k and moved to $\mathcal{C}_g^k$'s slot. The reservation becomes *slotted*.
- **C3** J_i acquires $\mathcal{C}_g^k$ when **(a)** its reservation R_j is slotted and **(b)** no other job is holding $\mathcal{C}_g^k$.

The significance of the slot concept may not be obvious at first, but will become more apparent after we introduce the delegation and abortion rules (in Sections IV-B and IV-C). In short, the slot mechanism serves to protect waiting reservations from excessive budget drain in the case of repeated budget depletions (by higher-priority reservations). More specifically, the slot will be essential to establishing Lemmas 5 and 6.

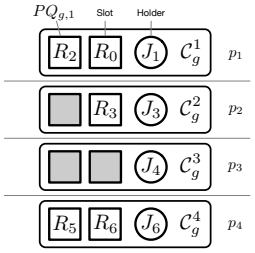
Phase 2. The *request-service phase* resolves inter-processor contention. During this phase, IPC context holders are sequenced and their requests processed by servers in a deadlock-free manner. Deadlock freedom is ensured by protocol rules and the server ticket abstraction (see Theorem 1 in Section V). Phase 2 starts when Rule **C3** takes effect.

Each server group contains a *group queue* of IPC contexts, denoted as GQ. As mentioned previously, IPC contexts track the state of C2S requests. The state stored in an IPC context $\mathcal{C}_g^k$ consists of a timestamp $ts(\mathcal{C}_g^k)$, a multiset of server tickets $\mathcal{T}(\mathcal{C}_g^k)$, and a call stack. We denote by $\mathcal{R}_{k,j}$ the j-th request on $\mathcal{C}_g^k$'s call stack. It follows that $\mathcal{C}_g^k$ tracks the progress of $\mathcal{R}_{k,1}$. We call the request on top of $\mathcal{C}_g^k$'s stack its *current request*.

During Phase 2, C2S and S2S requests are processed in essentially the same way. The only difference is that, before emitting a C2S request, the corresponding IPC context must be initialized and enqueued in GQ.

- **S1** Upon being acquired by job J_i at time t, the IPC context $\mathcal{C}_g^k$ is first initialized and then enqueued in GQ. After $\mathcal{C}_g^k$'s initialization, $ts(\mathcal{C}_g^k) = t$ and $\mathcal{T}(\mathcal{C}_g^k) = \mathcal{L}_o$. We assume a strict total order on timestamps.
- **S2** When `gip_invoke` is called to issue a request $\mathcal{R}_{k,j}$ to a server S_q, $\mathcal{R}_{k,j}$ is pushed onto $\mathcal{C}_g^k$'s call stack and S_q is notified of a request arrival. `gip_invoke` fails if $\mathcal{T}(\mathcal{C}_g^k)$ does not contain a ticket for S_q or S_q is already serving a request on $\mathcal{C}_g^k$'s call stack.
- **S3** A C2S request $\mathcal{R}_{k,1}$ becomes *committed* when $\mathcal{C}_g^k$ is enqueued in GQ and there is no IPC context $\mathcal{C}_g^b$ in GQ such that **(a)** $ts(\mathcal{C}_g^b) < ts(\mathcal{C}_g^k)$ and **(b)** $\mathcal{T}(\mathcal{C}_g^b) \cap \mathcal{T}(\mathcal{C}_g^k) \neq \varnothing$.
- **S4** A request $\mathcal{R}_{k,j}$ is served by the respective resource server when $\mathcal{R}_{k,1}$ is committed and $\mathcal{R}_{k,j}$ is on $\mathcal{C}_g^k$'s call stack.
- **S5** $\mathcal{C}_g^k$'s slot is freed when $\mathcal{R}_{k,1}$ becomes committed. We say the reservation *ceases to be slotted*.
- **S6** When `gip_reply` is called, $\mathcal{R}_{k,j}$ is popped from $\mathcal{C}_g^k$'s call stack and a ticket for S_q is consumed from $\mathcal{T}(\mathcal{C}_g^k)$.
- **S7** When J_i exits Phase 2 (*i.e.*, on $\mathcal{R}_{k,1}$'s completion), $\mathcal{C}_g^k$ is dequeued from GQ and released.

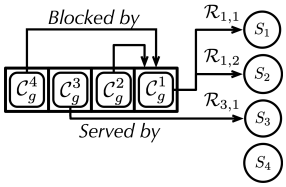
An example of Phase 2 is depicted in Fig. 2b. It illustrates the non-work-conserving behavior of the G(IP)^{2}C protocol: S_4 is stalling even though $\mathcal{R}_{2,1}$ is waiting for it. However, $\mathcal{R}_{3,1}$ is served even though a blocked request has been emitted earlier, showing that the G(IP)^{2}C protocol allows some degree of in-group parallelism. Fig. 2a shows that the slots of committed IPC contexts are not occupied by the reservation of their holder.

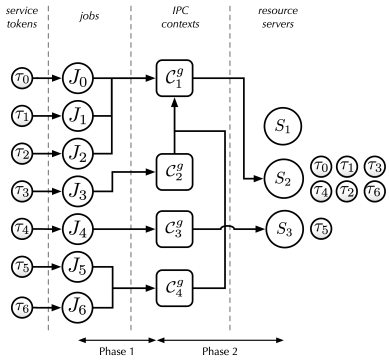


(a) IPC contexts and their slots and queues in Phase 1.

	ts	Tickets	Curr. req.
$\mathcal{C}_g^4$	7	$\{S_1, S_2\}$	$\mathcal{R}_{4,1} \to S_2$
$\mathcal{C}_g^3$	4	$\{S_3\}$	$\mathcal{R}_{3,1} \to S_3$
$\mathcal{C}_g^2$	3	$\{S_4\}$	$\mathcal{R}_{2,1} \to S_4$
$\mathcal{C}_g^1$	1	$\{S_1, S_2, S_4\}$	$\mathcal{R}_{1,2} \to S_2$

(b) The group queue used in Phase 2.





(c) Bandwidth-delegation graph.

Fig. 2: Example illustrating a snapshot of the different structures of the server group g during IPC. In this example, each reservation R_j contains a task T_j and R_i has lower priority than R_j if $i < j$. The resource server S_1 is blocked waiting for S_2 to reply to $\mathcal{R}_{1,2}$.

B. Bandwidth Delegation Rules

The core sequencing rules are insufficient as they do not consider any notion of progress. We now augment them with bandwidth delegation rules to support the passive server model. These rules define how jobs delegate service tokens to resource servers through different bandwidth proxies (as depicted in Fig. 2c): in Phase 1, jobs delegate their bandwidth to IPC contexts; in Phase 2, IPC contexts delegate their bandwidth to the IPC contexts blocking them or to resource servers.

Phase 1. Jobs simply delegate their bandwidth to the IPC context they hold or are waiting to acquire.

D1 J_i delegates bandwidth to $\mathcal{C}_g^k$ as long as R_j requires $\mathcal{C}_g^k$.

Phase 2. During Phase 2, we must ensure that resource servers receive bandwidth from the IPC contexts of the requests they are serving, and that not-yet-committed IPC contexts delegate their bandwidth to requests blocking them.

Before stating a rule for the latter case, let us introduce some terminology for blocking during Phase 2. Let $\mathcal{B}(\mathcal{C}_g^k)$ be the set of the IPC contexts preventing $\mathcal{R}_{k,1}$ from being committed, *i.e.*, IPC contexts that precede $\mathcal{C}_g^k$ in GQ and whose ticket multiset overlaps with $\mathcal{C}_g^k$'s. We say that $\mathcal{C}_g^b$ *blocks* $\mathcal{C}_g^k$ *directly* if it is the element of $\mathcal{B}(\mathcal{C}_g^k)$ with the latest timestamp. Since timestamps are ordered totally, only one IPC context can directly block $\mathcal{C}_g^k$. Blocking is transitive, thus any IPC context $\mathcal{C}_g^d$ blocking $\mathcal{C}_g^b$ also blocks $\mathcal{C}_g^k$. We then say that $\mathcal{C}_g^d$ *blocks* $\mathcal{C}_g^k$ *transitively*.

The delegation rules for Phase 2 are defined below.

D2 A committed IPC context $\mathcal{C}_g^k$ delegates its bandwidth to the server handling the request on top of $\mathcal{C}_g^k$'s call stack.

D3 If $\mathcal{C}_g^k$ is not committed and queued in GQ, it delegates its bandwidth to the IPC context $\mathcal{C}_g^b$ directly blocking it.

Fig. 2c shows an example of a bandwidth-delegation graph obtained during the two phases. Ultimately, resource servers receive the service tokens of all jobs directly or indirectly waiting for them to finish processing the currently served request.

C. Request Abortion Rules

There are two strategies for handling budget exhaustion during IPC: checking budgets beforehand or aborting requests emitted by jobs of exhausted reservations. The first strategy prevents budget exhaustion during IPC and does not require additional rules, but it places a significant configuration burden on the system integrator. The second strategy requires additional rules that we present now.

When a task exhausts its reservation budget during IPC, the top-level scheduler tries to abort the C2S request of the corresponding job. A C2S invocation request can be aborted only if it is not yet committed. Otherwise, the request must be carried out to completion.

The action taken to abort a request depends on the stage reached by the job in the invocation process.

X1 If R_j exhausts its budget while it is queued in PQ_g^k, then R_j is dequeued from PQ_g^k and J_i's request is aborted.

X2 If R_j exhausts its budget while it is slotted, then R_j ceases to be slotted.

X3 If R_j exhausts its budget while J_i is holding $\mathcal{C}_g^k$ and $\mathcal{R}_{k,1}$ is not committed yet, then $\mathcal{R}_{k,1}$ is aborted and $\mathcal{C}_g^k$ is dequeued from GQ and released.

With the protocol's main rules in place, we next analyze its properties. In particular, we bound the budget drain over the course of a C2S request and show that progress is ensured.

V. G(IP)^{2}C: Budget Drain and Deadlock Freedom

In the following analysis, we reuse the context in which the protocol rules have been expressed: job J_i is contained in a reservation R_j assigned to processor p_k and invokes a resource server S_q contained in group g to carry out operation o. We assume that `gip_reply` and `gip_wait` are performed atomically as part of a single `gip_reply_wait` system call.

We define the following times in the lifecycle of J_i's request.

- At $t = t_1$, J_i initiates Phase 1.
- At $t = t_2$, R_j is slotted.
- At $t = t_3$, J_i initiates Phase 2.
- At $t = t_4$, $\mathcal{R}_{k,1}$ is committed.
- At $t = t_5$, $\mathcal{R}_{k,1}$ completes.

We further denote by L^{max} the total length of the longest server operation in the group g, *i.e.*, the maximum end-to-end budget required in total by all servers involved in handling this operation in the absence of contention.

Requests that are pruned due to budget exhaustion lose all progress and must be reissued, which resets the analysis. In the following, we thus assume that the reservation under analysis R_j does not exhaust its budget during $[t_1, t_5)$, but place no assumptions on the behavior of any contending reservations.

First, we note the absence of deadlock.

Theorem 1. *The G(IP)^{2}C protocol is deadlock-free.*

Proof. By contradiction. Consider a resource server S_q waiting for a resource server S_r and suppose that S_r is directly (or indirectly) waiting for S_q (*i.e.*, there is a cycle in the *wait-for* graph). By the definition of server groups, the two servers belong to the same group g. Let $\mathcal{C}_g^k$ and $\mathcal{C}_g^l$ be the IPC contexts of the request served by S_q and S_r, respectively. By Rule **S4**, $\mathcal{C}_g^k$ and $\mathcal{C}_g^l$ both track committed requests. By Rules **S2** and **S6**, a server ticket is required to invoke a server, and consumed only after the corresponding request has been completed. Hence, the ticket multisets of both $\mathcal{C}_g^k$ and $\mathcal{C}_g^l$ include a ticket for S_r. By Rule **S3**, neither S_q nor S_r can process a request tracked by an IPC context with an earlier timestamp and an intersecting ticket multiset. It follows that $\mathcal{C}_g^k$ and $\mathcal{C}_g^l$ must be the same IPC context. However, by Rule **S2**, S_r is not serving any request currently on $\mathcal{C}_g^k$'s call stack, so $\mathcal{C}_g^k$ and $\mathcal{C}_g^l$ are distinct. Contradiction. □

Having observed that all requests *eventually* progress, we next consider the progress of J_i's request more closely. We say that J_i's request *makes progress* whenever **(i)** the server processing J_i's C2S request, **(ii)** a server processing an S2S request resulting from J_i's C2S request, or **(iii)** a server processing a request blocking J_i's request has a service token in its bandwidth that allows it to be scheduled.

Since the progress of J_i during Phase 1 depends on the progress of the job holding the IPC context J_i is waiting for, we proceed backward and start by analyzing Phase 2.

Lemma 1. *During Phase 2, the request of the job holding an IPC context $\mathcal{C}_g^k$ makes progress whenever a reservation requiring $\mathcal{C}_g^k$ drains budget.*

Proof. Let J_h denote the job holding $\mathcal{C}_g^k$. All jobs requiring $\mathcal{C}_g^k$ delegate their bandwidth to $\mathcal{C}_g^k$. Therefore, when a reservation of any of these jobs drains budget, J_h's request makes progress if $\mathcal{C}_g^k$ delegates its bandwidth to a resource server handling J_h's request or a request preventing J_h's request from being committed. If $\mathcal{C}_g^k$ is committed, then the server handling $\mathcal{C}_g^k$'s current request receives $\mathcal{C}_g^k$'s bandwidth (Rule **D2**) and the claim follows. Otherwise, there is a unique IPC context $\mathcal{C}_g^b$ directly blocking $\mathcal{C}_g^k$ and therefore receiving $\mathcal{C}_g^k$'s bandwidth (Rule **D3**). If $\mathcal{C}_g^b$ is not committed, it delegates its bandwidth to the only job directly blocking it. This process repeats until $\mathcal{C}_g^k$ delegates its bandwidth to a committed IPC context blocking it transitively. By Rule **D2**, the resource server handling this committed IPC context receives $\mathcal{C}_g^k$'s bandwidth. □

Building on Lemma 1, we observe that progress is guaranteed throughout the whole request sequence.

Lemma 2. *During $[t_1, t_5)$, whenever R_j drains budget, J_i's request makes progress.*

Proof. During $[t_1, t_5)$, J_i requires $\mathcal{C}_g^k$ and by Rule **D1**, it delegates its bandwidth to $\mathcal{C}_g^k$. By Lemma 1, the request of $\mathcal{C}_g^k$'s holder makes progress whenever R_j drains bandwidth. It follows that, during $[t_1, t_3)$, J_i makes progress towards IPC context acquisition. Then, during $[t_3, t_5)$, J_i holds $\mathcal{C}_g^k$, therefore J_i's request makes progress. □

We next bound the budget drained by J_i when it invokes S_q. For ease of analysis, we again proceed backward in time, starting with the interval $[t_4, t_5)$.

Lemma 3. *R_j drains at most $B_4 \triangleq L^{max}$ units of budget during the interval $[t_4, t_5)$.*

Proof. By Theorem 1, there is no deadlock. In particular, once $\mathcal{R}_{k,1}$ is committed, it is not delayed by other requests as, by Rule **S3**, there is no committed IPC context in GQ with an intersecting ticket multiset. By Lemma 2, $\mathcal{R}_{k,1}$ makes progress whenever R_j drains budget. As servers processing J_i's C2S request and any resulting S2S requests require at most L^{max} units of budget in total to complete the request, R_j drains at most L^{max} units of budget during $[t_4, t_5)$. □

Lemma 4. *R_j drains at most $B_3 \triangleq (m-1) \cdot L^{max}$ units of budget during the interval $[t_3, t_4)$.*

Proof. The IPC contexts preventing $\mathcal{R}_{k,1}$ from being committed are necessarily preceding $\mathcal{C}_g^k$ in GQ. Since at most m IPC contexts can be acquired at a time, and they are enqueued by timestamp order in GQ, at most $m-1$ IPC contexts precede $\mathcal{C}_g^k$ in GQ. Since $\mathcal{R}_{k,1}$ makes progress whenever R_j drains budget, it follows analogously to Lemma 3 that R_j drains at most $(m-1) \cdot L_{max}$ units of budget before $\mathcal{R}_{k,1}$ is committed. □

Lemma 5. *R_j drains at most $B_2 \triangleq L^{max}$ units of budget during the interval $[t_2, t_3)$.*

Proof. At time t_2, R_j is slotted. If J_i acquires $\mathcal{C}_g^k$ immediately due to Rule **C3**, then $t_2 = t_3$. Otherwise, another job is holding $\mathcal{C}_g^k$ at time t_2. By Rules **C3** and **S5** in the regular case, and by Rules **X2** and **X3** in the case of budget exhaustion, the request of the job holding $\mathcal{C}_g^k$ is already committed at time t_2. By Rule **C2**, analogously to Lemma 3, this request completes after R_j drains at most L^{max} units of budget. □

Lemma 6. *R_j drains at most $B_1 \triangleq m \cdot L^{max}$ units of budget during the interval $[t_1, t_2)$.*

Proof. After having drained $m \cdot L^{max}$ units of budget, either t_2 is reached and R_j is slotted, in which case the claim holds trivially, or R_j is still waiting to be slotted. In the latter case, if J_i is still waiting after R_j has drained $m \cdot L^{max}$ units of budget, then the job of the reservation occupying $\mathcal{C}_g^k$'s slot at time t_1 has been delegated sufficient budget to acquire $\mathcal{C}_g^k$ (analogously to Lemma 5) and have its request committed (analogously to Lemma 4). Consequently, the reservation now occupying $\mathcal{C}_g^k$'s slot has been slotted no earlier than at time t_1. According to Rule **C2**, this reservation thus has higher priority than R_j.

Therefore, R_j is not the highest-priority active reservation on its assigned processor and thus does not drain budget. □

Jointly, the above lemmas yield the end-to-end budget-drain bound $B \triangleq B_1 + B_2 + B_3 + B_4$, which we note as Theorem 2.

Theorem 2. *Under the G(IP)^{2}C protocol, R_j drains at most $B = (2m+1) \cdot L^{max}$ units of budget during $[t_1, t_5)$.*

Theorem 2 establishes *temporal isolation with bounded interference*: the maximum amount of budget drained when accessing a shared server is bounded by B irrespective of the behavior of any other clients of the server. In particular, note that Theorem 2 places no assumptions on the number of competing clients, their access patterns, on which processors they reside, and whether they exhaust their reservation's budget.

By design, the G(IP)^{2}C protocol ensures an even stronger property, namely *complete temporal isolation* at the group level: a reservation is entirely unaffected by contention in server groups its constituent tasks do not access.

Theorem 3. *Under the G(IP)^{2}C protocol, if task T_i does not invoke any resource server in group g, then IPC calls to resource servers in g do not affect R_j's budget consumption.*

Proof. By design, the only way for a resource server to drain budget on behalf of another task is through bandwidth delegation, which is managed at the server group level. Therefore, if T_i does not invoke a resource server in g, no resource server receives service tokens from T_i's pending jobs, and hence none of R_j's budget is consumed by servers in g. □

This concludes our analysis of the core G(IP)^{2}C protocol.

VI. G(IP)^{2}C: Protocol Extensions

In this section, we define additional rules adding support for reservations containing multiple tasks and background jobs.

A. Multi-Occupancy Reservation Support

Supporting multiple tasks per reservation requires managing contention at an additional level. To that end, we introduce a *Phase 0* in which jobs contend for a *right-to-invoke token* (RIT) before competing for an IPC context. There is one such RIT per server group in each reservation. An RIT represents a job's right to use its reservation budget to engage in IPC.

Priority inversion may occur during Phase 0, causing a job to consume budget while it waits for an RIT. We address this issue by introducing *RIT stealing*: an acquired RIT *can be stolen* unless its reservation is slotted. The highest-priority job of a reservation can *acquire* an already held RIT if it can be stolen. As we establish below, with this mechanism in place, a job does not consume more budget to acquire an IPC context than it would if it was the only job in its reservation.

Let us define the rules governing Phase 0. For each server group and each reservation, there is one RIT and an associated priority queue of jobs. We denote by λ_g^j reservation R_j's RIT for the server group g and by $RQ_{g,j}$ the priority queue sequencing access to it. Suppose J_i and J_k are two jobs in R_j competing for λ_g^j: J_i acquires λ_g^j under the rules below.

- **R1** J_i is *enqueued* in RQ_g^j when it starts requiring λ_g^j. Jobs in RQ_g^j are ordered according to the intra-reservation priority of their assigned service token.
- **R2** J_i acquires λ_g^j when **(a)** J_i is the head of RQ_g^j and **(b)** λ_g^j can be stolen or no job is holding it.
- **R3** J_i remains enqueued in RQ_g^j as long as it requires λ_g^j.
- **R4** When a job J_k is holding λ_g^j when J_i steals it by Rule **R2**, then J_k releases λ_g^j and J_i starts holding it instead.

A job delegates its bandwidth to an RIT from the start of Phase 0 until the end of its invocation request.

- **D5** J_i delegates its bandwidth to λ_g^j from the moment it enters RQ_g^j to the moment its invocation request completes.

The rules of Phase 1 need some adaptations. First of all, we must modify the acquisition and the delegation rules to consider RITs instead of jobs.

- **C3'** The job holding λ_g^j acquires $\mathcal{C}_g^k$ when **(a)** its reservation R_j occupies $\mathcal{C}_g^k$'s slot and **(b)** no other job is holding $\mathcal{C}_g^k$.
- **D1'** λ_g^j delegates bandwidth to $\mathcal{C}_g^k$ as long as R_j requires $\mathcal{C}_g^k$.

Then, additional rules are required to define when an RIT can be stolen and when it is released.

- **C4** λ_g^j cannot be stolen while R_j is slotted.
- **C5** J_i releases λ_g^j when its request is committed.

We assume that, if any job is waiting for λ_g^j when λ_g^j is released, then λ_g^j is reacquired and R_j is re-enqueued in PQ_g^k right away. In other words, no reservation with priority lower than R_j's is slotted while some job in R_j is waiting for λ_g^j and R_j does not deplete its budget.

We extend the budget-drain analysis presented in Section V to Phase 0. We say that job J_i *consumes its reservation's R_j's budget* when R_j drains budget and J_i is R_j's highest-priority pending job. Let t_0 denote the time at which J_i initiates Phase 0. We redefine t_1 to be the time at which J_i initiates Phase 1, and t_2 to be the first point in time at which R_j is slotted and J_i holds λ_g^j. After t_2, λ_g^j cannot be stolen anymore and the analysis remains unchanged. We assume the reservation under analysis R_j does not exhaust its budget during $[t_0, t_5)$.

Lemma 7. *If J_i consumes R_j's budget during $[t_0, t_1)$, then J_i does not consume budget during $[t_1, t_2)$.*

Proof. By contradiction. Suppose J_i has consumed R_j's budget at some point during $[t_0, t_1)$ and there exists a time $t \in [t_1, t_2)$ at which J_i also consumes budget. J_i consumes budget during $[t_0, t_1)$ only if it is the head of RQ_g^j and cannot acquire λ_g^j, which by Rules **R2** and **C4** implies that R_j *is* slotted at time t. Since $t < t_2$, by the definition of t_2, R_j is *not* slotted. Thus, some other reservation is slotted at time t. Since J_i is waiting for or holding λ_g^j throughout $[t_0, t_2)$, $\mathcal{C}_g^k$'s slot is not occupied by a lower-priority reservation at time t. It follows that $\mathcal{C}_g^k$'s slot is occupied by a higher-priority reservation. Therefore, R_j is not the highest-priority reservation on its processor and thus does not drain budget at time t. Contradiction. □

Lemma 8. *J_i consumes at most $m \cdot L^{max}$ units of R_j's budget during $[t_0, t_1)$.*

Proof. By contradiction. Suppose there exists a time $t \in [t_0, t_1)$ at which J_i is consuming budget and has consumed more than $m \cdot L^{max}$ units of budget in total. First observe that J_i consumes R_j's budget during $[t_0, t_1)$ only if it is RQ_g^j's head and cannot acquire λ_g^j, which implies that λ_g^j is held by a lower-priority job and λ_g^j cannot be stolen. Therefore, at any point in time that J_i consumes budget, R_j is slotted. At time t, R_j has thus been slotted for more than $m \cdot L^{max}$ time units. It follows analogous to Lemmas 4 and 5 that the job holding λ_g^j when J_i first required λ_g^j must have already released it by time t, but by the definition of t_1, J_i did not acquire it. Consequently, at time t, λ_g^j is held by a higher-priority job, and J_i thus does not consume R_j's budget. Contradiction. □

Lemma 9. *If another job steals λ_g^j from J_i at a time $t \in [t_1, t_2)$, then J_i does not consume R_j's budget during $[t, t_2)$.*

Proof. By contradiction. Suppose there exists a time $t' \in [t, t_2)$ at which J_i drains budget and t_2 has not been reached yet. If a job steals λ_g^j from J_i, then by Rule **R2**, J_i is not the highest-priority job in R_j. J_i thus does not consume budget until it again becomes the highest-priority job in R_j, which happens only after the higher-priority job that stole λ_g^j releases λ_g^j, which by Rules **C3'** and **C5** happens only if R_j is slotted. By the definition of t_2, however, R_j is not slotted at time t'. Recall that, when λ_g^j is released, no reservation with a priority lower than R_j can occupy $\mathcal{C}_g^k$'s slot if RQ_g^j is not empty. Therefore, at time t', since $t' < t_2$, $\mathcal{C}_g^k$'s slot is occupied by a higher-priority reservation. Therefore, R_j does not drain budget and J_i does not consume R_j's budget at time t'. Contradiction. □

Lemma 10. *J_i consumes at most $m \cdot L^{max}$ units of budget during $[t_1, t_2)$.*

Proof. If no job steals λ_g^j from J_i, then the claim follows analogously to Lemma 6. Otherwise, let t be the time at which λ_g^j is stolen from J_i for the first time during $[t_1, t_2)$. By Lemma 9, J_i does not consume budget after time t. Further, J_i consumes no more than $m \cdot L^{max}$ units of budget during $[t_1, t)$ since otherwise t_2 is reached (analogously to Lemma 6). □

Lemma 11. *J_i consumes at most $m \cdot L^{max}$ units of R_j's budget during $[t_0, t_2)$.*

Proof. By case analysis. If J_i consumes budget during $[t_0, t_1)$, then by Lemma 7 J_i consumes no budget during $[t_1, t_2)$ and by Lemma 8 at most $m \cdot L^{max}$ units of budget during $[t_0, t_1)$.

Otherwise, if J_i does not consume budget during $[t_0, t_1)$, then by Lemma 10 J_i consumes at most $m \cdot L^{max}$ units of budget during $[t_1, t_2)$. □

Lemma 11 replaces Lemma 6 in the sequence of bounds leading to Theorem 2, which establishes that temporal isolation is preserved even if multiple tasks share a reservation.

The next extension provides support for background tasks.

B. Background Tasks Support

Background tasks are low-priority tasks served in a best-effort fashion. Background jobs follow the same invocation process as regular ones, with two major differences. The first difference is that background jobs always have lower priority than non-background (*i.e.*, *regular*) jobs.

We also consider reservations whose RIT is held by a background job to have lower priority than regular reservations *for the purposes of the IPC protocol rules* (*i.e.*, the behavior of the reservation scheduler is *not* affected). We call such reservations *background reservations*.

The second difference is that background jobs with not-yet-committed requests cede any protocol resources they hold (RIT or IPC context) to regular jobs. A background job's request is aborted when it cedes a protocol resource and is later reissued.

Background job support is governed by the rules below. In these rules, J_b is a background job in reservation R_j.

- **B1** Background jobs have lower priority than regular jobs.
- **B2** During Phase 1, background reservations are considered to have lower priority than regular reservations.
- **B3** If J_b holds λ_g^j, and a regular job starts requiring λ_g^j, then J_b releases λ_g^j.
- **B4** If J_b holds λ_g^j, R_j occupies $\mathcal{C}_g^k$'s slot and there is at least one regular job requiring λ_g^j or $\mathcal{C}_g^k$, then R_j vacates $\mathcal{C}_g^k$'s slot, J_b releases $\mathcal{C}_g^k$ (if J_b is holding it), and R_j is reenqueued in PQ_g^k.

These rules ensure that background jobs do not increase the budget-drain bound established for regular jobs.

Lemma 12. *Under the above rules, Theorem 2 holds for regular jobs even in the presence of background jobs.*

Proof. Consider a regular job J_i and a background job J_b. During Phase 0 and Phase 1, unless J_b's request is already committed, J_b (or the reservation containing J_b) releases any resource it holds (RIT, slot, or IPC context) when they are required by J_i (or the reservation containing J_i). If J_b's request is already committed, then it is equivalent in effect to a regular lower-priority job's committed request and thus already accounted for. Therefore, J_b does not cause additional delay during Phase 0 or Phase 1. During Phase 2, J_b can delay J_i's request by preventing it from being committed. However, the bound on the maximum delay in Phase 2 derives from the limited number of available IPC contexts, which is not affected by J_b's background status, and hence still holds. □

Support for background tasks offers an alternative to request abortion in the event of a reservation exhausting its budget during IPC. When budget exhaustion occurs, the associated task is demoted to background status instead of aborting the request. When the reservation budget is replenished, the task becomes regular again. This strategy gives a chance for the request to complete in a best-effort manner before otherwise possible, and makes it unnecessary to expose the abortion and reissuing of requests to the programmer.

VII. Evaluation

To assess the proposed protocol and its analysis in a practical setting, and to experimentally justify the necessity of Phases 1 and 2, we implemented a prototype in LITMUS^{RT} [3, 9]. Specifically, we explored three questions: **(1)** how tight is the analytical budget-drain bound in practice, **(2)** what is the impact of eliding Phase 1, and **(3)** what is the impact of employing a work-conserving request selection policy during Phase 2? We first describe our evaluation platform, then introduce the employed benchmark and experimental protocol, and finally present three experiments answering the above questions.

Platform. We performed experiments on an Intel platform with a 4-core i5-4590 processor. We implemented the G(IP)^{2}C protocol in the latest release of LITMUS^{RT}, which is based on Linux 4.9. Our implementation is based on LITMUS^{RT}'s P-RES plugin, which implements partitioned reservation-based scheduling. Our prototype supports background tasks as described in Section VI-B and demotes jobs of depleted reservations to background status instead of aborting their requests.

Regarding budget exhaustion, our implementation slightly deviates from the protocol specification. Our prototype extends a pre-existing variant of the P-RES plugin supporting *non-transitive* bandwidth inheritance. We added support for transitive bandwidth delegation on top of the existing implementation by maintaining a delegation graph that is consulted to set up direct bandwidth inheritance between clients and servers as needed. Unfortunately, LITMUS^{RT}'s pre-existing *state-listener facility* used to notify the protocol implementation of budget exhaustion during IPC does not support the reconfiguration of bandwidth inheritance in the budget-exhaustion handler (due to invariants related to Linux's scheduler locks).

This limitation becomes an issue if a client waits to acquire the IPC context assigned to a reservation that depletes its budget. Instead of redesigning the existing bandwidth inheritance mechanism, we instead employed an alternative *lazy group-queue cleanup strategy*: IPC contexts of demoted requests are pruned from the group queue during `gip_reply`.

Since at most L^{max} budget units are drained before the queue is cleaned up, waiting for an IPC context to be removed from the group queue is equivalent to waiting for a committed IPC context to be released. Thus, the worst-case behavior of this workaround is equivalent to the worst-case behavior of the specified protocol. Note that this issue stems purely from our desire to reuse an existing codebase with certain design limitations; a clean implementation "from scratch" can proceed without such workarounds.

Benchmark. To easily compare different resource-server topologies and request patterns, we evaluated the G(IP)^{2}C protocol using a synthetic application exhibiting configurable runtime behavior. Our synthetic application involves clients invoking a chain of q resource servers. We call each such server in the chain a *link*. When the i-th server in the chain receives a message, it first invokes its successor (if any), and then spins before sending a reply.

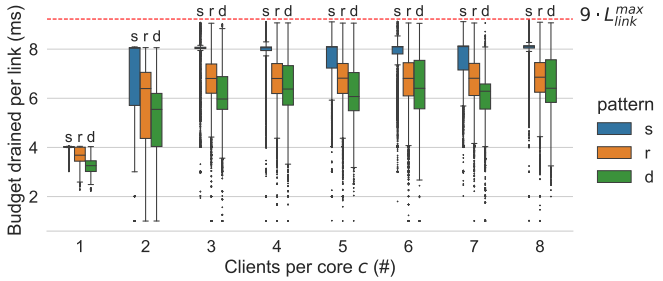


Fig. 3: Box plot showing the budget drained per link in Experiment 1. The bottom and top whiskers show the first and fourth quartiles, respectively; the box indicates the second and third quantiles. The median is shown as a horizontal line, and outliers below the first quartile are shown as points.

The spinning time depends on whether it is handling a C2S or S2S request. In the case of a C2S request, the server spins for i milliseconds. Otherwise, the server spins for one millisecond before sending a reply. Therefore, the budget needed to complete a C2S request is roughly equal to $L^{max} = q$ milliseconds regardless of the invoked server's position. This design allows us to compare the budget drained under different resource servers topologies (which is determined by the chain length) using a common metric: the *budget drain per link* (*i.e.*, the measured budget drain normalized by q). Correspondingly, the budget required per link is $L^{max}_{link} = L^{max}/q$, which by design is roughly one millisecond plus a small margin to account for overheads and spin-loop inaccuracy.

Experimental protocol. We measured the budget drained by a client under different contention scenarios. Our system is partitioned, and the same number of clients c is assigned to each core. Each client is contained in a sporadic polling reservation with a unique fixed priority. We measure the budget drained by the highest-priority client assigned to core 0. We call this client the *observed client* and the other clients the *contenders*. Unlike the contenders, the observed client checks that it has sufficient budget before invoking a resource server.

In all experiments, the observed client always invokes the first server in the chain. The server invoked by contenders is determined by three different *calling patterns*: under the s pattern, all clients call the first server in the chain; under the r pattern, contenders select a resource server randomly before each invocation; finally, under the d pattern, clients are distributed evenly along the chain by priority and then by assigned core. Specifically, using zero-based indexing, under the d pattern, the i-th client of the k-th core invokes the server at position $p = (k \cdot c + i) \mod q$.

The s pattern is a pathological case that does not allow for any in-group parallelism. Conversely, the d pattern allows for in-group parallelism because clients are distributed along the chain, and resource servers spin *after* receiving the reply to their S2S requests, which leaves time windows in which later-issued requests can be processed in parallel. Finally, the r policy falls between the s and d strategies.

We measured the budget drained by the observed client during each IPC request it sent to the server chain. We discarded

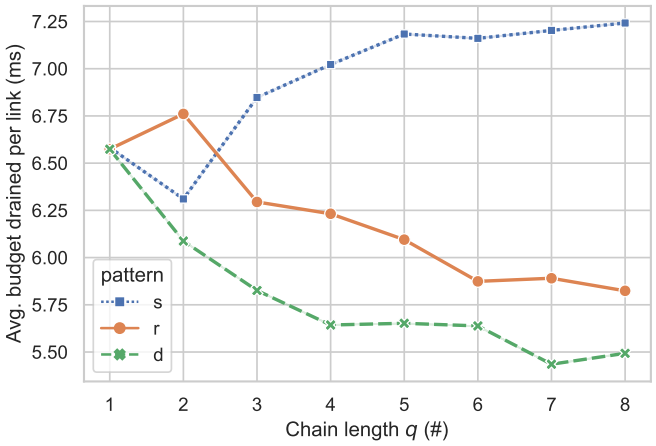


Fig. 4: Average budget drain per link in Experiment 1.

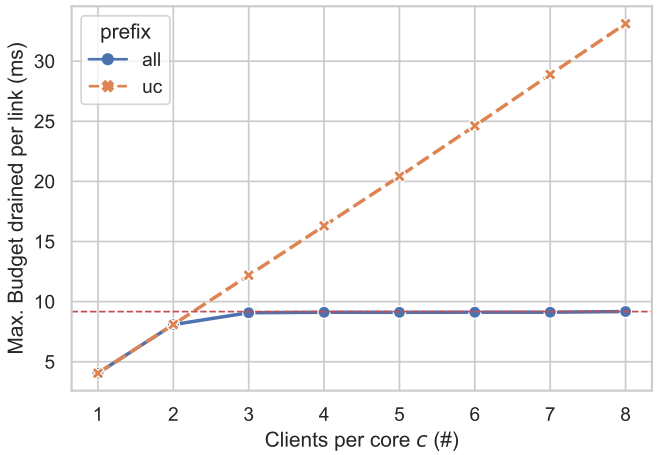


Fig. 5: Maximum budget drain per link in Experiment 2.

the first measurement since resource servers may not be ready to serve requests at the beginning of the experiment. Finally, we determined the budget required per link to be $L_{link}^{max} = 1.025\,\text{ms}$ by applying our experimental protocol without contenders.

Experiment 1: Tightness of the budget-drain bound. Our first experiment aimed at empirically assessing the tightness of the analytical budget-drain bound. We applied the experimental protocol for chain lengths $q \in \{1, \ldots, 8\}$, up to eight clients per core, and all calling patterns (s, r, and d). We call a combination of these three parameters a *contention scenario*. Each reservation was provisioned with enough budget for three IPC calls. In each of the 192 contention scenarios, the observed client sent 1000 messages, yielding 191808 samples in total.

The measured budget drain per link is depicted in Fig. 3. Foremost, we observe that samples close to the analytical upper bound of $(2m+1) \cdot L_{link}^{max}$ are indeed observable in practice, which suggests that the analysis is not pessimistic.

We can also observe that the protocol works as expected when there are only one or two clients per core. When there is only one client per core, no budget is drained during Phase 1. Therefore, the budget-drain bound per link is $4 \cdot L_{link}^{max} = 4.1ms$ in this case. If there are two clients per core, then the expected bound is $8 \cdot L_{link}^{max} = 8.2ms$. Indeed, the worst-case bound is reached only when a client waits for a slot occupied by a reservation waiting to acquire an IPC context. This scenario implies the presence of three clients on the same core, as validated by Fig. 3, which shows the observed maxima to reach a plateau close to the analytical bound for $c \geq 3$.

While the observed maxima do not vary with the calling pattern, the distributions of observed samples clearly differ. First, we observe that, in the case of the s pattern, most of the values are concentrated in the second and third quartiles (*i.e.*, the boxes are small in Fig. 3). This is not surprising since all clients call the first server, so all nested calls are serialized.

Notice also how, for each c, the highest and the lowest median values are always observed for the s and d patterns, respectively. This is again unsurprising because, as mentioned earlier, the s pattern does not allow for any in-group parallelism, whereas the d pattern does. Fig. 4, which shows the mean observed budget drain per link, confirms this intuition. The gap between the d and the s patterns clearly grows with c.

The r pattern performs slightly worse than the d pattern, but the average observed budget-drain per link still follows a similar trend. This result suggests that one may expect some benefits from parallelism also in practical systems with complex server topologies and irregular calling patterns.

Experiment 2: Effect of IPC context acquisition. A significant part of the budget drained during an IPC invocation is expended during Phase 1. To study the impact of bypassing this phase, we implemented a protocol version in which Phase 1 is elided. We call this version G(IP)^{2}C-UC. Instead of contending for a shared, processor-local IPC context, IPC contexts are picked from a global pool containing a sufficient number of IPC contexts to satisfy the demand of all clients. This modification has two effects: first, Phase 1 is elided; and second, the number of IPC contexts in the group queue is unconstrained and thus bounded only by the total number of clients.

We applied the experimental protocol to the G(IP)^{2}C-UC variant for the s pattern (*i.e.*, all clients invoke the first server in the chain), sending 1000 requests per contention scenario, giving us 63936 samples in total. The results are presented in Fig. 5, which shows the largest observed budget drain per link.

Under the the G(IP)^{2}C-UC variant, the maximum observed budget drain clearly scales with c, and is never lower than under the full G(IP)^{2}C as defined in Section IV. Furthermore, note that, in mixed-criticality systems, the total number of clients is not necessarily known in advance, and when it is, it is not necessarily reliable or trusted. Consequently, the observed scaling, and hence the implied dependency on a trustworthy number of clients, is undesirable in this context.

Overall, Experiment 2 clearly shows the benefits of Phase 1: sequencing intra-processor contention by providing only one IPC context per core renders the maximum budget-drain bound independent of the total number of contenders.

Experiment 3: Effect of non-work-conservation. The non-work-conserving request selection strategy employed in Phase 2, specifically the joint effect of Rules **S3** and **S4**, limits the degree of parallelism achieved by the G(IP)^{2}C. In the third and

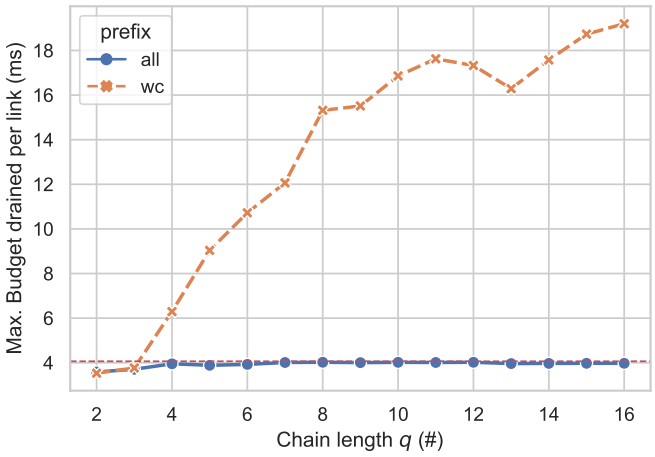


Fig. 6: Maximum budget drain per link in Experiment 3.

final experiment, we therefore sought to study how the system behaves if we employ a work-conserving approach instead.

To that end, we implemented a variant of the G(IP)^{2}C protocol that allows resource servers to process the first available request in the group queue without performing additional checks (*i.e.*, without waiting for the request to become committed), which we call G(IP)^{2}C-WC. Note that the G(IP)^{2}C-WC does *not* prevent deadlock, which however is avoided by design in our benchmark workload.

This experiment focuses on Phase 2. Consequently, our setup involved only one client per core. As a workload, we employed a slightly modified version of the r pattern in which contenders never call the first server. We evaluated server chains of lengths ranging from 2 to 16, where the observed client sent 10000 messages for each test case. In total, we collected 139986 data points for each protocol variant.

Fig. 6 shows the results of Experiment 3. Since there is only one job per core, no budget is drained during Phase 1, and thus the bound on budget drain per chain link is $4{\cdot}L_{link}^{max}$, as indicated by the dashed horizontal red line. The work-conserving variant G(IP)^{2}C-WC behaves like the regular, non-work-conserving G(IP)^{2}C for $q \leq 3$. Thereafter, however, the work-conserving variant drains significantly more budget, exceeding the G(IP)^{2}C bound by approximately 51% for $q = 4$ and 368% for $q = 16$.

Recall that, by the design of our experimental setup, the budget drained per link should not vary with the chain length. However, Fig. 6 shows that, under the work-conserving G(IP)^{2}C-WC, the maximum observed normalized budget drain increases massively with increasing q. The explanation for this trend is that, since no check is performed before serving a request, S2S requests can be delayed by any request associated with any IPC context enqueued in GQ. As q increases, there are ever more opportunities for later-arriving contenders to "sneak in" and cause delays, which makes the worst case significantly worse (compared to the full G(IP)^{2}C).

In conclusion, Experiment 3 validates the non-work-conserving design of Phase 2: while Rules **S3** and **S4** may seem unintuitive at first, they are actually essential to obtaining favorable and predictable worst-case behavior, which is clearly desirable in the context of critical real-time systems.

Overall, our prototype implementation of the G(IP)^{2}C in LITMUS^{RT} shows the protocol, despite its admittedly many rules, to be realizable in a real system. Additionally, Experiment 1 has shown the analytical budget-drain bound to be tight, and Experiments 2 and 3 have validated key design choices in the G(IP)^{2}C to offer clear benefits.

VIII. Limitations and Open Questions

We did not conduct a comparison of our G(IP)^{2}C prototype with prior IPC implementations in terms of low-level architectural overheads or achievable throughput. There are several reasons for this: first, there is no natural baseline in LITMUS^{RT}, since the design of the underlying Linux kernel is not focused on fast IPC. A direct comparison of our Linux-based prototype with a more nimble microkernel, however, would be heavily lopsided and thus not yield meaningful results.

Second, our prototype implementation is primarily a proof of concept, and as such not heavily optimized. In contrast, IPC paths in mature microkernels are typically some of the best-optimized, most-heavily scrutinized code in the entire system. Considerable engineering effort will be required to lift the G(IP)^{2}C protocol to a comparable level, which is beyond the scope of this paper and remains future work.

Ultimately, the throughput and latency limits achievable with the G(IP)^{2}C protocol in the context of a proper microkernel remains an open question at this point.

Nonetheless, the G(IP)^{2}C protocol undoubtedly imposes greater runtime overhead than more straightforward approaches such as FIFO and priority queues. For one, when emitting a C2S request (with `gip_invoke`), the relevant ticket multiset must be re-initialized. This overhead does not exist in simpler protocols and can become substantial for large multisets.

Additionally, the non-work-conserving behavior of the `gip_wait` operation is costlier than in other IPC protocols due to the need to check for intersecting ticket multisets. In the worst case, a resource server may traverse the group queue GQ completely in reverse order before findindg an eligible request. This is clearly costlier than simply dequeuing the head of a FIFO or priority queue. Fortunately, the traversal cost remains bounded and relatively small since there are at most m IPC contexts queued. Multiset intersection can be realized efficiently with bit strings in which the i^{th} bit indicates whether at least one ticket for the i^{th} resource server is present.

Finally, bandwidth delegation requires synchronization with the scheduler and thus imposes additional overhead. It is certainly not a lightweight technique, especially due to the frequent migration of servers among processors (with its implied loss of cache affinity). However, it is presently unknown whether it is even possible to realize synchronous inter-processor IPC with strong temporal isolation without relying on techniques similar to bandwidth delegation. In line with earlier considerations [4, 5], we conjecture the answer to be negative.

On a positive note, the G(IP)^{2}C protocol conceptually allows for a *fast path*, which is crucial to achieving performance approaching existing microkernel-based systems. A fast path

could allow for a direct context switch (*i.e.*, without involving the scheduler) from client to server whenever the server can process a request immediately. In particular, in a fully initialized system (with all servers running), *all* S2S requests could take this fast path. There is considerable potential in this direction.

Last but not least, a major constraint imposed by our protocol is its notion of server groups, which shapes overall system design. To maximize parallelism, it is desirable to obtain many small server groups. Naïve designs, however, with globally shared servers (*e.g.*, if all subsystems rely on a single, shared page allocator) can easily degenerate into a single group comprising all servers. Avoiding this requires careful resource partitioning (*e.g.*, separate memory pools).

Overall, the G(IP)^{2}C protocol offers a new tradeoff: unprecedented temporal isolation with $O(m)$ bounds, but at the cost of nontrivial overhead. In contrast, prior S2S-capable protocols are overhead-optimized, but lack temporal isolation and no applicable multiprocessor bounds are known for them.

IX. Related Work

This paper combines techniques stemming from mixed-criticality systems, real-time resource sharing, microkernel design, and reservation-based scheduling. As each of these areas is vast in itself, having received much attention for many years, we focus on the most closely related prior work.

Reservation-based scheduling is central to our approach to providing temporal isolation. The G(IP)^{2}C rules are not tied to any particular scheduler. The prototype presented in Section VII uses simple polling reservations [27], but more flexible approaches such as CBS [1] or RBED [7] can be adapted such that Rules **A1** and **A2** are satisfied, too.

Prior work on microkernel IPC has focused primarily on maximizing throughput in the absence of contention [23, 24, 30]. The sequencing of concurrent requests has received less attention, with prior work focused on priority and FIFO-ordered wait queues. For example, TU Dresden's Fiasco.OC microkernel [19] and seL4's mixed-criticality variant seL4-MCS [25] use priority queues to order IPC requests, while the MBWI protocol [11] (which can be trivially adapted into an IPC protocol) and "plain" seL4 [18] use FIFO queues. Mergendahl et al. recently showed that implementation choices surrounding IPC and budget enforcement can undermine temporal isolation guarantees even on uniprocessors [28].

We focus in this paper exclusively on resources and services shared at the software level. We do not consider implicitly shared hardware resources such as memory caches or DRAM controllers, which can give rise to substantial hardware-induced interference. This is a major problem in practice and has received much attention in recent years [14, 26], but it is largely orthogonal to the IPC coordination problem studied herein.

The structure of the G(IP)^{2}C protocol is inspired by the RNLP [36] and the GIPP [31]. The server ticket abstraction implements a form of DGL [37]. The G(IP)^{2}C protocol can be seen as the first RLNP instantiation to reservation-based scheduling, even though the RNLP is designed for systems using JLFP scheduling, unmediated direct resource access, and locking primitives. This change in setting results in a tighter coupling between the two phases of the G(IP)^{2}C than there usually is between the two main components of the RNLP. Further related protocols and techniques may be found in a comprehensive review of multiprocessor real-time locking protocols [2]. Tong et al. explored the complexities that arise when dealing with budget depletion in locking protocols [35].

Our approach to ensuring progress in the G(IP)^{2}C protocol, bandwidth delegation, is a form of allocation inheritance [17] and inspired by a number of earlier multiprocessors protocols employing similar techniques [4, 6, 11, 31]. Already more than two decades ago, a microkernel in the L4 family used process migration during IPC to "help" servers complete requests [15, 16]. Bandwidth delegation can also be seen as an emulation of the migrating-threads model [12], which has also influenced the designs of Composite OS [29] and seL4-MCS [25].

Service tokens are conceptually similar to scheduling contexts [20], TCaps [13], or scheduling context capabilities [25]. The G(IP)^{2}C protocol relies on the fact that service tokens embed the priority of their assigned job and the ability to revoke bandwidth delegation. Scheduling context capabilities [25] do not embed priority and rely on the immediate priority ceiling protocol [33] to implement the passive server model. Unfortunately, this approach by itself is insufficient to ensure temporal isolation in multiprocessor systems [4]. The TCaps design has not yet been extended to multiprocessors either.

Finally, the MC-IPC protocol [5] is in some ways a precursor of the G(IP)^{2}C protocol, but only conceptually, not in terms of technique. In fact, the G(IP)^{2}C is structurally rather dissimilar and uses substantially more advanced progress and sequencing techniques, as necessitated by the support for S2S requests.

X. Conclusion

We have considered the problem of temporally isolated synchronous IPC in the presence of S2S requests in partitioned, reservation-based multiprocessor real-time systems, and proposed the G(IP)^{2}C protocol as the first solution.

The G(IP)^{2}C protocol lifts the main limitation of MC-IPC [5] while preserving its analytical properties. From a protocol- and mechanism-design perspective, the G(IP)^{2}C protocol can be seen as the first RNLP [36] instantiation that is not a locking protocol, and not geared towards JLFP scheduling. Furthermore, it is the first variant with support for multi-occupancy reservations and background tasks.

We have shown the G(IP)^{2}C to be practically realizable with an implementation in LITMUS$^{\text{RT}}$. Using our prototype, we have experimentally demonstrated the tightness of the established budget-drain bound, and that omitting key elements of the protocol's design results in a loss of temporal isolation.

In future work, it would be interesting to focus on the low-level implementation details of the G(IP)^{2}C protocol. In particular, the integration of a fast path and the interplay between IPC and the scheduling infrastructure pose many optimization opportunities. Analytically, it would be an intriguing challenge to generalize the protocol to other multiprocessor scheduling models, including clustered and semi-partitioned scheduling.

Acknowledgements

We thank the anonymous reviewers for their insightful and constructive feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 803111).

References

[1] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 1998, pp. 4–13.

[2] B. B. Brandenburg, "Multiprocessor real-time locking protocols," in *Handbook of Real-Time Computing*, Y.-C. Tian and D. C. Levy, Eds. Springer Singapore, 2020, pp. 1–99.

[3] ——, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[4] ——, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *Proceeding of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013, pp. 292–302.

[5] ——, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *Proceeding of the 24th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2014, pp. 196–206.

[6] B. B. Brandenburg and A. Bastoni, "The case for migratory priority inheritance in Linux: Bounded priority inversions on multiprocessors," in *Proceedings of the 14th Real-Time Linux Workshop (RTLWS)*. Real-Time Linux Foundation, 2012, pp. 67–86.

[7] 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 *Proceeding of 24th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2003, pp. 396–407.

[8] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2000, pp. 295–304.

[9] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE, 2006, pp. 111–126.

[10] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 133–150.

[11] 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.

[12] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *USENIX Winter*, 1994, pp. 97–114.

[13] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *Proceedings of the 2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 56–67.

[14] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core COTS," in *Proceedings of the 34th IEEE/AIAA Digital Avionics Systems Conference (DASC)*. IEEE, 2015, pp. 8D4–1.

[15] M. Hohmuth and H. Härtig, "Pragmatic nonblocking synchronization for real-time systems." in *USENIX Annual Technical Conference (ATC)*, 2001, pp. 217–230.

[16] M. Hohmuth and M. Peter, "Helping in a multiprocessor environment," in *Proceeding of the 2nd Workshop on Common Microkernel System Platforms*, 2001.

[17] P. Holman and J. H. Anderson, "Object sharing in Pfair-scheduled multiprocessor systems," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2002, pp. 111–120.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating systems principles (SOSP)*, 2009, pp. 207–220.

[19] A. Lackorzyński and A. Warg, "Taming subsystems: capabilities as universal resource access control in L4," in *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems*, 2009, pp. 25–30.

[20] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT)*, 2012, pp. 93–102.

[21] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2001, pp. 151–160.

[22] Y. Li, R. West, and E. Missimer, "The Quest-V separation kernel for mixed criticality systems," *arXiv preprint arXiv:1310.6298*, 2013.

[23] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of the 14th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 1993, pp. 175–188.

[24] ——, "On micro-kernel construction," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 237–250, 1995.

[25] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the 13th EuroSys Conference*, 2018, pp. 1–16.

[26] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–38, 2019.

[27] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: An abstraction for managing processor usage," in *Proceedings of IEEE 4th Workshop on Workstation Operating Systems (WWOS-III)*. IEEE, 1993, pp. 129–134.

[28] S. Mergendahl, S. Jero, B. C. Ward, J. Furgala, G. Parmer, and R. Skowyra, "The Thundering Herd: Amplifying kernel interference to attack response times," in *Proceedings of the 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022, pp. 95–107.

[29] G. Parmer, "Composite: A component-based operating system for predictable and dependable computing," Ph.D. dissertation, Boston University, 2010.

[30] S. Peters, A. Danis, K. Elphinstone, and G. Heiser, "For a microkernel, a big lock is fine," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015, pp. 1–7.

[31] J. Robb and B. B. Brandenburg, "Nested, but separate: Isolating unrelated critical sections in real-time nested locking," in *Proceeding of 32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020, pp. 6:1–6:23.

[32] D. C. Sastry and M. Demirci, "The QNX operating system," *Computer*, vol. 28, no. 11, pp. 75–77, 1995.

[33] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[34] H. Takada and K. Sakamura, "Real-time scalability of nested spin locks," in *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1995, pp. 160–167.

[35] Z. Tong, S. Ahmed, and J. H. Anderson, "Overrun-resilient multiprocessor real-time locking," in *Proceeding of the 34th Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[36] B. C. Ward and J. H. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2012, pp. 223–232.

[37] ——, "Fine-grained multiprocessor real-time locking with improved blocking," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS)*, 2013, pp. 67–76.