Chaos: a System for Criticality-Aware, Multi-core Coordination *

Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, Zach Day
The George Washington University
Washington, DC
{phanikishoreg,peachg,gparmer}@gwu.edu

***Abstract*—The incentive to minimize size, weight and power (SWaP) in embedded systems has driven the consolidation both of disparate processors into single multi-core systems, and of software of various functionalities onto shared hardware. These consolidated systems must address a number of challenges that include providing strong isolation of the highly-critical tasks that impact human or equipment safety from the more feature-rich, less trustworthy applications, and the effective use of spare system capacity to increase functionality. The coordination between high and low criticality tasks is particularly challenging, and is common, for example, in autonomous vehicles where controllers, planners, sensor fusion, telemetry processing, cloud communication, and logging all must be orchestrated together. In such a case, they must share the code of the software run-time system that manages resources, and provides communication abstractions.**

This paper presents the Chaos system that uses *devirtualization* to extract high-criticality tasks from shared software environments, thus alleviating interference, and runs them in a minimal runtime. To maintain access to more feature-rich software, Chaos provides low-level coordination through *proxies* that tightly bound the overheads for coordination. We demonstrate Chaos's ability to scalably use multiple cores while maintaining high isolation with controlled inter-criticality coordination. For a sensor/actuation loop in satellite software experiencing inter-core interference, Chaos lowers processing latency by a factor of 2.7, while reducing worst-case by a factor 3.5 over a real-time Linux variant.

I. Introduction

Embedded systems are increasingly required to provide both complicated feature-sets, and high-confidence in the correctness of mission-critical computations. From self-driving cars and Unmanned Aerial Vehicles (UAVs) to CubeSats, software systems must provide more communication facilities and more complicated sensor fusion, while still maintaining the expected physical dynamics of the systems. This challenge is complicated by the trend in these and other domains that functionalities traditionally performed by disparate computational elements are consolidated onto less expensive and more capable multi-core, commodity processors. Unfortunately, current systems have difficulty in both supporting feature rich, general computation provided by large amounts of code, and the high-confidence physical control that often requires software simplicity while also providing high resource utilization.

In response to these trends, Mixed-Criticality Systems (MCS) [1] explicitly consider co-hosting software of various assurance-levels and mission-criticality on shared hardware. *Higher-criticality* tasks are those that are mission critical, and often impact human and equipment safety, while *lower-criticality* tasks include those that are desirable, but less safety critical (including, for example, logging, user-interface management, and cloud communications). Tasks are segregated by criticality to make explicit the desired service degradation properties: should the system not be able to meet all deadlines, lower-criticality tasks should receive degraded service first.

As high-confidence is required in high-criticality tasks, significant effort is often placed into assuring they behave according to specification. This is reflected in the *assurance-level* of the code that implements tasks. *High-assurance* code achieves certifications, has rigorous testing regimes, and, in the extreme, is formally verified. Such procedures are often expensive and thus are undesirable for complex software or for software whose failure is less impactful, including many lower-criticality tasks. Moreover, tasks of varying confidentiality execute within complicated software stacks composed of various *subsystems*. Each subsystem is a collections of code of the same assurance level (*e.g.* kernels, virtual machines (VMs), or applications), often isolated (for example, using hardware mechanisms) from surrounding subsystems. Importantly, system design for mixed-criticality, multi-assurance-level systems should provide *isolation of high-assurance subsystems executing high-criticality tasks from potential faults in lower-assurance subsystems*. This is particularly difficult as even high-criticality tasks wish to communicate with the broader system, and in doing so, harness the increased functionality of lower-assurance code. In contrast, traditional embedded systems eschew isolation and aim to ensure that all system software executes in a single subsystem and is at least the assurance-level required by the highest-criticality task.

Multi-core systems complicate the isolation between subsystems of different assurance levels. Abstractions shared between cores require *inter-core coordination* that can cause interference across assurance levels. Data-structures such as scheduler run-queues, shared between cores, can introduce significant interference as synchronization serializes operations. In contrast, Inter-Processor Interrupts (IPIs) enable preemptive message passing between cores, but high-priority interrupt execution interferes with preempted tasks. As all system software is dependent on the kernel, such coordination can disrupt high-criticality tasks, especially when generated by low-criticality (possibly faulty or compromised) code.

This research introduces Chaos, which is designed to *devirtualize* high-criticality tasks, a process that exports them out of possibly low-assurance subsystems that host low-assurance tasks, to remove interference due to the shared subsystem. Such tasks are imported into a ChaosRT execution environment with minimal controlled interference and

*This material is based upon work supported by the National Science Foundation under Grant No. CNS-1815690, and by ONR Awards No. N00014-14-1-0386, ONR STTR N00014-15-P-1182 and N68335-17-C-0153. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation nor ONR.

strictly predictable execution. This allows high-assurance, high-criticality tasks to execute simple I/O and perform computations predictably with a simple runtime, while still leveraging the high-functionality of low-assurance code. Devirtualization is enabled by asynchronous communication between proxies that ensure that communication has a bounded latency, and incurs a bounded interference on the surrounding system. The core focus of ChaosRT is to remove interference from shared subsystems of varying assurance-levels through *devirtualization*[1] and to bound the IPI interference using proxy-implemented *rate-limiting servers*.

Contributions.

- §II introduces a number of forms of interference that low-assurance tasks can have on those of high-criticality due to inter-core coordination.
- In response, §III and §IV detail *devirtualization* to extract high-criticality subsystems from lower-assurance legacy systems – while maintaining functional dependencies – and predictable inter-core message passing mechanisms.
- We introduce an *IPI rate-limiting* technique in §III which enables Chaos to bound the IPI interference and latency of notifications for inter-core coordination.
- Finally, in §V, we evaluate Chaos relative to both Linux and other reliability-focused systems.

II. Motivation

Software of various assurance levels that rely on the shared abstractions of a subsystem, is impacted by the overheads of the underlying mechanisms. The overhead, and inter-task interference of these mechanisms are particularly pronounced in multicore systems, which necessitate inter-core coordination. Here we study the impact of this sharing and coordination on the software running in Linux, seL4 [3], Fiasco OC [4], [5], and Composite [6]. Details of the experimental setup are in Section V.

A. Impact of Shared Memory on Predictability

Mutually exclusive access to shared memory data-structures serializes parallel operations, thus impacting the execution times of software across cores. We will use scheduling run-queues as an example as they are often synchronized using locks. Systems that use global scheduling contest a shared lock from all cores. More practically, many systems (*e.g.* Linux) use per-core runqueues and inter-thread communication between cores often requires taking locks for other core's runqueues. Though predictable sharing techniques can bound the latency of such operations, the required cache-coherency and serialization overheads can have a significant impact on common operations.

Figure 1 shows the impact of these shared runqueues on various systems. We run a high-criticality task on a target core making a system call and on a varying number of other cores, we run low-criticality tasks generating an adversarial

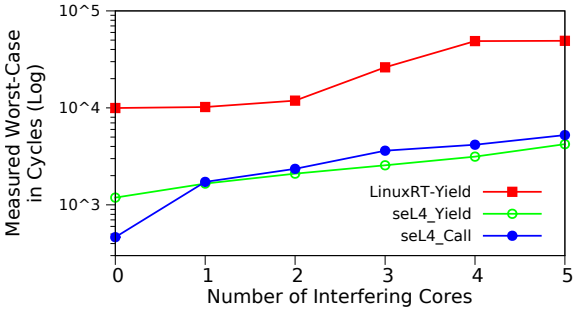

Fig. 1: Measured worst-case execution times of core-local system-call operation with adversarial workload on other cores.

workload that contests the target's runqueue's lock and shared data-structures in the kernel. The seL4_Call is a synchronous IPC system call in seL4 in which two tasks perform a synchronous rendezvous with one task calling seL4_Call API and the other calling the seL4_ReplyRecv. The seL4_Yield is a system call to donate the remaining timeslice to a thread of same priority, similar to sched_yield in LinuxRT-Yield. This experiment demonstrates that **low-assurance – possibly compromised and malicious code – can cause significant interference on the execution of high-criticality code, even when executed on a different core**. These problems become worse at scale, especially with NUMA. Linux kernel developers found that on a 4 way NUMA machine with 40 cores, runqueue contention cause *over 1ms of interference*[2]. These overheads are fundamentally determined by *hardware operation costs*, thus a new software structure is required to remove them.

It is important to note that if predictable locks are used (*e.g.* FMLP$^+$ [7], or simply non-preemptive FIFO spinlocks [8]), this overhead is *bounded*. However, these overheads can recur on each interaction with the shared structures (*e.g.* on each interrupt, each scheduling decision, and each communication operation), and since they increase with the number of cores, they significantly harm the overall utilization of the system. seL4 is an interesting example, as it has a non-preemptive kernel, and a full worst-case execution time (WCET) analysis has been conducted [9] on it. Even after optimization [10], this overhead is on the order of hundreds of μ-seconds. Though global locks are immensely valuable to maintain the verification guarantees of the system, an increasing number of cores multiplies the timing impact of the WCET on each kernel operation.

B. Inter-Criticality Interference via IPIs

An alternative to shared memory for inter-core coordination is using message passing via IPIs [11]. With this approach, data-structure access and modification is coordinated using message passing and IPIs for event notification. For example, an IPI is sent to activate a blocked, low-priority thread on another core. However, such IPIs cause high-priority interrupt execution on the target core which might be executing high-criticality tasks. Similar to shared memory, this interrupt

[1]We use the term "devirtualization" as it often involves exporting tasks out of *virtual machine* subsystems. We would like to disambiguate from devirtualization in programming languages [2], an optimization to convert *virtual* calls to direct calls.

[2]See the email by Mike Galbraith on the Linux Kernel Mailing List titled "Re: [RFC][PATCH RT 4/4 v2] sched/rt: Use IPI to trigger RT task push migration instead of pulling" on 21st of December, 2012.

execution represents interference caused by low-criticality tasks on high-criticality tasks.

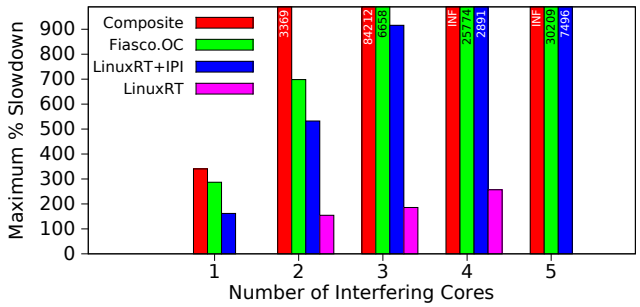

Fig. 2: Measured maximum %-slowdown of high-priority execution on a target core with a number of interfering cores attempting to unblock lower-priority tasks on the target core. Numbers on bars indicate the actual measured maximum %-slowdown for >1000% slowdown and INF indicates a livelock on the target core.

Figure 2 shows the impact of IPIs on the execution of a high-priority task with an increasing number of cores for a number of systems. A high-criticality task executes for 10 μ-seconds on a target core, and a number of low-criticality tasks on different cores attempt to unblock a lower-priority task on the target core. A 10 μ-seconds worload is chosen to emphasize the IPI kernel overheads, and their ability to cluster around a small timespan, thus generalizing to larger timespans. Similar to shared memory approaches, **IPI-based coordination caused by low-assurance code can induce highest-priority interrupt interference on high-criticality tasks executing on different cores**.

It is difficult to choose between using shared memory coordination – that has serialization and cache-coherency overheads that increase with larger numbers of cores – and IPI coordination – that can cause interference from high-priority interrupt execution overheads. For example, on using IPIs for such coordination, Thomas Gleixner points out that "...it avoids fiddling with the remote [runqueue] lock, but it becomes massively non deterministic."[3]. We show here that either choice has significant repercussions in that low-assurance code can increasingly impact the execution times of high-criticality tasks with rising core counts.

IPIs are used to coordinate many aspects of complex systems including synchronization primitives, work-queues, RCU quiescence [12], scheduling runqueue balancing, and TLB coherence. These experiments do not conduct an exhaustive study of these uses, instead showing that at least one of them can be used to cause significant interference on high-criticality tasks.

C. Overhead of Hierarchical Systems

An important goal of mixed criticality systems is to "reconcile the conflicting requirements of *partitioning* for (safety) assurance and *sharing* for efficient resource usage" [1]. Especially on multi-core systems, the efficient use of processing resources means allowing best-effort (often low-criticality) tasks to consume the remaining cycles after real-time tasks execute. Thus, partitioning (epitomized by separation kernels [13], [14]) is resource-inefficient.

[3] Email titled "Re: [RFC][PATCH RT 3/4] sched/rt: Use IPI to trigger RT task push migration instead of pulling" on the 11th of December, 2012 to the Linux Kernel Mailing List at https://lkml.org/lkml/2012/12/11/172.

Round-trip Comm.	Host ↔ Host	Host ↔ VM	VM ↔ VM
Same-Core	26670	105228 (295%)	15064*
Cross-Core	33155	91205 (175%)	77807 (135%)

Nanosleep	Host	VM
Wakeup	3602	13019 (261%)

TABLE I: Round-trip socket communication and timer notification overheads with virtualization in Linux. % numbers indicate the %-slowdown from Host-only operation. * is explained in text.

One means of addressing the sources of inter-core coordination interference, is to place all tasks of a specific criticality, or assurance level in a VM. Virtual machine infrastructures enable memory partitioning (*e.g.* separate runqueues across different VMs), along with the sharing of cores (between VMs). Within that VM, that kernel's interference should not impact tasks outside of the VM. Tasks of different confidentialities interact using inter-VM communication. Here we study the costs of communication involving VMs including the interrupt delivery latencies.

Table I shows (1) the overheads of round-trip UDP communication between a real-time Linux host process and a non-real-time Linux process in a VM (Host ↔ VM), between processes on the host (Host ↔ Host), and between processes within a VM (VM ↔ VM), and (2) the propagation latency of a timer from the hardware interrupt to a host process, and to a process in a VM. The communication between processes in the VM is faster (depicted with * in the table) as it is running a non-real-time Linux that does not incur the overheads of a real-time fully-preemptible Linux. Even systems that focus on removing the overheads of the virtualization hierarchy [15], still suffer from significant timer propagation delays. Additionally, virtualizing VM IPIs has significant overhead, even with hyperupcalls [16]. Though VMs enable isolation of high-criticality tasks from low-assurance code, **virtualization induces significant overheads in inter-VM and inter-core communication and timer propagation that significantly impact the execution times of the MCS tasks and the system schedulability**.

III. CHAOS DESIGN

A. Example MCS

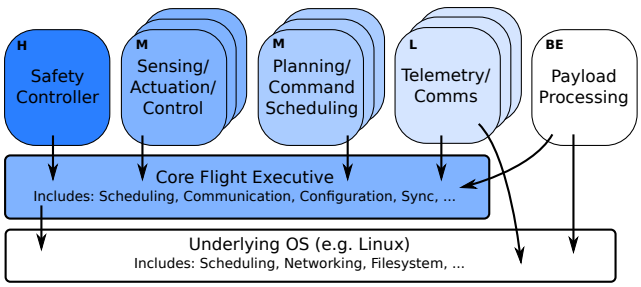

Fig. 3: Core software in a cFS system. The darker the software component, the higher assurance it is. Letters **H**, **M**, **L**, and **BE** denote high, medium, low, and best effort criticalities.

We use the software stack of an autonomous vehicle as an exemplar of a MCS. For this, we use the core Flight System

(cFS)[4] that is used in many NASA satellite missions, and has also been deployed in quad-copters. The cFS configuration we use includes 137,483 lines of code, and includes a number of applications, some of which handle control tasks, while others communicate with the ground station. cFS is low-level middleware that relies on an OS-specific backend that includes support for full-featured network communication with the ground-station and logging to disk. Our setup also includes a safety controller that maintains stability in the case of failure (similar to the Simplex model [17]). Figure 3 shows the logical subsystems on the system and the dependencies between them.

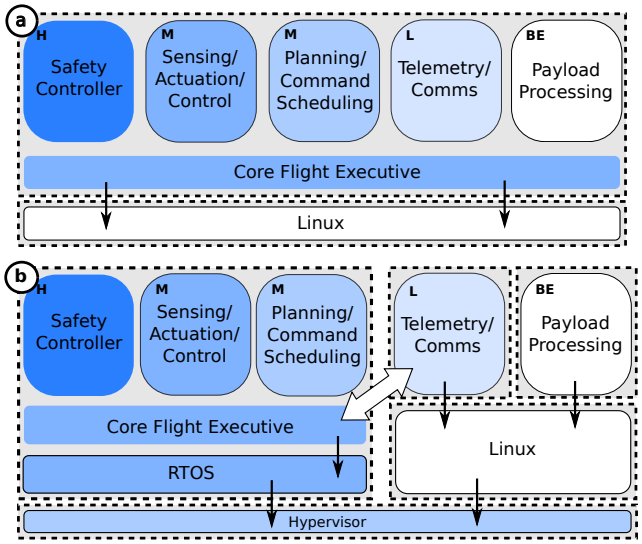

Fig. 4: Two cFS setups. Dotted lines indicate isolation boundaries. (a) cFS runs in a process shared with all its applications, on top of Linux. (b) Software split across two VMs.

Two concrete implementations of this system appear in Figure 4. In the first implementation in Figure 4(a) (which is a conventional cFS deployment), Linux provides POSIX services, and cFS is implemented with a Linux backend for its operations. Temporal or functional failures in the kernel can trivially impact cFS execution, and a failure in some of the more complicated (and network-facing) cFS tasks such as telemetry input and output can similarly impact the safety controllers. This, somewhat traditional, system structure motivates MCSes due to trivial interference between criticalities. Figure 4(b) isolates cFS into a separate VM, but constraints its functionality to that provided by a more minimal runtime RTOS. The best effort or low-criticality tasks remain in the Linux VM. This organization has the benefit that cFS tasks are insulated from failures in the Linux VM and from coordination interference. Unfortunately, expensive inter-VM communication constrains the ability to use VMs, and it might not be possible to move latency sensitive applications into separate VMs (the safety controller here). Most RTOSes do not have a full networking stack to perform virtualized network communication with other VMs.

[4]See https://cfs.gsfc.nasa.gov/ and http://coreflightsystem.org/. We use the OpenSatKit configuration (https://opensatkit.github.io/).

Importantly, interference between VMs might remain if the base kernel (hypervisor, in this case) demonstrates any of the overheads discussed in §II.

B. Devirtualization to Control Inter-Criticality Interference

Chaos enables the *devirtualization* of tasks of criticalities that are not compatible with the assurance levels of the subsystems they are dependent on. For example, the safety controller in Figure 4(b) should be higher assurance than the large cFS code-base, thus should be devirtualized. Devirtualization removes the task in question from the resource management domain of that subsystem, thus insulating it from potential interference via the abstractions and coordination within that subsystem. The task executes in a minimal Run-Time environment – that we call ChaosRT – that provides processing and memory facilities, and maintains transparent access to functionalities and APIs in the VM subsystem through *proxies*. These proxies insulate the high-criticality tasks from shared memory and IPI overheads within the low-assurance subsystem while routing requests for functionality to that subsystem. The safety controller sees the same functionality as it did in cFS, but avoids low-assurance subsystem interference. cFS subsystem sees the proxy as the safety controller.

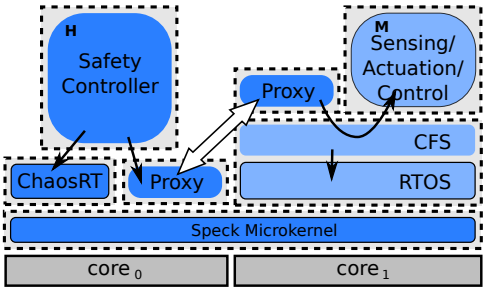

Fig. 5: Safety controller is exported from the cFS, executes in ChaosRT, but still predictably communicates with cFS via proxies.

Proxies to access low-assurance functionalities. Figure 5 depicts Chaos's proxies. They enable the *decoupling* of the low-level resource management of CPU and memory, from the functional requests (*e.g.* for network or filesystem I/O) for specific tasks. Proxies provide the routing machinery necessary to map specific functional requests made by a task into the communication mechanisms, and data-movement facilities provided by Chaos.

The high-criticality safety controller might depend on access to sensors, actuators, time-triggered processing, and sends sensor readings to further processing steps such as sensor fusion, planning, telemetry management, and logging. The sensor, actuator, processing, and memory management would be handled with tight predictability properties by ChaosRT, while the more complicated functions integrated into a larger, lower-assurance system would be sent through proxies to cFS that handles complex communication channel abstractions, and complicated software stacks. Similarly, applications in cFS that wish to send and receive telemetery via the network use proxies to the lower-assurance, best-effort POSIX environment.

Chaos requires that the underlying kernel that provides the virtualization and execution contexts not, itself, suffer from the shared memory and IPI interference problems demonstrated in §II. Composite's kernel [6] helps with this as it does not use mutual exclusion to protect its shared data-structures, thus avoiding much of the shared memory interferences in §II-A. It instead relies on wait-free operations based on raw atomic instructions. However, §II-B shows that Composite exposes asynchronous communication APIs that enable immense IPI interference. We address this in §III-D.

Devirtualization of legacy embedded systems. Many legacy embedded systems are specialized enough and have simple enough code bases that all software is compiled as a single binary with no memory isolation. This complicates interposing proxies that provide memory isolation. Regardless, the underlying RTOS exposes well-defined APIs used by each task to interact with the broader system. Past research has used these interfaces, along with stub code to emulate the C API, to separate the applications from the RTOS in separate protection domains [18]. Similarly, Chaos currently interposes proxies on the system call interface at the *bottom* of libc, or at the API-level of middleware. The former requires few changes to the POSIX applications and the latter is specific to the API.

In Chaos, subsystems require some code modifications that are common in paravirtualization approaches. We treat the complex cFS system's documented APIs as the interface between the applications and the core cFS system, and use Composite's thread-migration-based synchronous invocation to the server, cFS system. Proxies interpose on such invocations to the server in the cases where we need asynchrony. Though this breaks some backwards compatibility (for example, in the rare case when applications directly access each other's memory), most core cFS applications continue to function. Not all applications are at a different assurance level than the cFS core, thus they all do not require isolation. Regardless, we err on the side of isolation to increase reliability and security.

Interface routing between subsystems. An important part of our design is the ability to discriminate between devirtualized functional requests for the VM subsystem (*e.g.* complex I/O requests), and for the ChaosRT. Application binaries are linked by binding unresolved function symbols to the dependencies of the appropriate proxies. Currently, this enables functionality to be split based on functions, but extensions to allow routing based on parameters (*e.g.* file descriptors) are straightforward. As we are currently focusing on embedded systems (*e.g.* FREERTOS and cFS) that often have simpler APIs, so we have not investigated this extension.

C. Inter-Assurance-Level Communication

Tasks that are devirtualized and moved out of the VM must communicate via proxies to make functional requests to the VM. We consider the case when a task (the client) requests functionality from the VM subsystem (the server). If client and server have different assurance levels, Chaos must provide predictable service degradation. Here we analyze the required properties of this communication along the following dimensions: (1) synchronization properties of the communication (asynchronous or synchronous), (2) locality of communication (intra- or inter-core), and (3) the budget and priority of the threads involved in the communication.

Asynchronous vs. synchronous communication and locality. Synchronous communication semantics mimic function calls; the client does not resume execution until server computation is completed. In L4 variants, this is often achieved using the `call` and the `reply_and_wait` APIs in the client and server, respectively [19], while Composite kernel uses thread migration [20], [21] in which the same kernel abstraction migrates between separate isolated execution contexts. In either case, synchronous invocations couple the execution of the client with that of the server, thus complicating temporal isolation [22]. Should the server fail – for example, by experiencing unbounded execution – while servicing a client request, the failure propagates to the client thread. On the other hand, synchronous IPC has the significant benefit that in many implementations it is highly optimized (§V-A) and predictable. In contrast, inter-core synchronous invocations can be very expensive as they often rely on IPIs whose overheads dwarf those of typical synchronous IPC [23], and can cause interference (§II-B).

Asynchronous communication removes synchronization between the client and server execution. As client and server threads are executed with separate budgets and priorities, and only execute code within their protection domains, they insulate communicating threads from each other's execution properties. In the extreme, only a shared buffer to transfer data is required, and both client and server can poll the buffer to determine when to act. To maintain low-latency and remain compatible with existing software, we consider asynchronous communication that includes *event notification* – the activation of the destination task. This is important for real-time tasks as it enables activation of high-priority tasks awaiting communication. In UNIX, this might include sending a signal to another task, or writing to a pipe to activate a reading task. seL4 and Composite both include asynchronous notification end-points whereby a notification is sent to the end-point which unblocks the awaiting thread. As a client sending an asynchronous notification to a server continues execution, it is not temporally coupled to the server as in synchronous communication. This is a natural fit for notifications between parallel tasks as it avoids serialization, and mimics the parallelism of the underlying cores.

§II depicts two different implementations of this functionality. seL4 and Linux use shared data-structures across all cores and locks to synchronize parallel accesses to kernel data-structures. Thus, an asynchronous notification modifies the scheduling runqueue to wake up the receiving thread, and sends an IPI if it is now the highest-priority thread on the core. In contrast, Fiasco L4 and Composite rely on IPIs to avoid synchronization overheads, and to avoid accessing scheduling data-structures, respectively.

Chaos uses the underlying Composite kernel's synchronous invocations facility implemented with thread migra-

tion in the case where a client of the same assurance level as the server code attempts to harness the server's functionality. This enables efficient and predicable communication in the common case. If the server provides scheduling services (*e.g.* if it is a VM), then such client threads are scheduled directly by it, and the communication mimics conventional system-call semantics. Comparably, Chaos proxies use the Composite's asynchronous communication if there is an assurance mis-match between client and server, or if the client has been devirtualized onto a separate core. This enables predictable temporal fault degradation for clients in spite of server failures. Importantly, as we are leveraging the asynchronous notification facilities of the kernel, clients with higher assurance requirements can be scheduled in a *different environment* than that in the VM, thereby providing stronger temporal isolation for client execution. Thus, high-criticality tasks that require higher assurance than that of their default VM subsystem, can leverage the minimal ChaosRT scheduling and memory management facilities.

Scheduling context during communication. When communicating between different subsystems, end-to-end timing of execution is determined not only by the execution in each subsystem, but also by how client and server execution is prioritized, and with which budget they execute – in short, which scheduling context the client and server use for execution. Service degradation properties are also determined by how scheduling contexts are managed, as this determines, for example, the timing properties of client and server if the client generates an unbounded number of requests. Microkernels have handled this in a variety of ways [24], by either executing in different contexts, or by non-deterministically using the client's context until a timer interrupt occurs.

More recently, a variety of techniques unify client and server execution into the same context: thread migration facilities in the underlying Composite kernel explicitly use the same scheduling context for both client and server execution (though it switches page-tables, and C execution stacks), Credo [25] and Nova [26] decouple scheduling from execution context; and seL4 extensions [27] pass budgets between client and server threads. Thread migration completely avoids scheduling decisions during communication, thus enables scheduling policy and data-structures to be defined and implemented separately in each subsystem. This Composite facility provides the foundation in Chaos for devirtualizing high-criticality tasks and minimizing system coordination overheads.

Asynchronous communication has similar complications, especially in mixed-criticality systems [23], [27]. A high-criticality client task should be paired with a server task with scheduling properties that enable predictable, and sufficient progress within the necessary timing bounds. High-criticality tasks (especially those at a higher-assurance level) should generate a bounded workload on the server, so that traditional analyses are sufficient to choose a server execution budget and priority. However, such an allocation is necessarily pessimistic as it is based on worst-case executions in the server. In contrast, lower-assurance tasks might generate unbounded workload, either due to faults, or due to best-effort client behavior attempting to maximize throughput. This further complicates server scheduling parameter selection, and might pessimistically either decrease best-effort throughput, or lower aggregate real-time task utilization.

One of the fundamental challenges that asynchronous communication via proxies must address is how predictable communication can be performed between subsystems, even when those subsystems are controlled by different schedulers. Composite requires the definition of user-level scheduling policies [28], thus each of the subsystem's schedulers define their own timing properties and priorities.

Chaos takes three approaches to this problem. (1) Higher-assurance client's requests are scheduled on server tasks according to the server's scheduling policy. In NetBSD, this means running them in a real-time thread, while in cFS, this means choosing the appropriate fixed priority. Neither system provides budget-based servers, but so long as the client is higher-, or same-assurance-level as the server (NetBSD/cFS), the amount of execution requested should be predictable from the server's perspective. (2) Similarly, asynchronous invocations *between cores* use per-subsystem, per-core scheduling parameters. To avoid sharing the computational abstraction across cores in Chaos, cross-core notifications must run in the server context using a separate budget and priority. In the case of best-effort (unbounded) requests, conservative budget and priority selection is necessary. (3) Low-assurance clients requesting service from high-assurance/criticality servers must be treated carefully. Bounding the workload of these client requests, while enabling them to also utilize spare cycles is not straightforward. Chaos uses Temporal Capabilities (TCaps) [23] that require client proxies to *delegate* a span of time which the server uses for its computation. Computation in the server uses this time, and inherits the priorities from the client. TCaps are used to coordinate time management between untrusting subsystems, and importantly handles the transitive delegation of time across multiple subsystems by tracking different subsystem's priorities in a manner similar to how labels are tracked in distributed information flow systems [29].

D. Bounding IPI Interference and Latency

§II-B demonstrated that Composite provides an effective means for generating high frequencies of IPIs via its asynchronous communication primitives. The low-overhead mechanism for sending notifications easily livelocks a destination core with a striking amount of interference. Chaos, then, *bounds IPI interference* while also *bounding the latency of notifications*.

To bound the interference from IPIs from lower-assurance subsystems, we use *rate-limiting servers* to control the number of IPIs over windows of time. Specifically, for task i, over periods of time p_i^{ipi}, only e_i^{ipi} notifications can be sent. In the design of these servers, Chaos uses deferrable servers [30], thus defining p_i^{ipi} over fixed, periodic windows of time. Though more complex servers such as sporadic servers [31] can make tighter guarantees and limit the system to e_i^{ipi} IPIs

over a sliding window of p_i^{ipi}, the simpler code for deferrable servers, and the constant memory requirements are consistent with the high-assurance requirements of the code in ChaosRT and proxies. Though deferrable servers suffer from a "double hit" effect in which e_i^{ipi} IPIs arrive at the end of a window, immediately followed by another e_i^{ipi} IPIs at the start of the next – causing $2 \times e_i^{ipi}$ IPIs over a short window of time, they have been shown to have only a small impact on system utilization [32] for realistic workloads.

Perhaps surprisingly, Chaos silently *drops* any notification above the allowed rate. For real-time tasks that send a bounded number of notifications such over-runs should not happen. For faulty real-time tasks, and for best-effort tasks, such a policy is not sufficient. All event notifications for asynchronous communications are accompanied with data in shared memory, wait-free ring buffers, thus even when a notification is dropped, the corresponding data is still published. However, this can still lead to unbounded communication latencies: if data is queued but the rate is exceeded, and thereafter no additional notifications are sent, the receiving side will never be activated to handle the data. In response to these challenges, Chaos *augments notifications with polling* on the receive side of the communication. To bound the latency for processing asynchronous communication, this polling is performed with a period of p_i^{poll} by the same thread (in the proxy) that handles asynchronous activations, thus it uses the budget and priority discussed in §III-C. Thus, in the worst case, asynchronous communication processing is bounded by this polling latency, and the latency of e_i' notifications p_i^{ipi} is 0, and bounded by p_i^{ipi} otherwise.

E. Example MCS System in Chaos

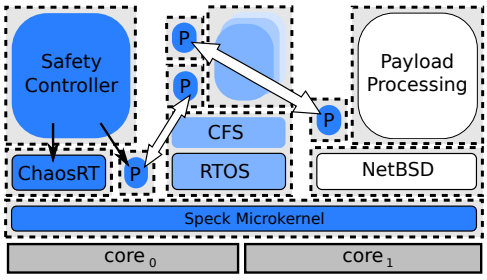

Fig. 6: The prototype cFS system in Chaos. Proxies (**P**s) aid in communication between the safety controller, cFS, and the best-effort system's networking subsystem.

Figure 6 shows the Chaos implementation of the system from §III-A. The cFS is devirtualized out of the POSIX VM (implemented using a base of a NetBSD `rumpkernel` [33]), and the safety controller is devirtualized out of both. Though cFS is a traditional embedded system with no spatial isolation between tasks, we devirtualize its tasks into separate protection domains, to enable mutual isolation between tasks. Though each subsystem can be assigned to multiple cores, higher-criticality tasks are isolated from the shared memory and IPI interference from lower-assurance code, yet functional dependencies are still resolved by proxies. The proxies use appropriate forms of communication given the assurance-level of the communicating code. The rigid hierarchy imposed by virtualization is replaced with a somewhat more chaotic, but predictable structure.

IV. Chaos Implementation

A. Leveraging Composite

The Composite kernel, Speck [6], uses capability-based access-control [34] to mediate operations on a small set of kernel-provided abstractions. A capability is an unforgeable token that conveys access to a system *resource*, ownership of which provides the ability to perform specific *operations* on that resource. The system resources accessible through capabilities include TCaps [23], threads, synchronous communication end-points, and asynchronous communication end-points (of which interrupts are a subtype). Composite is composed of a number of *components* that host user-level execution and consist only of a pair of a capability- and a page-table. Thus, components are a unit of isolation and the capabilities (entries in their capability-tables) constrain their scope of resource accesses. Capability-based isolation, and a fine-grained system decomposition enable the strict isolation required by the constraints in §III.

Chaos relies on a few key facilities of Composite:

- A lock-less kernel that uses wait-free kernel operations to update shared data-structures – *this avoids shared memory serialization and coherence overheads*. All shared data-structures are read-only for common operations, and synchronize using wait-free operations otherwise. This prevents the kernel from inhibiting the potential scalability of the component-defined policies.
- All kernel operations have bounded execution times – *which enables bounded higher-level subsystem operations*.
- All system resources are protected and referenced via capability-based access control [34] – *which enables strong isolation via confinement [35], [36]*.
- IPIs are not used for kernel coordination, instead only by optimized asynchronous communication between user-level end-points – *which enables rate-limiting policies*. Even TLB coherence is ensured via quescence [6], relying on user-level policies to ensure TLBs are flushed before reusing unmapped virtual ranges.
- Highly optimized communication mechanisms for synchronous and asynchronous communication – *enabling the finer-grained isolation*.
- The kernel does not have scheduling policy [28], thus scheduling policy is implemented in user-level – *enabling the separation of scheduling policy across subsystems*. Importantly, this moves shared scheduling data-structures such as runqueues out of the kernel, thus both removing system-wide interference from serialization of data-structure access, and the IPIs necessary for in-kernel coordination.
- *Components* provide spatial isolation by leveraging hardware page-tables and provide controlled access to a set of kernel resources through their capability table – *enabling a general mechanism for VM, ChaosRT and application protection*.

B. Virtualization in ChaosRT

We use the NetBSD `rumpkernel` infrastructure to support legacy POSIX requests for service. This gives us access not only to device drivers, but also file systems and a fully-featured networking stack. However, Chaos takes an aggressive stance on paravirtualization [37]: all system calls to the NetBSD "kernel" (which runs as a user-space component) are modified to explicitly use Composite synchronous invocations. This is done by redirecting the `libc` system calls to minimal stubs that are paired with corresponding stubs in NetBSD. In Linux this might be done by appropriately redefining the `vsyscall` page which would decrease the number of required code modifications. NetBSD context switches are modified to use Composite thread dispatch, thus implementing a user-level scheduler. Rumpkernels do not provide virtual memory facilities (*e.g.* `fork`, `mmap`), but given our focus on embedded systems, and the use of the best-effort VM subsystem as a provider of higher-level POSIX functionality, this has not been prohibitive.

C. ChaosRT, User-level Scheduling, and Coordination

Chaos avoids interference from shared memory data-structures on an increasing number of cores by devirtualizing higher-assurance tasks, thus moving them out of the scheduling domain of the VM subsystem into ChaosRT. Instead the core scheduling and memory interfaces are emulated in the minimal ChaosRT environment that avoids such overheads. As such, ChaosRT uses minimal libraries that provide fixed priority, round-robin scheduling, and memory allocation, and export these services through synchronous communication to the devirtualized task. ChaosRT is a fully multiplexing runtime and multiple tasks can be devirtualized into it.

One of the fundamental challenges that asynchronous communication via proxies must address is how to predictably communicate *directly* between subsystems, even when those subsystems are controlled by different schedulers. Composite requires the definition of user-level scheduling policies [28], thus each of the subsystem's schedulers define their own timing properties and priorities. An asynchronous notification through proxies of a thread in *another* subsystem must use the scheduling context as determined in §III-C. A notification will only cause a preemption, and immediately execute the receiving thread if all schedulers agree that the receiving thread has higher priority. This consensus decision between schedulers is implemented using TCaps, as each scheduler can associate its priority with a TCap. An asynchronous notification compares this vector of priorities for the client and the server, and only if the priorities are uniformly the same or higher in the server, does a preemption occur. Thus, TCaps act as an efficient and predictable means to make cross-scheduler CPU-allocation decisions, thus avoiding the hierarchical overheads of inter-VM coordination (§II-C, [15]).

Table I demonstrates that communication from within a VM to outside can have significant overhead. It also depends on the VM to orchestrate the communication that could entail scheduling decisions, access to shared runqueues between cores and IPIs, both within and outside the VM. In contrast, Chaos uses the direct, asynchronous communication between isolated subsystems via proxies. Proxies use asynchronous end-points in Composite and TCaps to properly schedule the cross-subsystem, and cross-core communication.

D. Proxy Implementation

Proxies are meant to directly pass data for co-ordination from a client to a server, while orchestrating control flow and scheduling according to the criteria in §III-C. A synchronous invocation is simply an activation of a capability corresponding to the function being invoked (*i.e.* the system call). This simple implementation has a significant benefit: the capabilities are polymorphic to each of the communication mechanisms. Chaos enables the *transparent interposition of proxies on the invocation path* using proxies on the client- and server-side ($Proxy_c$ and $Proxy_s$). Even legacy Composite code that directly uses asynchronous activations harnesses the same technique: Chaos replaces the asynchronous endpoint capability with a synchronous invocation to the proxy. Regardless if the client believes it is invoking a synchronous or asynchronous communication channel, the actual kernel resource referenced by the specific capability is controlled by Chaos, thus activating proxies. This effectively updates interposition agents [38] to a secure, capability-based system.

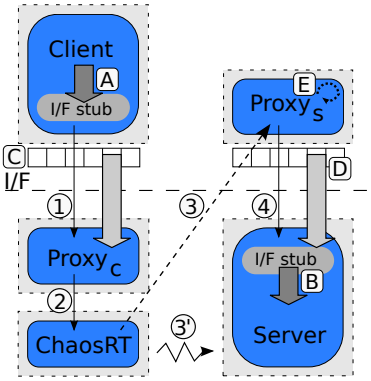

Fig. 7: Proxy interposition for a synchronous invocation. Step-by-step control-flow transfer is shown with numbers in the circles and the solid arrows. Letters indicate labels described in text.

Figure 7 shows the Chaos proxy interposition data- and control-flow for a client synchronously invoking a server's functionality. All proxies rely on client- and server-side interface (I/F) stubs to marshal and demarshal arguments and return values (labeled [A] & [B]). Thus, data is passed between client and server independent of the means of transferring control flow ((1) to (4)):

(1) Chaos replaces the synchronous invocation capability in the client's capability-table ([C]) with a $Proxy_c$-emulated API that makes the client synchronously invoke the proxy instead of the server.

(2) The $Proxy_c$ interacts with the ChaosRT scheduler,

(3) Asynchronous communication (with IPI rate-limiting interposition for inter-core co-ordination) is used to notify $Proxy_s$ of a request, and the thread awaits a reply. If the proxies execute on different cores, this involves sending an IPI ((3')).

(4) $Proxy_s$ then translates the asynchronous requests into the equivalent synchronous invocations ([D]) to the server.

E. Bounded Communication Interference & Latency

Devirtualized tasks indirectly interact with the VM through proxies that route their requests through asynchronous end-

Operation	Composite	seL4	Fiasco.OC	Linux w/ RT
Thread switch	307	327	245	1548
	431	**1231**	**470**	**9380**
Synchronous communication	741	934	1236	17801
	6930	**1428**	**3834**	**185551**
Cross-core communication	2386	4723	7615	8628
	2716	**4934**	**10495**	**52696**

TABLE II: Kernel operations measured in cycles (3200 cycles = μ-second). Synchronous communication is round-trip and cross-core communication is one-way. For each operation, the first row indicates the average costs and the second row indicates the Worst-Case Measured Time (WCMT).

points. If this communication spans cores, IPIs are generated which may interfere with high-criticality execution on the destination core (§II-B). As described in §III-D, Chaos proxies apply rate limits and polling (depicted as (E)) to asynchronous communication between assurance-levels. Instead of modifying the kernel to provide the rate-limiting policy (thus cementing deferrable server policies in the kernel, an undesirable design in a μ-kernel [19]), we observe that the overhead for the synchronous invocation to the proxy is significantly less than the hardware overhead of sending the IPI (see §V-A), and instead define rate-limiting policy in the proxies.

ChaosRT tracks the deferrable server, and asynchronous end-point meta-data information. When invoked by a proxy, ChaosRT uses Composite to directly pass the address of the corresponding end-point's meta-data (avoiding lookups and locks). The number of IPIs sent and the timestamp of the last budget replenishment (a multiple of p_i^{ipi}) are directly updated using atomic instructions to minimize overhead and interference. The proxy within the server environment (*e.g.* legacy VM) awakens either due to notifications, or due to periodic timeouts. Either way, it processes the pending requests placed into the shared memory queue.

V. Evaluation

Most of our evaluations are performed on a Dell Optiplex XE3 running a 3.20 GHz Intel Core i7-8700 8 GB physical memory (less than 256MB are used in Chaos evaluations) with Hyper-threading disabled. The number of physical cores enabled is varied in different experiments. In all our Linux experiments, we use Ubuntu 14.04 with the standard Ubuntu Linux 4.4.0-133 for the VM and a recompiled Linux 4.4.148 with the Real-Time(RT) Linux patch version 4.4.148-rt165 for the host Linux. We used a Intel desktop processor in our experiments to be able to evaluate different reliability-focused operating systems on a common, compabile, multi-core hardware.

This evaluation has a number of goals:

- To understand the performance properties of the underlying Composite kernel operations, and the overheads of Chaos over this Composite baseline.
- To understand the devirtualization overheads in Chaos comparable to Table I.
- To study the interference bounds, end-to-end latency guarantees and scheduling overheads for various rate-limiting server configurations w.r.t rate-limits and server polling periods.
- To understand the ability of ChaosRT to provide strong predictability guarantees to high-criticality safety controller and the real-time cFS subsystem in the presense of IPI interference from a low-assurance subsystem.
- Lastly, to understand the performance of best-effort computation in Chaos relative to existing systems.

A. Microbenchmarks

We conduct a set of micro-benchmarks to measure the average and measured worst-case costs of various kernel operations. Table II presents the costs of dispatch and communication operations in the underlying Composite kernel and comparable operations in other real-time systems. We used sel4bench benchmarking suite for measuring seL4 system call costs, and modified it to collect more samples, and not warm the cache. The average costs are measured over a million iterations (10K iterations in seL4). Synchronous communication uses thread migration in Composite and in L4 variants we use `call` and reply_and_wait. The cross-core communication uses the asynchronous end-points in Composite and `send` and `recv` equivalents in seL4 and Fiasco OC.

Discussion. These results show that the underlying Composite kernel is efficient, relative to existing optimized systems. Taken with Figure 1 where other real-time systems suffered from shared memory contention with an increasing number of cores, these results show that Speck is a strong foundation for Chaos to enable multi-core, predictable execution. However, the use of IPIs in Composite for cross-core communication could cause immense interference as shown in Figure 2. Chaos solves this problem by interposing proxies on the cross-core asynchronous communication and rate-limiting IPIs, as discussed in §III-D.

Table III depicts the indirect IPI rate-limiting costs in ChaosRT for interposing on invocations with proxies and the costs in ChaosRT user-level scheduling.

Thread yield (user-level scheduling)	654
	2166
Cross-core comm. (w/ proxy interposition)	2934
	3323

TABLE III: ChaosRT scheduling and cross-core communication costs in cycles. (Average cost in first row and WCMT in the second row).

Discussion. As expected, the cross-core IPC latency is the cost of raw asynchronous communication in the underlying Composite kernel plus the overhead of a synchronous invocation to the IPI rate-limiting server. This demonstrates that the proxy implementation of the IPI rate-limiting servers is efficient.

B. Chaos Devirtualization *Overheads*

The virtual machine infrastructure presents a number of overheads as studied in Table I. To evaluate the overheads in *devirtualization* in Chaos, we study the round-trip costs of asynchronous communication and interrupt thread activation in different subsystems. In this experiment, we have two threads that execute in, and are scheduled by either the *root*

Round-trip Comm.	$root \leftrightarrow root$	$root \leftrightarrow VM$	$VM \leftrightarrow VM$
Same-Core	1497	1513	1495
Cross-Core	5569	5562	5490

Timer Int. Activation	$root$	VM
	951	901

TABLE IV: Round-trip communication and interrupt activation costs in Chaos, comparing with the virtualization overheads Linux in Table I.

or the VM subsystem, similar to a host process and a VM process respectively in §II-C.

Table IV shows the average costs of round-trip communication between threads in different subsystems on the same core, and cross-core. It also shows the interrupt response time in different subsystems. This measures communication between threads that do not directly share a scheduler. To measure the round-trip communication latency, threads in subsystems $root$ and VM are associated with a receive endpoint and have a send capability to the thread in the other subsystem. To measure the interrupt delivery latency, a low-priority thread spins updating a shared timestamp variable and the high-priority thread is attached to the HPET interrupt in that subsystem.

Discussion. TCaps enable direct asynchronous notification delivery, regardless of depth in the scheduling hierarchy of the sender and receiver. This is shown here for both for inter-thread communication, and for interrupt delivery. Compared to Table I, Chaos is able to devirtualize tasks thus ensure bounded inter-assurance-level interference, without adding significant overheads. In this way, Chaos provides a light-weight alternative to VMs with strong, fine-grained isolation.

C. Bounding IPI Interference

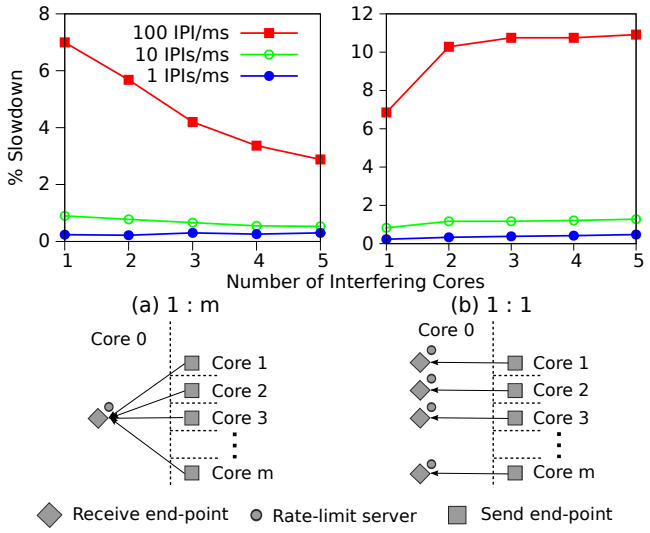

Fig. 8: Measured %-slowdown of high-priority execution on a target core with different IPI rate-limiting factors and an increasing number of cores sending to lower-priority tasks on the target core. (a) with aggregate IPI rate-limiting factors. (b) with per-core IPI rate-limiting factors.

In the Composite kernel, cross-core, asynchronous communication sends IPIs to the target core. ChaosRT interposes on asynchronous notifications, and implements *rate-limiting deferrable servers* at user-level. Chaos's *rate-limiting* mechanism bounds the IPI interference on high-criticality execution from possibly malicious low-assurance software running on other cores, as discussed in §III-D. To study the IPI interference bounds from different rate-limiting server configurations, we evaluate the %-slowdown of a high-criticality task monopolizing a target core with a number of interfering cores flooding with asynchronous notifications, similar to Figure 2.

Figure 8 shows two different setups and the %-slowdown of a high-criticality task on core 0, with m interfering cores each flooding with asynchronous notifications. (a) and (b) send notifications to 1 and m asynchronous receive end-point(s) on the target core respectively. The high-criticality, high-priority task on the target core executes a 1 millisecond workload in both (a) and (b). The 1 millisecond workload is chosen and not 10 μ-seconds similar to Figure 2, because the workload has to be small enough to be negatively impacted by the double-hit in the deferrable-server policy and large enough to demonstrate the impact of rate-limiting.

The interference and therefore the %-slowdown in high-criticality execution is expected to remain constant in (a) as we place an aggregate IPI rate-limit for any number of interfering cores and expect a linear increase in slowdown in (b) as we increase the number of interfering cores. However, Intel's APIC design limits IPIs to a maximum of two pending requests per interrupt line (in this case, interrupt line for asynchronous communication) and therefore the %-slowdown actually decreases with the increase in the number of interfering cores and plateaus in (b) for higher rate-limiting factors and/or higher number of interfering cores, as the IPIs are coalesced.

Discussion. As expected, in (a), increasing the number of cores shows a decrease in the %-slowdown of high-criticality execution because of the aggregate rate-limit plus the APIC limitation of two pending requests per interrupt line. (b) also shows the expected increase in the %-slowdown with the increasing number of interfering cores with per-core rate-limits, and plateaus for higher rate-limiting factors and higher number of cores. It is important to note that, while the Composite system exhibits unbounded interference and livelocks as shown in Figure 2, Chaos's *rate-limiting* technique enable the system to limit the interference from IPIs over windows of time thereby smoothing-out and bounding the amount of interference in that window.

D. IPC Latency Trade-off in Rate-Limiting

IPI *rate-limiting* requires that the receiving subsystem involved in the communication poll for requests as the requests that exceed a given rate do not cause IPIs or send notifications. To evaluate the latency, interference trade-off, Figure 9 studies how polling rates, and IPI rate-limits impact communication latency. The client on one core sends 50 notifications uniformly throughout a millisecond to a server running on another core.

Discussion. The average latency while the IPI rate or the send rate is lower than the rate limit is constant for different polling periods. Once it increases above the limit or the rate-limiting

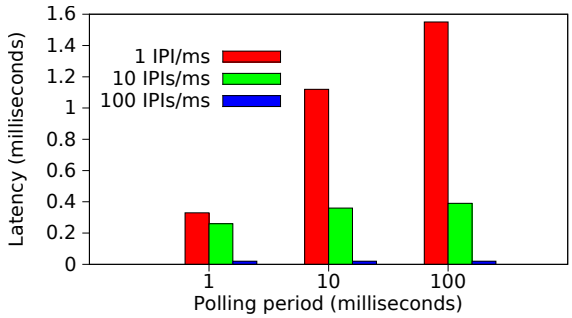

Fig. 9: Average latency with uniformly spaced asynchronous requests for different IPI rate-limiting factors with varying server polling periods.

factor is decreased to below the send rate, then the latency increases with the polling frequency. This result demonstrates that the polling rate enables bounds to be placed on the latency, while IPIs enable fast notifications.

E. Polling Overheads

Polling servers induce extra scheduling overheads in the system as the schedulers are required to track their timeout and activate the threads on their timeout expiry. To evaluate the polling scheduling overheads, we run a low-priority task with a 10 second workload and a high-priority task polling at different rates. We compare against the baseline of the workload with no polling.

Table V shows the %-slowdown in workload execution with different polling periods. We evaluate polling in the scheduler (to receive scheduler notifications from other cores), and proxies performing polling in another component.

Polling	Scheduler / Application	% Slowdown
1 ms	Scheduler	0.044
	Application	0.047
10 ms	Scheduler	0.039
	Application	0.038
100 ms	Scheduler	0.029
	Application	0.031

TABLE V: Polling overheads in schedulers and applications.

Discussion. The polling slowdown is small enough that it is essentially "in the noise" of the system. Given these measurements, we do not expect polling to cause prohibitive overhead.

F. Real-time Predictability

To evaluate the predictability properties of Chaos we conduct experiments that measure the round-trip time in sensor and actuator processing in cFS that is described in §III-A. The different systems that we evaluate: (1) LinuxRT running the cFS with all the applications co-hosted in the same protection domain as the cFS system (as in Figure 4(a)). (2) The Chaos system where the applications are spatially isolated and communicate with the cFS core subsystem using synchronous IPC. The cFS system's ground station communication harnesses the NetBSD networking services running on a separate core, through ChaosRT proxies. (3) The Chaos system with the high-criticality safety controller execution isolated from the cFS subsystem. The high-criticality safety controller interacts with the cFS subsystem using proxies. (4) The Chaos and LinuxRT systems with cross-core interference on the core running the cFS system and its applications. cFS applications are spatially isolated from the core cFS in Chaos and the interactions with the core cFS are through synchronous invocation. The cFS system interaction with the COSMOS ground-station is via OpenSatKit's KIT_TO (Telemetry Output) and KIT_CI(Command Ingress) applications. Normally, OpenSatKit interacts with a simulator of the physical model of satellite dynamics called 42. It is hard to achieve real-time behavior with the simulator in the loop, so we have emulated the sensor data by capturing a trace from the 42 Simulator. This sensor information is sent every 500 ms. We use HPET periodic timers to emulate the 500 millisecond periods and replay the sensor data to the cFS system. In LinuxRT environment, we use *timerfd* functionality to emulate the sensor periodicity. The Round-Trip Time (RTT) is measured from the activation of the interrupt thread to the end of sensor processing and actuator data output. We measure the RTT for a thousand iterations and Table VI plots the average and worst-case measured costs in cycles for each system.

System	Average	WCMT	σ
Chaos w/ NetBSD	98817	124033	5212.5
Chaos w/ NetBSD and safety controller *devirtualized* from cFS	101250	126214	5030.2
safety controller response time	2484	3206	88.4
LinuxRT	85478	247817	9084.1
Chaos w/ NetBSD w/ IPI RL=1/500ms w/ interference	98909	127819	5085.6
LinuxRT w/ interference	267584	450277	18265.0
slowdown vs Chaos	2.7x	3.5x	

TABLE VI: cFS Sensor Round-Trip Time (RTT) Average, WCMT and Standard Deviation (σ) costs.

Discussion. The measured RTT between sensor and actuator in Chaos is slightly higher than LinuxRT. The additional overhead is due to the increased application isolation – which includes a large amount of data copying – that Chaos added.

We saw in §II, that Composite suffers from IPI interference, and LinuxRT suffers from shared run-queue contention interference. Here we determine if this complex software infrastructure can be adversarially impacted by low-assurance tasks on other cores. The last two evaluations in Table VI depict these results. For Chaos, four cores generate asynchronous notifications, but the IPI (and polling) rate is set commensurate with the sensor rate. In contrast, for LinuxRT, we use a tight loop of `sched_yield` on four cores to generate runqueue contention.

Discussion. The LinuxRT cFS execution is slowed down significantly which affect both the measured worst-case and the average-case latencies. The rate limiting and polling overheads in Chaos do not impact the responsiveness of the system. It is important to note that Chaos lowers the average latency by a factor of 2.7, while reducing the worst-case by a factor of 3.5 over LinuxRT with interference.

G. Best-effort Throughput

In order to assess the best-effort performance, we study the throughput of the system. We study the network UDP throughput performance of Linux, NetBSD and Chaos running iperf3 version 3.1.3. As Chaos uses the NetBSD stack for networking and device drivers, we compare these similar best-effort software code-bases. While NetBSD's device drivers and the networking stack are in the kernel, Chaos runs a non-preemptive Rumpkernel that hosts NetBSD drivers in a user-level component. NetBSD 7.1 and Ubuntu 14.04 running Linux 4.4.0-133 are used in this experiment. The network controller used is a Intel 82571EB Gigabit NIC. All systems run iperf3 as a UDP server and we measure the throughput with a iperf3 UDP client running on a host machine.

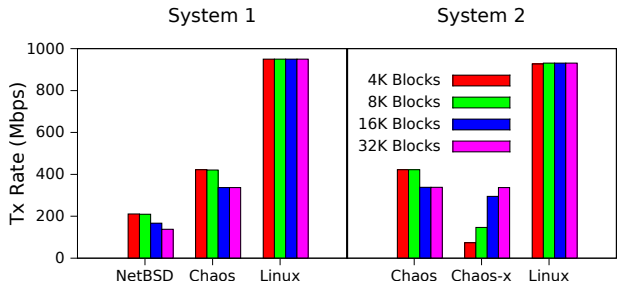

Fig. 10: Throughput of iperf3 server running on different systems.

Figure 10 presents the iperf3 throughput for different block sizes on two different hardware setups. Because of hardware compatibility issues with the NetBSD, we used an older hardware, system 1, running Intel Core i7-2600S processor running at 2.80 GHz with 4 GB of physical memory. We evaluate NetBSD, Linux and Chaos systems on system 1, and on system 2, which is the same Dell Optiplex XE3 used in the rest of the evaluations, we evaluate Linux, Chaos and Chaos-x. In Chaos-x, iperf3 is running on a separate core with cross-core communication through ChaosRT *rate-limiting* proxy interposing on NetBSD syscalls. Note that, the number of datagrams transmitted/received decrease with an increase in block size. The Linux system uses a far more efficient networking stack and the throughput measurements presented here are for context only.

Discussion. Chaos is at least as performant as an existing software code-base that has same device drivers and networking stack, NetBSD. We believe that the improved throughput of Chaos over NetBSD is due to (1) the best-effort NetBSD software in Chaos runs as a library thus avoiding virtual memory overheads. (2) the non-preemptive design of the Rumpkernel unikernel that cause fewer context switches.

The Chaos-x on the other hand shows better throughput for larger block sizes. With the large block sizes, Chaos-x is comparable to Chaos with iperf3 running on the same core. For larger block sizes, fewer datagrams are transmitted between the Rumpkernel subsystem and iperf3 running on a separate core, thereby incurring fewer cross-core communication overheads. For smaller block sizes, the hardware IPI overheads can cause significant performance degradation.

VI. Related Work

Chaos devirtualizes high-criticality tasks to remove interference, often harnessing inter-core communication. A number of other systems structurally encourage inter-core communication. To reduce overheads for using multicores, systems have partitioned kernel- and user-level across cores to minimize micro-architectural interference [39], dedicated a core to scheduling and accessed it via message passing [40], and designed kernels around message passing [41]. Systems such as MC-IPC [42] have added mixed-criticality constraints into the managing of parallel requests for service from a server. In contrast, Chaos takes a simple and practical approach to inter-core communication: efficient event notification, with rate limits to constrain interference, and polling to provide latency guarantees.

A strong form of isolation segregates hardware across software boundaries, either by running different criticalities on different cores [14], [13], or by multiplexing cores across VMs [43] with expensive context switches that include flushing the caches. Going further, shared hardware such as caches and memory can be carefully partitioned (as in MARACAS [44]) to further constrain interference. With these approaches, the interference between VMs is quite limited, but processing throughput is wasted, especially with best-effort tasks that wish to maximize throughput. In contrast, other systems [45] have shown the benefit of dynamically managing budget and locality to better use spare resources. In this vain, Chaos enables cores to host VMs of multiple criticalities, isolates the interference from inter-core communication, and removes inter-criticality, contention on shared data-structures.

VII. Conclusions

Chaos is motivated by the twin goals of effectively using the increased throughput of multi-core machines, and ensuring the necessary isolation between tasks of different criticalities and assurance-levels. Specifically, we have demonstrated that the inter-core coordination necessary in existing systems can cause undue and significant interference on high-criticality tasks due to shared memory contention on key data-structures, and IPI processing. To remove such overheads, Chaos devirtualizes high-criticality tasks by exporting them out of subsystems (*e.g.* out of VMs) and into a minimal ChaosRT environment. Yet Chaos maintains the efficient and predictable interaction between those tasks and the higher-functionality VM using proxies that bound both interference and latency. We have shown that Chaos is effective at removing interference due to cross-core coordination, while maintaining high performance. While existing systems suffer from significant cross-core interference from low-assurance tasks (either through shared memory synchronization, or IPIs), Chaos controls interference, while not resorting to partitioning. For a sensor/actuation loop in satellite software experiencing inter-core interference, Chaos lowers processing latency by a factor of 2.7, while reducing worst-case by a factor 3.5 over a real-time Linux variant.

References

[1] A. Burns and R. Davis, "Mixed criticality systems – a review, retrieved feb, 2016," *https://www-users.cs.york.ac.uk/burns/review.pdf*, 2016.

[2] P. Padlewski, "Devirtualization in llvm," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017.

[3] 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 Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 133–150.

[4] "The Fiasco microkernel: http://l4re.org, retrieved 10/6/17."

[5] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12, 2012, pp. 93–102.

[6] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[7] B. B. Brandenburg, "The fmlp+: An asymptotically optimal real-time locking protocol for suspension-aware analysis," in *26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[8] A. Wieder and B. B. Brandenburg, "On spin locks in AUTOSAR: Blocking analysis of fifo, unordered, and priority-ordered spin locks," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013.

[9] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov 2011.

[10] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys)*, 2012.

[11] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 3–19, 1979.

[12] "Userspace RCU: http://liburcu.org/, retrieved 6/16," 2016.

[13] J. M. Rushby, "Design and Verification of Secure Systems," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/800216.806586

[14] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Trans. Comput. Syst.*, 2016.

[15] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[16] N. Amit and M. Wei, "The design and implementation of hyperupcalls," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[17] L. Sha, "Using simplicity to control complexity," *IEEE Softw.*, vol. 18, pp. 20–28, July 2001. [Online]. Available: http://dx.doi.org/10.1109/MS.2001.936213

[18] E. Armbrust, J. Song, G. Bloom, and G. Parmer, "On spatial isolation for mixed criticality, embedded systems," in *2nd International Workshop on Mixed Criticality Systems (WMC)*, 2014.

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

[20] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2010.

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

[22] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2003, p. 251.

[23] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017.

[24] S. Ruocco, "A real-time programmer's tour of general-purpose l4 microkernels," in *EURASIP Journal on Embedded Systems*, vol. 2008, no. 234710, 2008.

[25] U. Steinberg, J. Wolter, and H. Hartig, "Fast component interaction for real-time systems," in *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 89–97.

[26] U. Steinberg, A. Bottcher, and B. Kauer, "Timeslice donation in component-based systems," in *OSPERT*, 2010.

[27] 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 Thirteenth EuroSys Conference (Eurosys)*, 2018.

[28] G. Parmer and R. West, "Predictable and configurable component-based scheduling in the Composite OS," *ACM Transactions on Embedded Computer Systems*, vol. 13, no. 1s, pp. 32:1–32:26, Dec. 2013.

[29] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2005, pp. 17–30.

[30] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.

[31] M. Stanovich, T. P. Baker, A.-I. Wang, and M. G. Harbour, "Defects of the posix sporadic server and how to correct them," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 35–45, 2010.

[32] G. Bernat and A. Burns, "New results on fixed priority aperiodic servers," in *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, Washington, DC, USA, Dec 1999, p. 68.

[33] A. Kantee, "Rump file systems: Kernel code reborn," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, 2009.

[34] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 26, no. 1, pp. 29–35, 1983.

[35] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[36] J. S. Shapiro and S. Weber, "Verifying the eros confinement mechanism," in *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2000, p. 166.

[37] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[38] M. B. Jones, "Interposition agents: Transparently interposing user code at the system interface," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993, pp. 80–93.

[39] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the conference on Symposium on Operating Systems Design & Implementation*, 2010.

[40] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[41] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Symposium on Operating System Principles (SOSP)*, 2009.

[42] B. B. Brandenburg, "A synchronous ipc protocol for predictable access to shared resources in mixed-criticality systems," *2014 IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[43] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *2013 IEEE Symposium on Security and Privacy (SP)*, May 2013, pp. 415–429.

[44] Y. Ye, R. West, J. Zhang, and Z. Cheng, "Maracas: A real-time multicore vcpu scheduling framework," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016.

[45] S. Groesbrink, L. Almeida, M. de Sousa, and S. M. Petters, "Towards certifiable adaptive reservations for hypervisor-based virtualization," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.