Slite: OS Support for Near Zero-Cost, Configurable Scheduling *

Phani Kishore Gadepalli, Runyu Pan, Gabriel Parmer
The George Washington University
Washington, DC
{phanikishoreg,panrunyu,gparmer}@gwu.edu

***Abstract*—Despite over 35 years of wildly changing requirements and applications, real-time systems have treated the kernel implementation of system scheduling policy as given. New time management policies are either adapted to the kernel, and rarely adopted, or emulated in user-level under the restriction that they must stay within the confines of the underlying system's mechanisms. This not only hinders the agility of the system to adapt to new requirements, but also harms non-functional properties of the system such as the effective use of parallelism, and the application of isolation to promote security.**

This paper introduces Slite, a system designed to investigate the possibilities of *near zero-cost scheduling* of *system-level threads* at *user-level*. This system efficiently and predictably enables the user-level implementation of configurable scheduling policies. This capability has wide-ranging impact, and we investigate how it can increase the isolation, thus dependability, of a user-level real-time OS, and how it can provide a real-time parallel runtime with better analytical properties using thread-based – rather than the conventional task-based – scheduling. We believe these benefits motivate a strong reconsideration of the fundamental scheduling structure in future real-time systems.

I. Introduction

Real-time systems have a broad breadth of requirements, many of which have only come to prevalence only in the last decade. These include strong security, reliability, effective use of multi-core, and the ability to meet the timing requirements of software of varying assurances and functionalities. Despite the flux in system capabilities and requirements, scheduling has maintained the same essential structure since the first systems that required temporal multiplexing. Due to the close interaction with system's concurrency and parallelism primitives – including the synchronization, interrupts, and coordination facilities – the scheduler is a fundamental service of the kernel. As scheduling is effectively baked into the system, this leads to a tension with the constant pressure to adapt the timing properties of the system to new environments and specifications.

On the other hand, the immensely varying systems *have* yielded new scheduling challenges and directions: multi-core systems have spawned compiler and library support for run-to-completion task scheduling exemplified by OpenMP [1], Cilk [2], and Intel TBB [3]; middleware systems used for coordination of applications have motivated user-level control of scheduling for QoS [4], [5], while coarse CPU share control is a fundamental aspect of container runtimes [6], [7]; mixed-criticality systems [8] require new scheduling and synchronization policies, whose adoption is complicated by the all-or-nothing replacement of existing policies; the invalidation of assumptions about non-preemptivity in virtual machines leads to degenerate spinlock behavior which requires cross-layer scheduler coordination [9]; motivated by security and architectural goals, non-preemptive, user-level control of dispatching between protection domains [10], [11]; information leakage security concerns in real-time scheduling and synchronization [12] require active obfuscation logic in the scheduler [13], while architectural support for speculation requires new dispatching mechanisms to maintain confidentiality [14]. Even the scheduling heuristics in monolithic POSIX systems are evolving [15] to the challenging multi-core landscape.

The continual introduction of new timing and coordination behaviors demands new system functionality and temporal semantics. It is the position of this paper that *the fundamental structure of scheduling should address the increasingly complicated timing landscape of modern systems*. It should be possible for *untrusted developers* to implement scheduling and synchronization facilities that have the potential to control system execution with the fidelity of a system scheduler.

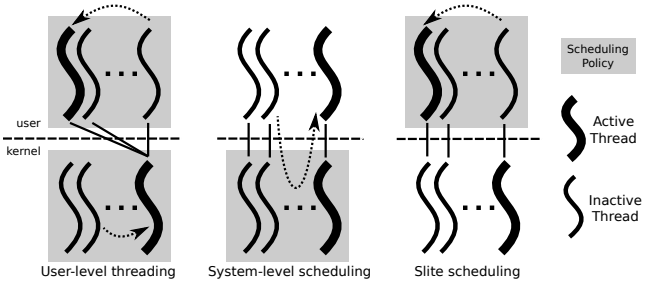

Fig. 1: Three scheduling models. Dashed arrows indicate the path for context switches. Green threads on the left contend with decoupled scheduling policies and threads. System-level scheduling manages threads across all applications. Slite defines policy only at user-level, and enables direct user-level dispatch. This allows inconsistency between which thread user- and kernel-level see as active.

Figure 1 depicts user-level threading (often called "green threads") which implements dispatching functionality and scheduling policy in a library at user-level, and normal system-level scheduling in which scheduling policy and dispatch logic are kernel-defined. The former empowers the user-level scheduler to control only threads within its own scheduler, and has limited visibility of system-wide thread state, thus complicating coordination between user- and kernel-level policies. This has led to complicated schemes for coordinating around kernel thread blocking and waking events [16], application allocation to cores [17], and efficient, predictable locking [9]. The latter

*This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS-1815690. 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 NSF.

2642-7346/20/$31.00 ©2020 IEEE
DOI 10.1109/RTAS48715.2020.000-9

makes application-specific scheduling logic implausible, and requires overhead (including system calls) to access scheduling and synchronization constructs.

We argue that the kernel-based implementation of scheduling policy is the key inhibitor to configurable policy which is preventing a tighter co-design of applications, systems, and schedulers. Elphinstone et al. [18] argue that time management is one of the last resource management policies to resist configurable implementation at user-level in μ-kernels. Though past work [19], [20], [21] has focused on this goal, it requires costly system calls and extensive application/kernel coordination.

Slite (on the right in Figure 1) – short for "scheduling light" – introduces a mechanism in which both the user- and kernel-level track the same set of threads, yet user-level defines scheduling policy and dispatching logic. In common cases, this enables both near zero-cost, direct thread dispatch without kernel involvement, and the integration of applications with timing policy. Real-time and time-sensitive systems benefit from this both due to the decrease in synchronization overheads (for example, blocking operations on locks and message queues), and the use of specialized policy for timely execution.

Slite allows direct user-level thread switches to cause *incoherence* with the kernel's notion of the currently active thread. Later, the kernel recovers coherence when activated. We pair this with non-blocking OS APIs, even for Inter-Process Communication (IPC), to enable the system-level scheduling of threads that execute across different protection domains. By providing efficient access to timing policy at user-level, Slite counters the intuition that extracting a policy from the kernel decreases performance. Similar to research that moves networking to user-level [22], Slite effectively implements "kernel-bypass" for scheduling.

This paper asks if the *user-level dispatching* of *system-level threads* has *near-zero cost*, can we more effectively meet the demands of modern applications by *integrating them with scheduling logic*? Thus, we focus on demonstrating the utility of co-designing the timing policy and the application. Toward this, we focus on two domains:

- increasing system security in a Real-Time Operating System (RTOS) by enabling the isolated, user-level execution of system services with the predictability and performance properties of a legacy, unprotected, kernel-resident RTOS; and
- enhancing a task-based, non-preemptive parallel runtime often used in real-time systems (*OpenMP*) that has significant worst-case synchronization constraints that prevent work-conserving execution, with a thread-based execution model that avoids these problems, thus enabling the use of high-utilization real-time schedulability analysis.

Both of these applications are only possible given the efficient thread dispatch and coordination facilities of Slite. They demonstrate that a re-imagining of the structure of system scheduling can have wide-ranging implications for dependable, parallel, and predictable systems.

Contributions. This paper's contributions include

- the design (§III) and implementation (§IV) of the efficient, predictable user-level scheduling of system threads,
- the application of Slite to a RTOS (§VI) focused on bare-metal performance while providing strong isolation, and a parallel runtime (§VII) to demonstrate the effectiveness of integrating scheduling with applications, and
- the evaluation (§VIII) of Slite and its application domains.

II. Background and Related Work

Configurable kernel scheduling (CKS). There is a long tradition of extensible frameworks in the kernel for scheduling policy customization. The frameworks range from hierarchical composition of schedulers [23], to languages for extensibility [24], to APIs for pluggable policies [25]. Though these frameworks demonstrate the utility of policy customization, these techniques focus on kernel-level policies (Figure 1, center) without a tight co-design with applications.

User-Level Scheduling (ULS). Real-time frameworks [5], and concurrency management within processes in response to IPC [26] use user-level scheduling (Figure 1, left) to manage threads *within that protection domain*. These approaches cannot schedule threads in other protection domains. ULS enables efficient thread dispatch, but requires consideration of and integration with blocking system calls. Scheduler Activations [16] define a protocol for dynamically binding kernel- to user-level threads based on blocking system call usage. Unfortunately, systems that have implemented scheduler activations – Solaris being the most prominent – have since removed them due to implementation complexity. Slite instead maintains a one-to-one correspondence between user and kernel contexts but enables active thread user/kernel incoherence.

Middleware Schedulers (MWS). An alternate approach uses the kernel's APIs around prioritization to indirectly control the active thread, for example, by explicitly assigning priorities so that only a single target thread has the highest priority [4], [27]. This enables the scheduling of threads even outside of the scheduler's protection domain by cleverly utilizing existing kernel policies and abstractions. These systems have a number of challenges: (1) These systems not only require kernel-mediated dispatch, but also have the overhead of double the kernel context switches (*i.e.* by switching to a scheduler thread) and of the APIs for priority management. (2) Thread's blocking on system calls, and more importantly, waking up at non-deterministic times provides a challenge for the scheduler to accurately track its state. Aswathanarayana et al. [4] trusts applications and modifies libc to catch all syscall returns, while Lyons et al. [27] narrows the scope to track only thread timeouts and cooperative blocking thus preventing, for example, general synchronous IPC with servers. Fiasco L4 [28] has kernel API support for directed yields between threads, and the association of prioritized CPU reservations with those threads. This simplifies user-level scheduler design, but – similar to [27] – the user-level schedulers are not integrated with the system-wide blocking semantics. This limits thread interactions with blocking kernel APIs (*e.g.* for IPC).

In contrast to these systems, Slite enables the direct dispatch between threads in the scheduler's protection domain. It also

System	Policy Location	Dispatch Mechanism	Scope of Concurrency Control	Preemptivity
CKS [23], [24], [25]	● kernel-level	● kernel-level	● inter-process, system scheduling	● preemptive
ULS [5], [26], [16]	● user-level	● user-level	● process-local†	● preemptive
MWS [4], [27], [28]	● user-level*	● API-driven dispatch	● inter-process, system scheduling	● preemptive
PSS (*e.g. OpenMP*)	● user-level	● user-level, closure activation	● process-local, cooperative	● non-preemptive
UL-SS [21], [19], [20]	● user-level	● kernel-level	● inter-process, system scheduling	● preemptive
Slite	● user-level	● user-level	● inter-process, system scheduling	● preemptive

TABLE I: A summary of configurable scheduling research. Acronyms reference the classifications defined in §II. Options are color-coded from efficient and flexible (●) to slower and less flexible (●). The † indicates complicated integration with kernel blocking/wakeup behaviors that exhibit additional overhead, and * denotes that these techniques rely on limited use of the blocking/waking APIs of the system.

leverages existing OS support in Composite to avoid blocking system calls even for protection domain traversing IPC, thus enabling the scheduler to define the blocking and waking system semantics.

User-Level, System Scheduling (UL-SS). Previous scheduling systems [19], [21], [20] were designed for the *user-level control* of *system-level* threads (*i.e.* across multiple protection domains). They enable the vectoring of thread block and wakeup events to schedulers [19], [21]. In contrast, Composite [20] takes a clean-slate approach and designs a minimal μ-kernel that avoids blocking APIs, instead relying on explicit communication with the scheduler *components* for concurrency. IPC between protection domains uses thread migration [29], thus avoiding scheduler interactions on IPC. However, such approaches require kernel mediation of thread dispatch, thus preventing its use in applications that require near-zero-cost scheduling.

Parallel system scheduling (PSS). Large-scale multi-core systems have challenged the ability of real-time systems to schedule and synchronize effectively. Applications that wish to leverage the system's parallelism to increase performance often use parallel runtimes, and significant work in real-time systems has used *OpenMP*. The scheduling of parallel tasks in such runtimes [30], [31], [32] for real-time processing offers a unique set of challenges.

To maximize the use of a large number of cores, parallel runtimes break computation into fine-grained *tasks* that execute non-preemptively and run-to-completion by default on a smaller number of system threads. They often define *scheduling points* when a task performs an operation that delays its progress (for example, awaiting another task's completion) at which point the runtime runs a pending task on the same thread. A set of common features complicate the ability of a system to maintain high utilization while analytically meeting deadlines: (1) scheduling points at which a task effectively blocks, (2) at which point the thread context is re-used to execute pending tasks, (3) while access to intra-task shared data-structures uses critical section primitives, and (4) tasks execute non-preemptively. Deadlock can easily occur if a task holding a critical section hits a scheduling point, and the same thread executes a task that attempts to enter the same critical section. *OpenMP* solves this deadlock using runtime-provided *avoidance*. To avoid this deadlock, all tasks are, by default, *tied tasks* [33], [34] that constrain the runtime from ever executing a task that could possibly attempt to take such a critical section. Unfortunately, this runtime constraint is particularly challenging in real-time systems, as it prevents work conservation [33], [34]. We investigate the maximum impact of the runtime constraint on worst-case deadline calculations for parallel tasks, and can *easily trigger worst-cases 100% larger than without the constraint.* For more details, see the Appendix A. Additionally, effective schedulability analysis for parallel runtimes [35], [30] (based on earlier versions of *OpenMP*) make a strong assumption of work conservation. This means that, effectively, *high-utilization analysis cannot apply to default OpenMP programs*.

In contrast, Slite focuses on a *preemptive scheduling* runtime based on *threads rather than tasks*. Each *OpenMP* task is executed in its own thread, thus avoiding this deadlock by design. SliteOMP, a Slite-based *OpenMP* runtime, leverages the preemptive task scheduling and Slite's near-zero-cost scheduling, and aims to be at least as efficient as traditional *OpenMP* implementations based on the efficient task model, while also providing work conservation.

III. Slite Design

The goals of Slite's design are:

- *User-level dispatch.* Thread context switches avoid system-calls in the common case. This enables scheduling libraries to be tightly paired with applications with near zero-cost scheduling, and enables scheduling components to efficiently manage threads.
- *Scheduling policy customization.* As the scheduling policy is library- or scheduling component-defined, applications configure it appropriately. The Slite infrastructure should make writing policies simple by abstracting the synchronization and thread management functions.

Slite's user-level dispatch causes the active thread (*e.g.* `current` in Linux) to be *incoherent* between user-level scheduling code, and kernel-level. The main challenges in the design of Slite involve enabling incoherence to empower the avoidance of user-kernel transitions. Fast user-level dispatching, thus user/kernel incoherence, requires (1) lazy user/kernel synchronization to re-achieve coherence when necessary, (2) user/kernel synchronization that avoids mutual exclusion as the kernel cannot trust user-level schedulers, and (3) synchronization facilities within scheduling libraries.

A. User-Kernel Synchronization

There are two causes for the inconsistency between user- and kernel-level's view of the active thread: (1) direct user-level switches update user-level scheduler's active thread, and

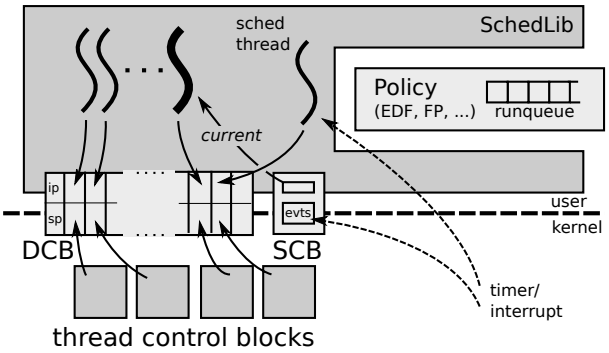

Fig. 2: The thread and scheduling data-structures in Slite. *SchedLib* enables configurable policy, and tracks threads and synchronizes with the kernel using shared structures.

(2) kernel-level interrupts lead to thread switches that update the kernel's active thread. In these cases, the kernel and user-level scheduler, respectively, need to re-achieve a coherent view of the active thread. To achieve this, Slite uses two shared memory regions between user-level and the kernel for synchronization: the Dispatch Control Block (*DCB*) which contains a set of entries, one per-thread, and the Scheduler Control Block (*SCB*) which contains per-core data. The *SCB* includes a token identifying the currently active thread (see *current* in Figure 2). The *current* value enables the user-level scheduler to asynchronously notify the kernel when it dispatches to another thread. Later, when a system-call, exception, or interrupt triggers kernel execution, it determines if *current* has changed, and updates its record of the active thread (*e.g.* to save the preempted registers into).

The *DCB* synchronizes each thread's register contents. As the user-level scheduler dispatches away from a thread, it saves its general purpose registers onto its stack, and its *stack pointer (sp)* and *instruction pointer (ip)* into the *DCB*. If the kernel later must switch to that thread (due to an interrupt, or through a cross-protection domain dispatch), it checks if the *DCB ip*/*sp* are non-zero, and if so, it restores those registers.

When the kernel activates a thread, it switches threads without scheduler involvement. In doing so, it sends an activation notification to the component scheduler including the number of cycles the previous thread ran before preemption. Slite uses a wait-free queue (labeled `evts` in Figure 2) in the *SCB* which the kernel uses to pass these events to the component scheduler. This queue ensures the independent progress of both the user-level scheduler and kernel. If the kernel needs to send more events than fit into the queue, a system call retrieves the next batch of events, thus amortizing the system call overhead.

When a thread is preempted, the kernel saves all of its registers. Thus, when the user-level scheduler wants to switch back to the preempted thread, it cannot use the user-level dispatch fastpath, and must do so using kernel dispatch. Real-time systems often bound the rate of preemptions, thus bounding the frequency of kernel-mediated dispatches.

Security concerns. As the kernel and user-level share the *DCB* and *SCB*, Slite's design is careful to prevent undue kernel manipulation. When a thread's *ip* and *sp* are in a scheduler's *DCB* – and are to be used when dispatching to that thread – the kernel validates that the thread was last executing in user-level in that address space, before using them. This validates that the kernel uses a thread's register contents in the *DCB* only if the scheduler was allowed to previously dispatch way from that thread. Only the kernel publishes events to the *SCB*'s event queue, and any user-level corruption can only delay or impair event deliver to that user-level scheduler.

B. Modular Scheduler Library

To enable real-time system developers to easily implement scheduling policies, we provide a *scheduling library* (*SchedLib*) that encodes the common logic to handle concurrency, parallelism, and synchronization at user-level. The *SchedLib* exports a number of functions for (1) creating and destroying threads, (2) blocking and waking threads, (3) periodic or one-shot timeouts, and (4) abstracting task parameters.

Concurrency control. *SchedLib* includes data-structures for threads and a timeout queue. Without access to instructions to enable and disable interrupts, *SchedLib* serializes access to these data-structures with a single per-core lock that implements priority-inheritance [36]. The lock is implemented using an atomic *compare-and-swap* instruction and user-level dispatch upon contention. As the lock variable is per-core, the `lock` prefix is not used on x86 to improve performance by preventing cross-core synchronization. A per-core *scheduler thread* is activated in response to kernel timer interrupts, and pending kernel events.

Extensible policy. Figure 2 depicts the configurable scheduling policy that manages its own runqueue. *SchedLib* enables extensible scheduling policy modules by requiring each policy to implement an API including functions to (1) add a new thread, (2) program and receive one-shot, cycle-accurate timers (hardware permitting), (3) modify thread's parameters (*e.g.* priority and periodicity), and (4) block or wakeup threads. We've implemented preemptive and non-preemptive fixed-priority (152 lines of code), earliest deadline first (265 lines of code), and rate-limiting servers (deferrable and sporadic).

IV. Slite Implementation

The implementation of Slite requires architecture-specific dispatch code and a tight interplay with the kernel. Regardless, we believe the core abstractions are general as we've implemented them on both x86 and ARM Cortex-M7 microprocessors, two dissimilar architectures.

The *SCB* and *DCB*. The *SCB* maintains per-core information that the scheduler and kernel use to synchronize. In addition to the *current* thread identifier and the kernel event queue, the *SCB* contains a *scheduler epoch*. The latter solves a race between when the scheduling policy chooses to switch to a thread, and when it actually dispatches to that thread. A preemption at which the kernel activates a higher-priority thread can change the policy's intended target, thus this epoch enables the detection of *stale* scheduling decisions.

Near zero-cost thread-switches. During a direct user-level context switch, the *SchedLib* must update (1) the thread's *ip*

```
1  ; Previously: read scheduler epoch, policy decision,
2  ;           confirm no kernel events, save registers
3  store restore_addr -> *%curr_thd_dcb.ip
4  store %sp -> *%curr_thd_dcb.sp
5  load *%next_thd_dcb.sp -> %tmp_sp
6  je %tmp_sp, 0, slowpath_addr
7  store %next_thd -> *scb_active_thd
8  move %tmp_sp -> %sp
9  jmp *%next_thd_dcb.ip
10
11 slowpath_addr:
12 call slowpath_dispatch(next_thd)
13
14 .align 8
15 restore_addr:
16 store 0 -> *%next_thd_dcb.sp
17 .align 8
18 kern_restore_addr:
19 ; restore registers, check for stale scheduler epoch
```

Fig. 3: Slite x86 implementation of direct user-level dispatch.

and *sp* in the *DCB*, (2) the *current* thread capability in the *SCB*, and (3) its local copy of the scheduler *epoch*. The variables are spread across multiple words which raises the question of how to atomically update all of them. To reason about this, we organize the system around some scheduling invariants.

Scheduling invariants. A combination of the scheduler (annotated usr) and the kernel (kern) ensure a number of *DCB* invariants:

- A thread's *sp* is zero when executing (usr and kern) or when preempted (kern).
- A thread saves its *ip* before its *sp* (usr), thus a non-zero *sp* implies a valid *ip*.
- A thread's *ip* and *sp* are only set to non-zero values by the thread itself (usr).
- A thread's *sp* is set to zero (usr) upon dispatching to it,
- If and only if *sp* (thus *ip*) is non-zero, the *DCB*'s *ip* and *sp* are used to restore the thread's context (usr and kern).
- The *current thread* in the *SCB* is always representative of a thread which can be activated (with valid *sp*/*ip*) (usr).

In addition, the kernel is modified in two ways. (1) When the kernel is entered, it updates its current thread by looking at an active scheduler's *SCB*'s current thread. (2) When the kernel switches to a thread in response to an event (*e.g.* interrupt), it checks the thread's *DCB* record and if the *sp* is non-zero, it restores only the instruction and stack pointer from the *DCB*.

The stylized assembly in Figure 3 details the Slite x86 implementation of direct user-level dispatch which implements the scheduling invariants. Registers start with "%", and have names according to their function. Thread's save their context to be later restored on lines 3 and 4. When switched to, that thread's execution will continue on line 15, and it will proceed to restore its registers. However, if the kernel restores the thread, it will restore the *DCB*'s *ip* + 8 (thus at `kern_restore_addr`) as the register for `%next_thd_dcb` is stale. Line 6 determines if the thread we are switching to can be directly dispatched (*i.e.* if it wasn't preempted) or if should use the kernel slowpath (at line 12). Otherwise, the current thread is updated, and the *ip* and *sp* of the target thread are loaded into registers (lines 8 and 9), thus completing the switch. The *scheduler invariants* result in a proper context switch if preemptions occur at any point in this sequence.

V. Slite Implementation in Composite

So far, §III and §IV have focused on efficiently dispatching to threads within the scheduler's protection domain. These techniques alone enable kernel/user-level scheduler coordination around concurrency. However, to schedule not only threads in the scheduler's protection domain, but also those outside, thus providing a system-level scheduling service, some OS support is required. Toward this, we build Slite on the Composite component-based operating system [20] (http://composite.seas.gwu.edu), and discuss Composite-specific design in this section.

A. Composite Background

Composite is a μ-kernel that focuses composing executable systems from user-level *components* each of which adds functionality to the system. The resources available to a component are strictly access-controlled [37]: each has a separate page-table – to subset accessible memory, and capability-tables [38], [39] – to subset access to abstract resources such as threads, and communication end-points.

User-level, component-based scheduling. Components are *schedulers* if they have access to threads (via capabilities). Activating a thread capability via system call triggers kernel-driven dispatch to that thread. Importantly, that context switch can execute the thread in the scheduler, or in another component where it was previously preempted. The ability to context switch to threads in other components (with sufficient capability-based permission) forms the basis for the ability of a user-level component scheduler to schedule system-wide execution, and also the ability to restrict which subset of threads each scheduler manages in a multi-scheduler system [40], [41].

Components communicate via *thread migration-based IPC* [29] enabled by synchronous invocation (`sinv`) communication end-point capabilities. Thread migration enables the same schedulable entity executing in a *client* component that activates a `sinv` capability to *invoke* a function defined by a *server* in which it continues execution. Later it can *return* (by invoking a `sret` capability) to the client. A per-thread component invocation stack in the kernel tracks these invocations. Unlike synchronous thread rendezvous between threads [42], [18] that involves thread blocking and activation, thread migration avoids such thread state transitions. This is significant: scheduling components – and not kernel – provide thread and synchronization interfaces to block and wakeup threads. Scheduling components have the power and responsibility to manage thread interleaving and synchronization.

Temporal Capabilities (*TCaps*) and interrupts. Despite the ability of schedulers to dispatch threads, interrupts significantly complicate user-level scheduling. In μ-kernels that export device drivers to user-level such as Composite, interrupt execution (with the exception of the timer-interrupt) causes the activation of an *interrupt thread* which handles the logic for the device. This exports device drivers from the kernel to user-level. However, when an interrupt occurs,

the kernel must decide if it should immediately switch to the interrupt thread, thus preempting current execution. Naively activating a user-level scheduling component to make this decision on each interrupt has significant overhead. Instead, Composite provides a mechanism, called "Temporal Capabilities" (*TCaps*) [41], that enables schedulers to pass relative prioritization information to the kernel that is used to make a scheduling decision upon interrupt.

Each interrupt thread is associated with a *TCap*, as is a thread when it is dispatched to by a scheduler. As such, all thread execution is associated a *TCap*. Each *TCap* is programmed by a scheduler with a priority. Priorities are in a 64-bit namespace, thus easily enabling dynamic priorities that might use the priority as a timeline (as in EDF). Thus, when an interrupt triggers, its handler compares the priority of the active thread's *TCap* with that of the interrupt thread and preeempts accordingly. *TCaps* enable user-level schedulers to control policy, while interrupt service routines are able to quickly make preemption decisions.

Mirroring Slite's active thread incoherence due to user-level dispatch, Composite's kernel interrupt thread dispatch switches threads causing incoherence. To re-achieve coherence, the Composite kernel provides an event channel of interrupt thread activations and their execution times that component schedulers retrieve via a system call. Thus, a component scheduler has an accurate record of active threads, and can maintain cycle-accurate accounting.

TCaps additionally provide means for multiple schedulers in the system to interact by delegating controlled slices of time among each other. Each *TCap* consists of a (cycle-accurate) slice of time, and a set of priorities, one per-scheduler. The former is expended as the *TCap* is used for execution, and the latter enables all schedulers that have delegated time to properly constrain preemption decisions. In this way, *TCaps* provide access control for the computation time of the system.

Multi-scheduler systems. One of the key benefits of Slite, is its ability to integrate customized policy with applications. This implies that *each* application has its own policy, and that a system-level scheduling component multiplexes applications.

Composite supports hierarchical scheduling [40] which defines parent and child scheduler coordination protocols. These protocols revolve around parents scheduling child scheduler threads, while timer-interrupts sent to the root scheduler enable global multiplexing. In contrast, *TCaps* enable even sibling schedulers to coordinate by delegating slices of time.

B. Slite in Composite

Slite adapts Composite by using a thread's capability to denote the *current*ly active thread, and replaces the Composite, in-kernel queue of interrupt events for a scheduler, with the Slite event queue. Slite works with the Composite design to additionally provide scheduling threads across the entire system, and to enable multiple Slite schedulers to co-exist.

System-level scheduling. Slite supports the system-level scheduling of threads not only in a scheduling component, but also those executing in other components. This is possible in Composite as a scheduler can dispatch to a thread preempted in another component, which not only restores the thread's register context, but also switches page-tables. Thread-migration-based IPC means not only that scheduling decisions aren't required on IPC, but also that blocking and synchronization abstractions are implemented in scheduling components and accessed via IPC. As such, to block or wakeup a thread, a component scheduler modifies its own runqueue, and dispatches away from, or to the thread. Thus, Slite schedulers manage threads outside of the scheduler component (as a *system service*), and define the concurrency primitives of the system. Slite enables efficient definition of these concurrency primitives using direct, user-level scheduling when possible, and kernel-based dispatch across page-table boundaries to provide full-system policy.

Scheduler security considerations. Slite differentiates between 1) trusting a component for scheduling services, and 2) trusting it with the ability to modify register contents (thus potentially causing faults). The schedulers control thread interleaving, but must *not* be able to alter register contents of the threads in other protection domains. Therefore, the kernel dispatch path must be used to switch between protection domains – thus to threads preempted while executing in other components, and if a thread's *DCB* entry has *ip*/*sp* values, the kernel validates that the thread is not, in fact, executing in another component. This constraints a Slite scheduler from unduely controlling the control flow integrity of a thread in another component.

Inter-core, scheduler coordination. To maintain the efficiency of Slite, we avoid any inter-core synchronization on the common scheduling path. Thus, our current integration of Slite into multi-core systems focuses on partitioned scheduling (*i.e.* separate runqueues per-core). The coordination between cores, orchestrated by the *SchedLib*, uses shared structures, and Composite support for Inter-Processor Interrupts (IPI) for timely notification.

VI. Slite Application: RTOS

A potential benefit of enabling efficient and predictable implementation of scheduling at user-level is the increased isolation afforded to the system. Traditional RTOSes implement all their services and often run applications in kernel mode to achieve the efficiency and predictability requirements. In such systems, the only scheduler is the kernel's which applications leverage through direct function calls. This design is most common in resource-constrained microcontroller systems ($<$ 200 MHz and $<$ 256 KiB SRAM). Unfortunately, the increasing exposure of our embedded systems to the network is threatening a constant stream of security compromises.

A. SliteOS Design

To enable systems to both leverage user-level isolation of different subsystems, *and* efficient scheduling, we implement SliteOS, a new RTOS. The goals of this system are to provide component-based scheduling with performance properties on

the order of a bare-metal OS, and to pair Slite with multi-component isolation for dependable embedded computation. SliteOS provides the abstractions of threads, access to interrupts (via interrupt threads), timers, communication, and synchronization between threads. These coordination facilities including bounded-size message queues for inter-thread communication, and mutexes for inter-thread synchronization.

SliteOS leverages the efficient scheduling infrastructure of Slite including its support for timeouts and integration with Composite interrupts to provide preemptive, priority-driven execution. Though previous research found [43] that user-level handling of interrupts has a small overhead, the cost on microcontrollers [44] was significant. Pan et al. [44] provided a virtualization environment for Cortex-M microcontrollers capable of running 8 VMs in 128 KiB SRAM. The goal of that work was to bolster the security properties of the system by increasing the system's inter-application isolation. Notably that work provided a systems scheduler that scheduled between VMs, and paravirtualized RTOS VMs that scheduled their own threads. Unfortunately, the scheduling infrastructure in that work used Composite user-level scheduling support that added significant overhead relative to *FreeRTOS*, a common, efficient RTOS. This forced system designers to choose between isolation, and efficiency. Thus, we first apply SliteOS to resource constrained microcontrollers to achieve both user-level, isolated execution, and interrupt response times on the order of *FreeRTOS*. SliteOS compensates for this overhead by enabling context switch costs on the order of unprotected, kernel-based RTOSes.

Second, we study the use of different isolation structures with Slite, and the integration of RTOS abstractions (notably message queues) with scheduling functionality. We consider two isolation structures: (1) process-style isolation of different system functionalities (*e.g.* drivers) that use RTOS services via IPC, and (2) a hierarchical scheduling system in which OS instances are co-located with applications, but isolated from each other. A root scheduler multiplexes the instances.

B. SliteOS Implementation

Cortex-M port. We port the Slite implementation on top of the Composite kernel to a ARM Cortex-M7 MCU-based architecture and implement the SliteOS API similar to *FreeRTOS*, including message queues (a *FreeRTOS* queue counterpart) and mutexes. The most significant difference between the x86 and Cortex-M port is the user-level context switch assembly routine implementation. Unlike x86 where an interrupt or exception causes the pushing of an activation record onto the *kernel stack*, Cortex-M7 processor hardware pushes a stackframe onto the *user-level stack* which contains the *ip* to return to when transitioning back to user-level. Thus, unlike on x86, a kernel return to user-level from the kernel expects to restore it from the user stack. This presents a challenge when the kernel switches to a thread that previously saved its register context during user-level dispatch. The kernel has no means to set the *ip* when switching back to the thread as the *ip* is in the stackframe (the kernel in Composite does not access user-memory directly). Thus, Slite redundantly saves the thread's *ip* in the *DCB* (for user-level dispatches), and in a stackframe on the thread's stack (for kernel dispatches).

Co-design of message queues and scheduling. We leverage the ability to customize and co-design scheduling logic with the surrounding runtime. Bounded message queues transmit messages between threads, thus involve the waking and blocking of threads. We optimize message queues in SliteOS such that whenever a producer publishes a message, they directly switch to the consumer (thus eliding scheduling policy execution). To maintain correctness, this optimization is only used if the message queue has a single consumer, and that consumer has a higher fixed priority than producers.

VII. Slite Application: Parallel Runtime

The effective use of multi-core processors in embedded systems is not optional in many domains. The sensor complexity in domains such as autonomous vehicles requires more computation than can be performed by a single core. As writing multi-threaded code with synchronization at a low-level is error prone and challenging, parallel runtimes such as *OpenMP*, Cilk, and TBB provide run-to-completion task abstractions, and higher-level constructs to orchestrate computation. *OpenMP* is widely used in the real-time literature, and runtimes exist for `gcc`, `llvm`, and Intel's `icc`. *OpenMP* is an extension to the C/C++ language that takes the form of `pragma` preprocessor directives, we refer the readers to the specification [45] for complete details of the *OpenMP* directives presented hereafter.

***OpenMP* execution model and `tied` tasks.** All tasks (created with `parallel` or `task` constructs) are run-to-completion, which means that they have a defined termination point, and no inter-task preemptions (only inter-thread). However, there are points in their execution when they are awaiting synchronization with other tasks (*e.g.* barrier synchronization or `taskwait`), and *OpenMP* specifies these as *task scheduling points (TSPs)*. At these points, another pending task (task B) can be executed using the same shared stack of the synchronizing task (task A). This creates an execution dependency between tasks as task A only continues execution *after* task B completes its execution. TSPs include task creation and completion, `taskwait`, `taskyield`, and `barrier` constructs.

By default `all` tasks that are created (with `task` construct) are *tied*. A Task Scheduling Constraint (TSC)[1] specifies that the "Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled *only if it is a descendent task of every task in the set*." (The emphasis is ours.) Effectively, this prevents the execution of "sibling" tasks in the same thread at TSPs. One of the motivations for the tied task scheduling constraint is that `untied` tasks can easily lead to deadlock (*i.e.* if they are not subject to the TSC). As described in §II (and detailed

[1]Specified in section 2.10.6 of the *OpenMP* specification [45].

in the Appendix A), a trivial *OpenMP* program's worst-case is practically inflated by 100% due to the TSC. This makes *OpenMP's TSC solution impractical for real-time systems*.

***GOMP* and work-conservation.** *GOMP*, the `gcc` *OpenMP* runtime, is implemented as a co-design between the compiler that generates a set of closures for the parallel tasks, and invocations to the *GOMP* library that orchestrates and schedules the execution of those tasks. The *GOMP* library interface is well-documented, and we implement a version of the library based on Slite, which we call SliteOMP. The goal of this library is to *integrate task with thread scheduling* to enable more flexible *OpenMP* execution. Specifically, tight theoretical results assume that the *OpenMP* runtime is *work-conserving*, that is, that if there are tasks to execute, they will always be executed on an idle core [35], [30]. This is more challenging than it seems: (1) the default behavior of all tasks being tied prevents work-conservation at TSPs due to the TSC as some tasks cannot be migrated to idle parallel threads, (2) tasks awaiting a critical section either spin awaiting access, or block the entire thread, and (3) the dependencies created at TSPs prevent the progress of a thread with context deeper into the stack, can delay their computation. Thus, SliteOMP co-designs the scheduler with the parallel runtime, and executes *all* tasks as separate threads. This removes the need for tied tasks as stack-based dependencies are avoided by design, and threads can block when awaiting critical sections.

An *OpenMP* example in Appendix A demonstrates a 100% increase in worst-case executions for runtimes that use the TSC to avoid deadlock, while SliteOMP avoids the same deadlock using a thread-based scheduling model that is work conserving. Thus, SliteOMP provides the required semantics for existing work on parallel fork/join, and DAG execution [30], [34], [46]. Critical sections and scheduling points are handled using thread blocking, thus enabling other tasks to execute in separate threads. In this way, it should be noted that SliteOMP explicitly does *not* adhere to the TSC.

A. *SliteOMP* Runtime Design

The SliteOMP runtime uses the Slite support for configurable scheduling policies by (1) using a FIFO scheduling module (within the parallel application) to minimize overheads, (2) integrating the work-stealing deque with predictable stealing into the runqueue logic, (3) using efficient access to system IPIs for barrier coordination between threads – for example, to wake a blocked master thread, and (4) using an efficient implementation of locks for critical sections. We co-design the SliteOMP parallel runtime with the Slite thread scheduling leveraging the traditional techniques for balancing parallel tasks (task closures) across cores, tracking parent-child and sibling relationships in the scheduling structures and using simple barriers for `parallel`. We use common task queue data-structures (*e.g.* work-stealing deques [47], [48]) to balance tasks between cores. SliteOMP provides threaded computation for all tasks, yet only dequeues pending tasks into a thread when necessary to maintain work-conservation. This is in stark contrast to existing operating systems that support thread migration using heavyweight protocols that require the migration of an entire thread's context, rather than a simple task closure. We believe that SliteOMP demonstrates the multi-dimensional value that near-zero-cost scheduling can provide to a runtime system by *simplifying the programming model, ensuring work conservation, and maintaining high performance and predictability.*

B. *SliteOMP* Runtime Implementation

We implement the work-stealing deque [47], [48] for explicit tasks generated using `task` construct. We only modify the deque operation to steal work from a deque for another core by stealing from random deques *that we haven't tried to steal from yet*, before assuming there are no tasks to steal. This guarantee is necessary to bound the execution time of the stealing operation, thus ensure predictability. Comparably, to ensure bounded data parallelism, removing work from a data-parallel tasks is wait-free and uses fetch-and-add. For simplicity, we omit the nested fork/join implementation details and disable nesting in the SliteOMP runtime.

Thread pool and scheduling. On each core, a pool of N threads are created upon initialization of the runtime that immediately block until asked to perform task computation. Thus, SliteOMP runtime explicitly controls the scope of parallelism. A single data-parallel task is active when a `parallel` construct is created, and it is executed by threads in the pool if there is more than one thread in the team. The scheduling policy is FIFO with one exception: if the runqueue of SliteOMP threads is empty, a low-priority idle task wakes up a thread, to execute a pending task, or a portion of the work in a data-parallel task. This policy focuses on executing tasks to completion where possible, thus leaving tasks and work in the work-stealing deque where can be efficiently migrated between cores. SliteOMP does not differentiate between tied and untied tasks as they are all run in a separate thread context.

Synchronization. SliteOMP converts task suspensions on barriers and `taskwait`, and mutex contention events into simple thread blocking. If a task was blocked in this way, and is awoken upon – for example, rest of the threads reaching a barrier, a child task completion, or a mutex release – cross-core messages and IPIs are used to promptly activate it.

SliteOMP runtime coverage of *OpenMP*. We implement SliteOMP as a library written to the *GOMP* API, and is linked directly into the applications. At the moment, SliteOMP supports most common *OpenMP* constructs including `parallel`, `for`, `single`, `sections`, `critical`, `barrier`, `task` and `taskwait`.

VIII. Evaluation

The x86 hardware environment used in the experiments is a Dell Optiplex XE3 running a 3.2 GHz Intel i7-8700 processor with six physical cores (with hyper-threading disabled) and 8 GB physical memory. In Linux experiments, we used a Ubuntu 14.04 32 Bit OS running Linux 4.4.148 with Real-Time (RT) patch version 4.4.148-rt165, which we call *LinuxRTp*.

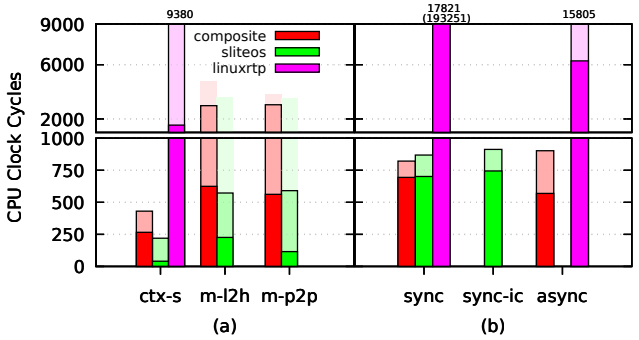

Fig. 4: Microbenchmarks for system primitives (labeled on x-axis) on x86 hardware. (a) OS Primimitives: `ctx-s` = One-way thread switch (Kernel primitives in Composite and LinuxRTp), `m-l2h` = Message Queues - sending from a low priority thread to a high priority thread (`m-p2p` – using One-to-One Message Queues). (b) Kernel Primitives: `sync` = Synchronous Communication (`sync-ic` – with lazy-coherence), `async` = Asynchronous Communication. Shaded bars without borders indicate the WCMT and with borders indicate WCMT with first measurement filtered out, and dark bars indicate average costs.

In the microcontroller experiments, we use an ARM Cortex-M7 microcontroller running at 216MHz (STM32F767IGT6), with 512KB embedded SRAM and 1024KB embedded flash. The 16KB instruction cache, 16KB data cache, and flash prefetch accelerators are enabled on the microcontroller. We use the `gcc` compiler version 4.9.4 for x86 with *OpenMP* 4.0 support and version 5.4.1 on microcontrollers, with the -O3 optimization flag for all cases.

A. Microbenchmarks

Figure 4 presents the costs of dispatch and communication operations in the underlying Composite kernel, SliteOS user-level scheduling, and comparable operations in LinuxRTp on the x86 architecture. The average costs indicated as darkest bars, are a measure over a million iterations and the lighter bars indicate the measured worst-case execution times (WCMT). The synchronous communication uses thread migration in both Composite and SliteOS, and pipe round-trip (RPC) in LinuxRTp. To understand the overhead of Slite's lazy coherence between user- and kernel-level on the IPC fastpath, also plot `sync-ic`, the IPC overhead with lazy coherence. The asynchronous communication uses asynchronous end-points in Composite, and pipes in LinuxRTp.

Discussion. Composite is an optimized μ-kernel and therefore its communication premitives are efficient compared to the general-purpose Linux. However, at 41 cycles (13 nanoseconds), it is clear that Slite is approaching zero-cost dispatching. The improvement of this value over a native Composite thread switch is mainly due to the inherent overheads in mode switching when making the system call. Importantly, when analyzing the *increase* in the cost of synchronous communication, we see that the kernel check for incoherence with the user-scheduler doesn't impose much common-case overhead (7 cycles), and even the lazy updating of the current thread adds only 50 cycles which is less than if the thread switch were conducted through the kernel (250 cycles). The

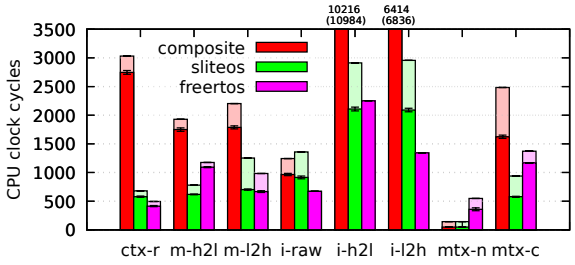

Fig. 5: Microbenchmarks for system primitives (labeled on x-axis) on ARM Cortex-M7 microcontroller. Shaded bars indicate WCMT, error-bars for standard deviation (SD) and dark bars for average costs.

SliteOS API for communication between threads in the same protection domain leverages user-level scheduling, and even the somewhat immature message queues show improvement over Composite. More importantly, the optimized one-to-one message queues show significant improvement in the average cost compared to Composite due to Slite near-zero-cost dispatching at user-level.

B. SliteOS Benchmarks

To study the efficiency and predictability properties of SliteOS together with the user-level isolation as compared against a bare-metal (kernel-mode, no isolation) RTOS, we measure the computational cost of primitive system services on the Cortex-M7 microcontroller. For response time analyses, these costs are integrated as system overheads into a schedulability analysis, thus directly translating into a decrease in system utilization. We compare the following systems: (1) Composite with the SliteOS APIs implemented using the kernel-mediated context switches, (2) SliteOS with Slite user-level dispatching, and (3) *FreeRTOS*, a widely deployed, efficient microcontroller RTOS.

Figure 5 depicts the average (the bottom darker bar), stdev (the error bar displayed on the average bar), and maximum (the lighter top bar) costs of these core operations over ten thousand iterations. The Composite and SliteOS measurements in this experiment are all intra-component and thus do not include protection domain switch costs. The different system primitives measured in Figure 5 are:

- *Context switches*: The round-trip context switch, `ctx-r`, is measured for `thd_yield(tid)` in Composite and SliteOS and `portYIELD()` in *FreeRTOS*. *FreeRTOS* does not allow the user to control which thread to yield to, while Composite and SliteOS provide directed yields, which avoids scheduler policy involvement.
- *Message queues*: Messages are restricted to 4 bytes (a single word) to minimize the transfer overhead in the measurements. For *FreeRTOS* we use `xQueueSend()` and `xQueueReceive()`. Sending from a high-priority to a low-priority thread (`m-h2l`), and vice-versa (`m-l2h`) are measured.
- *Interrupt response-time latency*: The interrupt handling is divided into "top half" execution in the Interrupt Service Routine (ISR) and the "bottom half" execution in a thread context. In Composite and SliteOS systems, the threads execute in user-level components, while for *FreeRTOS*, both ISR

and thread execute in kernel-mode. Interrupt latency (`i-raw`) measured is the time interval between the activation of the ISR, and the start of the corresponding interrupt handler thread's execution. In *FreeRTOS*, the ISR uses `xQueueSendFromISR()` to send to a queue, thus activating the interrupt thread. In Composite and SliteOS, the ISR uses "asynchronous send" (`asnd`) endpoint to activate the user-level interrupt thread waiting on the "receive" (`rcv`) endpoint.

It is common for an interrupt thread to perform device-specific operations, then send the interpreted data to an application thread. Thus, we study the end-to-end interrupt latency (1) when interrupt thread is higher priority (`i-h2l`), and (2) when the interrupt thread is lower priority (`i-l2h`) than the application thread, respectively. We measure the total time to relay the message from the ISR handler to the interrupt thread (in `i-raw`) and application thread (in `i-h2l and i-l2h`).

- *Mutexes*: `mtx-n` measures the total time to acquire and release a mutex without contention, which is the most common case. `mtx-c` measures the contention case in which two threads contend for a lock. In the latter case, this includes a context switch to the second thread that owns the lock (`mtx-c`). In *FreeRTOS*, we use `xSemaphoreTake()` and `xSemaphoreGive()`.

Discussion. The overheads of system calls on resource-constrained microcontrollers are significant. Slite enables isolated code to have overheads on the order of an unprotected RTOS code by avoiding expensive privilege-level switches.

As interrupts activate user-level threads in Composite and SliteOS, the raw interrupt latencies (`i-raw`) are higher when compared to *FreeRTOS*. However, when holistically looking at the end-to-end interrupt latency in the common case (the device driver at high priority (`i-h2l`)), we observe that the latency in SliteOS scheduling is lower than *FreeRTOS*.

Importantly, SliteOS demonstrates (compared to the comparably massive overheads of Composite), that *interrupt response times and OS abstraction costs comparable to a kernel-resident RTOS are possible in an isolated, user-level component*. Therefore, SliteOS constraints the scope of faults at a relatively small cost. We belive that SliteOS when paired with infrastructures for inter-component isolation [44], shows the value of user-level, configurable near zero-cost scheduling.

Isolation and Predictability. Many real-time systems structure software as a pipeline of filters and transformations, triggered from interrupt execution. For example, the core Flight System (https://cfs.gsfc.nasa.gov/) used in many NASA satellite missions and quad-copters facilitate a software bus that is used by pipeline of tasks for sensor fusion, actuator control and for telemetry application to communicate the system state with the ground station. Further, timely execution of real-time software is crucial in these embedded and cyber-physical systems that often have a pipeline of tasks processing data in a pipeline to predictably control the actuators [49]. To study the impact of SliteOS on software isolation, we measure communication latencies in software pipeline architectures, with the tasks using SliteOS message queues for communication.

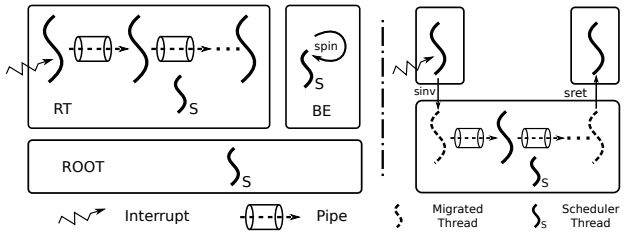

Fig. 6: Multi-component isolation structures. (a) Pipeline processing in hierarchical subsystem isolation. (b) Pipeline processing in process-style software isolation.

The end-to-end costs of communication impact the real-time system utilization when applying a schedulability analysis. Thus, we focus on benchmarking the system primitives in the following setups as they emphasize the overheads that directly impact the worst-case response times of the system. (1) a hierarchical system in which the real-time (RTOS) software is isolated via hierarchical scheduling from the best-effort software (Figure 6(a)). (2) a process-style isolation of different pipeline processing stages (*e.g.* device driver) from each other (Figure 6(b)). We conduct these experiments on the x86 architecture to compare with the existing process-based isolation in LinuxRTp. In Composite and SliteOS, we use the High-Precision Event Timer (HPET) to emulate a sensor interrupt. Interrupts arrive at an inter-arrival of 20 milliseconds, which triggers the pipeline processing in SliteOS and Composite systems. We measure the end-to-end latency from arrival of the interrupt to completion of pipeline processing.

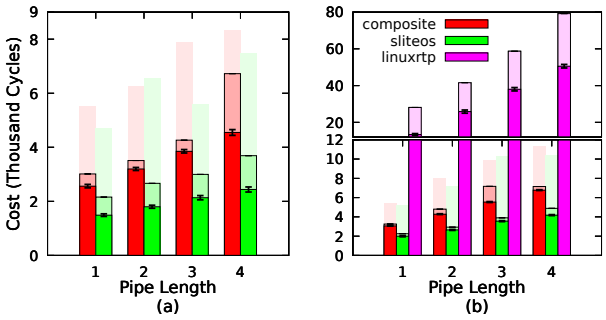

Fig. 7: Isolation and communication costs in different systems on x86. (a) Pipeline processing using separate, hierarchical subsystems for isolation. (b) Pipeline processing in process-style software isolation. Shaded bars without borders indicate the WCMT, shaded bars with borders indicate WCMT with first measurement filtered out, error-bars indicate SD and dark bars indicate average costs over hundred thousand iterations. Reported "pipe length" does not include the interrupt thread.

Figure 6(a) depicts a high-priority real-time subsystem containing pipeline processing, along side a low-priority best-effort subsystem running an infinite loop to cause interference. These subsystems are scheduled by the root scheduler.

Discussion. Figure 7(a) confirms that the SliteOS pipeline latency is lower than Composite, though the speedup is not as impactful as that of the microcontroller. More importantly, SliteOS does so while maintaining the strict isolation and

predictability that the underlying Composite system provides through hierarchical partitioning.

Figure 6(b) depicts a system that selectively isolates different pipeline stages. Different pipeline processing stages implemented in separate, isolated protection domains that access the message queues via synchronous invocation-based IPC, or as scheduler-resident computations (*e.g.* representing communication stacks). The pipeline processing stages alternate between isolated, and scheduler co-located tasks as shown in the Figure 6(b). The comparable LinuxRTp system uses inter-process pipe communication and we measure the end-to-end latency from an event generated by timer delivery (using `timerfd_create`) to the end of the pipeline computation.

Discussion. Figure 7(b) shows that the SliteOS pipeline latency based on Slite significantly lowers overheads compared to the general-purpose Linux pipes. More importantly, the SliteOS pipeline latency is lower than the underlying Composite μ-kernel despite the synchronous communication costs that are slightly higher due to kernel checking on active thread inconsistencies. SliteOS enables predictable and efficient end-to-end communication latencies together with the Composite-enabled isolated execution.

C. SliteOMP Benchmarks

In SliteOMP, the use of IPIs enable efficient coordination between threads blocked at a synchronization point instead of spinning in the synchronizing threads. This enables the system to efficiently schedule between multiple threads that could then steal work from a work-stealing deque. To study the performance benefits of Slite in SliteOMP and to understand the scalability properties of the nascent SliteOMP runtime, we evaluate and compare with the *GOMP* runtime in LinuxRTp on x86 architecture. In LinuxRTp, we measure the costs of both *spin* (gomp-spin) and *block* (gomp-blk) implementations in *GOMP* runtime with OMP_WAIT_POLICY set to ACTIVE and PASSIVE respectively and GOMP_SPINCOUNT=INFINITY for spin. The number of threads in the team of data parallel execution are set to be equal to number of cores enabled in the system. Nested fork/join parallelism is disabled in all runtimes.

SliteOMP microbenchmarks. To understand the performance of the SliteOMP runtime, we evaluate the microbenchmarks for the following *OpenMP* constructs shown in Figure 8. (a) `parallel` that creates a fork/join data-parallel execution, and (b) `parallel + task + taskwait` that creates a fork/join data-parallel execution and each thread in the team creates a task and waits for the completion of its child task.

Figure 9 depicts the average costs (solid bar) and WCMT (empty bar) over a thousand iterations for varying team-sizes on SliteOMP, gomp-spin, and gomp-blk runtimes.

Discussion. Though this is a relatively small scale multicore system, we believe it is representative of many in the field. Even at this scale, SliteOMP generally demonstrates fewer overheads with increasing cores than *GOMP*. These benchmarks measure the impact of thread-based scheduling in SliteOMP on the worst-case timing of the parallel runtime.

```
/* (a) empty parallel construct */
#pragma omp parallel {
}

/* (b) empty parallel + task + taskwait constructs */
#pragma omp parallel {
    #pragma omp task {
    }
    #pragma omp taskwait
}
```

Fig. 8: *OpenMP* constructs for microbenchmarks.

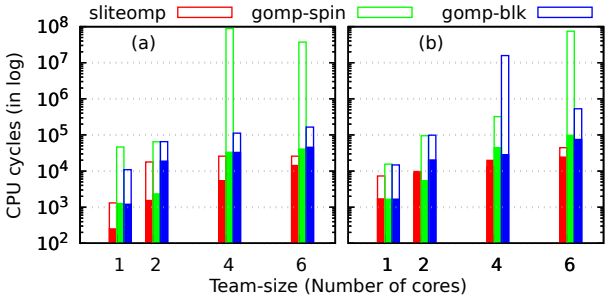

Fig. 9: *OpenMP* microbenchmarks for (a) and (b) in Figure 8.

It is necessary to control and bound the worst-case overheads such system effects when using *OpenMP* in real-time environments as those overheads must factor into a system schedulability analysis, thus overheads directly translate into utilization decreases. However, both *GOMP* implementations demonstrate significant WCMT spikes due to the complexity of the *GOMP* runtime and the underlying Linux system. At 6 cores, the SliteOMP runtime is about 3 times faster in Figure 8(a)/(b) benchmarks and the WCMT of Figure 8(b) benchmark with "tied" tasks is 8.7 times faster when compared to `gomp-blk` on LinuxRTp, while the `gomp-spin` shows significantly higher WCMT spikes. The SliteOMP runtime simplicity and the promotion of task- to thread-execution without significant decreases worst-case performance demonstrate the ability of SliteOMP to *remove the pessimism introduced by the deadlock-avoiding scheduling constraints in OpenMP, while maintaining work conservation so as to meet the assumptions of efficient analysis.* We believe this demonstrates that integrating Slite into SliteOMP enables significant gains in the efficiency and opens up the use of the tight analytical approaches necessary for practical real-time adoption.

***OpenMP* application benchmarks.** Tied tasks have been shown to have bad work-conservation properties due to the *OpenMP* specification's scheduling constraints (TSCs). SliteOMP uses a thread-based rather than a task-based parallel scheduling model. It is important to understand the average-case overhead that this imposes on the system. Though the worst-case in Figure 9 is necessary for real-time systems, the average case often determines the practical utility. We use conventional parallel data-structures so do not expect a significant speedup, but assess the overheads of the thread-based scheduling of SliteOMP over a task-based model.

To assess the SliteOMP performance, we study the performance of Barcelona OpenMP Task Suite (BOTS) benchmarks (http://www.github.com/bsc-pm/bots) that use "tied tasks".

The BOTS applications measured are, (1) `fib`: Computes fibonacci numbers. (2) `sort`: Uses a mixture of sorting algorithms to sort a vector. (3) `strassen`: Computes matrix multiplication using strassen's method. (4) `sparselu`: Computes the LU factorization of a sparse matrix. We use the `single` version for our benchmarks.

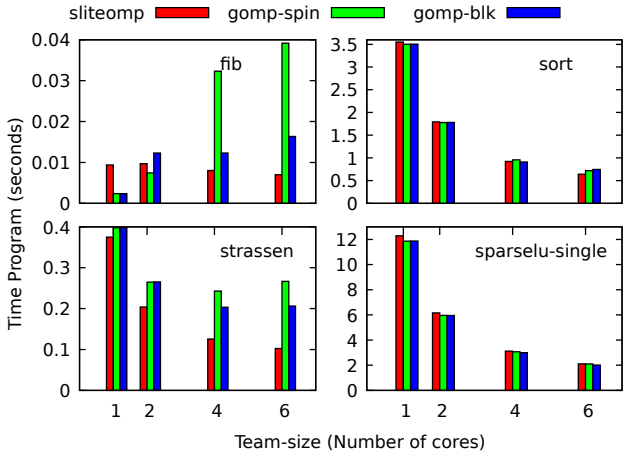

Fig. 10: Execution time (in seconds) of BOTS programs.

Figure 10 measures the execution time of the BOTS applications on SliteOMP, gomp-spin, and gomp-blk runtimes for increasing number of cores (team-size).

Discussion. The `fib` application uses very fine-grained tasks each with very little computation, thus significantly stresses the task runtime. For one and two cores, SliteOMP is *slower* than *GOMP*. For one core, *GOMP* has checks to avoid the runtime's data-structures, instead directly calling tasks. SliteOMP does not special-case one core. At up to six cores, SliteOMP is the only implementation to *lower* execution times. For the computationally intensive benchmarks (coarser tasks) `sort, strassen and sparselu`, SliteOMP is comparable to *GOMP* runtime if not better. The practical performance of SliteOMP is competitive and benefits from the simplicity derived from promoting all task execution to separate threads. More importantly, this enables a work-conserving runtime preventing degenerate task orderings that can increase worst-case execution.

IX. Conclusions

This paper has presented Slite, a system for the near zero-cost, user-level scheduling of system threads. The key mechanism for direct user-level dispatch (in ∼40 cycles on x86) enables incoherence between which thread user- and kernel- level believe is executing, with lazy synchronization using shared structures. We have applied Slite to create a parallel runtime in SliteOMP which promotes traditionally task-based scheduling to instead use full threads, thus removing inefficient non-work-conserving constraints that challenge effective real-time computation. We have implemented SliteOS, which enables isolated user-level computation with performance properties on par with the bare-metal *FreeRTOS* on microcontrollers. To emphasize the isolation that SliteOS enables, we demonstrate hierarchical partitioned system support and fine-grained process-based isolation support that both benefit from Slite support. We see this as a necessary step toward providing secure real-time, embedded systems. Slite effectively challenges the existing dominant structure of kernel-resident scheduling, and demonstrates that the user-level scheduling of system threads can be more efficient, predictable, and flexible.

Acknowledgments. We'd like to thank the anonymous reviewers, Bryan Ward, and Rick Skowyra for their helpful feedback and our shepherd for helping to significantly improve the quality of this paper.

Appendix A
Work Conservation in OpenMP Implementations

```
#pragma omp parallel {          /* a. fork                */
  #pragma omp single {          /* b. enter single        */
    #pragma omp task {          /* c. create task (A)     */
      #pragma omp task {        /* d. create task (B)     */
        spin_100ms()            /* e. spin for 100ms      */
      }
      #pragma omp taskwait      /* f. wait for B          */
    }
    #pragma omp task {          /* g. create task (C)     */
      spin_100ms()              /* h. spin for 100ms      */
    }
    spin_100ms()                /* i. spin for 100ms      */
    #pragma omp taskwait        /* j. wait for A and C    */
  }                             /* k. implicit barrier    */
}                               /* l. join                */
```

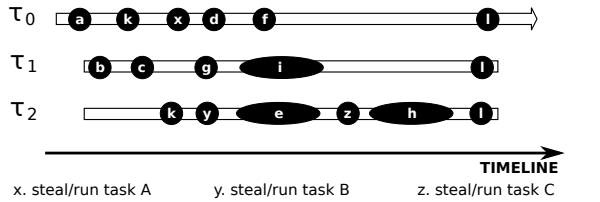

Fig. 11: An *OpenMP* program and a timeline for a valid interleaving of the work-conservation problem with "tied tasks" and the TSC.

Figure 11 shows a valid *OpenMP* program and an adversarial interleaving among three threads in a *GOMP* team to demonstrate that "tied tasks" and the TSC significantly impact the worst-case execution. The `spin_100ms` function synthesizes a 100 milliseconds workload in the tasks. Across three cores, we expect the total execution time to be around 100ms. We break computation into a number of labelled steps, and show a valid sequence of steps in Figure 11's timeline that leads to non-work conserving behavior due to TSC.

Discussion. Only a single thread (τ_1) executes the `single` construct creating tasks A and C while the rest of the threads wait at the implicit barrier at the end of the block. The thread, τ_0 schedules task A to create the task B before waiting at `taskwait` for its child task B executed by τ_2. However, the TSC prevents τ_0 from scheduling a non-decendant task C, causing WCMT of 200ms for 37% of the 1000 iterations in the *GOMP* runtime. We validate this program on the SliteOMP runtime and observe that the execution time is always 100ms (for 1000 iterations), thus preserving the work-conservation required in real-time systems.

REFERENCES

[1] "OpenMP: http://www.openmp.org, retrieved 9/21/12."

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Jul. 1995, pp. 207–216.

[3] "Intel Thread Building Blocks: http://threadingbuildingblocks.org/, retrieved 9/21/12."

[4] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill, "Design and performance of configurable endsystem scheduling mechanisms," in *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 32–43.

[5] J. H. Anderson and M. S. Mollison, "Bringing theory into practice: A userspace library for multicore real-time scheduling," in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 283–292.

[6] "Docker: https://www.docker.com/," 2018.

[7] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: a new facility for resource management in server systems," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 45–58.

[8] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[9] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont, "The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock)," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*, 2017.

[10] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An os abstraction for safety and performance," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[11] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "Spacejmp: Programming with multiple virtual address spaces," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[12] B. Engel, H. Hartig, C.-J. Hamann, and M. Volp, "On confidentiality-preserving real-time locking protocols," in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[13] M. Nasri, T. T. Chantem, G. Bloom, and R. M. Gerdes, "On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks," in *Proceedings of the Twenty Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '19)*, 2019.

[14] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference (EuroSys '19)*, 2019.

[15] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena, "The battle of the schedulers: Freebsd ULE vs. linux CFS," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[16] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1991, pp. 95–109.

[17] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, 2018.

[18] 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.

[19] J. Stoess, "Towards effective user-controlled scheduling for microkernel-based systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 59–68, 2007.

[20] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.

[21] B. Ford and S. Susarla, "Cpu inheritance scheduling," in *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*. New York, NY, USA: ACM Press, 1996, pp. 91–105.

[22] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015.

[23] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London, UK, Dec. 2001, pp. 3–14.

[24] L. P. Barreto and G. Muller, "Bossa: a language-based approach to the design of real-time schedulers," in *10th International Conference on Real-Time Systems (RTS'2002)*, Paris, France, mar 2002, pp. 19–31.

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

[26] "Scheduling in k42, whitepaper: http://www.research.ibm.com/k42/white-papers/scheduling.pdf."

[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 '18)*, 2018.

[28] 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.

[29] 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.

[30] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *Proceedings of the 2013 19th IEEE Symposium on Real-Time and Embedded Technology and Applications*, 2013.

[31] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. D. Gill, and C. Lu, "Randomized work stealing for large scale soft real-time systems." in *Proceeedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2016.

[32] W. Qi and G. Parmer, "FJOS: Practical, predictable, and efficient system support for fork/join parallelism," in *Proceedings of the 2014 20th IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, 2014.

[33] J. Sun, N. Guan, Y. Wang, Q. He, and W. yi, "Real-time scheduling and analysis of openmp task systems with tied tasks," in *Proceeedings of the IEEE Real-Time Systems Symposium (RTSS '17)*, 2017.

[34] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of openmp4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, (CASES '15) 2015*, 2015.

[35] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-federated scheduling of parallel real-time tasks on multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS '17)*, 2017.

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

[37] 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.

[38] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Symposium on Operating Systems Principles*, 1999, pp. 170–185. [Online]. Available: citeseer.ist.psu.edu/shapiro99eros.html

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

[40] 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.

[41] 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.

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

[43] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *In Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

[44] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.
[45] ""openMP application programming interface", version 5.0, november 2018, https://www.openmp.org/specifications/, retrieved may 2019."
[46] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global edf scheduling for parallel real-time tasks," *Real-Time Systems*, vol. 51, 2015.
[47] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2005, pp. 21–28.
[48] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and efficient work-stealing for weak memory models," in *Principles and Practice of Parallel Programming (PPoPP '13)*, 2013.
[49] Z. Cheng, R. West, and C. Einstein, "End-to-end analysis and design of a drone flight controller," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, 2018.