Temporal Capabilities: Access Control for Time*

Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, Gabriel Parmer
The George Washington University
Washington, DC
{phanikishoreg, robertgif, lbaier, gparmer}@gwu.edu

***Abstract*—Embedded systems are increasingly required to handle code of various qualities that must often be isolated, yet predictably share resources. This has motivated the isolation of, for example, mission-critical code from best-effort features using isolation structures such as virtualization. Such systems usually focus on limiting interference between subsystems, which complicates the increasingly common functional dependencies between them. Though isolation must be paramount, the fundamental goal of efficiently sharing hardware motivates a principled mechanism for cooperating between subsystems.**

This paper introduces Temporal Capabilities (TCaps) which integrate CPU management into a capability-based access-control system and distribute authority for scheduling. In doing so, the controlled temporal *coordination* between subsystems becomes a first-class concern of the system. By enabling temporal delegations to accompany activations and requests for service, we apply TCaps to a virtualization environment with a shared VM for orchestrating I/O. We show that TCaps, unlike prioritizations and carefully chosen budgets, both meet deadlines for a hard real-time subsystem, and maintain high throughput for a best-effort subsystem.

I. Introduction

Embedded and real-time systems are increasingly expected to provide a challenging combination of timing predictability, reliable software, and complex feature sets. For example, such systems often co-host more general purpose computation with timing-critical computation due to the twin forces of the (1) increasing functional requirements (*e.g.* computer vision, mapping), and (2) the pressures of cost and Size, Weight, and Power (SWAP) have forced consolidation of many disparate systems. Driven by these often conflicting goals, systems erect isolation boundaries between different bodies of software to constrain the impact of temporal and spatial faults. Significant research has focused on scheduling for mixed-criticality systems [1] to constrain the impact of temporal irregularities, and to explicitly differentiate timing properties for bodies of software based the designer's level of confidence in them, and their importance to the embedded system.

Unfortunately, less research has investigated the *interactions* between spatial and temporal isolation. The focus of spatial isolation has often been on providing separation kernel-style complete isolation between subsystems [2]. Though appealing from an isolation perspective, such approaches often require partitioning memory and I/O thus prohibiting inter-subsystem sharing. This paper squarely focuses on how isolated subsystems can both *coordinate* to provide services to each other, while also mutually *controlling the interference* that coordination can cause. We make few assumptions about these subsystems: they can have different scheduling policies, be implemented as an arbitrary number of spatial protection domains, and can handle interrupt processing.

This paper introduces Temporal Capabilities (TCaps), an abstraction that enables the user-level control of access to processing time that provides a principled means for *distributing and restricting* time management responsibilities between isolated subsystems. TCaps decouple scheduling decisions (*what* computation should consume time at any point) from the ability to consume time (*if* a computation has a timeslice and can be active now). TCaps enable the inter-subsystem delegations of a non-replenishable processing time and track multiple priorities from different subsystem schedulers that are involved in the delegation path. These multiple priorites are used to control preemption decisions to vector interrupts directly to their subsystems.

TCaps are built on the Composite OS [3] which does not have an in-kernel scheduler, instead enabling *user-level* definition of scheduling policy provided by a *dispatch* operation. This enables scheduling policy specialization and distribution of system scheduling responsibilities between multiple schedulers (*e.g.* as with virtual machines). In this setting, TCaps provide a number of key features necessary for full-system predictability: (1) *controlled subsystem coordination* by enabling schedulers to delegate extents of time to each other, thus enabling temporal flows that correspond with inter-subsystem functional dependencies; (2) *inter-subsystem isolation* as each subsystem uses only processing resources granted to it from other subsystems that are associated with a priority (of sorts), thus limiting the scope of its interference on other subsystems; (3) *control over timer notifications* is enabled for each individual scheduler within their TCap-provided slice of time; and (4) *predictable interrupt scheduling* via the association of the interrupt's execution with a TCap.

Figure 1 depicts a system containing multiple coordinating Virtual Machines (VMs). On the left, a traditional system that contains multiple VM schedulers, all orchestrated through the underlying hypervisor's scheduler. Timers and interrupts can only be delivered to the hypervisor subsystem which then orchestrates the control flow. On the right, TCaps are used to control and coordinate access to processing resources. Timers and interrupts can be directly vectored to subsystems as determined by TCaps and their delegation patterns. Different parts of subsystems can use separate TCaps, and

*This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675, ONR Award No. N00014-14-1-0386, and 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 or ONR.

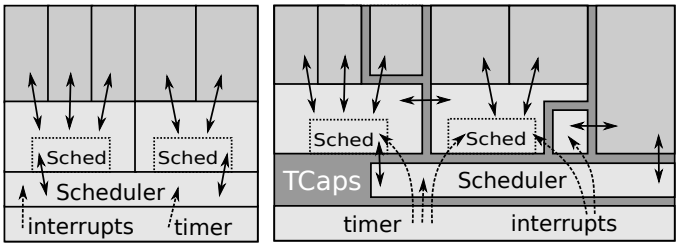

Fig. 1: Left: a traditional system with virtual machines. Applications are in darker grey at the top. Arrows designate control flow, and dotted arrows designate hardware interaction. Right: a system where the TCaps mediate and control interactions between subsystems. Darkest gray paths designate processing time flow through TCaps.

subsystems delegate and activate each other. In this example, the center subsystem is activated only by requests from the other subsystems with TCap processing-time delegations, thus is not necessarily scheduled by the global scheduler. This enables systems to accurately account for the time spent in processing the requests in the center subsystem to the surrounding subsystems.

TCaps use a combination of cycle-accurate budget tracking, one-shot timers, capability-controlled budget delegation, and priority tracking using an information flow tracking-like [4] system. Each TCap includes a budget of processing cycles. Schedulers charged with making fine-grained CPU management decisions can *delegate* time to a subsystem, thus enabling it to manage a slice of processing time, or use it to process a service request. TCaps *decouple* the conventional fine-grained scheduling of threads, from the temporal guarantees made between various subsystems. In managing these flows, TCaps enable preemption decisions to be conducted based on *all* involved schedulers.

II. Background and Related Work

TCaps employ budget, scheduling, and priority mechanisms that superficially resemble those in existing systems. The differences that we discuss here, though subtle, have a profound impact.

Scheduling contexts [5]. Scheduling contexts (SCs) in Fiasco.OC are kernel-level objects that provide an abstraction of processing time accessible through capabilities. SCs can be created and modified through the system call interface and are dependent on the kernel-level scheduler configured at compile-time. Each SC encloses a scalar priority value that is kernel-level scheduling policy dependent. Interrupts can trigger SCs based on a scalar priority and the processing time is accounted to these SCs. In contrast, TCaps are a kernel mechanism that enables user-level scheduling by allowing individual schedulers control over timer notifications within their TCap-provided time-slice. Each TCap encloses a non-replenishable processing time and tracks priorities for each scheduling policy involved in the delegation path. TCaps preemption decisions for interrupts and asynchronous notifications are based on priorities of all the subsystems involved in a delegation path, thus considering the scheduling policies spread across multiple subsystems. TCaps also allow inter-subsystem service requests (i.e. for I/O) to account for processing time in interrupts and device drivers to subsystems that make the service request for I/O.

Budget management. Budget servers enable the *rate of computation* to be constrained, thus providing varying temporal guarantees. We refer to these as rate-based servers as they constrain the rate of execution of windows of time. Rate-based servers are used in flattened hierarchical scheduling [5] in Fiasco.OC by attaching SCs to vCPUs subject to replenishment rules, RTXen [6], and Quest [2].

These rate-based servers are subject to both global scheduling and replenishment policies. This places the burden on each scheduling policy to convert its own priority into the kernel's version of priority, and to use implicit replenishments appropriately. We find that there is a semantic gap between these global policies, and the needs of systems with coordinating subsystems. For example, the interference caused by network interrupt execution can be controlled by executing them in a server [2], [5]. However, choosing the budget and replenishment is choosing between conflicting goals: smaller utilization lessens interference and is necessary for predictability, but might prohibitively constrain best-effort throughput. Better is to always execute those interrupts with a controlled processing share delegated from the best-effort subsystem at the time of service request. However, this is difficult if VMs don't share both identical scheduling policies and thread priorities. Though others have investigated priority inheritance for interrupts based around which tasks are requesting service [2], what is required is a form of budget inheritance, but that is difficult given the wide variation in interrupt rates, and I/O request rates. In Section VIII-E, we demonstrate the difficulties of choosing static budgets to satisfy both real-time constraints, and effectively manage best-effort throughput.

Concretely, the difference in mechanism between [2], [5], [6] and TCaps, are that (1) TCaps are a simple slice of time and do not have a replenishment period, and (2) TCaps track the priorities for each scheduling policy involved in the management of the time, instead of flattening priority for a single policy. The former enables *transient* delegations of time with a single finite amount of interference that matches activations of other schedulers (as in hierarchical scheduling), or functional request-based delegations where a finite amount of work will be done in the "server" subsystem. The latter enables the controlled coordination between subsystems with different scheduling policies, and different notions of a given computation's priority. This, for example, enables interrupts to be properly scheduled even in a hierarchical system including both fixed priority, and EDF schedulers.

Rate-based servers are a strong abstraction to provide an upper bound on a thread's execution. However, TCap budgets and delegations are a better fit for the fine-grained coordination required in a loosely coupled system. TCaps *generalize rate-based allocations* by using TCap-based servers that make delegations in accordance with replenishment policies.

Inter-protection domain interactions. When harnessing the functionality of another subsystem, passing budget and priority via synchronous Inter-Process Communication (IPC) [7], [8] has a number of complications: 1) asynchronous execution (*i.e.* for device driver execution during interrupt processing) cannot use a scheduling context that is transient, and 2) timing irregularities in the server directly impact timing and budgets of the client. TCaps differentiate between inter-subsystem dependencies that imply relatively a strong trust relationship, and those that require a weaker temporal coupling. For the former, Composite uses thread-migration-based IPC [9] where server execution uses the same scheduling context as the client. With the latter, subsystems interact in Composite using asynchronous notifications and TCap delegations.

Existing systems typically focus on priority inheritance on synchronous invocations (*e.g.* system calls [2], IPC [7]) where each protection domain shares a notion of priority. Priority inheritance on IPC between subsystems that each define separate scheduling policies or mappings between computations and priorities is more challenging. TCaps track the priority of all schedulers that delegate a slice of time, and makes preemption decisions based on *all* priorities to avoid undue interference.

User-level scheduling. The μ-kernel system design philosophy is based around a notion of system minimality in which only those system services that must be in the kernel, are implemented there [10]. Systems have successfully exported most necessary system-level services to user-level such as virtual memory management [10], kernel memory management [11], and device drivers, often with very little overhead. However, *processing time* management, though going through a large amount of turbulence [7], [12], [13], [11], [5], [14], is still implemented as a kernel-resident policy.

Providing user-level control over scheduling policy was proposed by Elphinstone et al. [11] as one of the long-standing problems in μ-kernel construction. Composite has long enabled the user-level customization of scheduling [15]. That work used shared memory regions between user-level schedulers and the kernel, so that the kernel could reference thread priorities to make scheduling decisions. Unfortunately, the asynchronous access of user-level schedulers and the kernel to these regions led to complex, inefficient, and buggy code. TCaps provide comparable functionality and provide the foundation for distributed scheduling with both fine-grained user-level control, and kernel-assured allocation guarantees.

CPU Inheritance Scheduling [16] provides a processor scheduling framework in which arbitrary threads can act as schedulers for other threads. However, their work requires traversing unbounded scheduler hierarchies and often multiple inter-process context switches for each event (interrupt, timer). In contrast, TCaps enables direct interrupt delivery, timer interrupt notification to any scheduler subsystem, and inter-subsystem coordination with delegations that strictly limit the scope of interference.

Composite features user-level scheduling that counter-intuitively provides overheads less than those for monolithic systems such as Linux (see Section VIII-B), and TCaps enable efficient and predictable, kernel-based interrupt scheduling based on the priorities assigned by all relevant user-level schedulers.

III. TCap System Model and Design

To understand the relationship between TCaps, scheduling in different subsystems, and thread execution, we introduce a simple model. A system is a composition of multiple *subsystems*, $s^x \in S$, that we treat as the *principals* of the system (consumers of time). A scheduler in each subsystem is responsible for managing the computational resources of all threads that execute within that subsystem, and of all subsystems that leverage its scheduling services. Conventional examples include VMs, hierarchical scheduling systems [17], and separation kernels [2]. A TCap must be *active* at all points in time, and all processing time (measured in cycles) is accounted to the active TCap. Each thread's execution is associated with a TCap dynamically by dispatching between both threads and TCaps.

Each subsystem s^x has capability-controlled access [3] to a set of TCaps, $T^x = \{t_1^x, \ldots, t_n^x\}$.

A. TCaps Budget

Each TCap, t_n^x, has a *budget*, B_n^x, which is the scalar size of the slice of time it provides to it's subsystem, in cycles. This budget is expended via thread execution. *Delegation* enables time to be transferred from one TCap to another through asynchronous end-points, often across different subsystems. When b time units are delegated from t_n^x to t_m^y, the natural adjustments to budget are performed:

$$\begin{aligned} B_n^x &= B_n^x - b \\ B_m^y &= B_m^y + b \end{aligned} \tag{1}$$

The budget in a TCap is *not* replenished by the system, and instead must be delegated by another TCap.

A single TCap we call Chronos, t^c, in a trusted subsystem, s^c, has $B^c = \infty$. Once all other TCaps have expended their budget, t^c is activated, and it delegates appropriately to replenish budgets. Since this replenishment is programmatic, the amount of replenishment for each TCap can vary over time.

B. TCaps Quality

Each TCap, t_n^x, has a *quality*, Q_n^x, which denotes a metric of the importance of the TCap's time. Q_n^x is a set of priorities for each subsystem, $Q_n^x = \{p_n^x, p_n^y ...\}$. A subsystem assigns a scalar priority to each of its TCaps using its own semantics. A TCap's quality includes the priority of each subsystem that has (transitively) delegated budget into t_n^x. *Lower* numerical values of p_n^x designate a *higher* quality.

The "higher quality" relation on TCap ($\succ$) is used to make preemption decisions. An asynchronous event (asynchronous *send*, or interrupt) that activates t_n^x should preempt execution of the current thread using t_m^y if and only if $Q_n^x \succ Q_m^y$. Quality is not used to determine which TCap to execute at all points

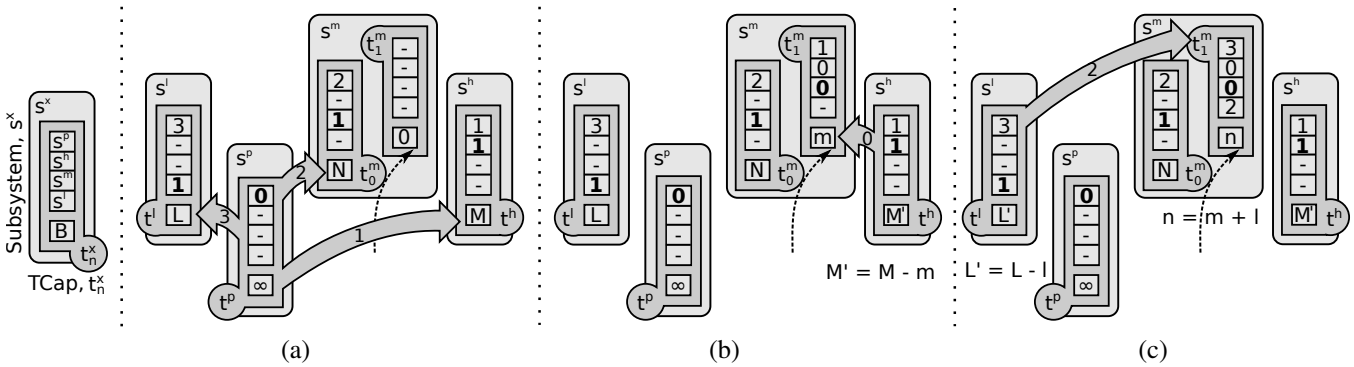

Fig. 2: TCap example. TCap structure is shown on the left, including the array Q^x_n with a priority entry for each of the subsystems, $S = \{s^p, s^h, s^m, s^l\}$ (the parent, high priority, medium priority and low priority subsystems), and budget B. The system includes five TCaps. The priority each subsystem has control over is bolded for clarity. The dashed arrow indicates an interrupt attached to TCap t^m_1. (a) The parent (s^p) delegates L budget at priority 3 (lower value = higher priority), N budget at priority 2, and M budget at priority 1, into s^l, s^m and s^h, respectively. (b) s^h delegates m cycles into t^m_1 in s^m with priority 0. (c) s^l delegates l cycles into t^m_1 in s^m with priority 2, which degrades the quality of t^m_1.

in time – it is *not* used to make scheduling decisions – so it is not necessary for it to define a total order between TCaps.

$$Q^x_n \succ Q^y_m \triangleq \bigvee_{s^z \in S, p^z_n \in Q^x_n, p^z_m \in Q^y_m} p^z_n \leq p^z_m \qquad (2)$$

Note that it is possible that neither $Q_n \succ Q_m$ nor $Q_m \succ Q_n$. In such a case, two system schedulers disagree on the relative importance of the time associated with a TCap. A fundamental goal of TCaps is to ensure that the delegation of a *local* priority limits the interference of the delegated budget on local threads. Thus, a preemption is made only if all schedulers agree that a preemption *should* be made.

To prevent undue interference upon delegations, the quality of the delegation is degraded to the lesser priority for each subsystem:

$$Q^x_n \sqcup Q^y_m \triangleq \bigcup_{\forall_{s^z \in S}\ p^z_n \in Q^x_n, p^z_m \in Q^y_m} \max(p^z_n, p^z_m) \qquad (3)$$

Equation 3 ensures that temporal priorities for a specific scheduler in a subsystem are maintained, and used in preemption decisions across transitive delegations. This is similar to how *labels* are tracked in distributed information flow [4] systems. The combination of Equations 2 and 3 leads to an important TCap property: *local* subsystem priorities have a *global* impact on preemptions.

C. TCap Usage Example.

Figure 2 depicts a simple system with four subsystems interacting in a simple delegation pattern. Figure 2(a) shows a parent delegating time to children, and Figure 2(b) shows a inter-child delegation (*e.g.* to accompany a s^h request for I/O from s^m). The implications of the assignment in (a) include: (1) s^m is given a lower s^p priority (higher numerical value) than s^h, an asynchronous send to it would *not* preempt s^h execution (*i.e.* $t^m_0 \not\succ t^h$); (2) the depicted interrupt to activate execution associated with t^m_1 would not cause a preemption as $B^m_1 = 0$; and (3) if these delegations were made every $T = M + N + L$ cycles, the utilization of the subsystems would be M/T for s^h, N/T for s^m and L/T for s^l.

After the inter-child delegation in Figure 2(b), in which s^h delegates m cycles into t^m_1 to be used for I/O processing on its behalf, interrupts associated with t^m_1 will preempt all execution in s^m ($t^m_1 \succ t^m_0$) and s^h ($t^m_1 \succ t^h$), but not s^p ($t^m_1 \not\succ t^p$). This inter-child delegation in (b) shows that s^m can preempt the execution in s^h for m cycles. This is significant because conventional prioritization (from the delegations in (a)) would never let s^m preempt s^h. s^h limits the interference that s^m can have on its own threads by carefully choosing m in accordance with the execution required for satisfying the I/O request in s^m. Note that any of s^p, s^h, s^m and s^l could – through delegations and re-prioritizations – prevent t^m_1 from preempting a subset of their threads, thus effectively integrating the timing of execution in *another subsystem* with their own for a time-slice as specified through the delegation.

This example demonstrates the ability of TCaps to enable fine-grained coordination between subsystems, while maintaining scheduler-local guarantees on limited interference. Even though s^p isn't involved in the delegation in (b), its timing constraints are still abided by in that only M cycles execute at its priority 1 every T cycles.

Figure 2(c) demonstrates subtle implications of delegations. Inter-child delegation from s^l to t^m_1 after (b) degrades the quality of t^m_1 for $n = m + l$ cycles. s^l delegates l cycles with lower s^l priority than t^l to t^m_1 so that t^m_1 cannot preempt s^l ($t^m_1 \not\succ t^l$). The s^p priority in s^l is the lowest among all TCaps, so an implication of this delegation is that t^m_1 inherits this low priority. Due to this, it cannot preempt any subsystem including s^m ($t^m_1 \not\succ t^p$, $t^m_1 \not\succ t^h$, $t^m_1 \not\succ t^m_0$). All of this time (n) is degraded in quality (even that which wasn't derived from s^l) which demonstrates the design of TCaps: quality is pessimistic, but guarantees that any delegating subsystem can upper-bound the interference the delegated time can have on it. Unfortunately, delegations from different subsystems have negatively impacted the overall quality of the TCap. Thus, an

important design constraint for systems where this is undesirable (for example, if they have isolation requirements between I/O requests from different subsystems) is to use different TCaps to receive delegations from different subsystems. Thus, TCaps are paired tightly with a protection medium (such as capabilities in Composite) that control which TCaps a given subsystem has permission to delegation to.

TCaps are an access-control layer that interpose on all scheduler operations and asynchronous communication. TCaps fundamentally decouple the allocation of processing time, and the consumption of time in a subsystem. This decoupling enables the distribution of the scheduling guarantees throughout the system while maintaining fine-grained control in schedulers. They provide *isolation* by limiting the amount of computation each subsystem can have, and limiting when each subsystem can cause temporal delays (preemptions) in another via $\sqcup$ and $\succ$.

We've avoided discussing the scheduling details of any specific subsystem. Which threads consume a subsystem's TCap at any point in time, is a policy decision within that subsystem and is, by design, *orthogonal* to the inter-subsystem guarantees that TCaps provide.

IV. TCaps Examples and Guarantees

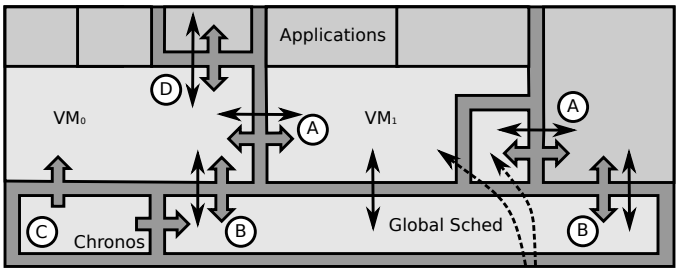

Fig. 3: TCap usage examples. Darkest gray is TCaps, and its arrows indicate temporal flows through delegation. Black arrows indicate control flow, and dotted arrows are interrupts.

By focusing on the collaboration and control of time between multiple subsystems, TCaps enable a variety of relationships between subsystems. The pattern of delegations that determines the flow of time through the system provides a number of trade-offs and design points. For example, budget delegated to a subsystem can derive from subsystems of varying levels of assurance, the granularity and priority of delegations impacts the recipient subsystem's interference, and which subsystems are charged with managing which threads and interrupts has a significant impact on their activation latency. Figure 3 depicts two virtual machines each having multiple applications, an application at the same level as the VMs, and a global scheduler. The VM_1, receives time delegations (*e.g.* for I/O computation) from the VM_0 and the application. Chronos that inserts time into the global scheduler and VM_0. We use Figure 3 as a means for describing various conventional system structures in the following.

Global Scheduling. Conventional system design uses a single subsystem with a kernel scheduler to manage system time. Analogously, the global system scheduler makes delegations and transfers control to subsystems in Figure 3 (B).

Separation Kernel. Separation kernels [2] seek to provide strong isolation between different subsystems by partitioning resources between them and minimizing the Trusted Computing Base (TCB) they rely on. This strictly controls information flow thus limiting attack surface, and provides strong temporal isolation. TCaps can be used by s^c to mimic a cyclic executive in which a *static*, table-driven schedule is emulated. The s^c delegates from t_c into each subsystem (Figure 3 in (C)), and transfers control to the last. Subsystem switches occur only when their budgets are expended, triggering the next subsystem, and s^c is activated once all budgets are expended (as $B_c = \infty$). Devices are partitioned across subsystems, and interrupt threads are given priorities via delegations from s^c such that no subsystem can preempt another.

```
const int nsubsys           = 2;
const int budgets[nsubsys] = {ms(3), ms(5)};
const int prio[nsubsys]    = {2, 1};
while (1) {
  for (int i = 0 ; i < nsubsys ; i++)
    delegate(t_c, i, budgets[i], prio[i], i == nsubsys-1);
}
```

Fig. 4: Chronos management code for guaranteeing specific rates to two, static separation kernel-style subsystems. The last argument is if the delegation should yield.

Figure 4 shows s^c's code that delegates 3 and 5 milliseconds of execution to two subsystems. The execution guarantee made to each subsystem is 3 and 5 milliseconds every 8 milliseconds, and the priorities guarantee that neither subsystem will preempt each other. Consistent with the separation-kernel philosophy, this minimal amount of trusted code ensures a strict partitioning by completely eschewing fine-grained scheduling between subsystems.

Hierarchical Scheduling. Hierarchical scheduling systems [18] (HSS) form a parent-child relationship between different subsystems. Any given parent subsystem, appropriately delegates and activates its child subsystems. These delegations can be made for a small portion of time commensurate with a single time-slice, thus enabling the parent to completely control the granularity of timer ticks for each subsystem. Each child might itself be a parent to a further layer of children as shown in Figure 3 with delegations of (B) and (D). When a child subsystem is idle, it can delegate the remaining time back to its parent.

Virtual Machines. Virtual Machines (VMs) are a restricted instance of HSSes, and the same techniques are used. However, practical VM systems such as Xen [6] create dependencies between different VMs. Most notably, Dom0 is a trusted VM in charge of multiplexing I/O (VM_1 in Figure 3). It receives asynchronous requests for service from other VMs. TCaps enable VMs that make functional requests via virtual device drivers to Dom0 to also delegate the time for the request's processing. The execution in Dom0 *on behalf* of a VM is conducted with that VM's time, thus properly accounting for

that asynchronous execution. In Figure 3, the system scheduler provides time to VMs (Ⓑ), and they make requests and provide TCaps for the I/O requests via Ⓐ.

Generality across scheduling policies. Scheduling policies are implemented in user-level logic, thus allowing arbitrary scheduling policies. However, asynchronous activations between subsystems, and interrupts vectored to subsystems use the TCap priority which is a single scalar value (for each subsystem). We implement this value as a 64-bit unsigned integer. TCaps, then, supports scheduling policies that can compress pairwise scheduling decisions into a single, large, uni-dimensional namespace. Fixed priority schedulers trivially satisfies this constraint, and UNIX timesharing schedulers linearize thread priorities into "nice" values. Dynamic scheduling priorities such as Earliest Deadline First (EDF) can also be trivially expressed by treating the priority value as a *timeline*, and setting the priority of a thread to its deadline on that timeline. This enables proper scheduling of event-triggered, dynamic scheduling policies.

V. Capability-Based OS Integration

Modern μ-kernel systems such as Fiasco.OC [5] and seL4 [11] protect access to system resources through capability-based access control. Access control policies define which *operations* a *principal* can perform on each system *resource*. These μ-kernel's capability systems focus on abstractions for memory, execution, and interrupts, and often address time management by enabling the parameterization of the fixed scheduling policy [8]. This often includes the mapping of scheduling policies into a global fixed priority namespace [5].

We implement TCaps in the Composite component-based OS [3]. Composite has a small number of simple abstractions that are all accessed through a capability-based kernel API. Capabilities are unforgettable tokens whose ownership denotes that specific *operations* can be performed on system *resources*. *Components* host user-level execution and are a collection of a capability table and page-table. All system resources are accessed through those tables, thus components provide a unit of isolation and their capabilities constrain the scope of resource accesses. Among the kernel resources accessed via capabilities are TCaps, components, threads, invocation end-points, and asynchronous coordination end-points (both for send and receive).

The Composite kernel *does not have a scheduler*. Instead, the system uses user-level, component-based schedulers for thread and interrupt scheduling, and TCaps for coordination between them. To enable user-level scheduling, the kernel provides a naive *dispatch* functionality in which any scheduler that has a capability to a thread can switch to it, even if it is executing in a different component.

Asynchronous end-points enable coordination between components, while loosely coupling their execution. They consist of a send end-point, attached to a receive end-point. The latter is associated with a thread to functionally handle the activation, and a TCap that is used both for the activation's execution, and to determine if the sending thread should be preempted (via $\succ$). Subsystems use asynchronous end-points as targets to *delegate* time from one of their TCaps, to the other subsystem's receive end-point's TCap using $\sqcup$ and budget transfer (Section III-B). This enables subsystems to both trigger events in each other – for example, child scheduler activation, or issuing a service request – and transfer budget for the subsequent processing. Interrupts in the kernel trigger asynchronous sends to a receive end point, thus enabling user-level interrupt processing[1].

A. Delegation and Revocation

Delegation is performed on an asynchronous send end-point by transferring budget and updating quality. Capability systems usually provide some means for *revocation* – the removal of access to resources that were previously delegated. This process is often recursive [10], with transitive delegations.

Perhaps unexpectedly, TCaps do *not* support revocation of previously delegated time. This is an important part of the system design as it enables the flow of time to be complex, and between mutually distrusting subsystems. Importantly, once a subsystem has been given a delegation, it has a guarantee of the full associated budget. Instead, TCaps enable limited budget delegations, and only degrade the quality (via $\sqcup$) with each delegation. This is inspired by distributed information flow systems [4] that accumulate taint with delegations to provide confidentiality, while TCaps use $\sqcup$ to constrain subsequent preemptions. The fundamental goal is to bound the interference from a subsystem that is delegated budget from another. The budget obviously limits execution, but the quality is important as it limits preemptions. For example, a global scheduler (Figure 3) making delegations to children (Ⓑ) associates those with different priorities to prevent undue control-flow switches into the lower-priority subsystem.

VI. TCaps Implementation

A. User-Level Scheduling

Syscall	Description
τ = thd_create(fn)	Create a thread executing fn
r = rcv_create(τ,$tcap$,r')	Create a receive end-point
s = snd_create(r)	Create an async send end-point
switch(τ, $tcap$, p, t, r)	Switch to a thread, execute using a $tcap$
snd(s)	Send a notification to a rcv EP
(τ, cyc, blk) = rcv(r)	Receive a notification, possibly blocking
hw_attach(hw_cap,$isr\#$,r)	Connect an ISR to a receive end-point

TABLE I. A summary of Composite system calls [3] for creating and modifying threads (τ) and asynchronous send (s) and receive (r) end-points (EPs). p, b, t are the priority, budget, and absolute timeout, respectively. Receive end-points retrieve events for the end-point, and for scheduling events. τ, and cyc are the thread that had the event, and the number of cycles it asynchronously executed, respectively. hw_attach connects execution of an ISR to activation of the thread associated with a receive end-point using the associated TCap.

[1] We use "interrupt processing" to refer to user-level interrupt processing. The "top-half" execution of the interrupts is in the kernel and includes only interrupt service routine processing, $\succ$ calculation, and possibly preemption.

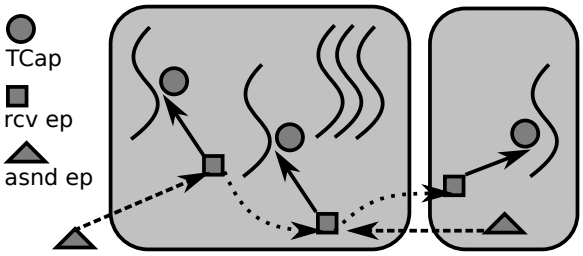

Fig. 5: TCaps data-structures, and their relationships. Solid arrows indicate the thread and TCap associated with a receive end-point. Dashed arrows show the connection between an `asnd` and `rcv` end-points. Dotted arrows indicate the scheduler thread/TCap.

Table I summarizes the relevant aspects of the Composite system call API. Component schedulers use `switch` to dispatch to a thread while activating a TCap whose budget will be consumed. `snd` and `rcv` form the basis of asynchronous notification in Composite and are complementary to synchronous thread migration that is the main form of IPC. Threads can suspend using `rcv`, awaiting a `snd`. A `snd` triggers $\succ$ calculation to determine if a preemption should occur for the thread associated with the receive end-point. Interrupts (including IPIs) are implemented as `snd` calls from the kernel interrupt handler which are attached to receive end-points using `hw_attach`.

Receive end-points deliver `snd` notifications to their associated thread, but can also send *scheduler notifications* to a scheduler receive end-point. A challenge in implementing user-level scheduling is determining if preemptions should occur, and if a thread blocks awaiting a notification, which thread to execute next. Thus, a call to `rcv` can trigger the execution of a per-subsystem scheduler thread. To support this, `rcv_create` takes another receive end-point (r') as an argument which is the scheduler's thread receive end-point. This explains why the `rcv` system call can return a thread τ, blk, and cyc. The scheduler thread will receive these values to determine if thread τ is blocked or not (blk), and if it is, how long it executed before blocking (cyc).

Figure 5 depicts the kernel's data-structures and their relationships. A receive end-point in the "child" subsystem on the left is activated (and possibly delegated to), which switches to its thread and TCap. When that thread does a `rcv`, the scheduler's thread associated with its receive end-point is activated which can dispatch between any of the four threads in its subsystem. Once it chooses to go idle and executes `rcv`, its parent scheduler in the parent subsystem on the right would be activated. If the parent wishes to reactivate the child subsystem's scheduler, it can use its `asnd` end-point to do so, possibly with a delegation of time.

Component schedulers use a combination of `switch`, and notifications from `rcv` to maintain their own data-structures and thread execution accounting. Schedulers coordinate through send and receive end-points by delegating time to each other as appropriate which will only cause control flow changes if TCap quality deems it so ($\succ$). Interrupts are vectored into subsystems, and cause preemptions based on TCap quality.

Syscall	Description
$tcap$ = `tcap_create()`	Create a new $tcap$
`tcap_delete(`$tcap$`)`	Deallocate a zero-budget TCap
`delegate(`$tcap$`,`s`,`p`,`b`,yield)`	Delegate via asynchronous channel
`transfer(`$tcap$`,`r`,`p`,`b`)`	Transfer budget $tcap \rightarrow r.tcap$

TABLE II. TCap operations. Budget (b) is transferred between TCaps, at specific priorities (p). Delegation is performed over an asynchronous notification channel (s), to a receive end-point (r) associated with a TCap ($r.tcap$). The associated thread is switched to if either it has a higher quality (via $\succ$), or if `yield` is $true$. Transfer enables a scheduler to transfer time between its own TCaps. Both delegation and transfer implement $\sqcup$.

B. TCap Interface

Table II shows the TCap interface. A TCap can make `delegation`s to a send end-point that will perform both a budget transfer with a priority p via $\sqcup$, and make a preemption decision via $\succ$. The `yield` flag is used to immediately switch to the receiving thread, and is commonly used by schedulers that want to unconditionally execute a child subsystem. This function targets a send end-point so that the receiving subsystem controls which TCap will gain budget. This enables loosely coupled subsystems to transfer time, while still controlling their own TCaps and execution. Alternatively, within a scheduler, a `transfer` via $\sqcup$ from one TCap to another can be made by directly addressing a receive end-point. This enables subsystems to maintain multiple TCaps– for example, to receive delegations from other subsystems, yet to still balance budget between them.

C. Computing and Comparing the Quality of Time

Each TCap is implemented in the kernel with a cycle-accurate budget tracked using the processor's cycle counter retrieved with `rdtsc`. Modern Intel x86 processor's cycle counter are "Constant TSC"s which are supported on all Haswell and later processors. These progress at a constant rate independent of the processor frequency, or sleep state.

The quality, Q^x, for a TCap must be implemented in a manner that enables efficient implementations of delegation ($\sqcup$) and comparisons ($\succ$) to determine if a preemption should occur. This is particularly important in Composite as the kernel is non-preemptive. Composite is a real-time system, thus every path in the kernel must be bounded and fast (*i.e.* no unbounded loops, no recursion) [3]. The implementation of delegation – $Q^a \leftarrow Q^a \sqcup Q^b$ – requires that each subsystem in Q^a is compared against the same subsystem in Q^b, and any subsystems not present in Q^a are added from Q^b. Determining preemptions – $Q^a \succ Q^b$ – requires that each subsystem in Q^a is compared against the corresponding subsystem in Q^b. Both, if implemented naively, take $O(N^2)$ as pair-wise subsystem comparisons are required.

In Composite, we implement both efficiently (1) by using an array in the TCap to store the (subsystem, priority) pairs to optimize cache locality, and (2) by ensuring that they are sorted

by a unique subsystem identifier to ensure that both operations are $O(N)$. Delegation simplifies to an algorithm similar to merging two sorted arrays, and determining preemption is a process of iterating two cursors through each array, comparing priority when a similar subsystem is found.

D. Temporal Faults and Chronos

When budget runs out, TCaps must determine what thread and with what TCap execution should continue. This is the only instant when TCaps must make what might be considered a scheduling decision. The intention is that it is almost never triggered as the fine-grained, component-based scheduling should adhere to the execution requirements of the system codified in TCaps.

However, to ensure the TCap guarantees are respected, a TCap with no budget must discontinue execution. First, the system checks if it can switch to the TCap's scheduler thread (*i.e.* the scheduler's TCap has budget). Otherwise, all TCaps with budget are tracked in a list ordered LIFO with respect to delegations, and the TCap at the head's scheduler is activated. Eventually, all TCaps will run out of time, and Chronos will be activated.

E. Scheduler Control over Timeouts

Timer management in Composite must consider TCap budgets to maintain temporal isolation, but should also enable user-level schedulers to control the frequency of timers for their computation. The frequency of timers can have a significant impact on system throughput and latency [6]. TCap budgets provide one means of controlling timeouts as when a TCap's budget expires, its scheduler thread is notified (Section VI-D). However, using this as the sole means for controlling timer notifications is inefficient and prone to scheduler race conditions. It is inefficient as it requires `transfer`s in addition to `switch`s, thus doubling the system calls required to dispatch to a thread. It is prone to races as preemptions between the `transfer` and the `switch` can lead to arbitrary computation in the mean time, thus requiring recalculation of when the timer should fire.

The TCaps API in Section VI-A passes a *timeout* parameter to `switch`. Timeout is specified on an absolute timeline (*i.e.* not relative to the current time), thus removing the possibility for any races and simplifying scheduler implementation. Two special cases must be considered. First, if the timeout is past when the TCap's budget will expire, the budget expiration is used for the timeout instead. Second, a preemption can, at worst, lead to a timeout being specified for a time in the past, which the kernel will return as an error (`EAGAIN`). The *priority* argument to `switch` is comparably motivated. Instead of requiring a `transfer` to change the priority of a thread's execution, it can be passed as an argument and is used for that TCap only during that thread's execution.

The kernel uses architectural *one-shot* timers that are reprogrammed when threads are dispatched or activated due to a `snd`. We have experimented with High Precision Event Timers (HPET), Local Advanced Programmable Interrupt Controller (LAPIC) one-shot mode timers, and the newer LAPIC TSC-deadline mode. These timers have trade-offs in programming time and precision. We found that on our machines, the HPET takes more than 10 μ-seconds to program, while both LAPIC methods take on the order of 100-400 cycles. One-shot timers have an internal clock and the kernel must transform a user-level specified timeout into that clock's domain. This has the impact of making timeouts more coarse-grained (on the order of μ-seconds), and requiring expensive calculations on thread dispatch. Thus, when it is architecturally available, we use the TSC-deadline mode which enables timeouts to be specified at a cycle granularity.

The overhead of programming the LAPIC methods does have a non-negligible overhead. The `switch` operation without reprogramming the timer is almost 70% faster than when it is programmed. Thus, we enable user-level schedulers to avoid that cost when possible, ideally once per actual timer interrupt. Schedulers can pass in a special value for the timeout which maintains the previous timer's programming when not switching between TCaps.

F. Intra-Subsystem Resource Sharing

Predictable resource sharing protocols limit priority inversion when a low-priority task mutually exclusively holds a resource that is contended by a higher-priority task. Though we omit details here, this is supported with TCaps by using such support in user-level schedulers while inheriting the TCap of the contending thread.

G. Multi-processor Considerations

This paper focuses on a single core's computation. TCaps are intrinsically a per-core, partitioned resource; processing time cannot generally be transferred between cores. Extending to multiple cores is beyond the scope of this paper. Even if limited to only a single core, the current results are useful given embedded system's difficulty in achieving predictable behavior out of multi-core machines. For example, the FAA [19] cites significant difficulties in certifying multi-core systems and limits systems that require strict isolation between untrusted subsystems to uni-processors.

VII. Virtualization System Design

To evaluate TCaps, we implement a virtualization system that shares much of its design with Xen [6]. As with Xen, we segregate device drivers into a more trusted domain that communicates with the other VMs in the system via virtual device drivers. We will adopt Xen's terminology and refer to the device driver domain as "Dom0", and the application VMs as "DomU". Xen does *not* account Dom0 execution to the DomU for which Dom0 computes, and it executes Dom0 effectively with high priority when it handles I/O and interrupts. A separate global scheduler multiplexes the CPU between the VMs (as in Figure 3 (B)), while Chronos delegates to it. DomUs request I/O through their virtual drivers to Dom0 using `snd` calls, and a large shared ring buffer per communication direction that is used to pass data between

Operation	Composite	Linux
Round-trip IPC	*787*, **3664**	pipe RPC:8678
`switch`	*365*, **2708**	yield:1157
`switch`, set timer	*418*, **2734**	
`switch`, inter-pgtbl	*542*, **3778**	
`asnd + rcv`	*770*, **5332**	pipe:3233
scheduler activation	*321*, **504**	
timer activation	*881*, **2608**	sigalarm:7331
`delegate`	*786*, **7480**	pipe:3233
TCap preemption calc. ($\succ$)	*70*, **616**	
`transfer` ($\sqcup$)	*562*, **4276**	

TABLE III. Composite kernel operations measured in cycles (2738 cycles = μ-second) – *Average Costs*, **Worst-Case Measured Time (WCMT)** and Average costs for Linux operations.

VMs. We will call this system cosXen as it is, in some ways, a port of the Xen architecture to Composite with the main design changes being that all our scheduling is user-level, and the use of TCaps for coordination.

TCaps are integrated by delegating time *along with* the functional requests from each of the DomUs to Dom0 (as in Figure 3 (A)). Dom0 maintains a TCap for each DomU that requests service, and it uses those TCaps to also handle interrupt processing. In this way, the end-to-end I/O processing due to asynchronous communication with Dom0 is accurately accounted to the VMs requesting that service all the way down the interrupt processing. The system scheduler is not involved in such delegations, and Dom0 does not need to trust the VMs making requests and providing execution budget. We call this second version of the virtualization system cosDist as the flow of time is distributed between VMs.

VM implementation. We build our virtualization platform on Composite and Rumpkernels [20]. Each VM, and the system scheduler are all implemented as separate subsystems in Composite. They are based on NetBSD and consist of the following changes: (1) virtual memory support is removed as the application is co-resident with the kernel services, and (2) scheduling, synchronization, and memory management are removed from the kernel, and provided by a library instead. Rumpkernels have multiple back-ends including those that execute in kernel-mode on the bare-metal.

We re-implement the library for scheduling, synchronization, and memory management to use the Composite primitives. Notably, the scheduling of application threads, kernel threads, and interrupt threads (*i.e.* software interrupts) are re-implemented to use TCaps. Device drivers are those provided by NetBSD and are used unmodified. We create a `cnic` pseudo-device sends packets between Dom0 and each VM. It includes a shared-memory ring-buffer for passing data between VMs, and a set of send and receive end-points that are used to trigger notifications and pass time (in cosDist).

VIII. Evaluation

All evaluations of TCaps are performed on a system with an Intel Core i7-2600S processor running at 2.80 GHz with 4 GB of memory (less than 800MB are used in the Composite cases). Only a single core is enabled in each test. The system has an Intel 82579LM Gigabit NIC directly connected to clients generating workload over a Gigabit Ethernet and we are using NetBSD version 7.99.17 in Rumpkernels for device drivers.

A. Complexity

TCaps form the basis for time management in Composite, and are fundamental in making temporal guarantees. It is important that this foundation is relatively simple. Though an imperfect metric for complexity, the Composite kernel is less than 7,500 lines of code, of which TCaps operations are 300. We believe this, along with a minimal Chronos, is small enough to have high confidence in the system.

B. Microbenchmarks

Table III evaluates the efficiency of each of the constituent TCap and time management operations, compared versus typical Linux operations, where applicable. Each of these costs is the result of averaging over 1 million different executions along with Worst-Case Measured Time (WCMT) costs for Composite kernel operations.

Discussion. Composite is an optimized μ-kernel, so it is no surprise that its communication primitives are efficient. Importantly, the dispatch overhead for the system, even though it necessarily involves a system call, does not appear to be prohibitive. TCap operations also seem relatively well optimized and `delegate`, which can also trigger control flow transfer, demonstrates relatively high performance compared to Linux pipes that don't transfer time. All TCap operations involve priorities from 3 subsystems. The WCMT costs for each operation is measured with a cold cache. We have observed occasional spikes in WCMT that are consistently around 400,000 cycles and we think they're due to Non-Maskable Interrupts (NMI) interference. We omitted these from our measurements. These microbenchmark experiments were rerun on a newer hardware Intel Core i5-6400 processor running at 2.70 GHz with 8 GB, and all the benchmarks were faster, but relatively similar across the different systems. Another important detail to note is that the newer hardware did not have the spikes of 400,000 cycles that we observed on the older hardware and attributed to NMI interference.

C. Scalability of TCap Operations

TCap operations increase in overhead as the time is delegated through multiple subsystems. Here we study the impact of this on the three key TCap operations: `dispatch`, `transfer`, and the computation of if a preemption should occur ($\succ$). Figure 6 plots the Average and WCMT costs of these three operations as an increasing number of subsystems is involved. The Composite kernel puts an upper-limit on the number of subsystems a TCap tracks – 16 in the current implementation – to maintain a predictable (bounded) kernel execution time, and to simplify the allocation of statically-sized TCap memory.

Discussion. As detailed in Section VI, the cost of each of these operations is linear in the number of subsystems

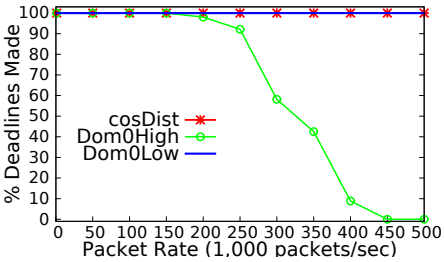

Fig. 7: % Deadlines made by DLVM for different packet rates. Packets are destined to IOVM and with no CPUVM interference.

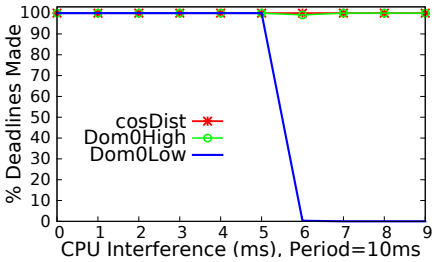

Fig. 8: % Deadlines made by DLVM in different systems for different CPU Workloads from CPUVM and no IO interference.

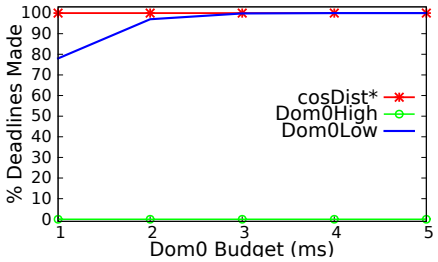

Fig. 9: % Deadlines made by DLVM in traditional systems for different DOM0 budgets with a ping flood to IOVM.

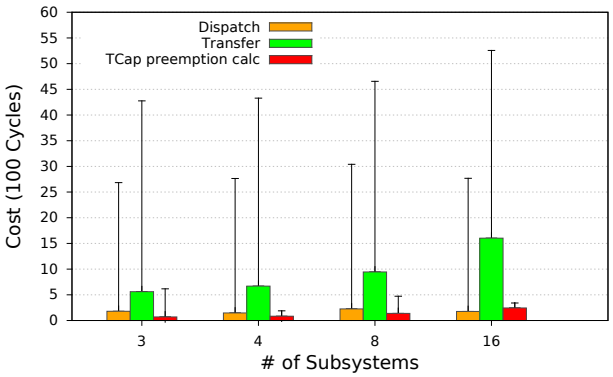

Fig. 6: The scalability of TCap operations as more subsystems are involved. errorbars are used to depict **WCMT** costs.

involved. `dispatch`, which only switches between TCaps has a constant cost. Notably, the $\succ$ operator is efficient, which is important given that it is evaluated for each interrupt.

D. TCaps and Priorities

To understand how systems handle the I/O dependency of multiple subsystems on a Dom0, we investigated Xen. Xen effectively gives Dom0 highest priority, and in our tests, this could lead to livelock situations, a phenomenon we investigate here. Some details of our Xen setup are in Section VIII-F. To understand the impact of different priority and budget assignments, and to understand the utility of TCaps, we evaluate three different system configurations in Composite.

All three setups have three VMs: Dom0 that handles interfaces with devices on behalf of clients; a best-effort VM designed both to generate interference and to test throughput; and a hard real-time VM performing event-driven computation. The best-effort VM is configured to either interact with Dom0 for networking I/O – we call this the IOVM – or to generate CPU interference – we call this the CPUVM – depending on the test we are conducting. The IOVM is running the Nginx webserver to handle HTTP workloads. The real-time VM is deadline-sensitive – the DLVM – and it is event-activated by a device in Dom0 through a service request. To ensure periodic activations of DLVM from the device so that we can best measure the real-time responsiveness of the system, we use the HPET timer whose period is configured to 10ms. In all cases, a global scheduler does Fixed-Priority (FP) scheduling.

The first two system setups we study are Dom0High and Dom0Low which rely on the global scheduler to control all processing time allocations to each of the VMs (Ⓑ delegations in Figure 3). These two systems differ only in their priority assignments. As Dom0 is relied on by both other VMs for I/O services, its priority is important. In Dom0High, Dom0 is set to high priority, DLVM to medium priority and IOVM/CPUVM to low priority. The motivation is to give Dom0 high priority as both DomUs require I/O from Dom0. However, high rates of I/O traffic served by Dom0 might interfere with DLVM deadlines. In Dom0Low, Dom0 is set to low priority, DLVM to high priority and IOVM/CPUVM to medium priority. The motivation is to give Dom0 low priority to limit the impact of high rates of I/O that can interfere with deadline completion. These setups enable us to investigate the trade-offs that these priority assignments make with respect to predictability and throughput.

The third system configuration we study is cosDist that leverages TCaps to delegate time between VMs (Ⓐ + Ⓑ Figure 3) – the DLVM is set to high priority and the IOVM/CPUVM set to low priority. Dom0 receives TCap delegations from those VMs with which to conduct its I/O. The global scheduler also delegates to it a minimal amount of medium priority time for its own service tasks. The HPET interrupt handler in Dom0 uses a separate TCap which receives DLVM delegations. Unless otherwise mentioned, the DLVM is set to have a utilization of 0.5 with a pipeline of threads that execute for a WCET of 5ms over HPET activation, and use an implicit deadline of 10ms.

In the first experiment in Figure 7, we evaluate how all three systems handle an increasing rate of network traffic going to an IOVM via Dom0. To prevent the IOVM from starving Dom0 in Dom0Low, the global scheduler constraints it to a budget of 4ms every 10ms. In this experiment, we use a UDP flooding client to generate the configurable workload.

Discussion. The results show that with increasing packet rates, the DLVM in the Dom0High fails to make its deadlines due to priority inversion caused by an increasing amount of network processing in Dom0 for the IOVM. The system does not properly discriminate between I/O performed for the DLVM

versus the IOVM, thus suffering significant unpredictability. .In the Dom0Low setup, DLVM makes most – but not all – of it's deadlines because the network interrupts, which are processed by Dom0, cause minimal interference on the higher priority DLVM.

The cosDist system does not miss any deadlines because Dom0's HPET interrupt processing is processed using the DLVM's budget and priority, thus preventing interference from the IOVM, and from the interrupt paths in Dom0 for network processing. This demonstrates the ability of TCaps to enable the controlled distribution of budget that is properly prioritized from application all the way to interrupt processing.

Figure 8 depicts the percent of deadlines made by DLVM with varying interference from a CPU-bound VM. We use the same setup as the previous experiment except that we disable the IOVM and introduce a CPUVM that produces a budget's worth of interference on lower priority subsystems.

Discussion. Contrary to the previous experiment, DLVM in Dom0High is now able to meet all deadlines as neither it nor Dom0 suffers any interference. Similarly, cosDist avoids any priority inversion as all processing in the I/O path for the DLVM is properly prioritized and budgeted. However, the DLVM in Dom0Low misses all of its deadlines once the CPU-bound interference on low-priority Dom0 prevents it from processing the HPET activations in time.

E. TCaps and Budget

A conventional technique for constraining high-priority interference is through rate-based servers. As discussed in Section II, choosing a budget for Dom0 can be difficult. On the one hand, a large budget can still cause significant interference, while on the other hand, a small budget can stifle throughput. In this section, we attempt to meet deadlines while maximizing by giving Dom0 a budget in both Dom0High and Dom0Low (note that Dom0 already has only a minimal budget in cosDist).

In Figure 9, we compare how Dom0High Dom0Low and cosDist behave under high I/O workloads with varying Dom0 budgets. We give the IOVM infinite budget as its execution is constrained by either Dom0's processing time in Dom0Low or, as in the two other setups, it's lowest priority, rendering its budget unimportant. We vary Dom0's budget from 1ms to 5ms (as more could impact the DLVM).

Discussion. Similar to the previous experiments, a high I/O workload still cases the DLVM in Dom0High to miss all deadlines. In this case, Dom0 will often run out of budget *before* the HPET even gets a chance to activate, thus delaying the notification to the DLVM till after deadlines have been missed. In contrast, cosDist accounts the I/O processing in Dom0 for the separate VMs to the budgets specifically delegated by those VMs, thus avoiding the need to budget Dom0. At low budgets, the Dom0Low system shows that Dom0 cannot process all interrupts (including the HPET) reliably in the time given. However, with larger budgets, it is able to reliably process the HPET in time to deliver it to the DLVM.

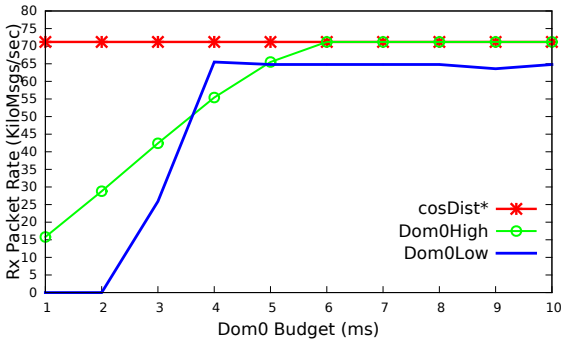

Fig. 10: Throughput measurements for IOVM running a UDP server and with smaller DLVM execution (C=0.5ms P=10ms), using a simple UDP client tool on the host.

To assess the potential for the system to not only meet deadlines, but also to best utilize spare capacity, we study the throughput of the system. Section VIII-F compares the throughput of the IOVM against Xen to determine if the TCap system provides throughput on a practical level.

Figure 10 depicts the throughput of IOVM at varying Dom0 budgets. We use a similar setup as the experiment in Figure 9 but modify the execution time in the DLVM to be closer to "average-time" than worst (0.5ms instead of 5ms). We have a simple UDP server application running in the IOVM for this experiment which recieves a 16Byte payload and replies to the client with a 16byte payload. Throughput is measured using a UDP client tool on the host PC that sends UDP packets at a very high rate (72K Packets/sec), each containing a 16Byte payload and displays the reception rate information on the console. The higher packet rate stresses the interrupt processing.

Discussion. Lower budgets for Dom0 do decrease its interference on the rest of the system, but we see the impact of that lower budget here. At low budgets, Dom0 doesn't have enough processing time to drive a high throughput for both Dom0Low and Dom0High systems. At high budgets, the throughput for the IOVMs in both Dom0Low and Dom0High plateau and we observe that the Dom0High throughput even for high budgets does not exceed cosDist system where TCaps are used extensively and I/O processing in DOM0 is accounted to IOVM.

F. I/O Throughput with different systems

Figure 11 studies the efficiency of the TCap system implemented in Composite. We compare I/O throughput with `nginx` 1.6.2 running in four different systems: cosDom0, cosXen, cosDist, and in a DomU in Xen. We use Xen version 4.4 running Linux (as the most supported Xen OS) version 3.16 in both Dom0 and DomU. We use ApacheBench (`ab`) 2.3 to test each system with and without the persistent HTTP connections (*e.g.* the "KeepAlive" flag). The transfer size used for this is 1KB, with concurrency set to 100 and number of connections to 100k.

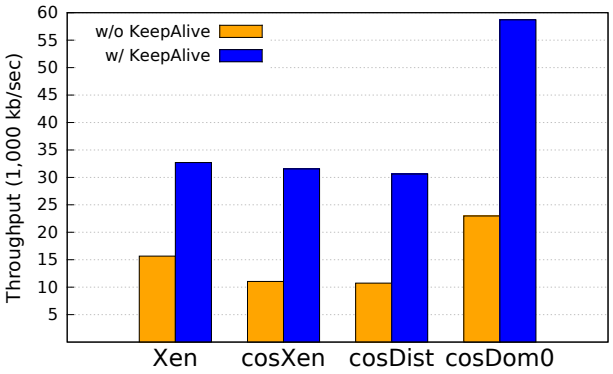

Fig. 11: Throughput for `nginx` in different systems.

Discussion. User-level scheduling is used in cosDom0 running `nginx`, and TCaps are used to schedule its networking interrupts. cosXen adds a separate VM serving clients and relies on the system scheduler to make quantum-sized delegations to the different VMs (Ⓑ Delegations in Figure 3). While using `ab` without KeepAlive, the throughput of cosXen and cosDist is 29% lower than Xen. However, with KeepAlive, we observe a similar throughput in Xen, cosXen and cosDist. Our virtualization system appears to interact badly with TCP connection creation. The KeepAlive results show that this issue is not likely attributable to the TCaps mechanism. Importantly, cosDist does not show a throughput decrease compared to cosXen. Our virtualization system is completely unoptimized, uses up to five memory copies for each packet during inter-VM message passing, has a trivial system scheduling policy (RR without blocking) that doesn't do I/O boosting, and uses the NetBSD networking stack for packet processing and routing. We believe that these results show the ability of the system to provide efficient user-level scheduling, and a low-overhead, pervasive use of TCaps that has a reasonable performance compared to commonly used systems.

IX. Conclusions

This paper introduces Temporal Capabilities that enable the decoupling of fine-grained scheduling decisions (that are made at user-level) from the temporal guarantees including predictable interrupt scheduling, isolation, and inter-scheduler coordination that is increasingly prevalent in modern systems. TCaps are an integral part of the solution to a long-standing challenge [11]: how to enable configurable, isolated, and efficient user-level definition of CPU management policies, and integrate them into a capability-based OS. We've demonstrated the ability of a system with dependencies spanning from application all the way to interrupts to correctly prioritize and budget both subsystems with strict timing requirements, and best-effort subsystems that don't, by both meeting all deadlines, and achieving high throughput, a combination no comparison system achieves.

References

[1] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.
[2] M. Danish, Y. Li, and R. West, "Virtual-cpu scheduling in the quest operating system," in *RTAS*, 2011.
[3] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *RTAS*, 2015.
[4] 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*, 2005.
[5] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *EMSOFT*, 2012.
[6] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *EMSOFT*, 2011.
[7] U. Steinberg, J. Wolter, and H. Hartig, "Fast component interaction for real-time systems," in *ECRTS*, 2005.
[8] A. Lyons and G. Heiser, "Mixed-criticality support in a high-assurance, general-purpose microkernel," in *WMC*, 2014.
[9] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *WTEC*, 1994.
[10] J. Liedtke, "On micro-kernel construction," in *SOSP*, 1995.
[11] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *SOSP*, 2013.
[12] U. Steinberg, A. Bottcher, and B. Kauer, "Timeslice donation in component-based systems," in *OSPERT*, 2010.
[13] S. Ruocco, "A real-time programmer's tour of general-purpose l4 microkernels," in *EURASIP Journal on Embedded Systems*, 2008.
[14] M. Vanga, F. Cerqueira, B. Brandenburg, A. Lyons, and G. Heiser, "Flare: Efficient capability semantics for timely processor access," Max Plank Institute, Tech. Rep., 2013.
[15] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS*, 2008.
[16] B. Ford and S. Susarla, "Cpu inheritance scheduling," in *OSDI*, 1996.
[17] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *RTAS*, 2011.
[18] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *RTSS*, 2001.
[19] "FAA CAST position paper 32." 2016.
[20] A. Kantee, "Rump file systems: Kernel code reborn," in *USENIX*, 2009.