SPECK: a Kernel for Scalable Predictability

Qi Wang, Yuxin Ren, Matt Scaperoth, Gabriel Parmer
The George Washington University
Washington, DC
{interwq,ryx,mscapero,gparmer}@gwu.edu

***Abstract*—Multi- and many-core systems are increasingly prevalent in embedded systems. Additionally, isolation requirements between different partitions and criticalities are gaining in importance. This difficult combination is not well addressed by current software systems. Parallel systems require consistency guarantees on shared data-structures often provided by locks that use predictable resource sharing protocols. However, as the number of cores increase, even a single shared cache-line (e.g. for the lock) can cause significant interference.**

In this paper, we present a clean-slate design of the SPECK kernel, the next generation of our COMPOSITE OS, that attempts to provide a strong version of *scalable predictability* – where predictability bounds made on a single core, remain constant with an increase in cores. Results show that, despite using a non-preemptive kernel, it has strong scalable predictability, low average-case overheads, and demonstrates better response-times than a state-of-the-art preemptive system.

I. INTRODUCTION

Multi-core systems have proven to be a double-sided sword for embedded and real-time systems. They provide increases in computational power that promise to not only consolidate previously distributed systems together, but also to increase the computational capability, thus intelligence and functionality, of embedded systems. However, these parallel systems present a significant challenge due to the interference between tasks caused by increased resource sharing between cores. For example, different cores often share hardware resources such as last-level caches (LLC) and memory buses. Past research has addressed each of these in turn by, for example, partitioning memory [1] or cache [2]. An inescapable challenge not addressed by these techniques is the interference caused by the *sharing relationships of data-structures within software due to cache coherency*. This problem is complementary to previous approaches, and it is particularly important: a store to a cache-line can (on our hardware) take three cycles, or *more than 27*μs, depending on coherency behavior.

If each load or store can have such high variance, depending on the sharing relationships in the software, it is reasonable to design the software to mitigate these effects. The impact of this overhead not only impacts a task's response time, but increases the interference between competing tasks. One task's data-structure access pattern in the kernel can increase the latency of another. This cross-talk makes temporal isolation difficult across cores. A high criticality task, tested in isolation on a system could suffer memory access latency spikes when a low criticality task is added to the system that contends a shared kernel data-structure.

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

This paper presents the SCALABLE PREDICTABILITY-ENABLED COMPOSITE KERNEL (SPECK), a kernel with the goal of not only scaling its performance, but also its worst-case latency with an increasing number of cores. We call this *scalable predictability*, and it is a strong form of scalability that focuses on the *worst-case overheads from cache-line coherency traffic* – in addition to the average behaviors that are often the focus of scalable systems – and on avoiding coherency traffic all-together. SPECK is a kernel that focuses on simplicity, and on exporting system policies from the kernel instead to be implemented in user-level components in the tradition of COMPOSITE [3], [4].

```
lock(&thd_lookup_lock);
t = thdlkup_find(thdid);
lock(t->thdlock);
thdlkup_remove(t);
unlock(&thd_lookup_lock);
thd_cleanup(t);
unlock(t->thdlock);
```

The shared data structures of kernels must be kept consistent despite parallel accesses from many different tasks on different cores. Locks are often used to ensure this consistency by serializing access to data-structures. For example, the simplified code to the right deallocates a thread and uses fine-grained locking: a lock protects the lookup structure, and a per-thread lock protects the thread structure. With increasing core counts, the overheads for cache coherence for the cache-line backing the `thd_lookup_lock` become quite large, especially in the worst case. Whenever the OS includes a data-structure that is accessible from within both low- and high-criticality tasks (as in the thread lookup structure in this example), both tasks' timing are impacted by the sharing.

To remove the overhead and interference from locks and shared kernel structures, SPECK takes a drastic approach: The kernel is *lock-less*, instead ensuring *consistency on shared objects with atomic instructions* on the granularity of kernel objects that either fit into a cache-line (*i.e.* the finest-possible granularity), or are core-local. This means that coherency overheads only occur when tasks on different cores (1) have *access* to the same object, and (2) actually attempt to modify the same object. This is in contrast to traditional methods of ensuring consistency that use locks where cache coherence overheads and interference impact (at least in the worst-case) *all* cores that must do lookups (thus, in our example, take the lock) of *any* resource.

Without locks, it is difficult to handle proper deallocation of resources. In the above example, no other core is accessing the lookup table while the thread is being removed from it, so after the `thdlkup_remove()` and `unlock` operations, the thread can be freed as no references to it exist. To avoid coherence traffic, SPECK instead uses quiescence-period based tracking

of references. Once removed from lookup structures, the system guarantees that no references (pointers) exist to resources after a bounded delay, thus enabling the memory to be reused. This quiescence-period is based directly on the worst-case execution of the non-preemptive kernel, effectively leveraging predictability guarantees of the system for the overall system design. SPECK's quiescence framework is also applied to virtual memory mappings. Thus, Translation-Lookaside Buffer (TLB) shootdowns (*e.g.* in `munmap`) that are traditionally extremely expensive on multi-core processors, have a constant worst-case overhead, thus enabling complicated memory mapping manipulations in real-time computation.

In addition to avoiding coherency overheads on accesses to different kernel objects, each lookup structure in the kernel is also strictly access-controlled along two dimensions: (1) modification access that can cause cache-coherency traffic (*i.e.* remote invalidations), and (2) kernel object access. Kernel data-structures are structured as simple radix tries called *resource tables* with embedded modification access bits. The capability to modify these tries is only granted through *higher-order resource tables*, references within resource tables *to* resource tables. Components of the system with heightened permissions (*e.g.* the booting component, similar to `init`) can access and modify all system resources. Such components create additional components that execute system and application tasks that have access to a *subset of the system's resources*, none of which are modifiable. This creates a container of resources whereby any execution therein is guaranteed to not create any modifications to kernel state that could cause expensive contention coherency traffic, thus ensuring scalable predictability for all of that component's system calls.

Contributions. This paper's contributions include:

- An introduction to the design and implementation of the SPECK kernel that provides both scalable predictablity, *and* low average-case overheads. This includes resource tables for lock-less kernel object lookup, access control, and fine-grained consistency management; and quiescence-based kernel object deallocation and reuse.
- A discussion of the support in SPECK that provides the foundation for the COMPOSITE component-based system. This requires that system policy is implemented in fine-grained user-level components (*e.g.* user-level scheduling). It enables system customization and fault tolerance, and efficient, low-level primitives for Inter-Process Communication (IPC) and memory mapping.
- An evaluation of the performance, predictability, and scalability properties of the SPECK kernel, and a comparison to a real-time L4 μ-kernel, Fiasco.

Paper organization. Section II investigates worst-case hardware cache-coherence overheads to motivate the goals of SPECK that are discussed in Section III. SPECK's design is covered in Section IV, and Section V translates how this design provides the necessary guarantees. Section VI evaluates the system and compares it to an L4 variant, Fiasco. Sections VII and VIII discuss related work and draw conclusions, respectively.

II. Background and Motivation

Cache-line Sharing Coherency Latency. Studies of the average-case scalability [5] of modern hardware have shown that data-structure construction, and synchronization primitives can have a large impact on overall performance. However, no comparable study has been performed on the *worst-case* scalability implications of different primitives. Even more-so, it is important to understand the limitations of the hardware coherence implementations as they place lower bounds on the latencies for sharing data between cores.

To study the hardware overheads for sharing cache-lines between cores, we run a series of experiments on a system consisting of Intel Xeon E7-4850 (10-core chip) clocked at 2.0 GHz, with four sockets. Hyper-threading is disabled, leading to a total of 40 cores. With 39 cores generating workload (one is used for system tasks and interrupts), we measure the latency of single load and store instructions. We measure with half store (writer) threads, half load (reader) threads, and, separately, with all but one writer thread. The results are consistent across both settings, and the average load and store latencies are close to 10μs which is similar to results shown in [5], but the *maximum* latency is 27μs. Fetch and add (`faa`) instructions yield similar results. Worse, compare & swap (`cas`) instructions that modify memory impose latencies of 17μs on average, and *50μs* in the worst case. Such results show that contention on shared and modified cache-lines can have significant overheads, and that a *single instruction can significantly degrade execution latencies*.

Do these results generalize to higher-level synchronization primitives? The Appendix details a study of the average-case and worst-case overhead experienced by a number of synchronization primitives. We survey a naive spinlock, a spinlock with geometric backoff, a trivial lock-free stack (without ABA protection), and two predictable spinlocks: a ticket-lock, and a MCS lock. In the worst-case, implementations with tight spins around shared memory do not scale. Surprisingly, the measured worst-case overheads for MCS locks, that are known to be scalable (in the average case) approach 50μs. When protecting a cache-line that is modified in the critical section, latencies increase further to over 65μs.

Discussion. In the worst case, the latency imposed by cache-coherency can significantly degrade the WCET of system code, and if such operations are in the path that determines system response-time, they place a strong lower-bound on that latency. If the WCET of a system is to be analyzed, assuming the worst case cache latency for every shared cache access will cause the WCET to be very pessimistic. In addition, the high cache-coherency overhead can also play a significant role in temporal interference between different criticality threads that access a shared system structure.

III. SPECK Goals

SPECK is designed to provide pervasive, efficient isolation, and scalable predictability guarantees for hard real-time tasks. Specifically, an application or system service that can be implemented scalably in a user-level component, must not

be limited by any kernel contention, particularly from lower-criticality interference. We seek a particularly strict sense of scalability: even in the worst-case the latency bounds made on a single core should hold with increases in core count.

G1 *Avoid cache-line contention for real-time tasks.* Given the overheads of modifying a shared cache-line, SPECK is designed to (1) ensure objects fit into a single cache-line and provide synchronization at that finest-possible granularity of contention, (2) ensure that all contention is explicit to user-level through the system-call API so that it can be avoided, and (3) ensure that the common-case uses of the kernel perform no shared cache-line modifications. One key to avoiding cache-line modifications in the common case is to provide liveness tracking of kernel objects. The key question of "when can an object be deallocated?" is complicated by parallel accesses to it, and common solutions such as reference counting do not scale.

G2 *User-level definition of policy, and pervasive inter-component isolation.* A fundamental goal of COMPOSITE is the extraction of resource management policy from the kernel to user-level components. SPECK takes this one step further, and enables the component-based control over all kernel scalability properties. This is essential to provide strong temporal isolation boundaries between components to match the spatial isolation provided by hardware protection.

G3 *Aggressive optimization of the fast-paths.* Optimizing for scalability often implies slowing and complicating the code. For example, fine-grained locking increases parallelism, but imposes more overhead [6]. SPECK is intended to not only provide latencies that do not increase with an increasing core count, but with latencies that are competitive on a single core with existing state-of-the-art systems.

IV. DESIGN AND IMPLEMENTATION OF SPECK

SPECK leverages a clean-slate co-design of the kernel's data-structures, interface, and contention mechanisms to achieve the required scalable predictability guarantees. This co-design enables the finest-gained synchronization at the level of individual kernel-object cache-lines, an explicit interface mapping from the kernel namespace to cache-lines that enables explicit cache-line contention avoidance, and a quiescence-based reclamation scheme based on the very predictability guarantees made by the kernel.

SPECK also maintains the fine-grained component-based isolation of COMPOSITE systems, and the implementation of all resource management policy in user-level components. *Components* here refer to a unit of code of data that 1) are memory isolated and execute in user-level, 2) export a set of functions that can be invoked by other components, and 3) have a set of functional dependencies on other components. A system is composed from a collection of specifically chosen components by satisfying all of their functional dependencies. Components in COMPOSITE include application processes and middle-ware in addition to low-level policies such as schedulers, memory managers, synchronization managers, file systems, and networking protocols.

A. SPECK *Kernel Abstractions*

SPECK has a small number of abstractions. These are focused around providing 1) the access control necessary to strictly control both the system resources accessible by each component, and the modification permissions to each of those resources to limit cache coherency operations, 2) efficient and bounded-latency access to those resources, and 3) the capability to define system resource management policies in user-level components.

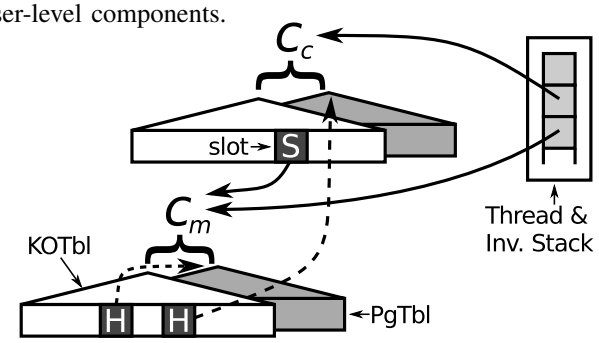

Fig. 1: SPECK's main data-structures. Components (here, a client C_c, and a manager, C_m) each include a KOTBL and PGTBL. C_c has a SINV object in a KOTBL slot (labeled S), that enables it to invoke C_m. In this case, C_m provides a memory mapping service, so slots (labeled H) in its KOTBL include higher-order PGTBLs. Thus, C_m can modify the PGTBL of the client by adding pages into it. A kernel thread structure includes an invocation stack that tracks a thread's execution as it migrates between components. In this case, the thread has invoked C_m from C_c (the stack grows down). System memory is referenced within the PGTBLs.

Resource tables (RESTBLS)**.** Each component is defined by a set of resource tables that index and control access to system resources. Each resource table is a radix trie consisting of nodes up to a fixed depth. Each node in the radix trie has a number of slots that hold references to other kernel objects. Slots in nodes that are internal in the radix trie (*i.e.* non-leaf nodes) hold references to nodes of the next level. Each radix trie maps an index into a *slot* in a leaf node that references a resource (*e.g.* a thread, memory frame, and other resource tables). This same data-structure is used for various system structures, each with different internal node radices, different numbers of slots in the leaves, and different node sizes. These include:

- KOTBL – the Kernel Object tables enable components to access and use kernel objects. The only system call in SPECK performs an operation on the kernel object at an index in the KOTBL.
- PGTBL – page-tables are the main mechanism for tracking not only virtual mappings for the component, but also access to physical memory used for kernel data-structures.
- LIVENESS TABLE – This structure is used to track the liveness and quiescence of kernel objects. This structure will be elaborated in Section IV-F.

The only mechanism to make modifications to resource tables is *higher-order resource tables* – references within resource tables *to* resource tables, themselves. Most components in the system do not have such access, thus make no modifications to shared kernel structures. Components with access to higher order resource tables are in charge of handling delegation of, and access to, resources throughout the system.

Even these management components only modify and access resource tables and resources reachable through their own resource tables. Higher-order resource tables are a significant departure from the previous COMPOSITE kernel and from existing μ-kernels. Figure 1 depicts some of SPECK's data-structures including RESTBLs. In contrast, capability-based systems (*e.g.* seL4 [7] and Fiasco L4 [8]) often provide kernel-supported delegation of resources between servers (components) via IPC. The bookkeeping for these delegations to support revocation of resources requires complex synchronization [6], and revocation execution time is bounded by the size of the bookkeeping data-structures. As SPECK's aim is to avoid all such locks and their associated impact on scalability and interference, and to bound kernel execution times, we instead use higher order resource tables managed by user-level components. Of note, the same policies in capability-based systems for delegation and revocation are implemented in versions of, for example, memory mapping management components.

Components. Components are the combination of the resources addressed by the namespaces of a KOTBL and PGTBL. Components are active only when a thread is created in them, or when one of their exported functions is invoked by another component. System calls are interpreted as lookups in the KOTBL for the active component, and all memory accesses are logically treated as PGTBL lookups. The latter are, of course, accelerated by the hardware page-table walker of our processor, and translations are cached in the TLB cache. A component's complete set of access rights are thus defined by the contents of their resource tables.

Communication channels. COMPOSITE requires highly optimized mechanisms for Inter-Process Communication (IPC). SPECK provides synchronous invocations between components with function call semantics (*i.e.* call-return) that can pass arguments in registers. The component to activate on invocation, and the instruction to initiate execution at in that component are stored in the SINV kernel object. The SRET object designates a return to the calling component. Asynchronous notifications, either within a component, or across components, are transmitted using ASND objects which activate threads blocked on ARCV objects on possibly different cores. Faults are routed through SINV objects, and interrupts are exposed to components as ASND objects that can be wired to a component's ARCV objects.

Threads. Thread structures are kernel objects also accessed through KOTBLs. Threads are orthogonal to components in COMPOSITE: a thread migrates [9] between components via invocations via SINV objects. Each thread tracks the sequence of components that have been invoked (*i.e.* including nested invocations) in the kernel thread structure in an *invocation stack*. The head of the stack is the component in which the thread is active. Thread kernel objects also store a register context for preempted threads.

Thread structures do not include a kernel execution stack. The SPECK kernel is non-preemptive, thus we use a *single kernel stack per-core* which is the root of all data-structures for that core. Most notably, a structure embedded with the kernel stack references the currently executing thread, and the active component is found on the top of its invocation stack.

System memory. Memory frames are accessed via RESTBLs just like any other system resources. SPECK supports the user-level management of kernel memory in a manner similar to seL4 [7]. The kernel does not have a memory allocator, and instead relies on user-level to provide memory to back kernel data. This has the significant benefits that the kernel is simplified, and that all memory is directly accounted to components. This is safe [7], and does *not* require trusted user-level components; the kernel ensures that user-level cannot access this memory, and that the kernel is memory safe (*i.e.* all pointers reference valid memory of the proper type). Toward this, each frame is *typed* and can be either user-level virtual memory, kernel memory used for kernel data-structures (including RESTBLs and threads), or untyped memory which can be retyped to and from the other types. PGTBLs hold not only references to user virtual memory, but also untyped memory. In the latter case, the present bit is not set in the page-table entry (PTE), instead one of the spare flag bits is used to denote the untyped memory. At boot, all frames not used for the initial kernel image are mapped into the initial `boot` component's PGTBL as untyped memory. Kernel memory is tracked through component's KOTBLs and backs all of the previously described kernel objects.

B. SPECK *Functionality and Operations*

All of a component's resources are accessed through one of the two namespaces associated with the component: virtual memory which is accessed through PGTBLs (*e.g.* via hardware paging accelerators), and kernel objects whose operations are accessed via a system call to the kernel. Fast-path operations are performed on objects indexed into the KOTBL. The following three sets of operations make no modifications to shared kernel cache-lines, and the majority of components have access only to them. This is the foundations for scalable predictability. These operations include:

- `sinv`(*args,...*) – A synchronous invocation from the current component, to another specified by the SINV resource. The previous component, and the return stack/instruction pointers are tracked in the invocation stack of the current thread. This is the main method of inter-component coordination and is paired with `sret`(*retvals,...*) that returns to the calling component. Invocations are tracked on the thread's invocation stack, and returning simply pops a component off. The `sinv` and `sret` pair forms a round-trip synchronous IPC between two components. Figure 1 shows how a `sinv` operation from a client to a manager is tracked in a thread's invocation stack.
- `asnd`() and `arcv`() - ASND objects send an asynchronous notification to an associated ARCV resource. A thread bound to an ARCV resource can block waiting for the notification. Each ASND and ARCV resource is bound to a specific core, and can only be used on that core. This enables partitioning algorithms to restrict the cores on which threads are activated. If send and receive are on different cores, then IPIs are used for the notification. This is the main mechanism for asynchronous inter-core communication, e.g. event notification.

• `dispatch`() – Scheduling policies in SPECK are defined by user-level schedulers. Thread dispatch operations are used by scheduling components to switch between two threads. If a thread resource is indexed in the RESTBL, this operation dispatches that thread. This is used only by schedulers that have been given access to the thread. The referenced thread resource can only be dispatched to on the owner core, thus enabling components to maintain strict thread partitioning.

Operations on higher-order resource tables. The previous operations are the common case, and most components will only ever use them. Components with a reference to a RESTBL in a slot within their own RESTBL have the opportunity to modify and add to that RESTBL (*e.g.* see the manager, C_m, in Figure 1). To create, delete, and duplicate kernel objects within this nested RESTBL, the following operations are provided to resource management components.

• `activate`(*rtid, rid, type,...*) – Activate a resource of a given type at a *rtid* × *rid* (that is, at a specific *rid* indexed into the RESTBL at index *rtid*). For example, a memory mapping manager component (*e.g.* C_m) activates a page as part of the page-table of a client component. This is used to allocate each resource type, and the arguments include type-specific initialization information that often include references to other resources used in the construction of the resource. When the object being activated requires significant memory resources beyond what the slot can provide, the index to kernel memory to be used to back the object is provided. For example, when a thread is activated, the backing memory is provided as an argument in the form of a kernel memory object.

• `deactivate`(*rtid, rid*) – Deactivate an already activated resource at *rtid* × *rid*. Any subsequent accesses to the slot in the RESTBL the resource was at will return failure, unless later reactivated.

• `copy`(*rtidto, ridto, rtidfrom, ridfrom*) – Duplicate a resource at *rtidfrom* × *ridfrom* to *rtidto* × *ridto*. This aliasing operation enables sharing between separate components (*e.g.* shared memory, or the ability to schedule the same thread as in [4]). For example, a memory manger component utilizes this operation to copy a memory mapping to construct shared memory for two components in different address spaces.

The kernel's RESTBLs are manually created using the following operations that use untyped memory.

• `cons`(*rtidto, ridto, rtidfrom*) – This function connects nodes in a resource table by extending a RESTBL node (*rtidto*) at level n at index *ridto* with the RESTBL node at *rtidfrom* at level $n + 1$. For example, this operation is used to connect one level of page table into the next level, and to comparably extend KOTBLs. The same RESTBL node can be aliased into multiple resource tables, enabling, where appropriate, efficient coarse-grained sharing. `decons`(*rtid, rid, rtidparent*) provides an opposite operation in which does of a RESTBL are disconnected.

• `retype`(*type, rtidto, ridto, rtidfrom, ridfrom*) – Each of the system's physical frames can be exactly one type, and can only be used in positions in the API that use resources of that type. Frames can be retyped if they are no longer used in any data-structures. Each frame starts as untyped memory, which is the only type that can be converted to the other types, and vice-versa. This is used by memory manager component to manage physical memory frames.

A final operation is required but rarely used.

• `introspect`(*rtid, rid*) – This operation is used to enable a component to gather information about the RESTBLs it has access to. This functionality is rarely used, but is important in cases when a manager crashes that is in charge of constructing kernel RESTBLs. A fault handling component can uses this operation to clean up the state by introspecting on the crashed component's RESTBLs, and `deactivate`ing/`decons`ing appropriately to reclaim the memory.

C. Fine-grained Kernel Synchronization

SPECK is designed to avoid the coherency costs imposed by locks in the kernel. However, without the consistency guarantees provided by lock's mutual exclusion, kernel data-structures must be synchonized using different mechanisms. All kernel data-structures aside from RESTBLs either fit into a RESTBL slot (*i.e.* they're embedded into the leaf of the RESTBL) – which are not larger than a cache-line, or are larger but core-local (*e.g.* threads). Thus, only slot access and modification requires synchronization. Due to quiescence-based reclamation of kernel structures (Section IV-E), RESTBL lookups do not require synchronization. In fact, after most kernel objects are created in a KOTBL slot (SINV, SRET, ASND, ARCV, and components), they are immutable, thus requiring synchronization only on `activate` and `deactivate`.

To provide activation and deactivation that are synchronized between cores, each cache-line in the leaves of the KOTBL includes an allocation bitmap, and a size field. Each of these cache-lines is broken into between 1 and 4 slots, depending on the value of the size field. Each slot additionally has a header with the type of kernel object held in the slot, and quiescence information. The allocation bitmap includes four bits that are zero if the corresponding slot is free, and one otherwise. Importantly, both this bitmap and the size field are in a single word, enabling atomic modification of them. Thus activation updates the allocation bitmap and size (size can only be changed if there are no slots allocated), initializes the kernel object, and sets the type of the slot. When a slot is deactivated, these operations are reversed. However, the current time of deactivation is recorded (see Section IV-E), and a subsequent activation of that slot must be delayed while any pending references to it exist (*i.e.* until the system has quiesced). If kernel objects grow beyond a cache-line, or cache-line sizes changed (on different architectures), the size and number of slots in these leaf items must be adjusted.

Threads are relatively large structures, but are still referenced through KOTBLs. The slot within the KOTBLs is managed identically to other kernel objects, but contains both a pointer to the thread structure (currently a page), and the core on which that thread can be accessed. Thus, synchronization of the thread structure is not required as it is accessed on a single core, and the kernel is non-preemptive. A thread contains a reference count as it might be accessed by multiple components, enabling multiple schedulers the ability

to context switch to, or away from the thread. This enables hierarchical user-level scheduling HIRES [4].

Like threads, the nodes that constitute the various levels of the RESTBLs are also referenced by slots in the KOTBL. `cons` simply attaches these nodes together to form RESTBLs, while `decons` breaks them apart. When a RESTBL node of level $n+1$ is `cons`ed to one at level n, the corresponding RESTBL is expanded.

This trivial support for RESTBL construction is sufficient for most systems. However, if the system wishes to efficiently alias part of the address space of a RESTBL with another, the second level in a resource table (*e.g.* PTE) should be referenced by two separate resource tables. Though not required by POSIX, this enables the controlled sharing of namespaces between cores as in shares and address ranges in Corey [10], and mutable protection domains in COMPOSITE [11]. Support for this requires that the RESTBL node's slot include a reference count for the number of aliases. Importantly, only a component that is trusted by the component being modified can perform these operations (as it has the proper higher-order RESTBL), so it can mitigate the impacts of the worst-case scalability problems, often by only performing the operations only on isolated cores.

D. Component Control over Scalability Properties

As all policy for resource and isolation management is exported to user-level components in COMPOSITE, it is required that control over the scalability properties of the kernel also be in component control with SPECK. A few invariants provide this control: (1) Components with no RESTBL nodes objects in their KOTBL *cannot make any cache-line modifications* that can be shared between cores. This represents the majority of components aside from the lowest-level, trusted management components. (2) Manager components that have access to higher-order RESTBLs mediate modifications to the RESTBL structures, or even delegate to other components by giving them access to subsets of the higher-order RESTBLs. (3) The mapping between the namespace that is used to index into higher-order RESTBLs explicitly maps to the slot's cache-lines. Thus managers can explicitly avoid false sharing. Specifically, SPECK guarantees that when indexing slots in a KOTBL, any indices with identical values for $\lfloor rid/4 \rfloor$ (given four slots per cache-line), share a cache-line. Manager components can avoid scalability bottlenecks by simply ensuring that any operations that modify a cache line on a specific core never use slots on a cache-line for another core, *i.e.* by striping the resource namespace across cores.

Cores in SPECK are tagged as one of two types: real-time or best-effort. A flag on the kernel stack (recall SPECK has single kernel stack per-core) is used to decide the type of the current core, which is used to limit access of best-effort cores to avoid interference: the data structures in the kernel used by real-time tasks cannot be modified by best-effort cores. Operations that might negatively impact kernel response-time (*e.g.* `cons` with RESTBL sub-trie aliasing) for a core handling real-time computation, must be exported to a best-effort core.

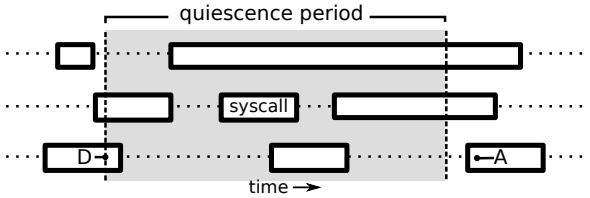

Fig. 2: Kernel object liveness example. Bars represent kernel system call execution, and the three timelines denote three cores. A kernel object is `deactivated` at D. The dashed (lightly shaded) region denotes the quiescence period (WCET_k+WCET_{cas}). Only those kernel invocations that start *before* `deactivate` can maintain references to the object (*e.g.* the first bar on the second core). A subsequent `activate` (denoted A) of the same slot must await completion of the quiescence period.

E. Quiescence-based Kernel Object Liveness Tracking

One of the most challenging aspects of implementing a kernel with no locks is determining when a kernel object's memory can be reused after being freed. Parallel references to an object must finish before the object is actually freed and can be reused. Were this not the case, an object that is freed and zeroed on one core, while being accessed on another, could cause faults as those accesses de-reference the newly zeroed values. Figure 2 depicts a timeline including kernel object `deactivation`.

Instead of using locks to solve this problem (by ensuring mutual exclusion, but causing coherency overheads), SPECK uses a *quiescence period*, the end of which determines when no references at the start of the period could persist. The important question in SPECK is what is an appropriate quiescence period? Here SPECK's non-preemptivity is useful: *no reference to a kernel object that has been removed from, or deactivated in* RESTBL*s can persist for longer than a kernel Worst-Case Execution Time (WCET_k).* Any references are released on return to user-level. When a kernel object is `deactivated`, the current time-stamp is taken and stored in the header of the object. If deactivate is called at time t, the kernel object can be reused at t+WCET_k+WCET_{cas} where WCET_{cas} is the worst-case latency for a compare and swap (`cas`) instruction that is used to mark the slot as inaccessible.

The LIVENESS TABLE is a separate data-structure that is used to facilitate tracking kernel object liveness. When deactivated, an offset into the LIVENESS TABLE is placed into the object's header. This level of indirection through the LIVENESS TABLE enables multiple kernel objects to share the same lifetime tracking information.

LIVENESS TABLE quiescence tracking. When `deactivate` is called on a kernel object, a index into the LIVENESS TABLE is provided. Each entry in the LIVENESS TABLE has a time-stamp value. This value in the LIVENESS TABLE entry associated with the kernel object is set to the deactivation time. There is no explicit call to "free" the slot in the KOTBL. Instead, a subsequent `activate` call is successful only if the quiescence period has passed from the liveness entry's time-stamp.

Time-stamps are taken using the `rdtsc` instruction on x86 processors. Modern x86 (and many other) processors support *invariant time stamp counters* that proceed at a constant rate, independent of frequency scaling or processor sleeping. This

enables a *global* notion of time across cores. For architectures without such support, emulated approaches, e.g. periodically updating a shared global variable on a single core, can be used to get a representation of time at coarse granularity. This hardware support enables quiescence periods to be globally applicable across all cores.

Quiescence-based TLB coherence. TLB shootdown is a means to achieve coherency between TLB caches on remote cores, and updated page-tables. We assume the shootdown flushes TLB entries of all address spaces on each core, for simplicity. A page-table frame mapping that is updated or removed typically requires an IPI be sent to all other cores so that they can flush their TLB caches. This is an extremely expensive operation that does not scale. SPECK introduces a second quiescence period: the latency between when a PGTBL is modified, and when all other cores have flushed their TLBs. Toward this, each core maintains a private time-stamp that tracks the last time it flushed its own TLB during a thread switch. Quiescence is reached when the time-stamp for each core is higher than that value.

We have implemented management components that use two mechanisms to unmap memory from page-tables. The first uses predictable, periodic TLB flushes to implement SPECK's worst-case scalable, TLB coherence protocol. Each core periodically executes a high priority thread in a dedicated TLB flush component that immediate blocks and forces a TLB flush. The second, traditional protocol uses synchronous TLB coherence provided by manually using ASND objects to trigger the TLB flushing threads (waiting on ARCVs) on each core. This approach is *not* scalable, but provides functionality that is backwards compatible with POSIX `munmap`.

LIVENESS TABLE TLB quiescence tracking. Tracking quiescence on virtual memory (VM) mappings is more difficult than doing so on slots in a KOTBL. When a VM mapping is removed from a PGTBL, it is replaced with an offset into the LIVENESS TABLE. At the same time (atomically), the *present* bit is unset, and one of the free bits in the PTE is used to denote pending *quiescence*. A comparable operation is performed when a `decons` is performed on non-leaf PGTBL nodes. These entries can only be reused when TLB quiescence has been achieved. The tracking mechanism assumes an available user-definable bit in the page-table entry (*e.g.* one of the three free bits on x86). However, on architectures with no such free bits (e.g. ARM), an additional resource table must be used to track quiescence states instead.

F. Scalable Physical Memory Management

Kernels must track physical memory so that it can be put to various uses. For example, `rmap` in Linux enables mapping from a physical frame, to the PTEs it is mapped into, and `struct page` contains a count of the number of mappings to the frame. As this count doesn't scale, RadixVM [12] introduces Refcache. This per-core cache provides scalability in the average case, but in the worst-case, still requires shared modifications.

Similarly, SPECK requires tracking of physical frames: if a frame is mapped into user-level, it cannot be used to back kernel data-structures, and vice-versa. This is the foundation for user-level management of kernel memory, and enables the removal of all memory management in the kernel. In other words, the `retype` operation can only be conducted on a frame when it is no longer used in its previous type's capacity.

Frames can be in one of three types: *Untyped*, in which case they are referred to in a single (non-*present*) PGTBL entry. Similar to the quiescence bit used for VM mappings, a free bit in the PGTBL entry is used to annotate such untyped memory. Thus, we assume a spare bit in the page-table entries. If this assumption is not valid, a separate RESTBL can be used to track such frames. *User virtual memory*, in which case a counter tracks how many times the frame is mapped into PGTBLs (*i.e.* with the *present* bit set). *Kernel memory*, in which case the frame is used in a kernel data-structure and referenced in a slot in a KOTBL. Retyping enables memory to be changed from one type to another.

A single counter and type variable per page is problematic due to scalability concerns. Clustered objects [13] enable distributed counters, local to each core, to be access in the common case, and only when a total count is required, is an expensive gather operation performed. Unfortunately, a straight-forward application of this technique requires too much memory. On a M-core machine with N physical frames, for example, $N \times M$ counters are required. Instead, SPECK takes advantage of the fact that user-level components can intelligently manage and partition frames for different uses, and maintains the counts for *bundles* of frames. Each bundle is a set of B adjacent frames (in the current implementation, $B = 32$), and the type of *all* frames in a bundle is the same. `retype`, then, retypes bundles of frames. Thus, the memory required for the counters is $M \times N/B$. The choice of B trades the granularity of frame typing for memory usage.

The per-bundle counters now track all frames in that bundle. If the pages are user virtual memory, the sum of the bundle's counters track the number of mappings of all pages in the bundle. If the bundle is typed to be kernel memory, then the counters hold the number of pages used in kernel data-structures. A *retype* operation is only possible when the sum of the count in all core's counters for a bundle is zero. A subtle additional constraint is that when retyping from user-level virtual memory to untyped, TLB quiescence is also required, and is tracked similar to how quiescence is tracked for user memory mappings.

The retyping operation is not scalable as it must read cache-lines modified on other cores. Management components still maintain control over the response time of the kernel: retyping should be rare (or non-existent on statically-configured systems), and can be performed on isolated cores. This implies that a number of cores should be devoted to such non-scalable operations, and the rest (the hard real-time cores) will benefit from low response times.

G. Scalable Asynchronous Notifications

The `asnd` and `arcv` operations are used for asynchronous notifications and are the primary mechanism for sending inter-core, preemptive notifications. Hardware Inter-Processor

Interrupts (IPIs) provide asynchronous inter-core notifications (via local-APICs on x86), and are used by these operations.

x86 processors do *not* provide message passing; data to accompany the remote interrupt is passed via shared memory. Thus, many operating systems (including Linux and Fiasco) use a lock-protected *queue per core* to pass messages with IPIs. To avoid the contention of $M-1$ cores modifying a core's message queue, SPECK uses *pair-wise, single-producer, single-consumer, wait-free ring-buffers* to deliver messages. Each queue has a head and tail offset, and a number of cache-line aligned data items. This provides predictable overheads as the sending core, and receiving core transfer three cache-lines (two written by the sender – head and data, one by the receiver – tail). The only data passed in the buffer is that which is necessary to identify the designated ARCV kernel object. No arguments are sent, as queuing them could require possibly unbounded memory.

The cost for this predictable overhead for message passing is in memory consumption for the the pair-wise queues. Though M^2 queues are necessary, they are each relatively small given the paucity of data passed.

V. SYSTEM GUARANTEES

SPECK's design meets the goals laid out in Section III. Here we emphasize a number of guarantees that are necessary for the correctness of SPECK.

Synchronized modification of kernel data-structures. Atomic instructions are used to modify individual kernel objects, and the activation/deactivation protocols (discussed in Section IV-C) are used for modifications larger than a word. All other accesses are to core-local data-structures (Sections IV-C, IV-F, and IV-E).

Component-controlled interference and latency bounds. Modifications to shared cache-lines are only allowed within components that have access to higher-order RESTBLs. Management components do not give out this capability (Sections IV-C and IV-A). Thus kernel contention is mediated only by trusted components capable of carefully managing resources. The explicit mapping of the resource namespace to cache-lines enables these components to avoid false-sharing (Section IV-D). The strongest guarantees are made by manager components by ensuring that cores with low response-time requirements don't modify *any* kernel cache-lines, instead outsourcing such modifications to other cores.

Kernel execution has a bounded latency. All paths in the SPECK kernel have bounded execution time. The entire kernel is lock-less. There are very few loops in the kernel, and all of them are bounded. RESTBLs all have a fixed depth, and lookup is constant-time. The longest loop in the kernel zeros a page to be used for a kernel data-structure, or when retyping it to be user-level-accessible (Section IV-F). Results on our architecture show that this page zeroing operation has acceptable latency. Were it not, we'd need to include preemption points [7]. There are no spin-locks in the kernel, and when contention between cores is detected (with a failed `cas` instruction), the kernel API returns a failure (Section IV-B and IV-C) instead of re-trying the operation. Thus, kernel execution time is trivially bounded on cores that only read kernel data-structures (*i.e.* hard real-time cores). It is bounded on cores that update data-structures by hardware-imposed coherency latencies.

Kernel memory is inaccessible to user-level. The retyping facilities of SPECK (Section IV-F) guarantee that each frame can be either typed as kernel memory (along with its bundle of frames), *or* as user virtual memory, but not both. This is tracked with distributed counters and is integrated with quiescence.

Kernel type and memory safety. SPECK guarantees that no kernel object can be deallocated or have its type changed while references to it exist. Key to maintaining this invariant is ensuring that a quiescence period has elapsed since the object is made inaccessible. Section IV-E discussed how these periods are maintained based on 1) the kernel WCET, and 2) on scheduled TLB flushes. Rare kernel data-structures that are aliased use specialized reference counting schemes (Section IV-C).

No kernel memory leaks. SPECK ensures that all kernel memory is referenced transitively from the initial component's RESTBL – *i.e.* that it isn't leaked and unrecoverable. This is provided by a combination of an API that only moves memory between structures (Section IV-B), and never removes references entirely. When a RESTBL node is being `deactivated`, it is scanned to ensure that it has no references to other kernel structures. Given these two guarantees, every kernel object should be reachable from the `boot` component's RESTBL.

Kernel support for user-level resource management. Scheduling is performed in user-level components using the direct dispatching support for thread kernel objects, and asynchronous end-points for interrupts (as in [3]). Memory mapping/kernel object management components have references to the PGTBLs/KOTBLs of the components they manage.

VI. EVALUATION

To evaluate and demonstrate the scalability and predictability of SPECK, we conduct μ-benchmarks of various operations in SPECK, and compare against Fiasco.OC (henceforth referred to as Fiasco), release-2014022815. Fiasco is from the L4 μ-kernel family. It is a fully preemptive kernel with low response times [8], which is designed for hard real-time computation. All experiments are run on the same 4 socket, 40 core, hardware from Section II. To avoid interference, core #40 is dedicated to handle non-benchmark workloads.

Measured Worst-Case Execution Time. In this section we try to evaluate the WCET of SPECK operations as well as average execution time. As mentioned earlier, all paths in SPECK have bounded length and therefore bounded WCET. Statically evaluating WCET itself is a complicated process, especially considering multi-core factors including cache coherency. To evaluate worst-case latencies, this section instead focuses on empirical measurements. Thus all reported "worst-case" measurements must be interpreted as "worst-case measured latency". Due to the difficulty of measuring worst-case latencies with cache-coherency effects, we use two empirical methods to measure latencies: 1) measured

execution time while flushing all caches, and 2) execution time with maximal cache coherency traffic due to modifications to shared cache-lines.

With the cache flushing case, a complete cache+TLB flush is done before measuring an operation. This emphasizes the impact of DRAM and cache latency on execution time. However, the high overhead of the flush operation itself reduces the degree of cache contention significantly (i.e., causing a very low cache contention *rate*). This case approximates WCET on a single core platform as cache-coherency has very limited impact. On the other hand, to emphasize the execution impact of cache coherency traffic, each core does measurements in a tight loop, therefore maximally contending on the shared cache-line. Neither of these techniques computes the absolute theoretical WCET. However, we believe the measured WCETs in both cases are reasonably representative and approximate the theoretical WCET.

A. Hard Real-Time Subsystem Scalability and Predictability

First, we implement a manager component that provides an execution environment for scalable predictability for HRT computation. It creates components such that all operations they perform are not on a higher-order RESTBLs, and it performs `activation/deactivation` to create/delete communication channels (capabilities, here) and map/unmap pages. It explicitly avoids shared cache-line contention by carefully using the RESTBL namespace (Section IV-D). These operations are tested in a management component (task) in both SPECK and Fiasco.

To demonstrate scalable predictability, all operations are measured under four different core counts. The minimal number of sockets is always used. In all cases, each operation is measured 1 million times in a tight loop; average cost, maximum cost and standard deviation are measured and reported (in CPU cycles). Irrelevant interrupts (*e.g.* timer interrupts and System Management Interrupts) are filtered out as they must be accounted for separately. We do *not* flush caches before each measurement. Flushing all caches is an extremely expensive operation that hides the scalability aspects of the test. We will investigate this in Section VI-B.

Core-local IPC. As one of the most common operations in μ-kernel based systems, IPC performance is critical in any μ-kernel, and we measure its round-trip cost between two protection domains. To assess scalability we measure IPC costs with 4 core counts (*i.e.* threads on that many cores IPC between the same pair of protection domains).

In SPECK, the IPC is via a SINV/SRET pair between two components. In Fiasco, two threads (with same priority, and from different tasks) on each core are used to measure IPC cost. The client thread invokes ipc_call and the server thread invokes ipc_reply_and_wait.

Discussion. The results of IPC benchmark are shown in Figure 3. Round-trip IPC cost is reported in cycles. SPECK achieves perfect scalability for both average and maximum overhead, and outperforms Fiasco in both cases. Fiasco maintains consistent, scalable average overhead on different core counts. However the max overhead increases to more than $11K$ cycles on 39 cores.

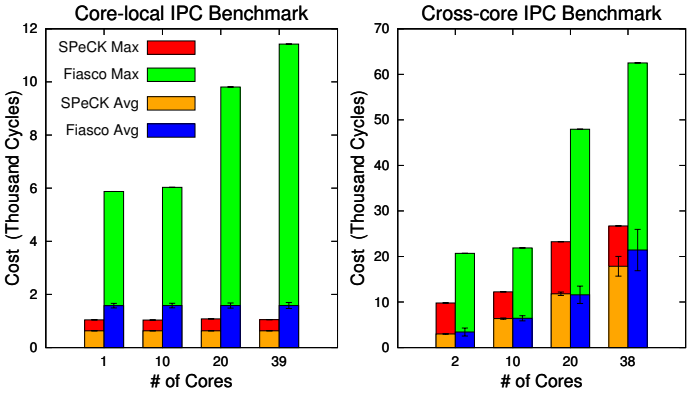

Fig. 3: IPC Benchmark Results

Cross-core communication benchmark. Cross-core communication is common and useful in multi-core platforms. As discussed in IV-B, ASND and ARCV provides asynchronous communication channels which is used for cross-core communication in SPECK via IPIs (Section IV-G). In Fiasco, the same IPC API as before is used, except one-way sends and one-way receives are used to mimic notification, rather than synchronous behavior. IPIs are used for event notification by both mechanisms.

The benchmark measures pair-wise cross-core communication costs. Each client thread on a core communicates with the dedicated server thread on the paired core. For configurations with 2 or 10 cores, all communication is intra-socket; for cases with 20 or 38 cores, all communication is inter-socket.

Discussion. Figure 3 includes the cross-core communication costs on both systems. Because hardware IPI generates contention on the APIC bus, higher overhead are observed for both systems when more cores are involved. SPECK and Fiasco have comparable average costs, while SPECK has relatively lower max overhead (more than 50% on 38 cores) and smaller standard deviation. The software overhead in SPECK remains constant across the core numbers here, so the overhead is hardware imposed. Fiasco sends IPIs in both directions for coordination reasons. Though these are ignored on the sender, they create more traffic on the APIC network and QPI links, which might account for some of the increase in worst-case overhead.

Memory mapping / unmapping. SPECK provides scalable and efficient memory mapping / unmapping mechanisms, which enables virtual address mappings to be built dynamically even for hard real-time computation. In this benchmark we measure the costs of memory mapping and unmapping with different numbers of cores.

For all configurations, only one thread conducts map / unmap on each core. However, all threads are in the same address space to emulate a manager component. Each thread maps a physical page to a dedicated, distinct region in its virtual address space. In SPECK, we explicitly avoid false-sharing, *i.e.* different cores modify different cache-lines in the page table.

Capability activation / deactivation. Similar to memory mapping and unmapping, SPECK allows scalable capability operations (*e.g.* activation and deactivation of SINV objects). The benchmark setup is also similar in both systems. Cache-

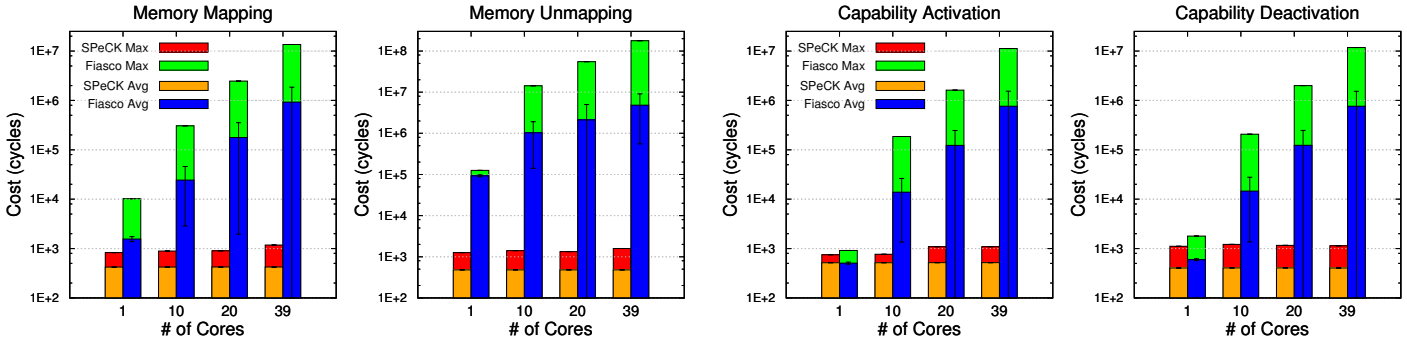

Fig. 4: Memory / Capability Operation Benchmark Results (in Log Scale)

line false-sharing is avoided in SPECK.

In Fiasco, system tasks of the L4 Runtime Environment interfered with execution, and manually re-prioritizing them did not have the desired impact. We manually modify the run-queue to remove them while the tests are running to avoid large perturbations in the worst-case costs.

Discussion. Results of memory and capability operations in Figure 4 show the benefits of the SPECK design: low average and maximum overhead, and small standard deviation across all operations. On the other hand, Fiasco suffers from a spin-based shared lock and unscalable TLB shootdown for memory unmapping, which result in extremely high overhead: millions of cycles on average, and hundreds of millions of cycles maximum. The high overheads of those operations make their use unrealistic in HRT computation. Even for non-RT applications, the overheads are a potential scalability bottleneck.

The cost of thread dispatch in SPECK is measured by two threads on the same core switching to each other. As thread objects are indexed through the KOTBL, and are core-local data-structures, thread dispatch cost in SPECK is very low – with average cost 463 cycles and maximum cost 528 cycles on 39 cores.

B. Response Time of Hard Real-Time Subsystems

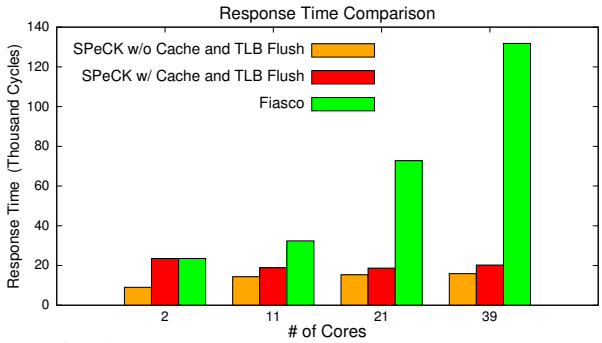

Fig. 5: Measured Worst-case Interrupt Response Time

In this subsection we evaluate the interrupt response time of SPECK. Based on the non-preemptive design of SPECK kernel, the response time of the system is determined by the longest execution time in kernel. We evaluate the impact of each of the operations in Section VI-A on interrupt response time. The response time measured in this section is the response time of Inter-Processor Interrupts (IPIs), which approximates I/O response time in a reasonable way. Like L4 and other μ-kernels, in COMPOSITE IRQs are mapped into user-level and delivered to handler components.

To measure the response time, a sender core sends IPIs using the system's cross-core communication mechanism to a high priority thread on a receiver core; low priority threads on all cores aside from the senders perform an operation to provide interference. The response time of the IPI to the high priority thread is measured on the sender core (a single word is written to shared memory when the notification is received).

At core counts $\geq$5, Fiasco deadlocks. This is due to a bug in the helping lock implementation – which uses thread migration – that leaves the state of the lock owner variables inconsistent. To avoid this deadlock, we prevent the low-priority thread on the receiving core from migrating (migration happens when preempted while holding a lock). This should, if anything, avoid the overheads of migration which could interfere with high priority execution. Thus we believe our modifications do not penalize Fiasco.

Discussion. Results of the worst-case interrupt response time are reported in Fig 5. Though Fiasco is a preemptive kernel, interrupts are briefly disabled in the lock implementation, and as the lock cache-line bounces between cores, the response time will rise. Thus, with more cores actively performing operations in kernel, the response time of Fiasco is impacted and increased to above $130K$ cycles with 39 cores.

Two cases of response time in SPECK are measured: with and without cache+TLB flushing. Results in the previous subsection show that scale in SPECK does not impact execution time in kernel (which is response time because of the non-preemptive kernel design). As shown in Fig 5, the response time in SPECK w/o cache+TLB flushing is low and consistent with more than 10 cores. To evaluate the impact of caches on the SPECK results, the same tests are executed with cache+TLB flushed (including entries with global bits set) on the receiver core. Due to the high overhead of cache flushing (millions of cycles on our hardware), each case is measured 10 thousand times. Results show that the response time is below $24K$ cycles, which is the worst-case response time in SPECK. Thus, even at 40 cores, the response time of SPECK is dominated by the same caching factors that equally impact the system on a single core. This is exactly the scalable predictability that is the main goal of SPECK.

An observation is, with cache+TLB flushing, the worst-case response time is the 2-core case. With more cores, the response time is actually lower. We believe this is because after the cache / TLB flushing, more cores executing will help warm up the shared cache faster (*e.g.* last-level shared cache), which lowers response time.

Operation	Cost (Cycles)
IPC	6024
Capability Activation	8079
Capability Deactivation	7464
Memory Map	10554
Memory Unmap	7770
KOTBL CONS	7290
KOTBL DECONS	9609
KOTBL Activation	14148
KOTBL Deactivation	17463
Memory Retype (to User)	8586
Memory Retype (to Frame)	13497

TABLE I: Measured Max Cost with Cache and TLB Flushing

Table I summarizes the maximum costs of each SPECK operation w/ cache+TLB flushing. Similarly, cache and TLB flushing is done before each measurement; each operation is measured $10K$ times. However, the cache / TLB flushing overhead itself limits contention from multiple cores as the flushing operation causes a large gap between two operations on a core. Thus the chance of contention are very low. Because of this, with cache+TLB flushing, we only measure and report the maximum overhead of each operation on single core. The worst-case overhead of HRT allowed operations is below $11K$ cycles (memory mapping).

Please note that Table I also includes operations that a component manager would not perform on a core devoted to HRT computations.

C. SPECK *Kernel Quiescence Period*

In this subsection, to evaluate worst-case kernel execution time, we relax the constraint of no false-sharing. Instead, we pursue maximum contention to measure the worst-case in all scenarios, including non-HRT cores with unconstrained operations. The resulting number is required to determine the kernel quiescence to use for liveness calculations.

For the completeness of the SPECK specification, all operations in SPECK are measured and reported in this subsection, including the ones not allowed in the HRT subsystem. Table II in Appendix show the results of different operations with contention (*e.g.* false sharing). Namely, the operations are `cons/decons` of KOTBL, `activate/deactivate` of KOTBL/PGTBL, and memory `retype`. In addition, memory mapping/unmapping and capability activation/deactivation are re-measured and reported because contention and false-sharing are allowed in this case.

From the results in Table II, KOTBL `cons/decons` operations are the most expensive operations, with average of $63K$ cycles and max of $277K$ cycles on 39 cores. This is caused by 1) shared reference counter and 2) high contention rate (if no memory is actually allocated after retyping, the next retyping can happen immediately with quiescence). If the worst-case overhead of these operations is not acceptable, RESTBL aliasing can be disabled as it is not an essential feature. When multiple reference to the same KOTBL object are not allowed, the contention of `cons/decons` is eliminated as well. The next most expensive operations are memory `retype`, which have average overhead $< 24K$ cycles, and max overhead $< 100K$ cycles on 39 cores. Please note that, as shown in Table I, the max overhead with cache and TLB flushing is $18K$ cycles (KOTBL `deactivate`), which doesn't contribute to worst-case execution time when contention dominates the overhead.

Though SPECK does include some operations that are *not* scalable, the capability for management components to explicitly prevent those operations from occuring on cores devoted to HRT computation demonstrates the utility of enabling an explicit mapping between the kernel namespace, and contention.

VII. RELATED WORK

Predictable shared resource mediation. Significant effort has been placed into predictable sharing of resources between processors (*e.g.* surveyed in [14]) and using improved sharing-aware analysis (*e.g.* surveyed in [15]). To avoid the overhead for locking that implies the coherency overhead from Section II, SPECK takes an orthogonal approach by preventing any contention in the case of kernel access by hard real-time tasks, and enabling contention to be on at the finest-granularity and mediated directly with atomic instructions to avoid locking.

System structure for scalability. Multikernels [16] propose a share-nothing approach to kernel design. Each core executes a separate kernel image, and manages disjoint memory sets, thus coherency contention is impossible, and message passing is used for inter-core coordination. Although no-sharing kernel design alleviates high overhead caused by cache coherency, it limits the flexibility and efficiency of kernel data-structure sharing. In addition, as the # of cores goes up, the # of messages, and the overhead of message passing / handling can also increase. Shared memory is still the fastest mechanism for inter-core coordination, and SPECK enables its use where appropriate, by enabling user-level components to define sharing policies (including share nothing if desired). Similarly, [17] specialize a core to perform global scheduling. Such a technique could be accomplished in SPECK using the asynchronous inter-core communication facilities. Corey [10] represents another system design for scalability that enables explicit control of mappings of namespace regions onto cores. This support can be emulated in SPECK by sharing select regions of RESTBLs between different components.

TLB coherency management. Previous systems have optimized the TLB shootdown procedure when access rights of a page mapping are reduced. As shown by the Fiasco results, such a shootdown is not scalable. One approach [6] maintains per-core bookkeeping to prevent some unnecessary TLB shootdowns. RadixVM [12] uses separate page-tables for each core to separately track accessed bits to determine if a TLB invalidation IPI must be sent. Unfortunately, these systems do not change the worst case: unscalable TLB shootdown. SPECK enables asynchronous TLB coherence with delayed virtual memory reuse, with a latency bounded by periodically scheduled TLB flushes, thus enabling the avoidance of TLB shootdowns even in the worst-case.

Increased complexity in user-level managers. The non-preemptive design of SPECK simplifies kernel design and enables scalable quiescence-based liveness tracking; higher-order resource tables enable managers to control parallel modifications to shared kernel objects. However, the trade-off for this approach is the increased complexity of user-level

Operation	Number of Cores											
	1			10			20			39		
KOTBL CONS	*332*	(2.00)	**441**	*1688*	(769.23)	**9669**	*16419*	(9732.71)	**84162**	*62518*	(34686.29)	**242928**
KOTBL DECONS	*358*	(2.00)	**1257**	*1381*	(691.16)	**7428**	*11361*	(9732.71)	**77523**	*48878*	(34591.28)	**276255**
KOTBL Activation	*1394*	(3.16)	**2355**	*1499*	(11.66)	**8349**	*1735*	(246.75)	**11352**	*2426*	(366.62)	**13548**
KOTBL Deactivation	*3494*	(9.70)	**6258**	*3580*	(19.87)	**9384**	*4045*	(228.50)	**11355**	*4501*	(391.49)	**14119**
Memory Retype (to User)	*490*	(2.24)	**1113**	*1516*	(131.64)	**8430**	*5118*	(4413.97)	**36580**	*19066*	(10497.42)	**87411**
Memory Retype (to Frame)	*1877*	(2.00)	**2385**	*1975*	(1131.50)	**9060**	*7432*	(5326.97)	**44166**	*23546*	(13563.87)	**96954**
Memory Map	*422*	(2.83)	**828**	*563*	(144.88)	**2160**	*1492*	(2053.86)	**25929**	*2749*	(2954.37)	**63219**
Memory Unmap	*481*	(2.24)	**1263**	*600*	(158.36)	**1641**	*1305*	(1705.90)	**25260**	*3248*	(3819.21)	**67092**
Capability Activation	*516*	(2.83)	**750**	*627*	(252.11)	**2766**	*1249*	(1008.75)	**15969**	*1480*	(1203.76)	**30570**
Capability Deactivation	*404*	(1.73)	**1113**	*637*	(204.93)	**2130**	*673*	(1044.40)	**20265**	*780*	(1144.15)	**44745**

TABLE II: SPECK Operation Benchmark – *Average Cost* (Standard Deviation) **Max Cost**

management components for tracking the RESTBL namespace, and for tracking when the slots for `deactivated` objects can be reused (*i.e.* they must track quiescence). Future work includes the creation of library support that simplifies much of this management burden for user-level policy components.

VIII. CONCLUSIONS

SPECK uses the co-design of the kernel interface, data-structures, and contention management mechanisms to yield a system with scalable predictability for cores with hard real-time tasks. Though the kernel is non-preemptive, this is used at scale to provide quiescence to enable common kernel execution paths with no modifications to shared cache lines. Though some kernel operations do perform cache-line modifications, they are controlled using higher-order RESTBLs for access control. Components with access have an explicit mapping from the kernel RESTBL namespace, to cache-lines, and are able to completely avoid inter-core contention. Consistent with COMPOSITE, SPECK provides fine-grained isolation, and component-based definition of system resource management policies.

We show that SPECK performs fundamental system management capabilities such as communication, thread management, and memory mapping with both low average-case overheads, and worst-case overheads, even at high core counts. SPECK scales perfectly, and is the first system we know of that provides worst-case scalable TLB coherence. We find that SPECK's response times are determined more by the cache effects that dominate single-core latency, than by coherency effects, thus demonstrating scalable predictability.

REFERENCES

[1] H. Kim, D. deNiz, B. Andersson, M. Klein, O. Mutlu, and R. (Raj) Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS*, 2014.

[2] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *ECRTS*, 2013.

[3] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS*, 2008.

[4] ——, "HiRes: A system for predictable hierarchical resource management," in *RTAS*, 2011.

[5] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *SOSP*, 2013.

[6] V. Uhlig, "Scalability of microkernel-based systems," Ph.D. dissertation, University of Karlsruhe, Germany, 2005.

[7] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *SOSP*, 2013.

[8] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *RTSS*, 2002.

[9] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *WTEC*, 1994.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *OSDI*, 2008.

[11] G. Parmer and R. West, "Mutable protection domains: Adapting system fault isolation for reliability and efficiency," in *TSE*, 2012.

[12] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *EuroSys*, 2013.

[13] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," in *OSDI*, 1999.

[14] B. Brandenburg, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *ECRTS*, 2013.

[15] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," *RTSS*, 2010.

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

[17] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing," in *RTAS*, 2014.

APPENDIX

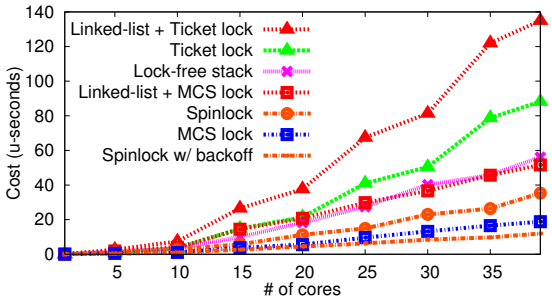

Fig. 6: Average Overhead of Synchronization Primitives

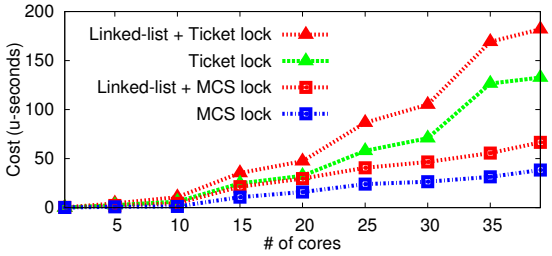

Fig. 7: Measured Max Overhead of Synchronization Primitives

Synchronization Primitive Scalability. Fig. 6 and 7 depict the average and measured worst-case overheads for different synchronization primitives running on the system detailed in Section II. Though some implementations do well on average, a rare view on these operations is with a worst-case perspective. The maximum values for even the scalable MCS locks show overheads close to 50μs. The use of such primitives can cause significant interference and negatively impact the latencies of hard real-time computations. The lock-free stack and two spin-locks (w/ and w/o back-off) are not shown in Fig. 7 as their measured maximum overheads are orders of magnitude higher due to their non-predictable nature.