Scalable Memory Reclamation for Multi-Core, Real-Time Systems

Yuxin Ren, Guyue Liu, Gabriel Parmer
The George Washington University
Washington, DC, USA
{ryx, guyue, gparmer}@gwu.edu

Björn Brandenburg
Max Planck Institute for Software Systems
Kaiserslautern, Germany
bbb@mpi-sws.org

***Abstract*—A core challenge in best utilizing an increasing number of cores in real-time systems is addressing the problem of efficient and predictable resource sharing. Traditional mechanisms for mutual exclusion, such as locks, limit parallelism due to serialized resource access. Relaxing mutual exclusion, reader-writer locks enable selective parallelism for a subset of accesses, but can suffer from increased implementation overheads. In all such implementations, the costs of cache-coherency alone can be prohibitive for an increasing number of cores.**

This paper investigates the use of techniques such as Read-Copy Update (RCU) to enable truly parallel access to data-structures. Such techniques optimize for data-structure read-paths, and can completely avoid stores to shared structures, thus avoiding cache-coherency overheads. We show that existing implementations of preemptive RCU aren't designed to provide real-time latencies, and require a potentially unbounded amount of dynamically allocated memory. Thus, we introduce two new implementations that are both predictable and efficient, and a matching analysis that establishes bounds on memory consumption. We additionally provide a schedulability analysis that demonstrates the effectiveness of scalable read-side operations, achieving consistently higher schedulability than existing techniques. We further apply the analysis to provide admission control for a soft real-time application to both achieve higher throughput than existing approaches (up to 40% higher) while limiting 99th percentile read-path latencies (4x lower than existing techniques).

I. Introduction

The increasing utility and availability of multi-core processors in embedded systems motivates efficiently and predictably harnessing their parallel computation capacity. Multi-core systems promise a decrease in size, weight, and power (SWaP) by effectively consolidating disparate functionalities onto a single processor, while possibly increasing a system's capabilities. However, the use of multi-core processors in real-time systems comes with a number of challenges, including contention in hardware resources such as caches, memory buses, and DRAM. In particular, prior work has devoted significant attention towards developing effective mechanisms and timing analyses for the controlled *sharing* of software data-structures in cache-coherent architectures. Such techniques are essential to effectively utilize multi-core systems. However, most of this research is based on locks that induce expensive cache-coherency traffic, and impose mutual exclusion that sequences parallel operations. To achieve high utilization with predictable behavior given an ever-increasing number of cores, techniques are necessary that *minimize cache-coherency operations and avoid sequencing otherwise parallel computation.*

In a parallel development, high-end computation in data-centers is seeing an increased need for predictable execution. A single request from a client often results in cascades of many sub-requests (often hundreds) [1]. The latency of the client's request is often dependent on some function of the *maximum* latency for any of the sub-requests. This has motivated work into reducing the *heavy tail* of latency in data-centers [2]. These systems already contain many cores (for example, HP Superdome systems with over 200 cores), often spread across multiple sockets. Thus, there is benefit to applying techniques that provide increased predictability while effectively harnessing the system's parallelism. From embedded systems up to the data-center, predictably utilizing parallel resources is a pervasive, important challenge.

This paper investigates real-time *Scalable Memory Reclamation* (SMR), which applies SMR techniques to real-time systems. SMR enables *unsynchronized* concurrent access of *readers* to data-structures. By eliding locks and other forms of synchronization, shared resource access requires no special treatment and is analytically an extension of sequential code. However, this lack of synchronization causes obvious challenges. Read-Copy-Update (RCU) [3] can be used for data-structure *updates* that are made by copying (a part of) the data-structure into newly allocated memory, updating the copy, and atomically replacing the old version with the new. This completely avoids synchronization in the read path, and minimizes it in the update path. Unfortunately, it presents a new problem: the memory for the old (parts of the) data-structure must eventually be reclaimed for reuse, but only after no reader is accessing it any longer. RCU *converts a scalability problem into a garbage collection problem.*

SMR techniques solve the garbage collection problem by conservatively ascertaining whether a memory node that has recently been disconnected from a data-structure is still *possibly* accessed by a concurrent thread via stale thread-local references. This is determined based on the concept of a *grace period* [3]. A grace period starting at time t_1, involves calculating – and often waiting for – a time t_2, where t_2 is a time when *certainly no references exist* to nodes removed from the data-structure before time t_1. A node made unreachable within the data-structure, and freed at time t_0 ($< t_1$) can only be reclaimed and reallocated after time t_2. SMR techniques calculate a *quiescence* point – the time when a grace period has elapsed. Before a quiescence

*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.

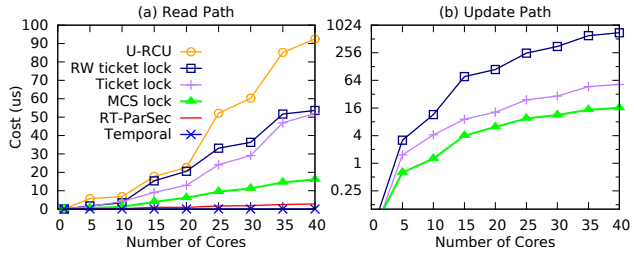

Fig. 1: 99th percentile measured cost of synchronization primitives

point is reached and previously freed memory is reclaimed, a potentially large amount of memory will accumulate. Clearly, to be practical in a real-time system, the maximum amount of accumulated garbage memory must be bounded a priori.

Contributions. This paper investigates three aspects of real-time SMR. First, we introduce two real-time SMR implementations and show that real-time SMR increases scalability and predictability by reducing worst-case blocking time. Second, we determine the timing properties of quiescence calculations, and integrate them into a response-time analysis of a hard real-time system. Third, we calculate memory bounds for the system and consider the trade-off between the overhead of frequent quiescence calculation, and the memory build up from infrequent quiescence calculation. With the goal of system schedulability, we introduce a quiescence frequency selection algorithm to minimize memory consumption.

Three SMR implementations are studied: (1) U-RCU [3] – a preemptive version of RCU quiescence detection[1], (2) RT-ParSec – a variant of ParSec [4] focused on scalability and adapted to real-time systems, and (3) Time-based quiescence – which has been used in a non-preemptive environment to achieve scalable predictability [5], and which we here adapt to preemptive environments. We investigate all three approaches as they each represent different trade-offs between the necessary hardware and system support, and the realized level of quiescence scalability.

In §VII, we evaluate these real-time SMR techniques in two environments. First, using the response-time analysis derived in §VI, we study their impact on system schedulability compared to conventional techniques such as non-preemptive spin-locks, while also investigating the resulting increase in memory consumption due to the SMR facilities. Second, we apply SMR to `memcached`, a popular cloud service that is used in latency-sensitive environments [6]. We combine admission control with our timing analysis to provide soft real-time guarantees *and* bounded memory consumption.

II. Background and Related Work

A. Locks and Reader-Writer Locks

Locks serialize critical section execution of data-structure operations. This makes analysis of the operation's correctness as simple as for sequential code at the cost of losing parallelism. Reader-Writer (RW) locks allow readers to access critical sections in parallel with each other, while updates are still mutually exclusive with other writers and readers. Past research in the timing analysis of locks has considered many different dimensions, such as should waiting tasks spin or suspend [7], the impact of preemptive versus non-preemptive critical sections [8], and how to analyze read-write locks [9].

[1]Note that RCU includes two separate functionalities. First, it provides an SMR implementation. This is the main focus of this paper. Second, it provides the read-copy-update namesake technique for *updating* data-structures. We assume this latter technique is used with *each* of the studied SMR implementations.

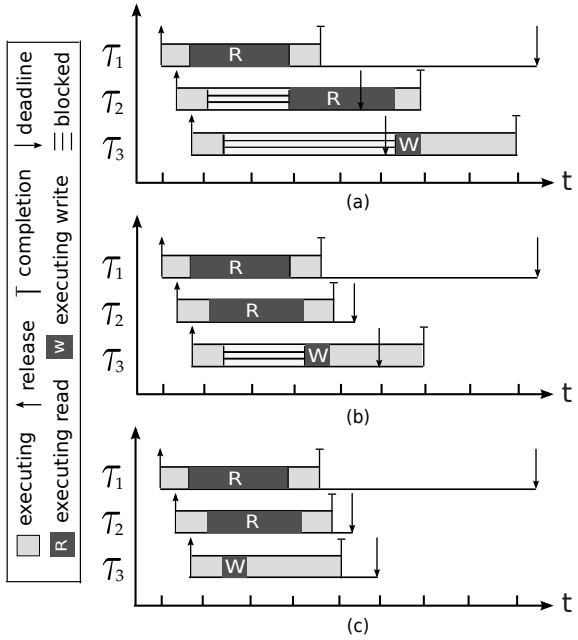

Fig. 2: Example schedules of two readers (τ_1, τ_2) and one writer (τ_3). (a) mutex; (b) RW locks; (c) real-time SMR.

B. Challenges with Locks

Cache-coherency overheads. An inherent limitation of locking algorithms is that they *explicitly* coordinate synchronizing tasks with *writes* to flags, counters, or pointers managed in *shared* control data-structures (*i.e.*, the lock), which can trigger a substantial amount of cache-coherency traffic, and these hardware overheads inflate the worst-case overhead of locks. Figure 1 details maximum overheads observed when using different synchronization primitives including a ticket-lock, an MCS lock [10], a RW ticket lock [9], U-RCU and two real-time SMR variants using the hardware detailed in §VII. The two real-time SMR primitives introduced in this paper incur the least cost in the read path, while the maximum cost of other lock-based approaches increases significantly as the number of cores in the system increases. For the update-path, as explained later, real-time SMR implementations utilize an MCS lock, and thus exhibit the same cost as MCS locks.

Worst-case blocking. SMR is unique in that it attempts to minimize synchronization between readers and writers. Figure 2 shows how the blocking time of locks compares to that of SMR. For example, with a mutex (Figure 2(a)), thread (or task) τ_1 delays both τ_2 and τ_3, and can cause them to miss their deadlines. While RW locks (Figure 2(b)) reduce blocking time incurred by τ_2, τ_3 still must block for all readers to finish, and still misses its deadline. For real-time SMR (Figure 2(c)), the blocking time of τ_3 is further reduced by running readers concurrently with writers. As a result, all tasks meet their deadlines. We next discuss how real-time SMR achieves this.

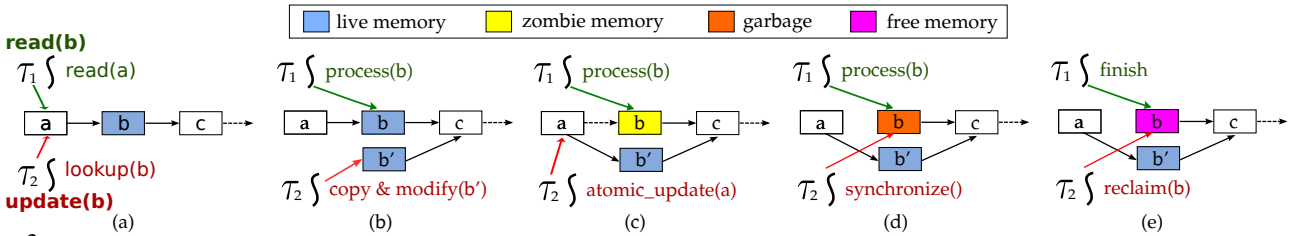

Fig. 3: Thread τ_1 reads node b while thread τ_2 concurrently updates node b. τ_2 locates node b to update. Instead of directly changing b, τ_2 allocates a new object b', copies b's content to b' and performs modification. During this modification, τ_2 does not interfere with readers as its private copy is not reachable within the linked list. After modifying b', τ_2 atomically both connects the new node b' and unlink the old node b from the list. The atomicity of this modification ensures data-structure consistency. Node b can be reclaimed after a grace period elapses, which guarantees no reader is still accessing b.

C. Non-blocking Data-structures with RCU

RCU (Read-Copy-Update) [3] is a technique that separates the *read path* that involves only *loads* from shared cache-lines, from the *update path* that modifies the data-structure. Thus the read path includes no explicit synchronization with concurrent modifications to the data-structure. However, since readers do not synchronize, data-structure nodes cannot be modified in a way that would provide readers with a snapshot of the data-structure that is inconsistent, or that has invalid pointers. Hence, care must be taken to make sure that updates to the data-structure are *atomic* with respect to readers. Figure 3 illustrates an RCU-based linked list example of parallel writers and readers. The update path for RCU (1) finds the node to be modified (Figure 3(a)), (2) copies that node into a newly allocated structure (Figure 3(b)), (3) modifies that node with the intended changes (Figure 3(b)), and (4) atomically both unlinks the old node from the data-structure while linking in the new node (Figure 3(c)).

RCU cannot trivially be applied to every data-structure as it has several constraints: (1) RCU requires writers to allocate and copy a node's memory, which can be prohibitive if nodes require large amounts of memory; (2) RCU requires updates to be performed by a single atomic instruction, which implies that nodes should only be reachable by a single pointer, a constraint that rules out, for example, doubly linked lists; (3) Updates are often mutually exclusive. To illustrate constraint (3), suppose that in Figure 3(b) another thread concurrently inserts a node after b. This inserted node would be lost as a result of disconnecting b (see Figure 3(c)). We therefore assume mutual exclusion among writers to rule out such race conditions. We note that research exists that lifts this assumption [11], [12]. (4) Lastly, memory reclamation (Figure 3(e)) is required, which is discussed next. Despite a restrictive programming model, RCU has been shown to scale well [3] and is used in more than 6500 API calls in the Linux kernel [13].

D. Scalable Memory Reclamation

Although RCU improves scalability, it raises a new challenge – reclaiming old unlinked objects. For example, in Figure 3, node b's memory needs to be reclaimed eventually. However, it cannot be reclaimed immediately after node b is unlinked, since a concurrent reader may potentially access the memory, as illustrated in Figure 3(c). Therefore, the memory reclamation has to be delayed to a point when no readers can potentially hold a reference to the old data. As shown in Figure 3(d), node b can be safely freed after τ_1 finishes reading b. Thus, memory keeps accumulating before it can be reclaimed, which obviously creates a challenge in real-time systems. Dynamic memory allocation is often avoided, and unless the amount of memory that accumulates awaiting reclamation can be bounded, it is difficult to characterize a system's worst-case behavior. The SMR implementation impacts the amount of accumulated memory by (1) the amount of time SMR takes to finish, and (2) the amount of memory SMR reclaims. Predictable memory utilization requires bounds on both factors. However, no prior SMR implementation provides such guarantees.

E. Memory States

In real-time applications, some portion of memory may never be freed (*e.g.*, a and c in Figure 3), but the amount of such memory has to be upper-bounded in any system. Instead of bounding total memory consumption, we focus on the dynamic memory consumption introduced by updates and real-time SMR. This part of dynamic memory goes through a sequence of states:

- *Live memory.* Memory that is both allocated and reachable in the data-structure, such as b and b' in Figure 3(b).
- *Zombie memory.* Memory that has been unlinked from the data-structure, but can still be referenced by others (pending a grace period), such as b in Figure 3(c).
- *Garbage.* Unreachable, freed memory that has yet to be reclaimed (b in Figure 3(d)).
- *Freed memory.* Memory has been reclaimed and is available to future allocation (b in Figure 3(e)).

F. SMR API

The primary goal of SMR implementations is to distinguish garbage from zombie memory. Toward this, SMR implementations require that readers use explicit code annotations when accessing the data-structure. Two common SMR interfaces are provided for such annotations:

- `enter()` to declare the start of a code section in which references to nodes can exist.
- `exit()` to declare the end of that section. No thread-local references to nodes can remain after this.

The section between `enter` and `exit` is referred to as a "read-side section" or "parallel section" interchangeably. Threads are said to be in a quiescent state when they are not inside a parallel section. A grace period is a time period during which every thread becomes quiescent at least once. Another SMR function is provided to detect the end of a grace period:

- `synchronize()` which enables the calling thread to wait for a grace period to elapse.

```
1 void read(D, identifier) {
2     enter();
3     process(lookup(D, identifier));
4     exit();
5 }
6 void update(D, identifier) {
7     n = alloc_node();
8     enter();
9     e = lookup(D, identifier); // find existing node
10    copy(n, e);                // copy e to n
11    modify(n);                 // update the contents
12    replace(n, e);             // add n and remove e
13    exit();
14    free_node(e);              // free after quiescence
15 }
16 void free_node(e) {
17    synchronize();
18    free(e);
19 }
```

Fig. 4: Usage of a typical SMR API with a data-structure D.

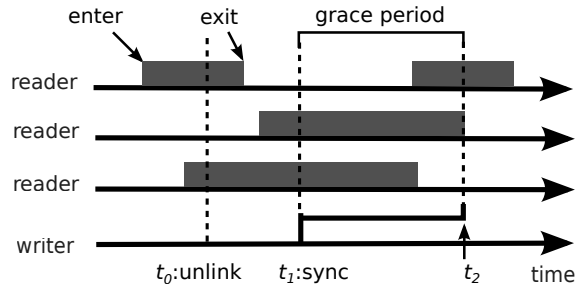

Fig. 5: Interaction between parallel sections, updates, & quiescence.

Figure 4 depicts example pseudocode illustrating the use of SMR. Note that all read-path operations are delimited by `enter` and `exit`, and `free_node` must contain logic to ensure a quiescence period has elapsed.

The key principle of SMR is that, if an object is unlinked from the shared data-structure, it can be safely reused after a grace period has passed. Because readers drop all references to the object after they exit the read-side section, and it is no longer linked within the data-structure, they cannot access the object anymore. Figure 5 depicts SMR's interaction between parallel sections and quiescence detection. Read paths are denoted as grey blocks. At time t_0, a writer unlinks a node, and at time t_1, it calls `synchronize` to await the passage of a grace period. The grace period extends past when any reader active at time t_1 has exited, and at that point (t_2), the system is quiescent. Thus, after time t_2, it is safe to reclaim the object unlinked at time t_0.

G. Dynamic Memory Management

Real-time SMR requires dynamic memory management (lines 7 and 18 in Figure 4), which is challenging in real-time systems as it is both difficult to (1) reasonably bound execution time, and (2) bound memory consumption due to fragmentation. With SMR in real-time systems, we assume size uniformity of the dynamic objects, and use a slab allocator [14]. To avoid competing for access to a single shared pool, the allocator uses hierarchical allocation with thread-local memory caches, and then from within a global allocation pool. That, combined with a slab allocator's ability to eliminate internal fragmentation, makes it appealing for systems that require memory bounds. We further improve the slab allocator by handling "remote frees" predictably. However, as real-time memory allocation is out of scope, these details can be found in the Appendix.

```
1 #define RCU_GEN_CTR 0x2
2 static pthread_mutex_t rcu_sync_lock;
3 int rcu_gen_ctr = 1;
4 int rcu_reader_ctr[NUMTHDS];
5 void rcu_read_lock(void) {   // enter
6    rcu_reader_ctr[thdid] = rcu_gen_ctr; // record count
7 }
8 void rcu_read_unlock(void) { // exit
9     rcu_reader_ctr[thdid]--; // indicate leaving
10    wake_up_waiters();
11 }
12 void synchronize_rcu(void) {
13    lock(rcu_sync_lock);
14    // suspend waiting for each reader to complete
15    wait_for_readers();
16    rcu_gen_ctr ^= RCU_GEN_CTR; // bitwise xor
17    wait_for_readers();
18    unlock(rcu_sync_lock);
19 }
```

Fig. 6: Simplified U-RCU implementation.

Summary. Although SMR avoids reader-side synchronization and enables reader-writer parallelism, memory cannot be reclaimed until quiescence has been achieved. Furthermore, bounding memory utilization requires SMR to efficiently reclaim garbage predictably. This makes it challenging to naively apply existing SMR implementations in real-time systems. Thus, this paper focuses on a real-time SMR implementation (§IV), and its impact on both response times and memory consumption in real-time systems (§VI).

III. SMR Implementation Case Study: U-RCU

First, we introduce the preemptive quiescence detection in User-level Read-Copy-Update (U-RCU) [3]. Figure 6 is the simplified pseudo-code for this U-RCU implementation. A global variable rcu_gen_ctr is initialized to 1 and a per-thread variable rcu_reader_ctr is initialized to zero. The read path simply records the global counter. When exiting, it unsets its local counter, which signals that it has left the parallel section, and wakes up threads waiting for quiescence (if any).

Toggled by `synchronize_rcu` (line 16), the global counter iterates through two states. These are used to determine if a read path started before or after the thread executing the write path started waiting for a grace period to elapse. The quiescing thread suspends in `wait_for_readers` waiting for existing read paths to complete. Thus, returning from `synchronize_rcu` indicates a grace period elapses and garbage can be reclaimed. For instance, in Figure 7, `synchronize_rcu` is invoked at t_3, and then blocks awaiting quiescence, which occurs at t_4. When `synchronize_rcu` returns, it is safe to reclaim all memory freed before t_3. A lock ensures that only a single thread calculates quiescence at a time (line 13). To reduce lock contention, a trick is used: concurrently quiescing threads queue in a wait-free queue, waiting for their quiescence to be calculated by the head quiescing thread (not shown in the code). A detailed explanation of U-RCU is provided by Desnoyers et al. [3].

Limitation. U-RCU has two challenges that limit its scalability and predictability. From an implementation perspective, when detecting quiescence, the modification of the global counter causes cache-coherency traffic for all readers, and the use of a lock increases overheads and blocking. From an analytical perspective, without knowledge of when an object became zombie memory, a quiescing thread has to suspend, waiting for all pre-existing readers to complete their parallel

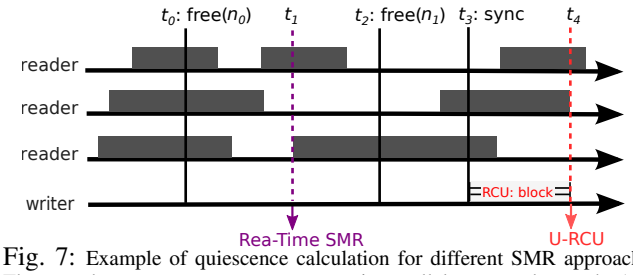

Fig. 7: Example of quiescence calculation for different SMR approaches. Three readers execute on separate cores, in parallel to an update path. At t_0 (t_2), node n_0 (n_1) is unlinked from the data-structure. At t_3, the updating thread attempts to quiesce (sync). U-RCU blocks awaiting quiescence which occurs at t_4 after all parallel sections active at t_3 have exited. In contrast, RT-ParSec calculates the most recent quiescence point, namely t_1. RT-ParSec does not block waiting for current readers to complete. Time t_1 is the last quiescence point: after that point a reader exists that prevents quiescence. After this quiescence calculation, U-RCU can recycle objects unlinked before t_3 (n_0 and n_1), and RT-ParSec reclaims all garbage unlinked before the calculated quiescence time t_1 (n_0).

sections. This couples the quiescence latency to *each* reader's worst-case response time.

IV. Real-Time SMR Implementation

We next introduce two real-time SMR implementations that are both predictable, and provide bounds on garbage memory. In contrast to U-RCU that *blocks* waiting for quiescence among the *currently active readers*, real-time SMR instead *calculates the time* most recently in the past when *quiescence was achieved*. All memory deallocated before that time can be reclaimed. Notably, this avoids the need for real-time SMR to block on parallel readers, but possibly decreases the amount of memory it can reclaim at a time.

A. RT-ParSec

RT-ParSec is a variant of ParSec [4] that has been designed to provide guarantees on memory bounds, at the expense of a decrease in scalability relative to ParSec. Similar to ParSec, RT-ParSec tags memory with the time that it was freed, and each thread maintains the timing information about when it enters and exits a parallel section. That timing information is *locally* accessed via the timestamp counter (*e.g.* Intel's *invariant timestamps* described in §17.14.1 of [15]), thus avoiding coherency traffic. To ensure that all queued garbage memory is reclaimed, thus providing predictable memory consumption, RT-ParSec calculates the most recent time in the past when quiescence was achieved. In contrast, U-RCU attempts await quiescence, thus it blocks waiting while all *current* readers have left their read-side sections.

RT-ParSec pseudo-code is shown in Figure 8. RT-ParSec and ParSec is the `ps_quiesce` implementation. Specifically, RT-ParSec only scans threads who are currently in their parallel sections and returns the earliest time they entered the parallel sections – the last quiescence time. This ensures that all memory made unreachable before that returned time point can be reclaimed (lines 27-30). In §VI, we show that by controlling quiescence detection frequency, the amount of collected garbage is bounded, which further guarantees that the execution time of the reclamation loop is bounded (lines 27-30). In ParSec, each thread caches the timing information of all other threads, and uses this cached information, where possible, to determine if quiescence is achieved. Using cached information avoids frequent access to remote cache lines, but

```
1  struct parsec {
2      struct thd_data {
3          tsc_t enter, exit;
4          struct quiesce_queue *quiesce_q;
5      } thd_info[NUM_THDS];
6  };
7  void ps_enter(struct parsec *ps) {
8      struct thd_data t = ps->thd_info[thdid];
9      t.enter = rdtsc();
10 }
11 void ps_exit(struct parsec *ps) {
12     struct thd_data t = ps->thd_info[thdid];
13     t.exit = t.enter + 1;
14 }
15 tsc_t ps_quiesce(struct parsec *ps) {
16     tsc_t q = rdtsc();
17     for (int i = 0 ; i < NUM_THDS ; i++) {
18         struct thd_data t = ps->thd_info[i];
19         if (t.exit < t.enter) q = min(t.enter, q);
20     }
21     return q;
22 }
23 void ps_smr(struct parsec *ps) {
24     struct thd_data t = ps->thd_info[thdid];
25     void *node = dequeue_peek(t.quiesce_q);
26     tsc_t q = ps_quiesce(ps);
27     while (node->free_tsc < q) {
28         free(dequeue(t.quiesce_q));
29         node = dequeue_peek(t.quiesce_q);
30     }
31 }
```

Fig. 8: RT-ParSec SMR implementation adapted from ParSec

```
1 void ps_enter(struct parsec *ps) {}
2 void ps_exit(struct parsec *ps)  {}
3 tsc_t ps_quiesce(struct parsec *ps) {
4     return rdtsc() - PS_GRACE_PERIOD;
5 }
```

Fig. 9: Temporal quiescence implementation. The `struct parsec` and `free_node` from Figure 8 are used directly. `PS_GRACE_PERIOD` is computed from the task model, and timing analysis (§VI).

leads to inaccurate quiescence calculation causing possibly unbounded memory consumption. In contrast, RT-ParSec directly gathers timing information from other cores, which guarantees accurate quiescence calculation. On the other hand, RT-ParSec causes more cache traffic, which decreases scalability. A comparison of quiescence detection behavior in U-RCU and RT-ParSec is depicted in Figure 7.

B. Temporal Quiescence

Hard real-time systems provide strong guarantees on the execution properties of tasks in the system. This approach to SMR calculates quiescence based entirely on the system task model. The maximum parallel section length is known a priori, and a response-time analysis can determine the longest possible grace period (based on all task periods). Thus, at a time `t`, the most recent quiescence point is simply `t` minus this a-priori-calculated grace period. The pseudocode is presented in Figure 9.

Temporal quiescence is used in the non-preemptive SPECK kernel [5] to control the re-use of kernel data-structures. However, an analysis of the worst-case data-structure access time is comparably simple in a *non-preemptive* environment. §VI demonstrates how to derive the longest-possible grace period given a response-time analysis in the general case.

V. System Model and Assumptions

Deferred memory reclamation will cause freed memory to accumulate, which both inflates memory consumption

and increases garbage collection interference. Thus a co-analysis of both memory consumption and response time is mandatory to derive bounds necessary for predictable SMR. This section introduces the system model, in preparation of the corresponding analysis in §VI.

A. System Model

We consider a real-time workload consisting of N sporadic tasks scheduled on M identical cores, $\tau = \{\tau_0, \ldots, \tau_N\}$. Each task consists of an infinite stream of jobs. Each task τ_i is characterized by a tuple (e_i, p_i, a_i). e_i is the worst-case execution time (WCET) of a job in τ_i, and p_i is the minimum inter-arrival time between jobs of τ_i. a_i is the maximum number of memory allocations made by one of τ_i's jobs. Each task has an implicit deadline equal to its period p_i. A task's utilization is defined by $u_i = \frac{e_i}{p_i}$. A job is pending from its release until it completes. The response time r_i denotes the maximum duration that any job of τ_i remains pending. We assume partitioned fixed-priority scheduling, as mandated, for example, by AUTOSAR [16]. Each task is statically assigned to a core, and each core schedules pending jobs in order of decreasing task priority. The set of tasks with higher (lower) priority than τ_i (on the same core as τ_i) is hp_i (lp_i).

B. Shared Resources

The tasks also share some global resources. When a job is going to observe (update) a shared resource, it issues a read (write) request, and is said to be a reader (writer). Γ_i^r (Γ_i^w) denotes the WCET of read (write) requests in task τ_i. Similar to task response time, we define request response time as the maximum duration from when a request is issued until it completes. Δ_i^r (Δ_i^w) denotes the read (write) request response time. For ease of notation, we make the simplifying assumption that (1) there is one shared resource, (2) a job issues at most one read and one write request, and that those requests are not nested within each other. However, multiple memory objects can be allocated or released during even one write request. Note that these restrictions are not fundamental. The presented analysis can be easily extended to multiple resources and multiple non-nested requests per job by adding up the blocking times of multiple requests or resource, as well as the overall memory usage.

As discussed in §II, real-time SMR enables read requests to run concurrently with other read or write requests. However write requests use mutual exclusion. Similar to the RW-FMLP [9], we assume non-preemptible task-fair mutexes (such as ticket locks or MCS locks) are used to serialize writers. When a job of task τ_i issues a write request, τ_i becomes non-preemptive and executes the corresponding locking protocol. The write request is satisfied once τ_i holds the write lock, and preemption is disabled until τ_i completes its write request. In contrast to the RW-FMLP under which both read and write requests are both non-preemptive, *only write requests are non-preemptive* in real-time SMR.

C. Memory Model

Hard real-time systems require predictable task execution times to ensure deadlines are met, *and bounded memory requirements to prevent memory exhaustion*. Though SMR techniques show promise to enable low-overhead, predictable parallelism, they do *require* both dynamic memory allocation, and delayed reclamation. To bound the memory consumption caused by SMR, every task must consume only a finite amount of memory. To capture this constraint, we use the release times of SMR memory operations to order all the memory allocation and deallocation operations performed by all tasks. L_i is the maximum time interval between the ith allocation and the ith deallocation (these two operations might not refer to the same memory object). We use L^* to represent the maximum value of L_i (*i.e.* $L^* = \max_{\forall i} L_i$). L^* captures the natural "ebbs and flows" of transient memory utilization in the system, which is application and schedule specific. Between the time the memory is freed, and the time it can be reused, delayed memory reclamation further increases the amount of required memory.

We assume that a job issues zero or more pairs of memory allocation and deallocation requests. This assumption is consistent with the typical use of RCU where a new node allocation is followed by the freeing of the old version of the node. While write requests are done within non-preemptive locks, memory operations are not necessarily protected by the lock. Thus, L^* is upper-bounded by the maximum response time of writer tasks. L_i can encode more complicated allocation and free behaviors such as producer/consumer or publisher/subscriber synchronization. Accurate L^* bounds can be computed with real-time calculus [17], but we leave such extensions for future work.

With respect to memory consumption, this paper focuses on bounding zombie and reclaimable memory, because the amount of live memory is only determined by the application's memory usage patterns. The amount of zombie memory is determined by the system's scheduling policy and task's parallel section WCET. The SMR implementation and activation times determine the amount of garbage. SMR detects quiescence, thus reclaims memory sporadically. The inter-arrival time of quiescence detection for task τ_i is denoted by q_i. We assume that the quiescence detection is integrated into the writer's execution (*i.e.*, as a function call). This is commonly the case. For example, the act of freeing memory can trigger quiescence detection (line 17 in Figure 4). Separating quiescience inter-arrivals from task inter-arrivals enables quiescence detection to be done infrequently to trade quiescence overhead for larger accumulated amounts of non-reclaimed memory. Quiescence detection could be implemented as a separate task.

We make no assumptions about the locality of memory allocations and frees, thus assume all memory is *remotely freed* in the worst-case. This means that the memory chunk is freed on a core other than where it was allocated, a situation that complicates per-core memory pools. We further assume that memory can be balanced between local pools immediately in case more memory is needed. This assumption is true for our current slab allocator modulo external fragmentation in each slab [14].

In the following analysis, we let r_σ^* denotes the maximum

response time of quiescence tasks, α is the quiescence detection implementation overhead, and β is the implementation cost of collecting a memory object.

VI. Memory and Response-Time Analysis

Given the system model in §V and the system operation overheads, we bound both the whole system's worst-case memory requirement and each task's response time.

A. Worst-case Memory Consumption

A well-known upper bound on the number of jobs issued by τ_i in any interval of length W is $\left\lceil \frac{W+r_i}{p_i} \right\rceil$ (*e.g.* see [18]), which implies a bound on the maximum memory needs.

Lemma 6.1: Within a time window W, the maximum number of memory objects task τ_i can allocate is

$$A_i(W) = \left\lceil \frac{W + r_i}{p_i} \right\rceil \times a_i. \tag{1}$$

and the maximum number of memory objects allocated by the whole system over a window W is

$$A(W) = \sum_{i=1}^{N} A_i(W). \tag{2}$$

Lemma 6.2: For any specific memory object allocation, there will exist a memory object that can be reclaimed after at most δ time, where

$$\delta = L^* + \Delta^* + q^*. \tag{3}$$

Proof Sketch: Based on the definition of L^*, for any allocation, there is a corresponding deallocation within L^*. That deallocated (zombie) memory will become garbage after no later than when a grace period (Δ^*) has elapsed. After this time interval, all threads will have exited all parallel sections. Hence, it will be collected by any following memory reclamation, which happens at the latest when the next quiescence task is released (q^*). We consider the maximum of all q_i to reflect that the memory could be reclaimed by any task. ∎

Because U-RCU will block for Δ^* as part of its quiescence detection (§III), a memory object can be reclaimed by U-RCU even if Δ^* has not elapsed before quiescence detection is performed. For example, in Figure 7, memory node n_1 can be reclaimed by U-RCU. Thus, we have a simplified result for the U-RCU case:

$$\delta_U = L^* + q^*. \tag{4}$$

Theorem 6.3: The maximum total amount of zombie and garbage memory objects in the whole system is upper-bounded by:

$$G^* = A(\delta + r_\sigma^*) = \sum_{i=1}^{N} \left(1 + \left\lceil \frac{\delta + r_\sigma^* + r_i}{p_i} \right\rceil\right) \times a_i. \tag{5}$$

Proof Sketch: Recall that r_σ^* captures the maximum amount of time quiescence detection takes to finish. From lemma 6.2, at any point in time t, the number of collected objects is no less than the number of memory allocations before $t - (\delta + r_\sigma^*)$. Therefore the worst-case memory consumption is the maximum amount of memory allocated by all tasks during a time interval of length $\delta + r_\sigma^*$. ∎

As an example, the maximum memory consumption is illustrated in Figures 3(b) - (d), where new objects have been allocated while old ones are not reclaimed yet, thus zombie and garbage memory keeps accumulating until SMR finishes quiescence detection and garbage collection.

B. Response Time Analysis

Audsley et al. [19] established that a bound on τ_i's response time r_i is given by the smallest positive solution of

$$r_i = e_i + b_i + \sum_{\tau_k \in hp_i} \left\lceil \frac{r_i}{p_k} \right\rceil \times e_k. \tag{6}$$

where b_i denotes the total blocking time incurred by τ_i. Thus, in order to calculate r_i, first we need to bound the interference from higher-priority tasks including SMR overheads, and then determine the blocking time caused by writer's mutex and SMR operations (in case of U-RCU).

SMR Interference. To account for SMR overheads, we treat quiescence detection as a separate task. For each writer task τ_i, there is a derived memory reclamation task $\sigma_i = (e(\sigma)_i, q_i)$, whose priority is higher than τ_i but lower than any task in hp_i. ghp_i represents the set of memory reclamation tasks with higher priority than τ_i on the same core as τ_i.

Memory reclamation contains two steps – detecting quiescence and reclaiming garbage. The cost of the second step depends on the amount of garbage. In the worst case, a deferred memory reclamation task will collect *all* garbage, thus leaving other quiescence tasks with nothing to reclaim. Hence, the total amount of work of all deferred memory reclamation tasks is upper-bounded by the maximum amount of garbage. Given the response time r_i, similar to theorem 6.3, the maximum amount of garbage collected is bounded by $A(\delta + r_i)$, which implies the following bound.

Theorem 6.4: Interference on task τ_i from local higher-priority SMR tasks is bounded by

$$I(\sigma)_i = \left(\sum_{\sigma_k \in ghp_i} \left\lceil \frac{r_i}{q_k} \right\rceil \times \alpha \right) + A(\delta + r_i) \times \beta. \tag{7}$$

Recall that Δ^* (the maximum read request response time) is used to determine zombie memory's liveness (lemma 6.2). In Eq 8, Δ_i^r is calculated similarly using Eq 6. In addition to inflating SMR interference accordingly, we need to replace e_i with Γ_i^r and set b_i to 0 (parallel section is non-blocking).

$$\begin{aligned} \Delta_i^r = \Gamma_i^r &+ \left(\sum_{\tau_k \in hp_i} \left\lceil \frac{\Delta_i^r}{p_k} \right\rceil \times e_k \right) + \\ &\left(\sum_{\sigma_k \in ghp_i} \left\lceil \frac{\Delta_i^r}{q_k} \right\rceil \times \alpha \right) + A(\delta + \Delta_i^r) \times \beta. \end{aligned} \tag{8}$$

Writer Mutex Blocking. Non-preemptive FIFO spin locks are used to protect writers, and they also introduce blocking overheads. [20] introduces an analysis framework based on integer linear programming (ILP) to accurately bound the maximum cumulative blocking time. This ILP-based method explicitly accounts for every potential lock contention pattern, and the resulting bound is directly used in equation 6 as a blocking term. To determine writer mutex blocking overheads

in real-time SMR, we modify this analysis to consider only write requests.

U-RCU Blocking. When trying to quiescence, neither ParSec nor time-based quiescence are blocking (§IV). However, in U-RCU, quiescence detection can be blocked by other tasks either in parallel sections, or also attempting quiescence. To make U-RCU more analysis-friendly, we consider a simplified case with only one memory reclamation task. Hence, this task only needs to wait for all readers to leave their parallel section, and this waiting time is bounded by Δ^*. Such self-suspension within the U-RCU memory reclamation task will cause increased interference on lower-priority tasks, which are not captured by the RTA (Eq 6). (Self-suspension problems are extensively studied in [21].) As a result, the presented analysis, though simple, will generate optimistic schedulability results for U-RCU. As shown in section VII-B, even with such optimism, U-RCU's schedulability falls short of real-time SMR. In contrast, real-time SMR incurs no self-suspension due to its non-blocking quiescence detection.

C. Quiescence Frequency Calculation

The above analysis requires quiescence periods to be given as input parameters. We present an algorithm for selecting quiescent periods that attempts to minimize memory consumption while maximizing schedulability. This is complicated by the fact that the quiescence frequency's impact on schedulability is not monotonic. While longer quiescence period causes less SMR interference, it incurs a higher garbage collection cost (see Eq 7). Therefore, we linearly, in steps of the minimum task period, search for the minimum quiescence period that results in a schedulable system. Additionally, the co-analysis of schedulability and memory bounds is also complicated by the inter-dependency among many parameters. For instance, SMR interference impacts Δ_i^r (Eq 8), which determines δ (Eq 3), and δ affects memory consumption, which will contribute back to the SMR interference (Eq 7). Thus, the schedulability test is performed by iteratively calculating blocking times, parallel section response times and task's response times until a global fixed point is reached. Figure 16 in the Appendix gives the algorithm's details.

VII. Experimental Evaluation

All experiments were run on a system consisting of four Intel Xeon E7-4850 sockets, each with 10 cores, clocked at 2.0 GHz. Version 3.10.10 of the Linux kernel was used and hyper-threading was disabled, leading to 40 cores in total. Though this system is not of the latest generation, it enables an evaluation of many sockets. Benchmarks were run on different core counts to measure scalability. The minimal number of sockets was always used.

A. Micro Benchmarks

We conduct a set of micro-benchmarks to estimate the worst-case overheads of various synchronization operations. For each core count, we enumerated reader/writer ratios to find the worst-case scenario. We use the 99th percentile cost across 10 millions runs to exclude non-maskable and other interrupts. U-RCU uses the `urcu-mb` implementation

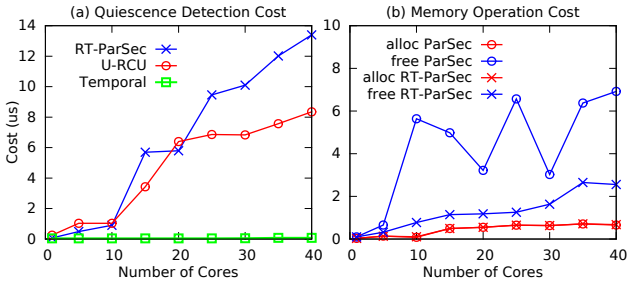

Fig. 10: 99th percentile costs of quiescence and memory operations.

from `liburcu` [22], and all locks are implemented in the Concurrency Kit [23]. Though [24] provides a more efficient queue-based phase-fair RW locks, the version we use still shows the useful trends in RW lock behavior, as they share analytical bounds.

Read path overheads are show in Figure 1(a). U-RCU incurs the highest overhead because it uses Linux `futex`es to wake up blocked quiescing threads. The temporal quiescence approach has the lowest cost as it does not access any shared cache lines at all. Though the tested RW lock incurs more overhead than a ticket lock, the overhead of the queue-based variant [24] is expected to be closer to the MCS lock.

Update path overheads have similar trends to the read path. Both U-RCU and real-time SMR serialize the writers using an MCS lock, which is the most scalable as shown in Figure 1(b),

Quiescence detection costs are shown in Figure 10(a). When calculating quiescence, RT-ParSec iterates through all other thread's local data, which causes significant cache-coherency traffic and inhibits its scalability. U-RCU incurs less overhead because our model assumes there is only one quiescing thread, while in RT-ParSec, the worst-case happens with multiple concurrent quiescing threads (as explained in §VI-B).

Memory operation overhead (Figure 10(b)). As mentioned in §II-G, SMR relies on dynamic memory management. Thus, we need to confirm that memory related operations are also predictable. The results demonstrate the improved predictability resulting from the slab allocator optimizations detailed in the Appendix, and that allocation and free operations scale relatively well even in the 99th percentile.

B. Schedulability Tests

Task set parameters. We generated task sets with 10 tasks per core using Emberson et al.'s task set generator [25]. Periods were uniformly selected from [10ms, 100ms]. Task execution costs were calculated based on utilizations and periods. The locking, SMR, and memory allocation overheads inflate task execution cost. Only one global resource is shared across all tasks. Readers (writers) are uniformly assigned to tasks, they issue one read (write) request per period. We study the following parameters. We consider platforms with $M \in \{1, 5, 10 \ldots 40\}$ cores. We vary the per core task set utilization across [0.6, 0.9] in steps of 0.05. The number of concurrent readers (writers) is varied across {0, 5, 10, 15, 20}. The duration of read (write) requests is uniformly distributed in [1μs, 300μs]. In total, we evaluated over 400 configurations. It is clearly not feasible to present all results, so we chose a set of default parameters, and vary only one

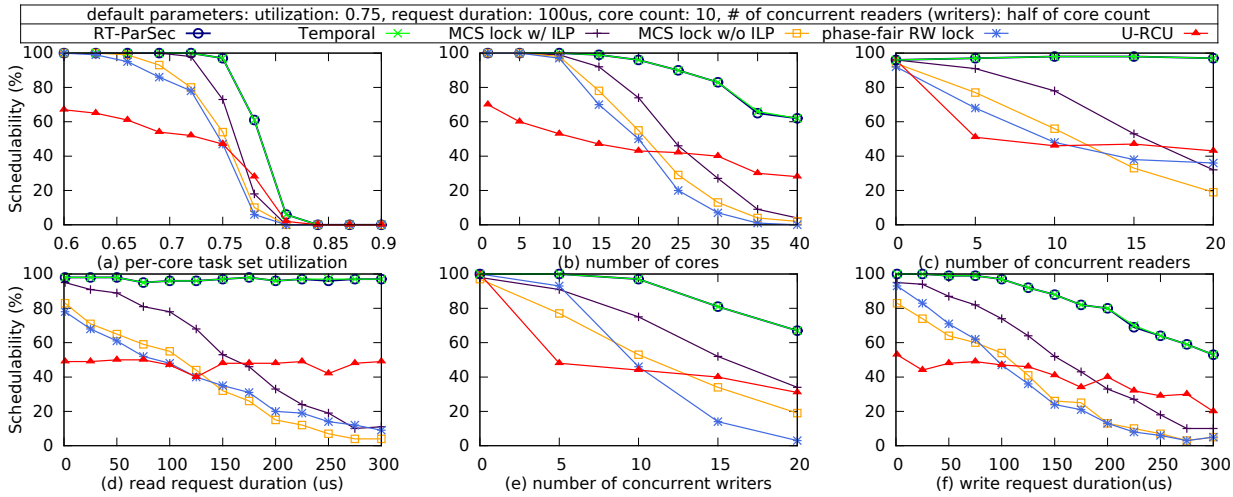

Fig. 11: Schedulability results for various configurations. The curves of RT-ParSec and Temporal largely coincide.

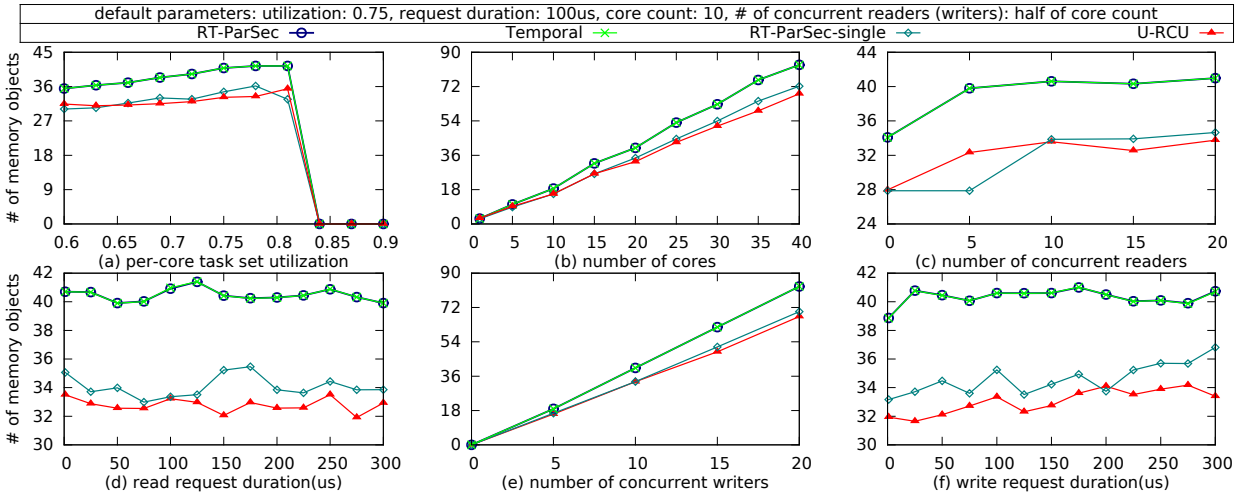

Fig. 12: Memory consumption result. The curves of RT-ParSec and Temporal largely coincide.

parameter in each graph reported herein. The default core count is 20 cores, half of them executing readers and the other half executing writers. Such reader/writer ratio is commonly considered to be *write-heavy*, and is not an obvious match for SMR techniques that are usually only used in *read-heavy* workloads. However, we choose this value to avoid artificially favoring SMR, which excels at read-mostly workloads, thus determining *if it can compete even in write-heavy workloads*. The per core task set utilization is 0.75, and the default request duration is 100μs. Unless otherwise stated, the quiescence period is determined using the algorithm in §VI-C and each writer allocates one memory object per period. For real-time SMR implementations, the highest-priority writer on each core detects quiescence and reclaims memory. While for U-RCU, we randomly select only one core to collect garbage.

Experiment methodology. For each configuration, we randomly generated 500 task sets to measure the schedulability (the fraction of schedule task sets in the generated task sets). For the schedulable task sets, we also calculated their memory consumption in the SMR cases. The schedulability test and blocking analysis code are taken from the SchedCAT framework [26]. We used two mutex blocking analysis: the inflation-based analyses [9], and the ILP-based approach [20], while for RW locks, only an inflation-based analysis exists. We try both techniques to analyze mutex blocking time. For the SMR techniques, as only writers need mutual exclusion, the difference between the two approaches is negligible. Thus, we only show the results of ILP-based analysis in SMR cases.

Schedulability results. Figure 11 shows the schedulability results for various configurations. The two real-time SMR variants, RT-ParSec and temporal quiescence, are very similar, and achieve the highest schedulability in all configurations. Somewhat surprisingly, even in write-heavy workloads, the reduced mutex contention due to readers avoiding serialization surmounts the allocation and quiescence overheads. Although U-RCU increases parallelism in best effort environments, its blocking quiescence detection causes the invoking task to miss deadline, leading to low schedulability for real-time tasks. Due to the pessimistic analysis and write-heavy workload, RW locks have lower schedulability than mutex in most cases with the exception being for low writer counts. RW locks are not designed for such write-heavy workloads, and we further investigate reader/writer ratios later.

The real-time SMR implementations exhibit high relative scalability. With increasing core counts (Figure 11(b)), real-time SMR has the smallest relative schedulability degradation. With core counts above 25, lock-based synchronization has lower schedulability than U-RCU, illustrating that the impact of blocking between readers and writers at scale. Predictably, Figures 11(c) and (d) show that real-time SMR schedulability

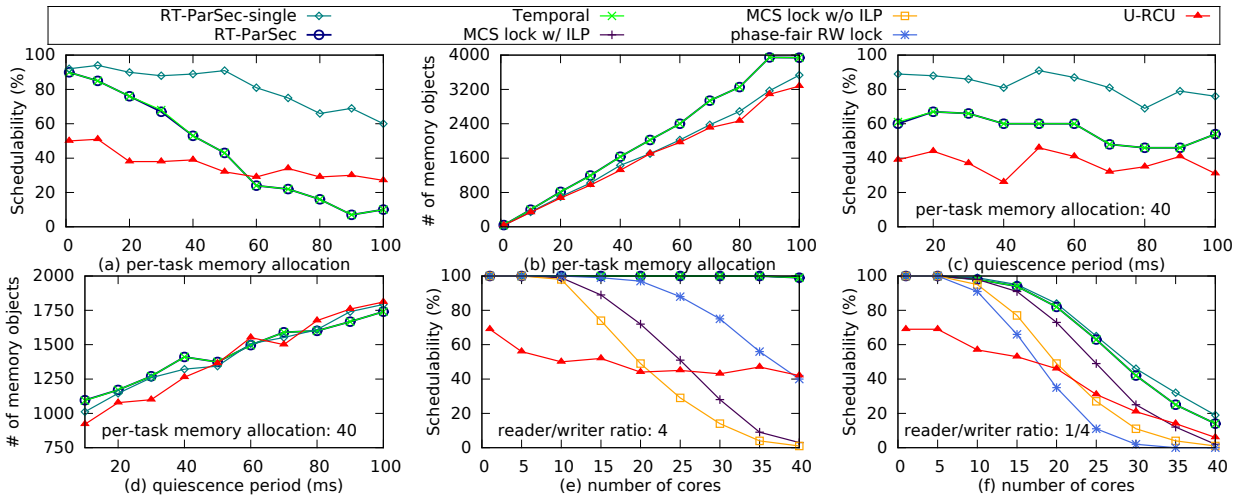

Fig. 13: (a) and (b) study the impact of per-task memory allocation; (c) and (d) show the impact of quiescence period; (e) and (f) show the schedulability under different reader/writer ratios. The curves of RT-ParSec and Temporal largely coincide.

is unaffected by the number of readers or the read request length. With increasing read contention, both U-RCU and RW locks increase schedulability relative to mutexes. With update-mostly workloads, real-time SMR implementations decrease in schedulability as write request contention becomes the dominating factor. Figure 11(e) and (f) demonstrate this trend.

Memory and quiescence results are presented in Figure 12 and Figure 13. In studying the memory requirements of the SMR implementations, Eq 5 gives us three factors that determine the total memory consumption. They are the number of writers (N), the maximum number of memory allocations during update (a_i) and the amount of time memory keeps accumulating ($\delta + r^*_\sigma$). Memory bounds increase linearly with those factors as expected (Figure 12(b), Figure 12(e), Figure 13(b) and Figure 13(d)). In the default setting, real-time SMR consumes more memory than U-RCU. This is because the maximum response time of the quiescence task (r^*_σ in Eq 5) in real-time SMR is larger than in U-RCU, as U-RCU has only one thread for memory reclamation, while real-time SMR has one quiescing thread per-core. If real-time SMR also uses one memory reclamation thread (the line labeled as "RT-PARSEC-single"), its memory bound is roughly the same as U-RCU. In the meantime, schedulability decreases with increasing memory usage due to collection costs (Figure 13(a)). This confirms the empirical conclusion that SMR can be prohibitive if updates require large amounts of memory. By affecting the amount of time memory keeps accumulating ($\delta+r^*_\sigma$), a longer quiescence period causes more memory to be accumulated (Figure 13(d)). However, as analyzed in §VI-C, quiescence frequency impacts schedulability in complex ways, which is demonstrated by Figure 13(c). Last, we study how schedulability scales under different reader/writer ratios. In read-heavy cases (Figure 13(e)), real-time SMR, U-RCU and the RW lock perform better than mutex locks. However, with write-mostly workloads (Figure 13(f)), all those techniques behave similarly.

C. *Application Study:* `memcached`

In this section we illustrate how real-time SMR provides useful bounds for *latency-sensitive applications*, of which we use `memcached` (https://memcached.org/) as a representative. `memcached` implements a concurrent hash table providing `get` and `put` requests for a cache of key-value pairs.

Memcached task model. In order to apply real-time SMR, we create a task model (§V) to characterize `memcached`. Formally, on each core, there are two tasks, τ_r and τ_w. τ_r (τ_w) consists of a stream of jobs to send `get` (`set`) requests periodically. The concurrent hash table is modeled as a global shared resource shared by all tasks. τ_r (τ_w) issues non-nested read (write) request for the hash table. Since a read-heavy workload is a typical real-world workload [27], we assign higher priority to τ_r. As `memcached` is not mission critical, we consider its tasks to be soft real-time. According to [7], [9], under partitioned scheduling, we can schedule an implicit-deadline soft real-time task set by viewing it as hard, but using average-case overheads. Thus, we apply the same analysis in §VI to the `memcached` task set with the measured average task and synchronization overhead.

Experiment set-up. To concentrate the evaluation on predictability, we run `memcached` in a single process, bypassing the network and kernel. To avoid potential unpredictable cache replacement policy, we also disable cache replacement and give it sufficient memory to avoid cache eviction. Similar to [28], a workload trace with 20% set requests is generated using YCSB [29], following a zipfian distribution. The trace contains 10 million requests (16-byte key and 32-byte value), which is partitioned across `memcached` tasks on different cores. `memcached`'s concurrent hash-table uses a separate (unpredictable) spin lock for each entry (and the associated linked-list of items). We modify `memcached` to use FIFO MCS locks so as to not unfairly penalize it. We compare three `memcached` implementations: using an MCS lock, a phase-fair RW lock and RT-ParSec. We omit RCU given the results in §VII-B. As the temporal quiescence SMR approach requires a hard real-time environment, it cannot be used here.

Strict admission control. To make `memcached` tasks periodic, we apply admission control based on the model, thus limiting the rate of jobs. According to Eq 5, the rate of jobs is the essential factor bounding memory utilization. Thus, the key question is how to choose the period of `get` and `set` requests to maximize throughput while still bounding

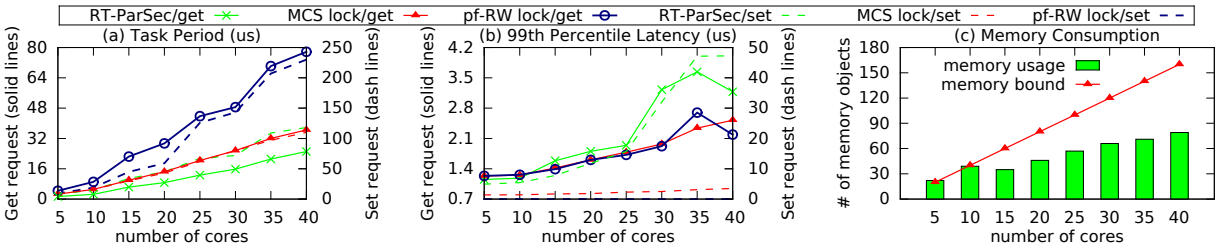

Fig. 14: Memcached evaluation results: strict admission control.

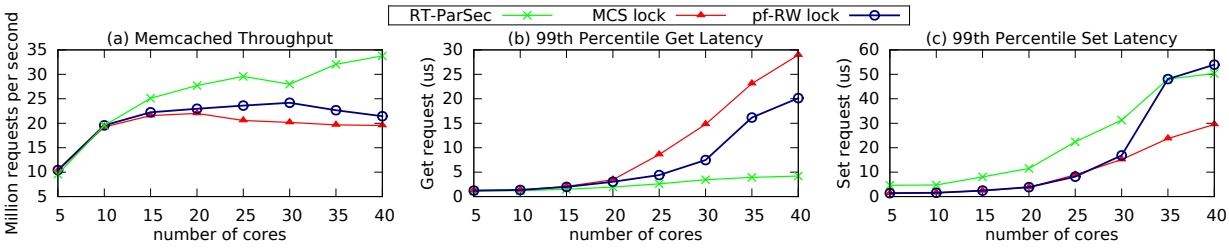

Fig. 15: Memcached evaluation results: relaxed admission control.

memory consumption. Based on the analysis (§VI), when the update period is fixed, we can determine the minimal reader's period that makes the task set schedulable if it exists. This provides an upper rate on the jobs for both updates and reads, and thus bounds the system's throughput. While larger update periods decrease `set` request throughput, they allow higher rates of readers, thus increasing `get` request throughput. Thus, for each configuration, we choose the minimal update period that maximizes the total throughput. Figure 14(a) shows these calculated periods. RT-ParSec archives the smallest `get` period (thus, largest `get` throughput), resulting from its efficient read-path synchronization. The `set` period in RT-ParSec is similar to that in MCS locks, as they use the same lock to serialize updates. MCS locks have the same period for `get` and `set` request, as it treats them the same. Figure 14(b) gives the measured 99th percentile latency for `get` and `set` requests. RT-ParSec has a larger `get` request tail latency than both MCS and RW locks. The higher rate of parallel `get` requests in RT-ParSec causes significantly more contention on the memory bus and controller. The non-uniform memory access in our system exacerbates this. RT-ParSec `set` request tail latency is high due to its coherency-operations on quiescence. In summary, per-hash-table entry locks combined with admission control result in less contention on shared cache lines for MCS and RW locks, leading to lower tail latency, while RT-ParSec achieves a higher rate of `get` requests. Figure 14(c) verifies that the amount of memory consumed by RT-ParSec is within its theoretical bounds.

Relaxed admission control. Although strict admission control lowers tail latency, its achieved overall throughput is not practical (*e.g.*, throughput of MCS lock on 40 cores is only 1.4 million requests per second). On the other hand, even if we use average-case overheads for the analysis, the analysis is still rather pessimistic. Given this, there is a large amount of slack time that can be used to serve more requests. However, we need to carefully use slack time to avoid interference on existing requests. For instance, such slack time cannot be used for `set` requests, as they need locks and modify shared cache lines. Hence, we relax admission control on `get` requests, allowing them to be served as long as there is slack time. The corresponding results are shown in Figure 15. Figure 15(a) and (b) show the most important benefit offered by RT-ParSec: RT-ParSec enables unsynchronized readers to run in parallel with all other requests. Thus increasing `get` request rate has little impact on tail latencies. For example, with 40 cores, tail latencies increase only 30% and 6% for `get` and `set` request respectively. On the contrary, because of lock contention for MCS and RW lock, both `get` and `set` request tail latency increase significantly. For instance, with 40 cores, `get` request tail latency increases 11 times and 9 times, for MCS lock and RW lock respectively (Figure 15(b)). `Set` request tail latency increases 8 times and 15 times, for MCS lock and RW lock respectively (Figure 15(c)). By sacrificing tail latency, MCS and RW locks increase their throughput. However, their throughput fails to scale with more than one socket as shown in Figure 15(a). In contrast, RT-ParSec not only has the highest throughput, but also scales better than the other two approaches.

VIII. Conclusions

We've studied SMR techniques for mediating shared data-structure access. Since existing techniques such as U-RCU are not a good fit for real-time systems, we introduced two implementations that demonstrate high schedulability. We introduced bounds on memory consumption that enable their use in real-time systems. Unexpectedly, they perform well even in the presence of write-heavy workloads. We have also shown their applicability to control the tail latency while maintaining high throughput in `memcached`. Though SMR-based data-structures do require a restrictive programming model, we believe they represent a promising mechanism for resource sharing for scalable real-time computation.

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

References

[1] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[3] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, 2012.
[4] Q. Wang, T. Stamler, and G. Parmer, "Parallel sections: Scaling system-level data-structures," in *Proceedings of the ACM EuroSys Conference*, 2016.
[5] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.
[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.
[7] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.
[8] B. Brandenburg, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *Proceeedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.
[9] B. B. Brandenburg and J. H. Anderson, "Reader-writer synchronization for shared-memory multiprocessor real-time systems," in *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.
[10] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, 1991.
[11] M. Arbel and H. Attiya, "Concurrent updates with rcu: Search tree as an example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14, 2014.
[12] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *EuroSys*, 2013.
[13] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole, "Rcu usage in the linux kernel: One decade later," *Technical report*, 2013.
[14] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, 1994.
[15] *Intel 64 and IA-32 Architecture Software Developers Manual, Volume 3B: System Programmers Guide*, Intel Corporation.
[16] "AUTOSAR os specification: http://www.autosar.org/."
[17] S. Chakraborty, S. Kunzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *DATE*, 2003.
[18] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.
[19] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.
[20] A. Wieder and B. B. Brandenburg, "On spin locks in AUTOSAR: Blocking analysis of fifo, unordered, and priority-ordered spin locks," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013.
[21] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley *et al.*, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," *Technical Report, TU Dortmund, March*, 2017.
[22] "Userspace RCU: http://liburcu.org/, retrieved 6/16," 2016.
[23] "Concurrency Kit: http://concurrencykit.org, retrieved 9/21/12."
[24] B. B. Brandenburg and J. H. Anderson, "Spin-based reader-writer synchronization for multiprocessor real-time systems," *Real-Time Systems*, vol. 46, no. 1, pp. 25–87, 2010.
[25] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.
[26] "SchedCAT schedulability test collection and toolkit: http://www.mpi-sws.org/bbb/projects/schedcat."
[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *NSDI*, 2013.
[28] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *NSDI*, 2013.
[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[30] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *ISMM*, 2006.

APPENDIX

A. Predictable Slab Allocator

We briefly introduce the memory allocator used in real-time SMR here. We adapted the slab allocator from ParSec. Most modern multi-core memory allocators [30] optimize for locality by using hierarchical allocation with thread-local memory caches, and then from within a global allocation pool. In addition, when memory is allocated on core i, and freed on core j, it is placed a "remote" freelist (called a "remote free") in core i's data-structures. When thread-local freelist cannot satisfy an allocation request, a memory is reclaimed from the remote freelist. Thus remote free is more expensive than local free, and would cause contentions on the remote freelist's cache lines. In the ParSec slab allocator, each core maintains separate remote freelists (also in separate cache lines) for each other socket. Although highly scalable, this optimization still suffers from contentions from the different cores within the same sockets. To eliminate this unpredictability, we change to per-core remote freelist. But we consolidate multiple remote freelist headers into a single cache line. In this way, different sockets still touch different cache lines, which guarantees scalability, but different cores from the same socket use different freelist headers within that cache line, which avoids contention, thus, provides predictability. As shown in Figure 10(b), CAS contention in original slab allocator causes some anomalies when measuring the worst-case overhead of free. This is because the list header is updated within a CAS loop whose number of CAS retries is unbounded due to contention. Such contention is eliminated after we use per-core remote freelist. More importantly, memory allocation performance is unaffected even with more list headers. This is because multiple list headers are consolidated into one cache line without extra cache line access.

B. Quiescence Frequency Calculation Algorithm

```
1 bool schedulability_test(TaskSet ts, int q) {
2     old_δ = 0;
3     calculate δ based eq (3) or (4)
4     while δ ≠ old_δ:
5         while there exist r_i ≠ old_r_i:
6             apply ILP analysis to bound mutex blocking
7             foreach τ_i in ts:
8                 calculate r_i using eq and (6) and (7)
9         while there exist Δ_i^r ≠ old_Δ_i^r:
10            foreach τ_i in ts:
11                calculate Δ_i^r using eq (8)
12        foreach τ_i in ts:
13            if (r_i > p_i): return false;
14        old_δ = δ;
15        calculate δ based eq (3) or (4)
16    return true;
17 }
18 int select_quiescence_period(TaskSet ts) {
19    // minimum (maximum) task period in ts
20    int min_q, max_q;
21    for qp in range(min_q, max_q):
22        if (schedulability_test(ts, qp) == true):
23            return qp;
24    return non-schedulable;
25 }
```

Fig. 16: quiescence period selection algorithm