Parallel Sections: Scaling System-Level Data-Structures [*]

Qi Wang
The George Washington University
interwq@gwu.edu

Timothy Stamler
The George Washington University
timstamler@gwu.edu

Gabriel Parmer
The George Washington University
Hewlett Packard Enterprise
gparmer@gwu.edu

Abstract

As systems continue to increase the number of cores within cache coherency domains, traditional techniques for enabling parallel computation on data-structures are increasingly strained. A single contended cache-line bouncing between different caches can prohibit continued performance gains with additional cores. New abstractions and mechanisms are required to reassess how data-structure consistency can be provided, while maintaining stable per-core access latencies.

This paper presents the Parallel Sections (PARSEC) abstraction for mediating access to shared data-structures. Fundamental to the approach is a new form of scalable memory reclamation that leverages fast local access to real-time to globally order system events. This approach attempts to minimize coherency-traffic, while harnessing the benefit of shared read-mostly cache-lines. We show that the co-management of scalable memory reclamation, memory allocation, locking, and namespace management enables scalable system service implementation. We apply PARSEC to both memcached, and virtual memory management in a microkernel, and find order-of magnitude performance increases on a four socket, 40 core machine, and 30x lower 99th percentile latencies for virtual memory management.

1. Introduction

Multi- and many-core systems are increasing in prevalence not only in server-based back-end systems, but also in embedded and real-time systems. Effectively harnessing this increasing parallelism enables continued performance increases as transistors are committed to increasing cores. However, writing software that can harness the capability afforded by an increasing number of cores is challenging.

Shared memory multi-processors, and multi-core systems, are a common architecture to harness parallelism. Though the shared memory abstraction is useful to provide a unified version of data-structures across cores, the requisite cache-coherency traffic can prohibit scalability. A cache-line modified on a single core, while accessed or modified in parallel on other cores, can result in significant overheads on the order of microseconds [27]. Consistency of such data-structures often involves lock hierarchies, and liveness of nodes requires reference counting. Both locks and reference counts have been significant impediments to scalability in existing systems [6].

Solving the problem of the scalable implementation of system data-structures is particularly important. If the computational infrastructure (*e.g.* kernels) doesn't scale, then application scalability can be artificially upper-bounded. Similarly, latency spikes suffered due to cache-coherency traffic within the system, contribute to the "heavy tail" in data-centers [19], and prevent usage in real-time systems. This paper presents Parallel Sections (PARSEC), an abstraction for parallel execution of operations on system-level structures. PARSEC attempts to avoid stores to shared cache-lines when accessing data-structures, until they are absolutely necessary (*e.g.* to modify a data-structure node). Toward this, PARSEC includes a form of Scalable Memory Reclamation (SMR) to provide memory reclamation for parallel systems, a modified locking API that inter-operates with SMR, and a mechanism for efficient and scalable lookup within data-structures.

We use SMR (not to be mistaken with *Safe Memory Reclamation* [23]) as a catch-all name for the family of techniques that provide delayed memory collection and reuse for parallel systems that lack garbage collection. SMR implementations include epoch-based memory reclamation [17], U-RCU [13], and hazard pointers [23]. In expanding on these techniques, PARSEC eschews globally-scoped locks, reference counts, and many other traditional techniques. In doing so, it attempts to provide a foundation for implement-

[*] 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.

ing system-level structures that will scale with increasing hardware capabilities.

Previous SMR techniques avoid reference counts on data-structure nodes by instead ensuring that data-structure accesses are surrounded with library calls to denote the start and end of data-structure accesses. The goal is that access to the data-structure proceeds in parallel, as opposed to with mutual exclusion. This makes it difficult to know when a node is being accessed, thus when its memory can be reclaimed once `freed`. To determine if a node is referenced, SMR implementations ascertain if *any* parallel thread is accessing the data-structure when the node is freed. If it is possible that a reference exists (a thread is accessing the data-structure), then the reclamation of the node's memory must be delayed. This technique is conservative as a parallel thread that is accessing the data-structure might not be accessing the specific freed node. However, this conservative technique enables implementations with no shared cache-line stores on the read-path. However, data-structure modifications must be synchronized with these parallel accesses, and with each other. RCU often serializes modifications to avoid races, and updates the data-structure atomically by first copying a node, then using a single atomic operation to update it in the data-structure (hence the name, Read-Copy-Update). However, the update path introduces shared cache-line modifications that inhibit scalability (as shown in Section 5).

PARSEC SMR uses *local* access to *global* real-time to scale, even in the presence of modifications, and a specialized locking API to synchronize modifications at the granularity of individual nodes. Real-time is used to establish an ordering between data-structure node deallocation, and each thread's access to the data-structure to prevent the reuse of the memory for a freed node that could still be referenced.

Contributions.

- We introduce PARSEC's SMR mechanism that aggressively avoids stores to non-core-local memory for scalable data-structure access and modification.
- We detail the design of the PARSEC abstraction, including the integration of SMR, memory management, and namespace management.
- We provide an evaluation of PARSEC's SMR implementation in isolation, and applied to a popular application (`memcached`).
- We detail and evaluate the application of PARSEC to the memory mapping management subsystem of a kernel, and compare to Linux.

This paper is organized as follows. Section 2 provides a survey of approaches to parallel data-structure construction. Section 3 introduces PARSEC and discusses its design, while Section 4 details PARSEC's use in a virtual memory mapping manager. Sections 5, 6, and 7 evaluate PARSEC, discuss related work, and conclude.

2. Parallel Data-Structures Background

PARSEC consists of two complementary components: first, a scalable mechanism for tracking the existence of references into data-structures to determine when memory can be freed without requiring stores to contended cache-lines; second, a collection of capabilities that use this facility to ease the implementation of complicated data-structures. These include scalable flat namespace lookup and management, and integration with fine-grained locking.

We introduce a simple abstract data-structure to understand achievable scalability properties. The data-structure, D consists of a set of nodes $\{n_0, \ldots, n_M\}$. Operations provided on the data-structure include:

- `observe`(n_i) which reads a node's memory;
- `modify`$(n_i, \ldots, n_j)$ which issues stores to a node's, or a series of node's memory;
- `add`(n_i) which adds n_i to D;
- `delete`(n_i) removes it from D, and frees it.

These operations are almost always paired with `lookup` (*e.g.* `modify`(`lookup`(p))) where

- n_i = `lookup`(p) where p is a predicate satisfied by a single node.

Operations executed in parallel are denoted as $o_0|o_1$. Given this, PARSEC aims to enable scalable execution of *all* operations *except* for:

- `modify`$(n_i, \ldots)$ | `modify`$(n_i, \ldots)$

If the same node is modified with high frequency by multiple cores, the unavoidable cache-line bouncing will prevent scalability. In such cases, message passing [4] is likely the main alternative.

Notably, the following *should* execute scalably:

- `modify`$(n_i, \ldots)$ | `modify`$(n_j, \ldots)$
- `observe`(n_i) | `observe`(n_i) | `observe`(n_j)

Modification to the structure of D via `add`(n_i) and `delete`(n_i) is intrinsically data-structure specific. PARSEC attempts to provide the tools to implement these scalably, and we investigate its application to complex data-structures in Section 5.

Unlike lookup structures as studied in [12], PARSEC mainly focuses on data-structures whose structure directly corresponds to the semantic organization of the represented data. For example, we apply PARSEC to implement a virtual memory subsystem that uses the recursive address-space model in [20]. For this structure, D is a set of nodes, each representing a specific page mapping into an address space. These mappings are organized into trees, which can be stored as hierarchical linked lists, where node deletion removes an entire subtree, and individual nodes are looked up in constant time. Similarly, virtual address tracking in Linux (via `vm_area_structs`) and `dentry`-based in-memory file-system organization are all similarly complex structures. PARSEC's techniques, especially for scalable memory reclamation, are complementary to the structures in [12] that require SMR.

2.1 Mutual Exclusion and Non-Blocking Data-Structures

Per-data-structure locks trivially ensure consistency (a semantically valid ordering of operations) among node operations by sequencing access to all nodes in D. However, this prevents scalable execution even of `observe`(n_i) | `observe`(n_j); Amdahl's law puts a hard limit on the achievable speedup. Fine-grained lock hierarchies use separate locks to protect access to each node, or groups of nodes. This enables parallel access to different nodes, and attempts to minimize the hold time of locks while performing `lookup`. However, as the number of cores increases, the simple modification of a lock's cache-line dwarfs the sequential hold-time of the lock, thus preventing scalability.

On the other hand, non-blocking data-structures [12] avoid the use of locks, instead directly relying on atomic instructions such as fetch-and-add (`faa`) and compare-and-swap (`cas`) for synchronization. These include both lock-free structures that guarantee progress for *some* operation, and wait-free operations that guarantee progress for *each* operation. The correctness of such algorithms are defined by *linearizability* – a point when the sequencing of parallel modifications to memory is guaranteed [18]. Non-blocking algorithms use SMR to provide scalable lookup (`observe`(n_i) | `observe`(n_i) | `observe`(n_j)). However, `add` and `delete` usually involve complicated, data-structure specific logic that considers all parallel interleavings.

ParSec uses a scalable implementation of SMR to provide scalable lookup, and a collection of wait-free algorithms to support the parallel processing of complex data-structures. ParSec locks integrate with SMR to enable locks to be taken only for the node(s) to be modified.

2.2 Scalable Memory Reclamation and ParSec

We posit that SMR is a *key to scalable data-structure access*. The intuition is that scalable performance is only possible when cache-coherency traffic is minimized, and such coherency traffic (especially for `observe`(n_i) | `observe`(n_i) | `observe`(n_j)) is avoided only by completely eliding stores to globally visible cache-lines. This implies that cores access D in parallel without any explicit synchronization. However, this complicates reference tracking and memory reclamation, which motivates SMR.

ParSec Interface. The ParSec interface for accessing a data-structure, and for memory management follow.

- `ps_enter(struct ps *)` – Declare the start of a section in which references to nodes in D exist.
- `ps_exit(struct ps *)` – Declare the end of that section. No references to nodes within D can remain.
- `void *ps_alloc(struct ps *, size_t sz)` – Allocate memory for use within the data-structure.
- `ps_free(struct ps *, void *node)` – Deallocate a node.

SMR implementations fundamentally exist to ascertain if a freed node that as been disconnected from D is still possibly accessed by a parallel thread. In addition to the equivalent functions to `ps_enter` and `ps_exit`, SMR implementations such as RCU and epoch-based memory reclamation implement a function, `synchronize` that is based on the concept of a *grace period* [13]. It is called at time t_1, and enables this thread to block waiting for a *grace period* between times $[t_1, t_2]$ where t_2 is a time when *no references exist* to nodes removed from D before t_1. A node n_i made unreachable within D, and freed at t_0 $(< t_1)$ can be reclaimed and reallocated after t_2.

ParSec does not expose this synchronization interface, instead using a modified version directly within its memory deallocation logic. Additionally, ParSec finds grace periods for the *time in the past*, when a specific node was freed.

- `int ps_quiesce(struct ps *, tsc_t` t`)` – Return *true* if all parallel threads have quiesced (*i.e.* a grace period has elapsed) at a point in time in the past, t. That is, return *true* if and only if
$$\forall_{c \in cores}\ t^c_{enter} > t \vee t^c_{enter} < t^c_{exit}$$
where t^c_{enter} and t^c_{exit} are the times (for example, measured using the CPU time-stamp counters) for core c's entry and exit from the parallel section. The time t corresponds to the when `ps_free` was called, and is compared against the `ps_enter` and `ps_exit` times of each core. It determines if each thread has `ps_exit`ed all parallel sections that could have been active at t.

In contrast to the traditional `synchronize`, freed memory is queued so this t is in the past, thus increasing the chance of quiescence. Additionally, `ps_quiesce` is *wait-free* – it does not block spinning on the state of other threads. This last fact means that memory is allocated if no memory can be reclaimed (*i.e.* a grace period hasn't elapsed) to avoid the significant performance and latency impact that blocking could incur.

In this paper, we use the SMR *read path* to denote threads that use only `ps_enter()` and `ps_exit()`, and the *update path* to include the freeing of memory and the use of `synchronize`/`ps_quiesce` after an update to D.

ParSec locks. With traditional locking, freeing a node (n_f) requires taking a lock containing nodes with references to n_f, removing the references, unlocking, and freeing n_f's memory. Parallel access to the data-structure are synchronized with the free via the lock. With ParSec SMR-mediated access to data-structures, `lookup`s in the data-structure are *not* synchronized with parallel frees, thus the locking API must be subtly different than a traditional interface (*i.e.* `POSIX pthreads`). Specifically, the *references* to a node to be freed are not protected by locks, thus to handle the case where a thread tries to acquire a lock on an already freed object, ParSec locks are specialized to check if the node has already been freed. Thus, the ParSec lock inter-

```
void observe(D, identifier) {
    ps_enter(D->ps);
    n = lookup(D, identifier);
    // process on the node
    ps_exit(D->ps);
}

void modify(D, identifier) {
    ps_enter(D->ps);
    node = lookup(D, identifier);
    if (ps_lock_take(node, node->lock)) {
        // modify the node
        ps_lock_release(node, node->lock);
    }
    ps_exit(D->ps);
}

void delete(D, identifier) {
    ps_enter(D->ps);
    node = lookup(D, identifier);
    if (ps_lock_take(node, node->lock)) {
        // cleanup & update D
        ps_lock_release_free(node, node->lock);
    }
    ps_exit(D->ps);
}
```

Figure 1. Example PARSEC pseudocode for `observe`, `modify`, and `delete`.

face includes slight variations on the typical take (lock) and release (unlock) operations to support interoperability with SMR.

- `int ps_lock_take(void *node, struct ps_lock *)` – Attempt to take a lock for a node, `node`. Return *false* if the node has been freed, thus cannot be locked.
- `void ps_lock_release(void *, struct ps_lock *)` – Release a taken lock for a node.
- `void ps_lock_release_free(void *, struct ps_lock *)` – Release a node's lock, and simultaneously free the node.

`ps_lock_take` will fail if the node being locked has been previously freed due to a parallel deletion. Such a case could result from a race between (a) chasing the pointer to the node and accessing the node, and (b) freeing and removing reachability to the node. `ps_lock_release_free` atomically marks the node as freed, and releases the lock. This ensures that any other thread contending the lock will fail to achieve access upon seeing it freed. The integration of locking with SMR is the key to this interface. The `node` is passed to each function as the memory header of the node includes the status of the node (*i.e.* free or allocated).

Figure 1 shows an implementation of three operations of D implemented using PARSEC. Each uses `lookup` to find the node of interest and either processes on it, modifies it, or frees it. Though modification and deletion use locks, they are used to provide mutual exclusion only on those nodes that require modification. In this example, these locks are not used to serialize access to the nodes for read-only (`observe`) access, nor for `lookup`s. As such, care must be taken to preserve intra-data-structure links (*i.e.* `next` pointers in linked lists) so that `lookup`s always find valid nodes, just as with RCU-enabled data-structures. We have found that this constraint is not generally difficult for system data-structures. File system hierarchies, virtual memory nodes, and hash-tables don't require complex changes in node linkages. PARSEC includes `lookup` facilities to map between a scalar identifier, and a node in the data-structure to ease this burden. To avoid locks in `observe`, `modify` must ensure that changes are made atomically (which RCU handles with copy and update). Where this atomic modification of nodes is difficult, locks can also be used in `observe`. Figure 1 motivates the specialized PARSEC lock API. A thread can attempt to `modify` a node while another thread has the node's lock and is in `delete`. Though `modify` followed a valid pointer to the node, when the lock is released, the node is no longer active – it has been freed. Thus, `modify` is notified of this case via `ps_lock_take`'s return value. Section 3.6 provides a complete treatment of the consistency properties of PARSEC.

3. PARSEC Design

PARSEC Goals. PARSEC focuses on the following goals:

G1 *Scalable-by-default parallel access.* For cases where data-structure consistency does not require explicit synchronization (for example, `observe`(n_i) | `observe`(n_j)), PARSEC must avoid these expensive operations.

G2 *Support for general workloads.* Many parallelization techniques make trade-offs in favor of read-heavy workloads. Though read-heavy optimizations are essential for efficient lookup, updating the structure is also quite common in systems code.

G3 *Integration with traditional consistency methods.* When synchronization on specific nodes within the data-structure is required, PARSEC must integrate with conventional synchronization techniques, which generally means locks. In this case, PARSEC should still provide scalable `lookup` of nodes within the structure before utilizing the traditional techniques.

G4 *Predictable latency.* Common operations in PARSEC should be wait-free, thus avoiding the introduction of low-probability, high-latency spikes. This is essential for real-time systems, but is additionally increasingly important in data-centers to avoid the "heavy tail" [19].

3.1 Scalability of Existing SMR Techniques

To understand the PARSEC implementation, we briefly discuss existing SMR techniques. Epoch-based scalable memory reclamation [17] and U-RCU [13][1] take core-local snapshots of a global value (epoch counter and quiescence period, respectively) in their equivalent of `ps_enter`. These snapshots must be committed to memory before accessing the data-structures to guarantee that other cores properly observe the subsequent access to the data-structures[2]. This path

[1] We use the default `urcu-mb` variant that can be used by general, preemptible applications.

[2] A memory barrier is used to avoid store-to-load re-orderings. This paper assumes an architecture such as x86 that supports the Total Store Order (TSO) memory model. Extrapolation of the algorithms to looser models is relatively straight-forward.

is scalable in the read-only case as threads read only global values in cache-lines in the *shared* state, and store only to local cache-lines. Consequently, observe(n_i) | observe(n_i) | observe(n_j) should scale.

Read-side scalability. When a thread synchronizes (i.e. when D is updated, and nodes are freed), it first updates the global value. This has the effect of invalidating all other core's cached data for that cache-line, and causing the corresponding coherency traffic when they next load the global variable. This impedes the scalability of the *read* path, especially for high update rates. For example, with observe(n_i) | delete(n_j), observe latency increases significantly (Section 5.1).

Update-side scalability. synchronize iterates through all other core's core-local structures, and it must block (or spin) until some predicate is satisfied by their local variables. Though this predicate is different for epoch-based SMR and U-RCU – progression through an epoch, or a quiescence period, respectively – the impact is the same: loads are issued to core-local cache-lines that are frequently modified by the other cores. This causes significant cache-coherency traffic, and impedes scalable performance of the update path. In practice, this means that observe(n_i) | delete(n_j) and delete(n_i) | delete(n_j) are *not* scalable. SMR techniques often focus on read-mostly workloads for this reason.

Factors that impact SMR latency. Minimizing tail-end latency is important in real-time and cloud systems [19, 27]. The goal is to both provide *wait-free* behavior for each operation in the SMR interface, and to minimize the coherency traffic required to make progress so that the latency of each operation scales. We assume that the minimum latency for a synchronize is lower-bounded by the latency a of a read-side section. Existing implementations either block waiting for all threads to elapse a grace-period, or don't guarantee progress by returning quiescence failure. In the former case, update latency is compounded by the coherency-traffic latency.

3.2 ParSec Scalable Memory Reclamation

ParSec introduces a SMR technique that alleviates cache-coherency traffic hot-spots on the read and update paths discussed in Section 2.2 using a number of techniques. Key to these techniques is the use of a simple mechanism provided by recent processors: consistent, or invariant Time-Stamp Counters (TSC). The time-stamp counters are monotonically increasing (modula wrap-around) cycle counts read with specific instructions (*i.e.* rdtsc on x86). Invariant TSC guarantees a constant cycle rate, regardless of each core's operating frequency. Invariant TSCs are available on recent (all Nehalem and later) Intel x86 processors (as indicated by the CPUID instruction, and described in the Intel 64 and IA-32 Architecture Software Developers Manual, Volume 3B: System Programmers Guide, 17.14.1), on which a constant cycle rate is guaranteed regardless of ACPI P-, C-. and T-states. The key to ParSec SMR is that the *thread-local time-stamp counter provides a global ordering of events* that otherwise would requires synchronization through shared cache lines. The events we are primarily concerned with are (1) when a thread is in a parallel section, and (2) when a node is made unreachable within D. A total ordering between these events provides all the information required to know if reclamation of the node is possible. The synchronization API is slightly modified to provide the more general notion of *quiescence at a specific TSC in the past*. In ParSec, the TSC for which we must determine that no existing references can exist corresponds to when a specific node was freed. This enables a number of optimizations within the implementation of ps_quiesce that cache *other core's* quiescence values to avoid loads to remote memory.

Using rdtsc for globally ordering events on x86. ParSec uses TSCs to provide *local* access to a *global* ordering of events. On all Intel x86 processors running Linux that support an invariant TSC that we tested on, the TSCs on different cores are offset by a small, but constant amount (on the order of a couple thousand cycles). The OS can synchronize TSCs using the IA32_TSC_ADJUST Model Specific Register (MSR). If necessary, the ParSec library could account for any remaining offset between cores with a constant by populating an offset table at initialization. Given that the absolute differences on our systems are small, we apply a simple offset for comparisons between core's TSCs.

Compensating for micro-architectural effects. As rdtsc is not a serializing instruction, and the rdtscp serializing variant can have significantly more overhead, ParSec must compensate for the possibility that out-of-order execution will result in an inaccurate TSC. When entering into a parallel section, the TSC is guaranteed to represent a time *before* data-structure access due to a memory barrier that ensures stores (including the TSC being written) commit. When a node is freed, the TSC is again taken with the possibility of pipeline reordering. In this case, reordering the rdtsc later in the instruction stream is not a correctness issue as it cannot result in reusing the node earlier. Reordering the rdtsc *earlier* in the execution stream can result in a TSC that overlaps with actual node access. This is avoided by using either memory barriers, or by adding an additional offset to TSC comparisons at least as long as the maximum reordering latency. ParSec, motivated by simplicity, uses the former.

Simplified ParSec SMR implementation. Figure 2 depicts a simplified version of quiescence detection in ParSec. thdid() returns the current thread id, and mem_barrier() is a memory barrier to flush the store buffer.

Entrance into, and exit from the library are marked by simply recording the corresponding cycle counts into that thread's per-thread structure[3]. Quiescence detection includes simply iterating through all other core's structures and wait-

[3] We use per-thread and per-core interchangeably. Each system will chose their most appropriate partitioning.

```
struct parsec {
    struct thd_data {
        tsc_t enter, exit;
        struct quiesce_queue quiesce_q;
    } thd_info[NUMTHDS];
};

void ps_enter(struct ps *ps) {
    struct thd_data *t = ps->thd_info[thdid()];
    t->enter = rdtsc();
    mem_barrier();
}

void ps_exit(struct ps *ps) {
    struct thd_data *t = ps->thd_info[thdid()];
    t->exit = rdtsc();
}

// A simple ps_quiesce (without any optimizations)
int ps_quiesce(struct ps *ps, tsc_t tsc) {
    for (int i = 0 ; i < NUMTHDS ; i++) {
        struct thd_data *other = ps->thd_info[i];
        if (i == thdid()) continue;
        if (other->exit > other->enter) continue;
        if (other->enter > tsc)         continue;
        return 0; // not quiesced
    }
    return 1;
}

void ps_free(struct parsec *ps, void *node) {
    struct thd_data *t = ps->thd_info[thdid()];
    mem_header(node)->tsc = rdtsc();
    mem_barrier();
    enqueue(&t->quiesce_q, node);
    while (keep_reclaiming()) {
        node = dequeue_peek(&t->quiesce_q);
        if (!ps_quiesce(ps, mem_header(node)->tsc))
            break;
        reclaim(dequeue(&t->quiesce_q));
    }
}
```

Figure 2. A simplified implementation of PARSEC.

ing for them to either (1) exit their read-side section (*i.e.* when their exit timestamp is great than enter), or (2) enter the library *after* the time at which we're waiting for quiescence. This implementation demonstrates a key benefit to using TSCs: the read-side accesses only thread-local data, and the machine's TSC. Unlike other SMR techniques, there is no global value that is read on the read-side, that is modified when quiescing. In this simplified version, however, update does *not* scale. Each core that calls `ps_quiesce` generates cache-coherency proportional to the number of cores in the system.

`ps_free` in Figure 2 adds the memory to be freed into a quiescence queue, to be reclaimed (and re-used) at a later point. It depicts how the freeing of a node must interact with the PARSEC SMR. The memory is freed by placing it into a quiescence queue (`quiesce_q`) that is naturally ordered by TSC, then the system attempts to reclaim memory that has quiesced. Note that `dequeue_peek` does not remove the node from the queue, just retrieves it. Each invocation of `ps_free` there is a limit (encoded in `keep_reclaiming`) on how many nodes are reclaimed to bound the cost of `ps_free`.

Though omitted here for brevity, a simple proof by contradiction ensures that two threads both attempting to quiesce cannot deadlock.

3.3 PARSEC SMR Implementation Optimizations

This implementation is simple, but has a number of inefficiencies that we remedy in the PARSEC implementation. These optimizations include:

1. **Avoid `rdtsc` in PARSEC exit.** The overhead of the `rdtsc` instruction is around 20 cycles on our processor. Though this cost as been declining over the years (on Pentium 4 processors, it was over 80 cycles), we optimize for using only a single `rdtsc` on entrance. Upon exit, we simply set the `exit` time to `enter + 1`. The key observation that enables this is that quiescence only compares `enter` and `exit`, thus to signal that a thread has exited PARSEC `exit` need only be greater than `enter`.
2. **Leverage quiescence already performed by other threads.** A particularly impactful optimization is that each core stores its last computed quiescence value in the same cache-line as enter/exit. This is directly used in future calls to possibly completely avoid accessing other thread's cache-lines. Further, when walking through other thread's structures, *their* computed quiescence TSC is used if it is sufficient. This optimization means that `ps_quiesce` likely will *not* iterate through all other thread's structures, thus avoiding significant coherency overhead.
3. **Avoid redundant calculations of quiescence.** Iterating through each other thread's local structures causes cache-coherency traffic in the likely case that the remote cache-line is in the modified state. Additionally, they will be bounced back to the remote core when modified next. To optimize, we locally cache data for each of the remote threads. This data includes their last-observed `enter` and `exit` values, and the time when this thread cached those values (stored as `cached`). When determining if a thread has quiesced at a given time, t, the local values of `enter` and `exit` can be used if `cached` $\geq t$. The trade-off this approach makes is in memory usage. Each thread caches data for each other thread, thus requiring a total of $O(N^2)$ memory where N is the number of threads. For OS-level code, we believe this overhead is acceptable as such tracking can be performed per-core rather than per-thread, thus limiting the amount of tracked data.
4. **Spread coherency traffic across cores.** Given the previous optimizations that avoid accessing *all* thread's structures in `ps_quiesce`, thread `0`'s structure is disproportionately accessed by other cores. A simple optimization is applied wherein each thread starts accessing other thread's structures at a fixed offset based on its thread id. This alleviates the hot-spot of the first thread structure's cache-line.

In this way, the PARSEC runtime trades memory consumption for wait-free allocation assuming that there is no resource starvation. In Section 5.3, we see the case where

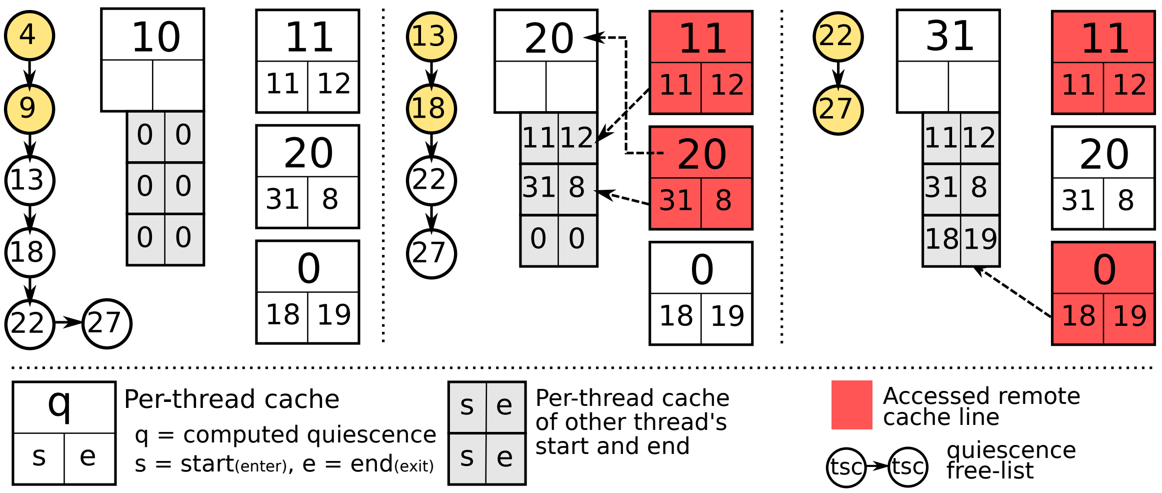

Figure 3. Example of a thread (on the left in each frame) attempting to reclaim various nodes on the quiescence free list. The *start* and *end* values are the PARSEC *enter* and *exit* TSCs. The left frame shows the second optimization were the last computed quiescence time (10) is used to reclaim the first two nodes in the free-list without accessing other thread's cache-lines. The middle frame shows the thread iterating through the first two other threads, caching their start and end values for future use (for the third optimization). The second thread has computed a quiescence time (20) that is sufficient to free the next two items, thus avoiding the coherency traffic for touching the third thread's cache (via the second optimization). The right frame shows the use of the local cache of the other thread's start and end values (third optimization). We must access the first thread's values since our cached values were saved before the time we're attempting to quiesce at (using the `cached` value). We can avoid accessing the second thread as the cached start is *after* the time we're attempting to quiesce at (22), even though it could currently be in its parallel section (end $<$ start). When accessing the third thread, we notice that it has exited its parallel section (end $>$ start), thus we can set our quiescence to the most recent possible TSC (31), and reclaim the remaining two nodes.

such starvation does happen. Wait-free allocation/deallocation is an important guarantee (**G4**) as PARSEC seeks to be *predictable*, thus avoiding latency spikes that can result in outlier response times that are prohibitive not only in embedded systems, but also in data-centers [19]. Figure 3 gives an example of some of these optimizations.

Nested parallel sections and granularity. Different parallel sections (*i.e.* using different `struct parsec`) maintain separate TSCs, and separately manage free memory. Thus, parallel sections can be nested. An alternative implementation of PARSEC could also enable nested instances of a single parallel section by maintaining a count of the number of times the parallel section has been nested, and maintaining the PARSEC TSC meta-data for the outer-most instance. In general, we believe that parallel sections should be relatively coarse (one per-system data-structure, or per-system). Not only does this save memory on PARSEC structures, but also simplifies system implementation. The main downside of this is that if some system paths take significantly longer than others, they might increase parallel section access latency.

3.4 PARSEC Memory Management

PARSEC includes a slab allocator [5] for allocating and freeing memory with a typical interface. A slab allocator is used as opposed to a replacement for the more general `malloc/free` interface as system data-structures commonly use well-defined nodes with consistent sizes. Each allocation includes a header consisting of (1) a TSC field that, when non-zero, indicates that the memory has been freed, (2) a pointer used to track the memory in the quiescence list, and (3) a pointer to the containing slab. The TSC is used in PARSEC's SMR facilities. PARSEC's slab allocator uses many of the accepted techniques for scalable memory management [25] including thread-local allocation lists, and optimized treatment of "remote frees".

Figure 2 includes a simplified version of the code for `ps_free`. A quiescence queue holds all memory that has been `ps_free`d, but has not been reclaimed for reuse. All such memory is tagged with the time it was freed which is used to assess quiescence. The quiescence queue is naturally sorted by quiescence times. The longer this queue is, the higher the chance that more memory will have quiesced, having accessed the fewest remote cache-lines. This leads to a memory versus scalability trade-off. We haven't thoroughly evaluated this trade-off, and we leave that as future work.

3.5 PARSEC Locking and Consistency

Node modifications must synchronize with respect to each other, and they must also be cognizant of concurrent lookups. This requires that care is taken when modifying pointers that connect nodes, as lookups proceed in parallel and follow those pointers. A number of techniques to consider this synchronization already exist. These include: (1) Direct atomic updates to single word fields, where such a simple solution is applicable. (2) Data-structure pointers can be updated using Read-Copy-Update which copies a node, modifies the copy, and uses a single atomic instruction to make an updated copy of a node visible. Many RCU data-structures require mutual exclusion between writers, thus prohibiting concurrent modification. For specific data-structures [1], this limitations has been lifted. Read-Log-Update (RLU) [21] is a recent technique to atomically make multiple data-structure alterations visible using fine-grained locks, and a means to coordinate the redirection of readers. Both approaches benefit from the scalable TSC-based SMR in PARSEC. In this work, we consider another possibility: locks at the finest possible granularity – one for each node. This is a practical solution for complicated data-structures and is the most accessible and common to programmers.

Section 2.2 describes the locking API for PARSEC that integrates locks with SMR. As previously described, the pointers that bind nodes together must be valid at all points (even after a node is freed) so that `lookup`s don't fail. RCU data-structures share this constraint, thus are directly applicable. Modifications to a node must be atomically visible either by using direct atomic instructions, copying and updating as in RCU, or by using PARSEC locks in `observe`.

The different operations on the data-structure D synchronize with each other using different means:

- `lookup`(n_i) | `modify`(n_i) – `lookup` issues only data-structure loads, while `modify` requires lock acquisition. Synchronization between these operations is implicit: pointers and data required by the `lookup` procedure must be treated as immutable, or atomically update-able (*i.e.* using atomic instructions with RCU). Access to more complex data requires a lock. However, this lock is at the finest-possible granularity, causing cache coherency traffic *only* to threads accessing the same node.
- `modify`(n_i) | `delete`(n_i) – The locking API in Section 2.2 enables explicit synchronization between lock acquisition and memory free. Lock acquisition can fail if a contending thread frees the node, and a thread that owns a lock frees the node by both releasing the lock, *and* freeing the memory. The node is marked as free before the lock is released, and a contenting thread takes the lock, then checks if the node is freed.
- `lookup`(n_i) | `delete`(n_i) – PARSEC's SMR ensures that a node's memory is not reclaimed until all concurrent `lookups` have completed. This maintains a consistent state within the node, compensating for races between deletion, when a node is removed from the structure (*i.e.* made unreachable), and concurrent reads.

Both `modify` and `delete` require a thread to own the lock for an object, so they are trivially consistent with other like operations.

The modifications made while a node is locked, and the locking requirements for different modifications, differ for each structure. For example, a tree structure might require that a parent is locked to perform modifications to its immediate children.

3.6 PARSEC Namespace Management

PARSEC includes facilities for namespace management. Namespace management is included in PARSEC not by necessity (as with memory management), rather because they are commonly used for `lookup` in system data-structures, and are efficiently implemented using PARSEC SMR. A PARSEC namespace manages a flat scalar namespace of identifiers (ids) and is used to associate these ids with nodes in the data-structure. System-level, request-driven code must often translate between an opaque identifier (*i.e.* a safe token exposed to user-level), and a node within a kernel data-structure. Allocation and management of file-descriptors has been shown [6, 10] to be required for system scalability. The Virtual Memory (VM) subsystem provides another example . When page-faults occur within Linux, the `vm_area_struct` corresponding to the address at which the fault occurred is located via a lookup in a red-black tree protected via a read-write lock. The synchronization around this lookup structure inhibits scalability [8, 9].

PARSEC namespaces address the scalability problems of traditional implementations by providing lookup that does not require stores to non-local memory, and scalable allocation/free of identifiers using the same PARSEC techniques as memory allocation. The namespace facilities are implemented using lock-less radix tries to put tight bounds on lookup latency (proportional to the height of the tree). Leaf entries contain identifier structures that are tracked as memory (*i.e.* they share the same header as slab memory objects). When an identifier is deallocated, it is only reused *after* quiescence has been achieved for the time when it was deallocated. This enables references within the data-structure to the ids of other nodes. Separate allocation lists track different sizes of id extents within a namespace, thus enabling the allocation of multiple contiguous ids. This is useful, for example, when multiple contiguous pages are requested by `mmap`.

Some applications might not utilize the namespace facilities of PARSEC (for example, see Section 5.2). However, they provide a reasonably common functionality, with a focus on scalability by using the very support that PARSEC provides for reclaiming deallocated namespace identifiers. Additionally, they emphasize a focus of PARSEC: synchronization-free lookup. Thus we believe they are an integral aspect of the PARSEC abstraction.

3.7 Custom Grace Periods in PARSEC

Grace periods around memory reclamation are based on the TSCs for all thread's entry into and exit from parallel sections. PARSEC provides the ability to provide a function that provides a customizable semantic for grace periods. When the allocator ascertains that a normal grace-period has passed, and is going to reuse a previously allocated item, this function is queried to determine if it can currently be reused. This is used in conjunction with the grace periods required by the SPECK [27] system to manage kernel resources, as detailed in Sections 4 and 5.3.

4. PARSEC Example: Virtual Memory Mapping Management

We investigate the application of PARSEC to the management of process virtual address spaces within a the virtual memory management code of a μ-kernel. This application will be evaluated in Section 5, but we introduce its structure here. We focus on a traditional μ-kernel interface for the recursive model of address spaces [20]. The general semantics of this structure have been adopted in capability management systems as well [14], showing its broad relevance. The API enables the granting, aliasing, and (recursive) revocation of memory pages across address spaces. The API follows:

- `vaddr_t grant(size_t npages)` – Request the allocation of `npages` new pages into the requesting address space. Return the virtual address (`vaddr_t`) of the base of the mapping.
- `void alias(pid_t from, vaddr_t mapfrom, pid_t to, vaddr_t mapto, size_t npages)` – Create shared memory (aliasing `npages` physical frames) from the calling address spaces into another specified address space.
- `void revoke(pid_t p, vaddr_t mapat, size_t npages)` – Remove *all* mappings that have been aliased from the given `mapat` `npages` virtual addresses in the processes `p`, including any recursive mappings.

This API is generally used in a μ-kernel for servers providing some service to clients to transfer mapped memory to the clients, or vice-versa. When a process terminates, or reclaims memory, it `revoke`s `grant`ed and `alias`ed pages.

The `revoke` call warrants an intelligent, tree-based backing data-structure consisting of `mapping` structures. This structure is often called the "mapping data-base". Each alias creates a parent/child relationship, and revoking a mapping must unmap the entire subtree (all children, grandchildren, etc...). Each physical frame is represented by a `frame` structure which is the root of each of these mapping trees. A similar structure is used to track memory in L4 μ-kernels [26], and Linux tracks similar relationships using its `vm_area_struct`s.

To maintain consistency for parallel accesses to this data-structure, the most naive implementation uses a lock for the entire structure. A lock per-process for general operations is insufficient as mappings between components refer to each other. A lock per-frame is reasonable, but a mapping between a process and virtual address to the actual frame requires a data-structure with consistency guarantees. This is required to implement O(1) `alias` and efficient `revoke`. As with many tree structures, fine-grained locking can be used. However, this often requires either root-first walks through each frame's mapping tree to adhere to parent-first lock-ordering protocols, or per-mapping node reference counting to enable direct pointers to mapping nodes, and per-virtual address space locks.

Consistency in the mapping-database. Though the tree structure can be walked directly, PARSEC namespaces are used to enable direct lock- and reference counting-free lookups of mapping nodes. At this point, traditional concurrency mechanisms such as locks, or atomic operations on simple pointers can be used. If traversals of the mapping tree can proceed in parallel with mapping update (*e.g.* modification of the list of child alias mappings), then care must be taken to make modifications to the data-structure such that readers will not see the structure in an inconsistent state. This is directly analogous to how RCU-based lists are managed [13], for example. In our case, we err on the side of ease of implementation, and rely on PARSEC namespace lookups to find the target `mapping` node, and only traverse the tree structure once we've taken a lock for the node. PARSEC namespaces are used to allocate/deallocate, and lookup virtual addresses (one namespace per-process), and physical frames.

The use of `mapping` locks is similar to traditional fine-grained locking structures *except* that the lock is taken only on the node where it is required. Importantly, read-side parallel sections enable wait-free, scalable and efficient lookup through the multiple namespaces to find the proper `mapping` node, and locks are only used at that *finest-granularity*. Parallel modifications impacting different mapping structures will scale (*i.e.* $\texttt{modify}(n_i) \mid \texttt{modify}(n_j)$). Given that proper memory management discourages parallel `munmap` or `mmap` to the same virtual address (as one of the parallel requests must fail), this scalable access is the common-case .

5. Experimental Evaluation

The goals of our evaluation of PARSEC include:

- to micro-benchmark and understand the performance and scalability properties of the PARSEC SMR techniques, and compare them existing mechanisms,
- assess the scalability of PARSEC's memory and SMR facilities when integrated into an application; and
- to evaluate the full complement of PARSEC's capabilities including the management of namespaces and different quiescence periods in a system-level service.

5.1 Evaluation of PARSEC Quiescence and Reader Overhead

Here we conduct a set of micro-benchmarks to evaluate the overheads of PARSEC quiescence and "read-side" sections. We compare against general-purpose User-space RCU (henceforth referred to as U-RCU), epoch-based SMR implementation, read-write locks (specifically the brlock variant that optimizes for core-local read locks), and MCS locks as a baseline for scalable locking. We utilize the epoch, brlock, and MCS locks that are implemented in the Concurrency Kit [7]. The goals of this evaluation are to (1) determine the overhead and scalability of these ParSec operations, and (2) to compare them to comparable operations of existing approaches. The U-RCU implementation is general in that it can be used in preemptible, user-level code, and only relies on the user to harness the previously introduced, simple API. This implementation is described in Section 2. We also compared against the version of U-RCU that uses cross-core signals for quiescence, but found that the update-path results very quickly degenerated to 10s of ms latencies, so we don't include those results here. Both the U-RCU and epoch SMR implementations generate cache-coherency traffic on both the read and the update paths when there are any updates.

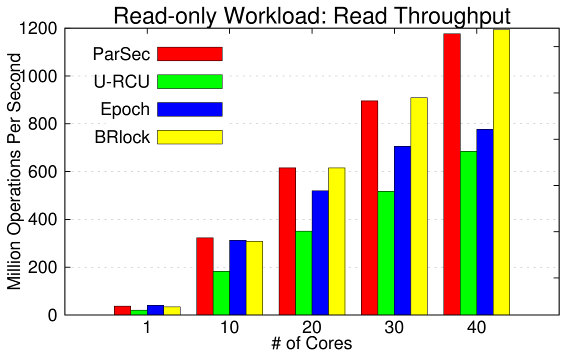

Figure 4. ParSec μ-benchmark: Read-only Workload

Benchmark configuration. To evaluate the performance of U-RCU and ParSec, we conduct experiments with no data access inside the read-side section, to measure only the pure overheads from each technique. Two types of operations are considered:

1. read operations, which evaluates the overhead of the read-side logic which includes `ps_enter` and `ps_exit` for ParSec, `rcu_read_lock` and `rcu_read_unlock` for U-RCU, and comparable functions for epoch; and
2. update operations, which generally involve triggering the `synchronize` interfaces for U-RCU and epoch, and involves the freeing and allocation of a node (a cache-line in size) in ParSec.

These update operations emulate the synchronization needed by a data-structure modification, *e.g.* delete a node from a linked-list and then insert a new one, or adding a node to a data-structure. The allocation operation in ParSec detects quiescence; but different from U-RCU, the quiescence in ParSec is relative to the time point in the past for the memory at the head of the quiescence queue. We also include brlock and MCS locks in the evaluation as a point of context.

Since an update operation involves quiescence state detection, which often comes with cache-coherent traffic, it impacts not only update performance, but also read performance. To evaluate the overhead of read / update under different workloads, we generate request traces with different read / update ratios. For each trace with a specific average update ratio, 10 million requests are generated following a uniform distribution. To avoid expensive random number generation function calls within the evaluation loop, we pre-generate the trace of which operation (read vs. update) should be conducted, and each thread iterates through a separate partition within that trace. Thus more memory traffic is generated than for the SMR operations in isolation. We do not believe this significantly impacts the results as most overhead results from coherency traffic.

Evaluation platform. All experiments in the paper are conducted on a system consisting of four sockets of Intel Xeon E7-4850, each one has 10 cores, clocked at 2.0 GHz. Hyperthreading is disabled, leading to 40 cores in total. Benchmark is run on different core counts (namely 1, 10, 20, 30 and 40) to measure scalability. The minimal number of sockets is always used.

Benchmark Results. Figure 4 shows the read throughput of both ParSec and U-RCU with a read-only workload. The "Speedup" is relative to the cost of ParSec on a single core, thus should only be directly related to the ParSec lines. With no updates, both U-RCU and ParSec achieve near-linear scalability as no shared cached-lines are modified. The timestamp based quiescence-state tracking of ParSec has a straightforward read operation, which contributes to lower overhead. All operations take under 100 cycles, so this overhead would likely be dwarfed by data-structure latencies in a real system. This experiment mainly confirms that the overhead for using `rdtsc` in ParSec does not preclude its competitiveness, even in read-only workloads.

Figure 5 includes the read / update throughput for three different update ratios: 10%, 50%, and 90% update requests respectively. All the graphs reporting the update throughput use a log-scale due to the wide gap between different system's throughput. With 10% updates, ParSec maintains roughly the same read performance as the read-only case, because the update operation in ParSec rarely loads the cache-lines that are modified by readers (in their core-local structures) because of the optimizations described in 3.4. This is to say that the update operation likely doesn't traverse all other core's structures, and when it finds a quiescence TSC, it can apply it to significant portions of its already-freed memory. With 50% and 90% updates, the read-path throughput does decrease (from around 1150 million requests/second down to 950) as it becomes more common for the reader to write the TSC into a cache-line in the shared state. Due to the quiescience optimizations, the update operation in ParSec is relatively efficient and scalable, even up to 90% updates.

On the other hand, the read performance of the other SMR techniques is negatively impacted because the update operation frequently modifies shared cache-lines, and moves into a shared-state, cache lines that are soon to be stored to. The throughput of the update operations of the other SMR techniques is also significantly impacted by increased parallelism. This behavior quickly degenerates into each core

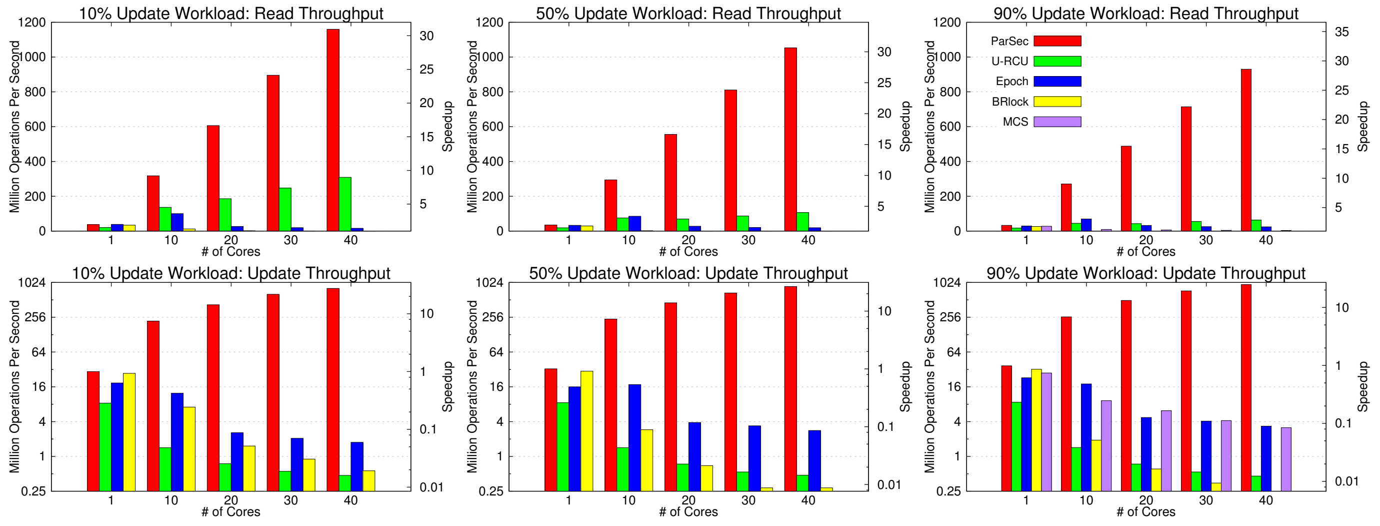

Figure 5. ParSec μ-benchmark: Read / Update Throughput

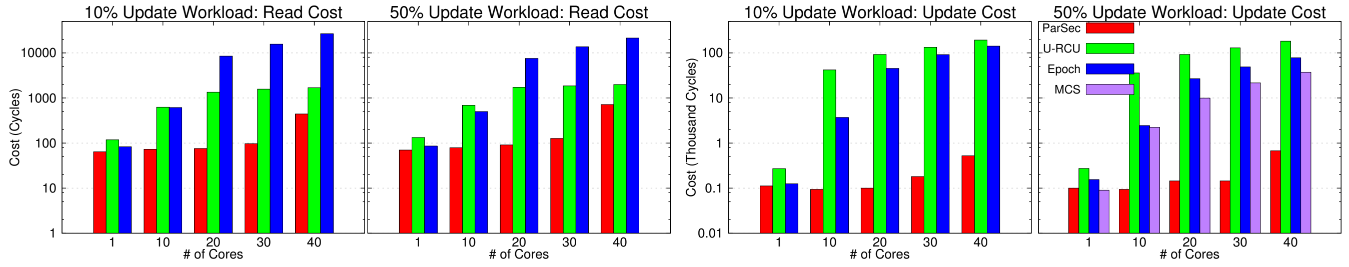

Figure 6. ParSec μ-benchmark: 99th Percentile Latency for Read / Update Workloads

loading cache-lines that are soon to be modified within the core-local structures of each other core. Neither U-RCU nor epoch-based SMR implementations are expected to scale with significant update rates, however, it is impressive that epoch maintains performance on-bar with MCS locks. Both brlocks and MCS locks are present for context for the performance and scalability of popular read-write, and scalable spin-locks.

Figure 6 shows the 99th percentile latency of read and update operations with 10% and 50% updates. The most important factor here in maintaining a low latency is that ParSec is wait-free by design. This improves the latency and predictability of both read and update operations. This wait-free design, compared to the `synchronize` operations in U-RCU and epoch, combined with the update-path optimizations in ParSec result in a significant reduction in tail-end latency.

5.2 Memcached

To evaluate the use of ParSec in an existing system, we use `memcached`, an in-memory cache commonly used in data-centers [24] to cache data-base results. We believe this application is representative of many caching workloads (*e.g.* the directory cache in kernels), and gives guidance for how to adapt an existing code-base to use ParSec.

As pointed out in [15], the stock memcached has a number of significant scalability bottlenecks caused by locks, reference counting, and LRU list maintenance. These bottlenecks are sufficient to prevent scalable execution (see "Stock MC" in Figure 7). The main optimization for scalability that `memcached` performs is to use a concurrent hash-table where each hash-table entry (and associated linked-list of items) has a separate lock. Thus, before applying ParSec to memcached, we first improve the scalability by using the following existing techniques from [15]: (1) use the CLOCK cache replacement policy instead of the LRU in the stock implementation which prevents the modification an item's cache-line *every* time it is accessed; and (2) use MCS locks to replace the simple (unscalable) spin-locks. Both techniques improve the scalability considerably. However, scalability is still rather limited – as shown by the "Modified MC" in

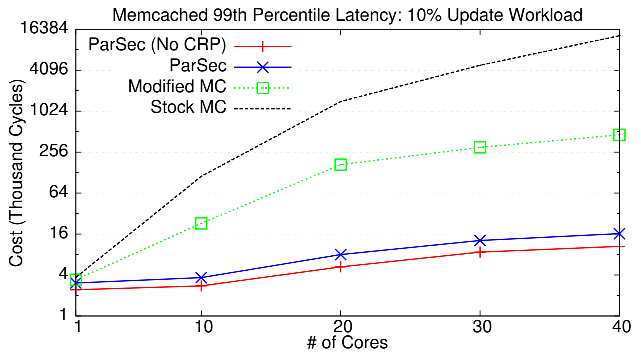

Figure 8. Memcached 99th Percentile Latency in log-scale. CRP = Cache Replacement Policy

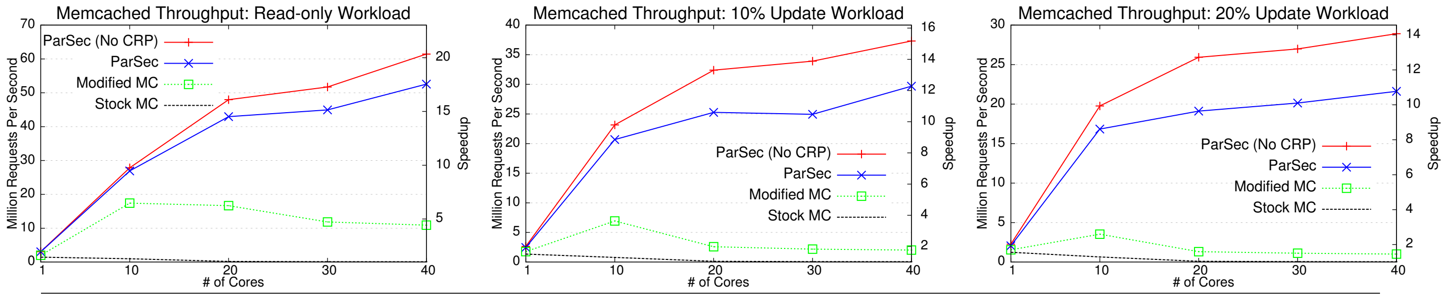

Figure 7. Memcached Throughput w/ Different Workloads. CRP = Cache Replacement Policy

Fig. 7. PARSEC is used to further increase the scalability of the system by utilizing its SMR and memory management facilities.

PARSEC memcached. Cache memory is managed by PARSEC. New items are allocated from the PARSEC allocator, while freeing an item returns it to the allocator. Same as in the stock `memcached` implementation, updating an existing key is a 2-step operation: 1) allocate an item for the new key, link it into the hash table, and after that 2) release the unlinked old item. One notable difference in implementation of the PARSEC-based `memcached` is the cache eviction: when eviction is required by the addition of new items, instead of the existing behavior which reuses that memory for the new item, the evicted item is returned to the free-list; then another allocation is made to get memory for the new item. The amount of memory for the PARSEC allocator is relatively small ($<$ 1,000 items), but the list is long enough to enable wait-free quiescence, thus efficient reclamation of items. Our initial plan was to replace the hash-table with a manually managed namespace from PARSEC, however the benefit in this case was not significant enough to warrant the additional changes it would require.

After changing memcached to use PARSEC managed memory, an additional improvement is made to alleviate the contention caused by the CLOCK replacement policy (using a lock-protected linked-list): instead of using a single CLOCK list, 64 separate lists are created utilizing part of the PARSEC namespace management. Hashing is used to determine the target CLOCK list. Higher partitioning than 64 didn't result in improvements.

The result of these changes is that `get` requests are satisfied without any lock acquisition or synchronization beyond `ps_enter` and `ps_exit`. Updating the hash-table's linked lists still requires the acquisition of the hash-table entry's lock.

To evaluate the PARSEC-based memcached, we compare four different implementations: 1) stock memcached, 2) modified memcached (w/ MCS lock and CLOCK replacement), 3) PARSEC memcached (w/ all improvements above) and 4) PARSEC memcached with no cache replacement policy. The last one removes the cache replacement policy entirely, to show the upper bound of an ideal, scalable replacement policy. All implementations are based on `memcached` version 1.4.22. Similar techniques to [15] are used in the evaluation: traces with `get` / `set` requests are generated using YCSB [11] with the specified `set` percentage, following a zipfian distribution. Each trace contains 10 million requests (16-Byte key and 32-Byte value), which will be partitioned by multiple threads on different cores. To concentrate the evaluation on cache scalability, the throughput is measured locally in the single `memcached` process, bypassing all the network related layers. Memory capacity is made sufficient for the cache contents. One thread per-core processes the generated requests by invoking `memcached` in a tight loop.

Fig. 7 shows the cache throughput with 3 different configurations: read-only, 10% `set` and 20% `set`. The 20% updates is already higher then the typical real-world workload reported in [24]. The speedup shown in graphs is relative to the single-core PARSEC case. Note that in the read-only case, PARSEC with no cache replacement policy is a completely read-only workload – the only memory writing required in the PARSEC-based `get` operation is for the CLOCK policy. In that case, the performance is limited by the memory bandwidth, instead of cache-coherency. In all cases, PARSEC memcached tracks the no replacement policy cases reasonably. Higher `set` ratios impact PARSEC throughput, but not as dramatically as other approaches. Fig. 8. depicts the 99th percentile latencies for the systems. The PARSEC-based approaches have lower latency by a factor of 32 on 40 cores.

5.3 Virtual Memory Management

To evaluate the full range of SMR, memory management, namespace management, and customizable grace-periods, we adapt the virtual memory manager in the COMPOSITE component-based system [27] to use PARSEC. For the purposes of this paper, SPECK is the kernel of COMPOSITE, and is a modern μ-kernel with virtual memory mapping management defined in a user-level component (the process-equivalent in COMPOSITE). That component has access to the kernel's APIs that enable it to map and unmap to the page-tables of the components it manages. This virtual memory management's component data-structures have already been described in Sections 4. We focus on extending Wang et al.'s work on SPECK as the kernel uniquely uses internal grace-periods to track kernel data-structure liveness, and TLB entry liveness. PARSEC's unique facilities for managing resource around different grace periods (Section 3.8) enable the use of the kernel's scalable operations. This appli-

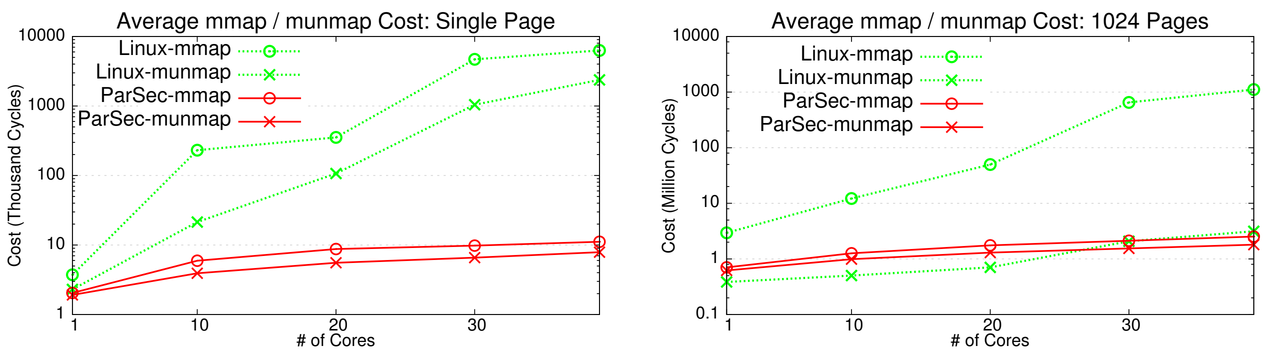

Figure 9. Average Cost of `mmap` / `munmap` in Linux and COMPOSITE

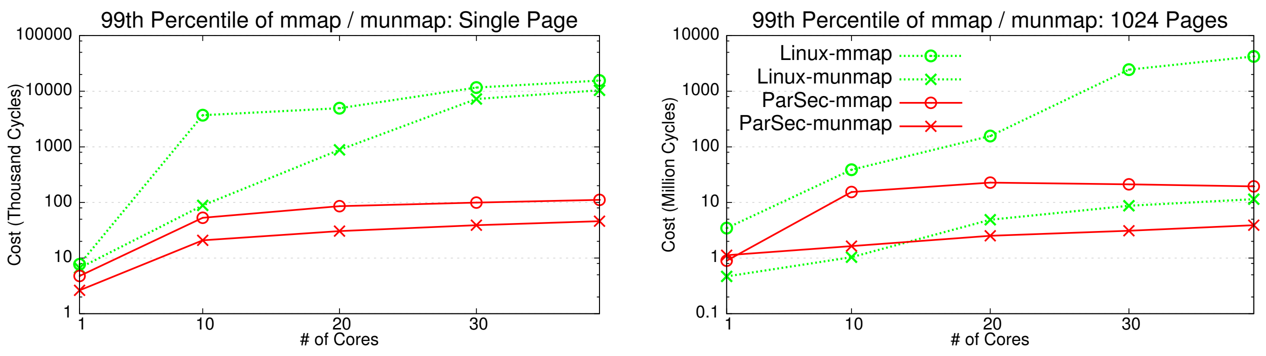

Figure 10. 99th Percentile Latency of `mmap` / `munmap` in Linux and COMPOSITE

cation of PARSEC harnesses the flexible support for multiple quiescence periods as required by the SPECK μ-kernel.

This subsection evaluates the PARSEC-based memory management component in COMPOSITE. The data-structure used are detailed in Section 4. PARSEC is used to support SPECK-specific supports the configuration of grace-periods used to reclaim different resources (Section 3.8). SPECK requires the following grace periods to be used when activating and deactivating different kernel resources: (1) kernel memory resources cannot be reused until a grace period has elapsed commensurate with a kernel worst-case execution time (maximum latency), and (2) the physical memory and virtual address for a mapping, when *unmapped*, cannot be reused until all TLBs have been flushed. Though this can be done eagerly, COMPOSITE schedules predictable TLB flushes, which removes the need for non-scalable TLB shootdowns. A grace-period for these resources is commensurate with the inter-TLB flush latency (10ms in our system). Namespace management in PARSEC is used to manage physical memory allocation, virtual memory allocation (per-client address space), and to lookup the namespace for a specific client. Note that SPECK supports safe user-level management of kernel memory [27] as in [14]. However, the kernel relies on the manager to decide which of the physical frames the manager has access to, to use for any given mapping. Memory for the `mapping` structures is provided using PARSEC's memory management functions.

We capture the memory mapping trace from Metis [22] – a shared-memory MapReduce library – that computes word inverse indexing (similarly used in [9]). As the memory allocator [25] used by Metis never returns memory to OS, memory unmap is not performed by the program. To evaluate both `mmap` and `unmap`, we simply unmap all the mapped regions after executing the trace once, and repeat the trace again. The trace is based on a input size of 128MB, which allocates ~650MB of memory during MapReduce. Linux kernel 3.10.10 is used to run the same traces for comparison. Memory is touched after each `mmap` call to ensure physical memory is committed. The map/unmap trace is repeated 10 times (*i.e.*, for each core count in Linux and COMPOSITE). The last core is reserved for system tasks, leaving 39 cores used for the benchmark.

The trace contains two types of mapping requests: single page and 1024 pages. Less than 5% of the requests are for 1024 pages. Fig. 9 reports the average cost of each case separately, while Fig. 10 reports the 99th percentile latencies. In all cases, PARSEC in COMPOSITE has lower overhead than the Linux equivalent, with the exception of unmapping on ≤ 20 cores. This is because the Linux kernel delays the expensive operations (*e.g.* TLB shootdown) to future `mmap`s, which dominate the overhead in Linux.

In COMPOSITE with PARSEC, the overhead also increases with more cores, but less severely. Though there is little shared cache-line contention, overheads still increase due to the sharing of other resources (*e.g.* LLC and memory bus), which are triggered by accessing a large amount of memory concurrently. Linux arbitrates parallelism over `vm_area_structs` and the page management structures using conventional means which carry significant scalability overheads [6]. Though it has improved in avoiding TLB shootdowns when possible, the ParSec-based VM manager still demonstrates performance at higher core counts with an improvement of more than a factor of 100.

6. Related Work

Mission: scalability. Both the Laws of Order [3], and the scalability commutivity rule [10] provide a insight into a core factor that determines an API's or a data-structure's scalability. Generally, if operations within the API commute, they *can* be implemented scalably. If not, they require expensive operations that modify shared cache lines. ParSec demonstrates that a scalable version of SMR can be implemented, and uses that SMR (along with configurable grace periods) to reclaim and reuse namespace descriptors, a key to enabling the scalability of file-descriptors and virtual memory pages. This is, in a sense, automating the advice from [10].

Quiescence and memory reclamation. Read Copy Update (RCU) and its user-level variant in U-RCU provide quiescent-state detection [13]. RCU focuses on low overheads for read-mostly workloads, while modifications must be require atomically visible and often rely on quiescence. Epoch SMR [17] uses the passage of epochs, and per-epoch limbo-lists to track grace-periods. To increase RCU scalability, predicate RCU [2] attempts to lower the cost of quiescence by altering the RCU API. Citrus trees [1] are binary trees that use RCU and enable concurrent writers. ParSec SMR scales well and uses the notion of ascertaining quiescence at a specific time in the past, paired with timestamped frees to provide wait-free SMR.

Other implementations of SMR exist including quiescent-state detection on context-switches in non-preemptible Linux kernels, and the non-default API in U-RCU that uses quiescence (i.e. `urcu-qsbr`) Such approaches can often remove the memory barrier in `ps_enter`, and rely on system, or application-specific calls to declare a lack of references into the data-structure at which points memory is reclaimed. ParSec only relies on applications to wrap data-structure accesses in enter/exit calls, thus using a more conventional API.

Parallel data-structures. ASCY [12] provides a survey of lock-free, lookup data-structures. They provide guidance for efficient parallel data-structure implementation by relating sequential memory accesses to the parallel structure's. RLU [21] enables multiple data-structure modifications that atomically activate, thus co-existing with parallel lookups. Both of these papers rely on SMR and some of the specialized ASCY structures also use locks.

Parallelism support within operating systems. Key scalability bottlenecks in existing systems such as Linux often center around liveness monitoring (*e.g.* reference counting), and concurrency control (*e.g.* locks) [6]. Tornado [16] uses distributed reference counts and clustered objects to provide a programmatic framework for scaling. However, such techniques often still rely on shared cache-lines which ParSec avoids. Corey takes a different approach by enabling applications to control the namespaces in the kernel used to access resources. ParSec shares the observation that namespace lookup and management significantly impact scalability, but provides a *framework* for the implementation of scalable system services.

RCU relies on SMR and uses atomic updates of copied, then modified, versions of nodes to handle updates. Updates are often mutually serialized, and along with the overheads of the RCU SMR quiesce, the focus RCU is on read-mostly workloads. Read-Log-Update (RLU) [21], on the other hand, updates structures using a mechanism to make multiple stores atomically visible along with locks to coordinate writers. In this paper, we've focused on accommodating a consistency API similar to traditional locks. Data-structures that require atomic modifications to many nodes would benefit from RLU. Both RCU and RLU rely on SMR and could benefit from ParSec SMR.

Bonzai trees [8] use RCU to implement scalable parallel lookup of virtual memory structures in the Linux kernel. Radix VM [9] applies a radix trie structure to the VM lookup data-structures with embedded locks to shrink the granularity of concurrency control. Scalability research in L4 [26] introduces dynamic lock granularity on mapping trees, with custom, kernel-specific lookup mechanisms. ParSec provides a mechanism enabling generic data-structures to have fine-grained locking with scalable lookup and memory management. It is informed by the previous work, and expands on those ideas (*e.g.* radix tries and RCU).

7. Conclusions

This paper introduces the *parallel sections* abstraction, that enables the implementation of data-structures with finest-grained application of consistency mechanisms such as locks, while enabling parallel lookups with no explicit synchronization. ParSec SMR provides memory reclamation with liveness based on a *global* ordering of events, based on *local* access to time. ParSec namespace management enables scalable descriptor allocation, and efficient lookup. All ParSec operations are wait-free, thus avoiding latency spikes due to contention. Results show that ParSec SMR scales better and has higher performance for read and update heavy workloads than existing SMR mechanisms. A ParSec-enabled `memcached` has a throughput between 2.5x and 5x higher, with 99th percentile latencies 30x lower. A virtual memory mapping manager in a μ-kernel that uses ParSec outperforms Linux by up to a factor of 100x.

Acknowledgments. We'd like to thank our shepherd, Robert Morris, and the anonymous reviewers for their feedback that has significantly improved the quality of this paper. We'd also like to thank Samy Bahra, the author of the Concurrency Kit, for his useful feedback throughout the process.

References

[1] M. Arbel and H. Attiya. Concurrent updates with rcu: Search tree as an example. In *PODC*, 2014.

[2] M. Arbel and A. Morrison. Predicate rcu: An rcu for scalable concurrent updates. In *PPoPP*, 2015.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011.

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

[5] 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.

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010.

[7] ck. Concurrency Kit: http://concurrencykit.org, retrieved 9/21/12.

[8] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. In *ASPLOS*, 2012.

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

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[12] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[13] 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.

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

[15] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, 2013.

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

[17] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12), 2007.

[18] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. ISSN 0164-0925.

[19] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *SoCC*, 2012.

[20] J. Liedtke. On micro-kernel construction. In *Proceedings of SOSP*, 1995.

[21] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[22] Metis. Metis: https://pdos.csail.mit.edu/archive/metis/, retrieved 10/22/15.

[23] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 2004.

[24] 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.

[25] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM*, 2006.

[26] V. Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *SIGOPS Oper. Syst. Rev.*, 2007.

[27] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer. Speck: A kernel for scalable predictability. In *RTAS*, 2015.