Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2014-019 September 16, 2014

OpLog: a library for scaling update-heavy data structures

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

OpLog: a library for scaling update-heavy data structures

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

Abstract

Existing techniques (e.g., RCU) can achieve good multi-core scaling for read-mostly data, but for update-heavy data structures only special-purpose techniques exist. This paper presents OpLog, a general-purpose library supporting good scalability for update-heavy data structures. OpLog achieves scalability by logging each update in a low-contention per-core log; it combines logs only when required by a read to the data structure. OpLog achieves generality by logging operations without having to understand them, to ease application to existing data structures. OpLog can further increase performance if the programmer indicates which operations can be combined in the logs.

An evaluation shows how to apply OpLog to three update-heavy Linux kernel data structures. Measurements on a 48-core AMD server show that the result significantly improves the performance of the Apache web server and the Exim mail server under certain workloads.

1 Introduction

Systems software often maintains data structures that are frequently updated but rarely read. LRU lists, for example, are usually updated with every access, but read only during eviction. Similarly, Linux reverse page maps are updated whenever a physical page is mapped, but read only when swapping pages out to disk or truncating a mapped file.

Update-heavy data structures can pose a multi-core scalability bottleneck. Locks protecting them force serialization; even updates with atomic instructions incur expensive transfers of contended cache lines. As we show in §3, the Linux kernel suffers from such scalability bottlenecks at high core counts.

An existing approach is to apply updates to per-core instances of the data structure, thereby avoiding serialization and cache line transfers [10, 13]. This problem has been studied for the specific case of a counter "data structure": SNZI [15] and Refcache [9] demonstrate how to merge updates to per-core counters that are frequently updated but rarely read.

The programmer work required to modify a data structure so that it is factored into per-core instances can be substantial. The hard part is usually designing the read strategy: the per-core data must be combined to produce a consistent value, one that could have been produced by the original serial implementation. As an example, consider reading a global LRU list that has been factored into per-core lists. Finding the globally least-recently used object is likely to require comparing timestamps of the head elements of all the n per-core lists. This kind of code has to be designed and written by hand; we do not know of any general-purpose, scalable tools to help programmers factor data structures while preserving consistency.

This paper presents a general-purpose approach for scaling data structures under update-heavy workloads, and an implementation of this approach in a library called OpLog. "Update-heavy" means that most operations are updates without return values. The key insight is that such updates can be deferred until required by operations whose return values require reading the global state. OpLog's approach is to initially record each update in a per-core log. Before a read operation, OpLog applies the updates in all the logs to the data structure to bring it up to date. OpLog time-stamps each log entry with the current CPU cycle counter (§6.1) and applies entries from the logs in timestamp order.

The OpLog approach has three main benefits. First, it ensures that updates scale, since each update only appends to a local per-core log, incurring no lock contention or cache line transfers. Second, logging and deferred execution can be transparently applied to a wide variety of existing operations. Finally, the per-core log representation makes some optimizations easy which might be hard to apply to ordinary data structures, particularly absorption.

OpLog faces some challenges. First, applying it to an existing data structure might be difficult. Second, it might impose high overhead for log manipulation. Third, storing all the log entries, for every object in the system, might require a large amount of memory. OpLog addresses these challenges using the following ideas.

First, OpLog provides an API that is easily layered on top of an existing object type. OpLog intercepts method calls to each object. It turns each update call into an append to a local per-core log associated with the object. For a read method, OpLog first calls the underlying update methods recorded in the logs in timestamp merge order, and then calls the underlying read method.

Second, the programmer can direct OpLog to combine operations in the log when allowed by the semantics of the data structure. For example, this might allow OpLog

to implement a set "delete" operation by removing a previous "insert" from the local log. This "absorption" can keep the logs short, reducing the cost of the next read operation.

Third, OpLog reduces memory use by adaptively enabling itself only when needed. That is, if a data structure receives many updates, OpLog stores the updates in per-core logs; otherwise, it maintains only the underlying data structure, and immediately applies updates to it.

To demonstrate that OpLog can improve the performance of real systems, we implemented two prototypes: one in C++ for user-space applications, and one in C for the Linux kernel. We applied OpLog to several data structures in the Linux kernel: the global list of open files, the virtual memory reverse map, the inotify update notification queue, and the reference counter in the directory name cache. A performance evaluation on a 48-core AMD server shows that these uses of OpLog improve the performance of two real applications. OpLog removes contention on reference counts in the directory name cache, encountered by the Apache web server, allowing Apache to scale perfectly to 48 cores when serving a small number of static files. OpLog also removes contention on the virtual memory reverse map, encountered by the Exim mail server, improving its performance by 35% at 48 cores over a lock-free version of the reverse map under the workload from MOSBENCH [7].

The rest of the paper is organized as follows. §2 reviews related work. §3 provides examples of update-heavy data structures in the Linux kernel. §4 outlines OpLog's approach, and §5 describes OpLog's library interface. §6 describes OpLog's implementation and discusses modern processor support for synchronized clocks. §7 evaluates OpLog. §8 identifies some areas of future work, and §9 concludes.

2 Related work

The best route to scalability is to eliminate unnecessary inter-core sharing altogether. For example, Linux's `CLONE_FILES` eliminates the contention involved in file descriptor allocation by giving threads separate descriptor spaces. OpLog targets cases where it is not convenient to eliminate sharing.

OpLog is in spirit similar to read-copy update (RCU) [26], which is heavily used in the Linux kernel [27]. Both provide infrastructure to help programmers implement low-contention shared data structures. OpLog targets update-intensive data structures, while RCU targets read-intensive data structures.

Flat combining reduces contention costs [18] by having one core apply the updates of all the cores contending to write a datum. Flat combining avoids performance collapse from lock contention and can improve locality. OpLog delays applying updates until needed by a read operation, which gives it a higher degree of batching and locality than flat combining. The delaying also enables OpLog to perform optimizations such as absorption and therefore scale better (e.g., near-ideal scaling if most operations are absorbed); flat combining has no equivalent of absorption. In flat combining, every operation requires accessing a shared memory location, which can be a scalability bottleneck. OpLog defers execution of updates, and uses timestamps to establish a global order of operations in a scalable way, to eliminate accesses to shared memory locations when logging an update.

Tornado's Clustered Objects [16], which were also used in K42 [3], help programmers improve scalability by allowing dynamic choice of representation. Each "virtual" object can have a per-core partitioned, distributed, or replicated implementation at runtime, depending on the workload. The programmer must implement a strategy for combining the per-core data consistently, e.g. using an invalidation and update protocol [16]. OpLog's logs, on the other hand, provide a single consistency strategy that works for any data structure. By focusing on a specific consistency strategy, OpLog is able to implement optimizations such as batching and absorption. OpLog shares some implementation techniques with Clustered Objects; for example, OpLog's per-core hash table to avoid using per-core logs for all object instances is similar to the per-processor translation tables in Clustered Objects. K42's online reconfiguration [30] could be used to switch between different implementations of a data structure at runtime, such as switching between OpLog when updates are common and a more traditional implementation when updates are rare.

OpLog can be viewed as a generalization of distributed reference counter techniques [3, 4, 14, 25]. Applying OpLog to a reference counter creates a distributed counter similar to Refcache [9]. OpLog makes it easy for programmers to apply the same techniques to data structures other than counters.

OpLog applies some distributed systems ideas to multicore software. Per-core logs of operations are similar to Bayou's per-device operation logs [31]. OpLog also borrows Bayou's re-ordering of certain logged operations. EPaxos [29] and Tango [5] defer execution of operations with no return value by adding them to a log. OpLog applies this idea to shared-memory data structures, and uses synchronized clocks to order operations without explicit communication.

Barrelfish [6] and Fos [34] organize a multicore kernel as a set of semi-independent cores interacting only via message-passing; their goal is to reduce contention by eliminating shared-memory data structures. OpLog is aimed at multi-core software that shares data.

3 Problem

In order to illustrate situations where OpLog can help, this section explores the scalability of three example data structures in the Linux 3.8 kernel: the reverse page map (`rmap`), the filesystem change notification queues (`inotify`), and the pathname lookup cache (`dcache`). These data structures are update-heavy, are limited by multi-core contention under some workloads, and are often updated in situations where the update can be deferred: that is, they are good candidates for OpLog. As a preview, we compare the "stock" Linux implementations of these examples with OpLog implementations (described in §4–6). We focus on Linux because it has been extensively optimized for multi-processors [7].

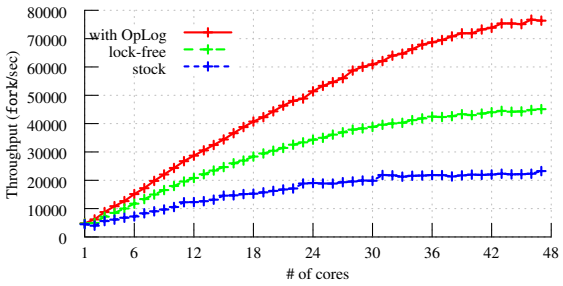
The stock implementations use spinlocks. Linux's spinlocks suffer from performance collapse under contention [8], and their cost dominates performance for our benchmarks. Scalable locks [28] or lock-free data structures would increase performance. They would provide only limited scalability, however; cores would still be bottlenecked by contention to write shared data. We demonstrate this by re-implementing several of the examples using atomic instructions rather than locked critical sections. These lock-free implementations are limited by contention from concurrent writes to shared data, and OpLog is still faster.

We present results from a 48-core x86 machine, composed of 8 6-core AMD Opteron Istanbul chips. Experiments on a large Intel machine generate similar results. All measurements in this paper are averages of three runs; there is never much variation.

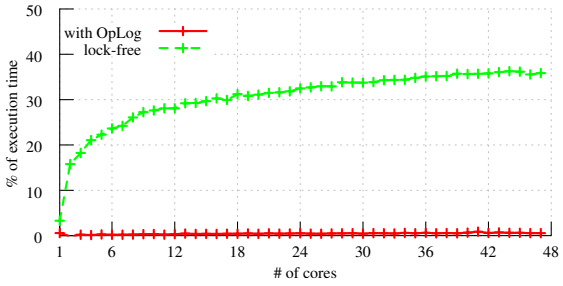
Example: reverse page map (rmap). Linux's reverse map (rmap) records, for each physical page, all page table entries that map that page. Linux reads these reverse mappings when it truncates a file or swaps a physical page out to disk, in order to find (and then delete) all page table entries that refer to the deleted page(s). The `fork()`, `exit()`, and `mmap()` system calls update the rmap, but don't need to read it.

The Linux designers have heavily optimized the rmap using interval trees [21, 22, 24]. Each file has an associated interval tree protected by a lock. A file's interval tree maps intervals of the file to virtual address mappings. Each virtual address mapping maps one or more pages from the file. The Linux rmap implementation can become a bottleneck when many processes simultaneously try to map the same file. For example, simultaneous creation of many processes is likely to cause contention for the lock protecting the interval tree of the libc library, because creating a process entails creating virtual address mappings for libc and inserting the mappings into the libc rmap interval tree.

The `fork` benchmark measures the performance of the rmap. The benchmark creates one process on each core. Each process repeatedly calls `fork` to create a child process that immediately calls exit. This stresses the rmap, because `fork` and `exit` modify rmap interval trees by inserting and removing virtual address mappings.



(a) Performance.



(b) Percent of execution time spent waiting for cache line transfers in the rmap.

Figure 1: `fork` benchmark results, exploring reverse map (rmap) scalability.

Figure 1(a) shows the result. The x-axis shows the number of cores and the y-axis shows the throughput in forks per second. The "stock" line shows the performance of stock Linux; the limiting factor is contention for the lock on the libc rmap interval tree. The libc rmap interval tree lock is most contended because the benchmark processes have more virtual address mappings for libc than any other files, so `fork` and `exit` insert and remove from the libc rmap interval tree more than other interval trees.

In order to find out whether the locks were the only reason for rmap's failure to scale well, we wrote a new rmap that replaces the interval trees with lock-free lists. The "lock-free" line in Figure 1(a) shows that its performance achieves about twice the performance of the stock rmap. The limiting factor is the cost of fetching cache lines that have been recently updated by another core. Figure 1(b) shows the percent of total execution cycles the `fork` benchmark spends waiting for cache lines in the lock-free rmap code. With 40 cores, roughly 35% of the execution time is spent waiting for cache line transfers.

To show there is room for improvement, the "with OpLog" line shows the performance of the rmap implementation described in §4.3, Figure 7, and §5.2. OpLog

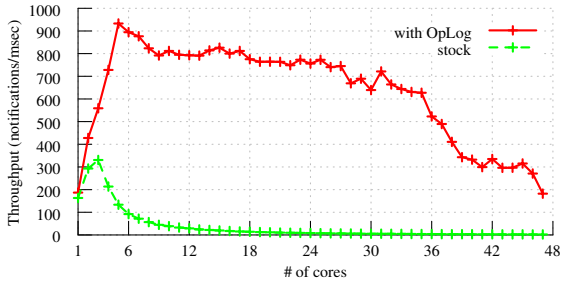


Figure 2: `inotify` benchmark results.

arranges this version's data so that most writes are to cache lines used by just one core; thus the writes don't require cache line invalidation and movement, and are cheap. As a result the OpLog rmap is considerably faster than the lock-free rmap. §7 explores the performance of the OpLog-based rmap further.

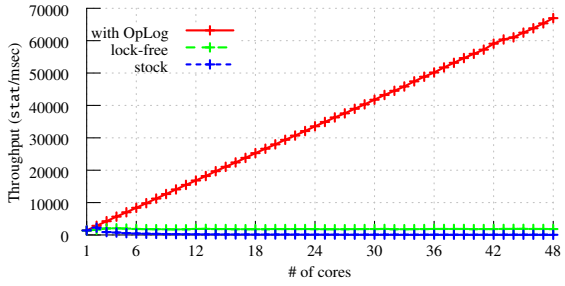
Example: inotify queues. Inotify is a kernel subsystem that reports changes to the filesystem to applications. An application registers a set of directories and files with the kernel. Each time one of them is modified, inotify appends a notification to a queue, which the application reads. File indexers, such as Recoll [2], rely on inotify in order to re-index new or modified files.

Inotify is interesting because the notification queue is both updated and read from frequently. The kernel serializes updates to the queue with a spinlock. This helps preserve the correct order when operations occur on different cores, e.g., one core creates a file, while another deletes it.

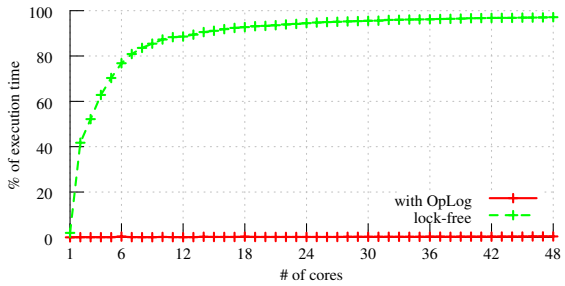
Our `inotify` benchmark creates a set of files, registers the files with inotify, and creates a process that continuously dequeues notifications. The benchmark then creates one process on each core, each of which repeatedly modifies a random file in its non-overlapping subset of the files.

Figure 2 presents the performance of the benchmark. The "stock" line shows that throughput increases up to three cores, then collapses due to spinlock contention caused by multiple cores simultaneously trying to queue notifications. The "with OpLog" line demonstrates that even with frequent reads it is possible to improve the queue's performance using OpLog. Performance decreases with six or more cores because each OpLog read must check more per-core logs.

Example: `dcache` and pathname lookup. In order to speed up pathname lookups, the Linux kernel maintains a mapping from directory identifier and pathname component to cached file/directory metadata. The mapping is called the `dcache`, and the entries are called `dentry`s. Each file and directory in active or recent use has a



(a) Performance.



(b) Percent of execution time spent waiting for cache line transfers in reference counting.

Figure 3: `stat` benchmark results, exploring pathname lookup (`dcache`) scalability.

`dentry`. A `dentry` contains a pointer to the parent directory, the file's name within that directory, and metadata such as file length.

The `dcache` is well parallelized using RCU [23], with one exception. In order to allow lock-free consistent reads of multiple slots of a `dentry`, pathname lookup reads a per-`dentry` generation number before and then after reading the `dentry`'s fields; if the number changes, the `dentry` has changed, and the kernel must re-try the reads. If the kernel decides to use the `dentry`, it must increment the `dentry`'s reference count; it uses a spinlock to ensure the atomicity of the generation number check and the increment.

The `stat` benchmark creates one thread per core; all threads repeatedly call `stat` on the same file name. Figure 3(a) shows the resulting performance. The x-axis shows the number of cores and the y-axis shows throughput in stat operations per millisecond. The "stock" line shows that Linux's implementation achieves no parallel speedup; it is limited by the spinlock mentioned above.

To see whether the spinlock was the only scaling limit, we implemented a lock-free scheme. The lock-free version packs the generation count and reference count into a single cache line, and uses an atomic compare-and-exchange instruction to conditionally increment the reference count. This lock-free implementation also scales poorly (see the "lock-free" line). The reason is contention in the compare-and-exchange; the cores spend much of

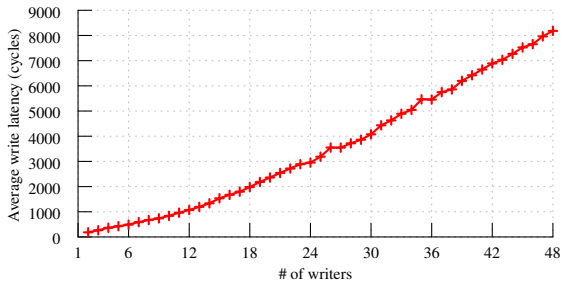


Figure 4: Average time for a write to a shared cache line, as a function of the number of simultaneous writing cores.

their time re-trying the instruction. Figure 3(b) supports this point, showing the fraction of total run-time that is spent in the reference counting code; with more than a few cores, this code accounts for almost all of the total time.

To see how much room there is for improvement, the graphs include the performance of a version that uses a distributed reference counter [9] optimized using OpLog (see §5.2). It increments a local reference count; if it then sees that the generation count changed, it decrements the local count and retries. The "with OpLog" line in Figure 3(a) scales nearly perfectly, and shows the potential for improvement by avoiding contention on locks and shared cache lines.

3.1 The cost of write contention

The examples above spend much of their time waiting to fetch cache lines during writes; the amount of time increases with the number of contending cores. This section explores the cost of contended writes, and notes that even a single contended write can take as much time as an entire system call.

Figure 4 shows the time to complete a single write when a varying number of cores are simultaneously writing the same location. Each core uses a `cpuid` instruction to wait for its write to complete. As the number of writers increases from 2 to 48, the average time to execute a store increases from 182 to 8182 cycles. We also measured the average time to execute a contended atomic increment; it is only about 1.5% longer than for a `mov`.

Figure 4 shows that the time for a write is roughly linear in the number of cores writing the same location. The fundamental reason is that the cache coherence system serializes the writes: only one core at a time is allowed to write, and the others must wait their turn [11, 19]. Between one turn and the next, the cache line must be moved from one core to the next; this movement takes between 124 cycles (between cores on the same chip) and 550 cycles (between cores on different chips that are separated by multiple interconnect hops). The slope in Figure 4 is roughly 160 cycles per core, which is consistent with these inter-core transfer times.

System call	Latency
open	2618 cycles
close	770 cycles
mmap	849 cycles
dup	387 cycles

Figure 5: Single core latency of several Linux system calls.

The write times Figure 4 can easily dominate time spent in the kernel, in the sense that even a single highly contended write can take much longer than a typical (uncontended) system call. Figure 5 illustrates this with some typical uncontended system call times from Linux 3.8.

4 OpLog design

OpLog helps update-intensive data structures scale well on multi-processors by reducing contention. It is intended to be easily and transparently layerable on existing data structure implementations, to make few assumptions so that it works with a wide variety types, and to support a number of programmer-enabled optimizations. This section describes OpLog's design, and concludes with an example of its use.

4.1 The OpLog approach

OpLog timestamps each update operation, appends the operation to a per-core log, and only merges and executes the logged operations before the next read operation. Almost any "deferrable" update can be safely logged for later execution, which allows OpLog to avoid having to know anything about the underlying data type. A deferrable update is one whose semantics allow its side-effects to take place after it returns. Blind update methods typically have "void" return values, so that the caller can't immediately tell whether the method call did anything.

By default OpLog executes logged updates in temporal order. For example, consider a linked list. If a process logs an insert operation on one core, migrates to another core, and logs a remove operation, the remove should eventually execute after the insert. OpLog relies on timestamps from a system-wide synchronized clock to tell it how to order entries in different cores' logs[1]. This ordering ensures linearizability, making OpLog compatible with existing data structure semantics. Reading the clock is fast, much faster than (for example) maintaining the shared-memory data required to provide causal consistency.

When a developer applies OpLog to a type, OpLog interposes on this "underlying" type's methods. For each object of the OpLog-augmented type, OpLog creates an internal instance of the underlying type, and a log for each core. OpLog requires the programmer to indicate which

[1]Modern Intel and AMD processors provide synchronized clocks via the `RDTSC` and `RDTSCP` instructions; see §6.1 for more details.

methods of the underlying type are deferrable updates, and which involve reads. Instead of executing an update method immediately, OpLog appends an entry indicating the method, its arguments, and the current time to the local per-core log. OpLog's interposed read methods first execute the operations from all the logs in timestamp order, calling the underlying update methods on its internal instance of the object, and then calls the underlying read method.

Update operations in OpLog are fast because they incur no contention. Each core appends to a log dedicated to that core. That log's cache lines are likely to remain in the core's cache in exclusive mode, so that writes to them are very fast. OpLog acquires a per-core lock protecting the per-core log; this lock is not usually contended, and its cache line usually stays exclusive in the local cache, so the lock acquisition is fast.

OpLog only incurs contention and cache line transfers during a read operation. The reading core must acquire all the locks protecting the per-core logs, and must fetch the cache lines containing the logs. Then the core executes all the logged operations. While the locks and cache line movement may be expensive, their cost is usually amortized over the execution of many logged operations. This amortization often causes OpLog read operations to be faster than directly executing each operation on shared data; in the latter case, every operation involves a contented lock acquisition, invalidation of other cores' copies of the shared data, and movement of the shared cache lines.

4.2 OpLog optimizations

OpLog facilitates several optimizations that reduce inter-core communication.

Batching updates. OpLog's deferred execution of updates leads it to execute them in batches. This improves locality, since the data being updated remains in the cache of the processor that applies the updates. The main cost is pulling the log entries to the core that is processing the logs; this is often less expensive than it would have been to move the data to each of the cores executing the original operations. Flat combining [18] has similar benefits, but OpLog can batch more by deferring updates for longer.

Absorbing updates. It is often possible (under programmer direction) for OpLog to take advantage of the semantics of the target data structure to "absorb" log operations, in two ways.

First, if a new operation cancels the effect of an operation that is already in the local per-core log, OpLog removes the existing operation rather than adding the new one. For example, if the underlying type is a set, then a new remove operation may cancel an existing insert of the same item.

Second, if multiple operations can be represented by a single operation, OpLog combines the log entries. For example, OpLog can combine multiple increment operations on a counter into a single "add *n*" operation.

Both forms of absorption reduce the size of the logs, so that the read that eventually executes them has less work to do.

Allocating logs. Often only a few instances of a data structure are contended and will benefit from OpLog. To avoid the costs of logging for low-contention instances, OpLog only allocates log space for recently used objects. If an object has not been updated recently on a given core, OpLog applies all updates to that object from other cores' logs (if any), and then frees the local log space.

4.3 Example: a logging rmap

We illustrate the use of OpLog by applying it to the Linux reverse map (rmap) discussed in §3. Three rmap operations are relevant: "add" and "remove," which are called when a process maps or unmaps a page from its address space, and "truncate," which is called when a file is truncated and the file's pages must be removed from all address spaces. "Add" and "remove" are deferrable updates and are common; "truncate" reads but is relatively rare.

An OpLog-rmap consists of a single internal underlying rmap instance, a lock protecting it, a log per core, and a lock for each log. The OpLog-rmap "add" and "remove" methods append a timestamped operation to the current core's per-core log. The OpLog-rmap "truncate" acquires all the per-core log locks and the rmap instance lock, merges the logs by timestamp, applies all log records to the internal rmap by calling the underlying rmap's "add" or "remove," truncates the logs, and then calls the underlying "truncate" on the internal rmap.

For increased performance, the programmer can direct OpLog to use absorption to reduce the number of log entries. If a core is about to log a removal of a region, it can check its log for an earlier addition of the same region; if one exists, OpLog can remove the addition from its log and not log the removal.

5 The OpLog library

This section presents the detailed design of the OpLog library. The design addresses two main challenges:

- How can developers apply OpLog without changing existing interfaces or data structures?
- How can developers expose the optimization opportunities like absorption to OpLog without having to customize OpLog for each use case?

Method call	Semantics
`Object::log(Op* op)`	add `op` to a per-core log, implemented by a Log object
`Object::synchronize()`	acquire a per-object lock, and call apply on each per-core Log object
`Object::unlock()`	release the per-object lock acquired by `synchronize()`
`Log::push(Op* op)`	insert `op` into a per-core Log object
`Log::apply()`	sort and execute the operations from Log
`Log::try_absorb(Op* op)`	try to absorb an `op`
`Op::exec()`	execute the operation

Figure 6: OpLog interface overview.

5.1 OpLog API

There are two OpLog implementations, one in C++ for user-level programs, and one in C for the Linux kernel. This section presents the C++ interface because its syntax is clearer.

To use OpLog, a developer starts with the uniprocessor implementation of a data structure type, called the "underlying" type. The developer wraps an instance of the underlying type in a new type that will be publicly visible, called the OpLog-augmented type. The augmented type must be a sub-class of OpLog's `Object` class. Figure 7 shows an example, in which a new `Rmap` type is wrapped around the underlying `IntervalTree<Mapping>` type.

The programmer then creates a method for each kind of deferrable update. This method must create a sub-type of `class Op`; each log entry is an `Op`. Each kind of `Op` stores that operation's arguments. The update method calls `log()` to add the `Op` to the local per-core log. The `Op`'s `exec()` function calls the corresponding method of the underlying type, applying it to the internal underlying instance wrapped in the augmented type. OpLog calls the `exec()` functions when reading the logs.

Finally, the programmer creates a method wrapping each of the underlying type's non-deferrable-update methods. The wrapper must call `synchronize()` to read the logs and apply them to the internal instance, and then call the underlying type's method on that instance.

OpLog's `log()` and `synchronize()` functions hide the logs from the developer, simplifying the developer's task and allowing OpLog scope for optimization. By default, `synchronize` acquires locks on all per-core logs, merges the logged operations by timestamp, calls `try_absorb` on each operation (see §5.2), and calls `exec` for each operation. The locks ensure that at most one core is reading an object's logs. To avoid consuming too much memory, OpLog invokes `synchronize` if a core's log grows too long.

5.2 Type-specific optimizations

OpLog allows the programmer to optimize log handling by parameterizing the `Object` type with a customizable `Log` type. A `Log` type describes a per-core log, and must support the `push`, `apply`, and `try_absorb` methods shown in Figure 6.

```
struct Rmap : public Object<Log> {
public:
  void add(Mapping* m) { log(AddOp(m)); }
  void rem(Mapping* m) { log(RemOp(m)); }

  void truncate(off_t offset) {
    synchronize();
    // For each mapping that overlaps offset..
    interval_tree_foreach(Mapping* m, itree_, offset)
      // ..unmap from offset to the end of the mapping.
      unmap(m, offset, m->end);
    unlock();
  }

private:
  struct AddOp : public Op {
    AddOp(Mapping* m) : m_(m) {}
    void exec(Rmap* r) { r->itree_.add(m_); }
    Mapping* m_;
  }

  struct RemOp : public Op {
    RemOp(Mapping* m) : m_(m) {}
    void exec(Rmap* r) { r->itree_.rem(m_); }
    Mapping* m_;
  }

  IntervalTree<Mapping> itree_;
}
```

Figure 7: Using OpLog to implement a communication-efficient rmap.

Consider a reference counter supporting 3 operations: increment (`inc`), decrement (`dec`), and reading the count (`read`). There is no need for true logging in this case; very specialized per-core "logs" consisting of a simple counter are enough. To do this, the programmer implements a `CounterLog` class, as shown in Figure 8. The `CounterLog`'s `push` just immediately applies the `inc` or `dec` `Op` to the local counter. When the counter needs to be read, OpLog's `synchronize` calls `CounterLog`'s `apply`, which sums the per-core counters. Executing operations directly on per-core counters reduces storage overhead and in most cases performs better than queuing operations. Furthermore, the dynamic allocation of logs ensure that if a counter is not heavily used, the space overhead is reduced to just the shared counter.

The OpLog version of a distributed counter has the same performance benefits as previous distributed counters, but is more space efficient when it is not contended.

```
struct Counter : public Object<CounterLog> {
  struct IncOp : public Op {
    void exec(uint64_t* v) { *v = *v + 1; }
  }

  struct DecOp : public Op {
    void exec(uint64_t* v) { *v = *v - 1; }
  }

  void inc() { log(IncOp()); }
  void dec() { log(DecOp()); }

  uint64_t read() {
    synchronize();
    uint64_t r = val_;
    unlock();
    return r;
  }

  uint64_t val_;
}

struct CounterLog : public Log {
  void push(Op* op) { op->exec(&val_); }

  static void apply(CounterLog* qs[], Counter* c) {
    for_each_log(CounterLog* q, qs)
      c->val_ += q->val_;
  }

  uint64_t val_;
}
```

Figure 8: Using OpLog to implement a distributed reference counter.

This OpLog distributed counter is the one that §3 measured (see Figure 3(a)).

The logging rmap design can also benefit from a type-specific `Log`, to support absorption. `push`, when called with a `rem` `Op`, would check if the mappings to be removed have an `add` in the local core's log; if so, `push` would remove the logged `add` instead of logging the `rem`. Thus mappings that are added and deleted on the same core will be particularly efficient. To ensure that combining `add` and `rem` operations is safe, the logging rmap must prevent a region identifier (pointer to a `vm_area_struct` object) from being reused if that identifier appears in some core's log; it does this by incrementing the reference counts of `vm_area_struct` objects in the log.

For rmap, it is not uncommon that one core maps a file and another core unmaps it; for example, the parent might map, and a child unmap on another core. To deal with this case, the OpLog rmap performs absorption during `synchronize` by overriding `try_absorb`. If the operation passed to `try_absorb` is a `RemOp` and OpLog has not yet executed the `AddOp` that inserts the `Mapping`, `try_absorb` removes both operations from the log of operations. `try_absorb` can tell if OpLog executed a particular `AddOp` by checking if the `Mapping` member variable has been inserted into an interval tree.

6 Implementation

Both the C++ and the C implementations of OpLog are about 1000 lines of code.

To implement dynamic log space allocation, OpLog maintains a per-core hash table for each type. There can be at most one object in a table for each hash value. Only objects in a core's table can have a log on that core. When an object needs a log but doesn't have one, OpLog evicts the object currently in the same hash slot, after `synchronize()`ing it. The result is that the table will tend to contain recently-used objects.

6.1 Synchronized timestamp counters

OpLog assumes that cores have synchronized CPU timestamp counters. We were unable to ascertain whether counters are in fact synchronized on modern processors despite consulting the specifications, several Intel and AMD engineers, and several computer architecture researchers. In lieu of an official guarantee, we ran an experiment to determine if we can observe timestamp counter skew. In our experiment, we chose pairs of cores, and repeatedly issued `RDTSC` on the first core, stored the result in a shared-memory location, read the value on a second core, and compared it with the value from `RDTSCP` on that core (which will not be re-ordered before the memory read).

Running this experiment on two different 8-socket machines (one with a 10-core Intel Xeon E7-8870 CPU in each socket, and one with a 6-core AMD Opteron 8431 CPU in each socket), we observed that the second timestamp (`RDTSCP`) is always higher than the first timestamp (`RDTSC`), across all pairs of cores and many iterations. This result suggests that hardware in practice provides the guarantee needed by OpLog.

Furthermore, we observe that modern hardware already has active clock synchronization between cores and between sockets. Both machines have been running continuously for about 45 days at the time of the experiment, with timestamp counters starting at zero on all cores. Since the latency for uncontended communication through shared memory is at most $\sim$1000 cycles, our experiment demonstrates that the clock skew was less than 1000 cycles over 45 days (10^{16} cycles), which is an accuracy of about 10^{-13}. Quartz crystal oscillators achieve accuracies of at most 10^{-10} [33: §2], and Intel CPUs can operate on system reference clocks with just 10^{-4} accuracy [20: §2.8]; thus, there must be some protocol synchronizing timestamp counters across cores and sockets to achieve 10^{-13} accuracy.

We believe OpLog provides compelling evidence that synchronized timestamp counters have important software applications, and should be guaranteed by hardware; indeed, some other software systems already assume this is the case [32]. For hardware that cannot provide strict guarantees but can bound clock skew, OpLog could use a

technique from Spanner [12], delaying commits for the maximum skew interval.

7 Evaluation

This section answers the following questions:

- Does OpLog improve whole-application performance? Answering this question is challenging because full applications stress many kernel subsystems; even if some kernel data structures are optimized using OpLog, other parts may still contain scalability bottlenecks. Nevertheless, we show for two applications that OpLog benefits performance. (§7.1)
- How does OpLog affect performance when a data structure is read frequently? We answer this question by analyzing the results of a benchmark that stresses the rmap with calls to `fork` and `truncate`. (§7.2)
- How important are the individual optimizations that OpLog uses? To answer this question we turn on optimizations one by one, and observe their effect. (§7.3)
- How much effort is required of the programmer to use OpLog compared to using per-core data structures? We answer this question by comparing the effort required to apply OpLog and the per-core approach to three subsystems in the Linux kernel. (§7.4)

7.1 Application performance

Since OpLog is focused on scaling updates to data structures that have relatively few reads, we focus on workloads that generate such data structure access patterns; not every application and workload suffers from this kind of scalability bottleneck. In particular, we use two applications from MOSBENCH [7]: the Apache web server and the Exim mail server. These benchmarks stress several different parts of the kernel, including the ones that the microbenchmarks in §3 stress.

Apache web server. Apache provides an interesting workload because it exercises the networking stack and the file system, both of which are well parallelized in Linux. When serving an HTTP request for a static file, Apache `stat`s and `open`s the file, which causes two pathname lookups in the kernel. If clients request the same file frequently enough and Apache is not bottlenecked by the networking stack, we expect performance to eventually bottleneck on reference counting the file's `dentry`, in the same way as the pathname microbenchmark in §3 (see Figure 3(a)).

We configured the Apache benchmark to run a separate instance of Apache on each core and benchmarked

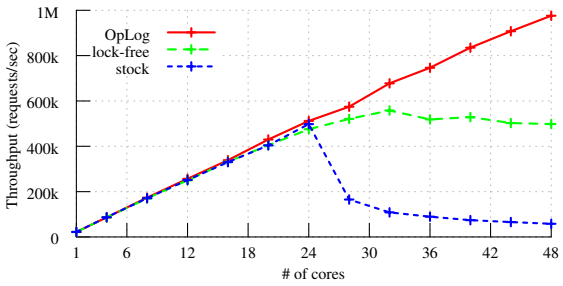


Figure 9: Performance of Apache.

Apache with HTTP clients running on the same machine, instead of over the network. This configuration eliminates uninteresting scalability bottlenecks and excludes drivers. We run one client on each core. All clients request the same 512-byte file.

Figure 9 presents the throughput of Apache on three different kernels: the stock 3.8 kernel and the two kernels used in the pathname example in §3. The x-axis shows the number of cores and the y-axis shows the throughput measured in requests per second. The line labeled "stock" shows the throughput of Apache on Linux 3.8. The kernel acquires a spinlock on the `dentry` of the file being requested in order to atomically check a generation count and increment a reference counter (see §3). This line goes down after 24 cores due to contention on the spinlock for a `dentry`.

The line labeled "lock-free" shows the throughput after we refactored the code to remove the lock with a conditional compare-and-exchange instruction (as described in §3). This line levels off for the same reasons as the line in Figure 3, but is higher because Apache does a lot of other work in addition to checking the generation count and incrementing the reference count of a `dentry`.

The line labeled "OpLog" shows throughput when using reference counters implemented with OpLog (as described in §4), which scales perfectly with increasing core counts. The distributed counters built with OpLog apply increments and decrements to per-core counters, avoiding any inter-core communication when incrementing and decrementing a reference counter.

Exim mail server. We configure Exim [1] to operate in a mode where a single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming mail, queues it in a shared set of spool directories, appends it to the per-user mail file, deletes the spooled mail, and records the delivery in a shared log file. Each per-connection process also forks twice to deliver each message.

The authors of MOSBENCH found that Exim was bottlenecked by per-directory locks in the kernel when creating files in spool directories. They suggested avoiding these

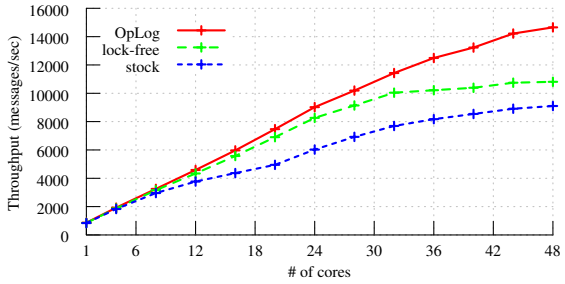


Figure 10: Performance of Exim.

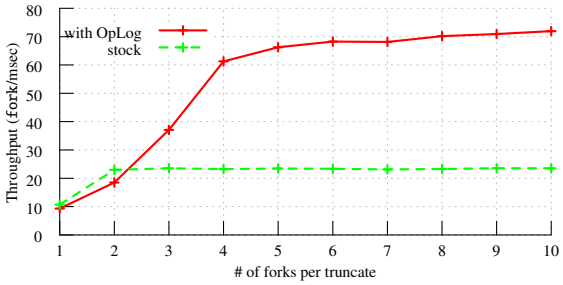


Figure 11: Performance of the `fork-truncate` benchmark for varying truncate frequencies.

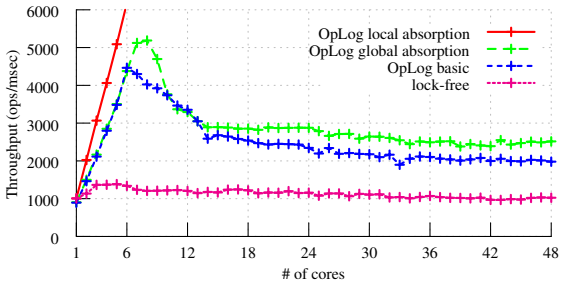


Figure 12: Performance of a lock-free list implementation and three OpLog list implementations.

locks and speculated that in the future Exim might be bottlenecked by cache misses on the rmap in `exit` [7]. We run Exim with more spool directories to avoid the bottleneck on per-directory locks.

Figure 10 shows the results for Exim on three different kernels: the stock 3.8 kernel and the two kernels that the fork and exit microbenchmark uses in §3 (see Figure 1). The results in Figure 10 show that the performance of Exim plateaus on both the stock kernel and the kernel with a lock-free rmap. Exim scales better on the kernel with OpLog, but at 42 cores its performance also starts to level off. At 42 cores, the performance of Exim is becoming bottlenecked by zeroing pages that are needed to create new processes.

7.2 Read intensive workloads

One downside of using OpLog for a data structure is that read operations can be slower, because each read operation will have to execute `synchronize` to collect updates from all cores, and OpLog will not be able to take advantage of batching or absorption.

To help understand how reads affect the performance of OpLog, we wrote a version of the `fork` benchmark that calls `truncate` to trigger reads of the shared rmap. This `fork-truncate` benchmark creates 48 processes, one pinned to each core, and 48 files, and each process `mmap`s all of the 48 files. Each process calls `fork` and the child process immediately calls `exit`. After a process calls `fork` a certain number of times, it truncates one of the `mmap`ed files by increasing the file length by 4096 bytes, and then decreasing the file length by 4096 bytes. A runtime parameter dictates the frequency of truncates, thus controlling how often the kernel invokes `synchronize` to read a shared rmap. The benchmark reports the number of `fork`s executed per millisecond. We ran the benchmark using truncate frequencies ranging from once per fork to once per ten forks.

Figure 11 shows the results of the `fork-truncate` benchmark. The stock version of Linux outperforms OpLog by about 10% when the benchmark truncates files after one call to `fork` and by about 25% when truncating after two calls to `fork`. However, we find that even when `truncate` is called once every three forks, OpLog outperforms the stock kernel. This suggests that OpLog improves performance even when the data structure is read periodically.

7.3 Breakdown of techniques

We evaluate the individual OpLog optimizations from §4 with user level microbenchmarks we wrote using the OpLog C++ implementation. To drive the microbenchmarks we implement a singly-linked list using OpLog, which the benchmark adds and remove entries from concurrently. Benchmarking a singly-linked list is interesting because it allows us to compare OpLog versions of the linked list with a lock-free design [17].

We study how much each technique from §4 affects performance using three different OpLog list implementations. The first is a OpLog list without any type-specific optimizations ("OpLog basic"). Comparing throughput of OpLog basic to the throughput of the lock-free list demonstrates how much batching list operations can improve performance. The other two list implementations demonstrate how much absorption improves performance. The second OpLog list uses type-specific information to perform absorption during `synchronize`. `synchronize` executes an add operation only if there is not a remove operation logged for the same element ("OpLog global absorption"). The third list takes advantage of the fact that the operations do not have to be executed in order: it performs absorption immediately when a thread logs an operation ("OpLog local absorption").

We wrote a microbenchmark to stress each list implementation. The benchmark instantiates a global list and creates one thread on each core. Each thread executes a loop that adds an element to the list, spins for 1000 cycles, removes the element from the list, and spins for another 1000 cycles. We count each iteration of the loop as one operation. We chose to delay 1000 cycles to simulate a list that is manipulated once every system call.

If operations on the lock-free list cause cache contention, we would expect OpLog basic to provide some performance improvement. OpLog global absorption should perform better than basic OpLog, because it ends up performing fewer operations on the shared list. OpLog local absorption should scale linearly, because when a thread logs a remove operation, OpLog should absorb the preceding add operation executed by the thread.

Figure 12 presents the results. The lock-free line shows that the throughput of the lock-free list peaks at 3 cores with 1364 operations per millisecond. OpLog basic throughput increases up to 4463 operations per millisecond on six cores, then begins to decrease until 1979 operations per millisecond on 48 cores. Throughput starts to decrease for more than 6 cores because the benchmark starts using more than one CPU node and inter-core communication becomes more expensive. OpLog global absorption throughput peaks at 5186 operations per millisecond on eight cores. Similar to OpLog basic, OpLog global absorption throughput decreases with more cores and is 2516 operations per millisecond on 48 cores. OpLog local absorption throughput scales linearly and achieves 49963 operations per millisecond on 48 cores.

OpLog allocates per-core data structures dynamically to avoid allocating them when an object is not heavily used. To measure the importance of this technique we measure the space overhead for `dentry`s with regular distributed reference counters instead of OpLog's space-optimized distributed counters. Each `dentry` is 192 bytes, and adding an 8-byte integer per core incurs a 200% overhead per `dentry` (384 bytes) for a regular distributed counter on our 48-core machine. As an example, we examined the `dcache` on our experimental machine after it had been running for some time; it had 15 million `dentry`s. The absolute storage overhead without OpLog's optimization would thus have been 5 GBytes; OpLog's optimization reduces this to a small constant-size overhead.

7.4 Programmer effort

To compare the programmer effort OpLog requires to the effort per-core data structures require we counted the number of lines of code added or modified when applying OpLog and per-core data structures to three Linux kernel subsystems. We implemented per-core versions of the rmap and inotify and used the existing per-core version of the open files list. The lines of code for the OpLog implementations do not include type-specific optimizations, only calls to `synchronize` and `log`.

Subsystem	Per-core LOC	OpLog LOC
rmap	473	45
inotify	242	11
open files list	133	8

Figure 13: The number of lines of code (LOC) required by per-core implementations and OpLog implementations.

Figure 7.4 presents the number of lines of code. The rmap required the most lines of code for both the per-core and OpLog implementations, because the rmap is not very well abstracted, so we had to modify much of the code that uses the rmap. The inotify and the open-files implementations required fewer code changes because they are well abstracted, so we did not need to modify any code that invoked them.

The OpLog implementations of all three subsystems required fewer changes than the per-core versions. The OpLog implementations do not modify operations, except for adding calls to `synchronize` or `log`. After calling synchronize, the existing code for read operations can safely access the shared data structure. The per-core implementations, on the other hand, replace every update operation on the shared data structure with an operation on a per-core operation and every read operation with code that implements a reconciliation policy.

The number of lines of code required to use OpLog is few compared to using per-core data structures. This suggests that OpLog is helpful for reducing the programmer effort required to optimize communication in update intensive data structures.

8 Discussion and future work

The machine used in the paper implements the widely used MOESI directory-based cache coherence protocol. The exact details of the cache-coherence protocol do not matter that much for the main point of the paper, but improvements in cache-coherence protocols could change the exact point at which the techniques are applicable. An interesting area of research to explore is to have the cache-coherence protocol expose more state.

Another future direction of research is to extend OpLog with other optimizations. For example, OpLog could exploit locality by absorbing state in near-by cores first, and arranging the cores in a tree topology to aggregate the state hierarchically. This design can reduce the latency of executing `synchronize`.

9 Conclusions

Data structures that experience many updates can pose a scalability bottleneck to multicore systems, even if the

data structure is implemented without locks. Prior approaches to solving this problem require programmers to change their application code and data structure semantics to achieve scalability for updates. This paper presented OpLog, a generic approach for scaling an update-heavy workload using per-core logs along with timestamps for ordering updates. Timestamps allow OpLog to preserve linearizability for data structures, and OpLog's API allows programmers to preserve existing data structure semantics and implementations. Results with a prototype of OpLog for Linux show that it improves throughput of real applications such as Exim and Apache for certain workloads. As the number of cores in systems continues to increase, we expect more scalability bottlenecks due to update-heavy data structures, which programmers can address easily using OpLog.

References

[1] Exim, May 2010. `http://www.exim.org/`.

[2] Recoll, July 2013. `www.recoll.org/`.

[3] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[4] H. G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), Sept. 1994.

[5] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proc. of the 24th SOSP*, Farmington, PA, Nov. 2013.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc. of the 22nd SOSP*, Big Sky, MT, USA, Oct. 2009.

[7] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proc. of the 9th OSDI*, Vancouver, Canada, Oct. 2010.

[8] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proc. of the Linux Symposium*, Ottawa, Canada, July 2012.

[9] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference (EuroSys 2013)*, Prague, Czech Republic, Apr. 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 *Proc. of the 24th SOSP*, Farmington, PA, Nov. 2013.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, Mar. 2010.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of the OSDI 2012*, Hollywood, CA, Oct. 2012.

[13] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proc. of the 24th SOSP*, Farmington, PA, Nov. 2013.

[14] J. DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Nov. 1990.

[15] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC 2007*, Portland, Oregon, USA, Aug. 2007.

[16] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of the 3rd OSDI*, pages 87–100, 1999.

[17] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

[18] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, Thira, Santorini, Greece, 2010.

[19] Intel. An introduction to the Intel QuickPath Interconnect, Jan. 2009.

[20] Intel. Intel Xeon processor E7-8800/4800/2800 product families, Apr. 2011. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e7-8800-4800-2800-families-vol-1-datasheet.pdf.

[21] Jonathan Corbet. The object-based reverse-mapping VM, Feb. 2003. http://lwn.net/Articles/23732/.

[22] Jonathan Corbet. Virtual memory II: the return of objrmap, Mar. 2004. http://lwn.net/Articles/75198/.

[23] Jonathan Corbet. Dcache scalability and RCU-walk, Dec. 2010. http://lwn.net/Articles/419811/.

[24] Jonathan Corbet. The case of the overly anonymous anon_vma, Apr. 2010. http://lwn.net/Articles/383162/.

[25] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Prog. Lang. Syst.*, 28(1), Jan. 2006.

[26] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf.

[27] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proc. of the Linux Symposium*, pages 338–367, Ottawa, Canada, 2002.

[28] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[29] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of the 24th SOSP*, Farmington, PA, Nov. 2013.

[30] C. A. N. Soules, J. Appavoo, K. Hui, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. of the 2003 USENIX Annual Technical Conference*, pages 141–154, June 2003.

[31] D. B. Terry, M. M. Theimer, K. Petersen, and A. J. Demers. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th SOSP*, 1995.

[32] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. of the 24th SOSP*, Farmington, PA, Nov. 2013.

[33] J. R. Vig. Quartz crystal resonators and oscillators for frequency control and timing applications - a tutorial (rev. 8.5.5.3), May 2013. http://www.ieee-uffc.org/frequency-control/learning-vig-tut.asp.

[34] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2): 76–85, 2009.