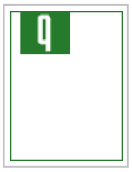
Performance
Vol. 12 No. 1 – January 2014



Scaling Existing Lock-based Applications with Lock Elision
Lock elision enables existing lock-based programs to achieve the performance benefits of nonblocking synchronization and fine-grain locking with minor software engineering effort.

Multithreaded applications take advantage of increasing core counts to achieve high performance. Such programs, however, typically require programmers to reason about data shared among multiple threads. Programmers use synchronization mechanisms such as mutual-exclusion locks to ensure correct updates to shared data in the presence of accesses from multiple threads. Unfortunately, these mechanisms serialize thread accesses to the data and limit scalability.

by Andi Kleen

Eventually Consistent: Not What You Were Expecting?
Methods of quantifying consistency (or lack thereof) in eventually consistent storage systems

Storage systems continue to lay the foundation for modern Internet services such as Web search, e-commerce, and social networking. Pressures caused by rapidly growing user bases and data sets have driven system designs away from conventional centralized databases and toward more scalable distributed solutions, including simple NoSQL key-value storage systems, as well as more elaborate NewSQL databases that support transactions at scale.

by Wojciech Golab, Muntasir R. Rahman, Alvin AuYoung, Kimberly Keeton, Xiaozhou (Steve) Li

The API Performance Contract
How can the expected interactions between caller and implementation be guaranteed?

When you call functions in an API, you expect them to work correctly; sometimes this expectation is called a contract between the caller and the implementation. Callers also have performance expectations about these functions, and often the success of a software system depends on the API meeting these expectations. So there's a performance contract as well as a correctness contract. The performance contract is usually implicit, often vague, and sometimes breached (by caller or implementation). How can this aspect of API design and documentation be improved?

by Robert Sproull, Jim Waldo

acmqueue

Scaling Existing Lock-based Applications with Lock Elision

Lock elision enables existing lock-based programs to achieve the performance benefits of nonblocking synchronization and fine-grain locking with minor software engineering effort.

Andi Kleen, Intel Corporation

Multithreaded applications take advantage of increasing core counts to achieve high performance. Such programs, however, typically require programmers to reason about data shared among multiple threads. Programmers use synchronization mechanisms such as mutual-exclusion locks to ensure correct updates to shared data in the presence of accesses from multiple threads. Unfortunately, these mechanisms serialize thread accesses to the data and limit scalability.

Often, lock-based programs do not scale because of the long block times caused by serialization, as well as the excessive communication overhead to coordinate synchronization. To reduce the impact on scalability, programmers use fine-grained locking, where instead of using a few locks to protect all shared data (coarse granularity), they use many locks to protect data at a finer granularity. This is a complex and error-prone process[10,11] that also often impacts single-thread performance.

Extensive work exists on ways to improve synchronization performance. Lock-free data structures support concurrent operations without mutually exclusive locks.[12] Such algorithms are often quite complex.

A rich variety of locking algorithms of varying complexity exist.[10] Other approaches such as optimistic locking avoid synchronization overhead—for example, by using sequence numbers to protect reading shared data and retrying the accesses if necessary. While effective under certain conditions, extending these approaches to be generally usable is quite difficult.

Transactional memory[5] proposed hardware mechanisms to simplify the development of lock-free data structures. They rely on mechanisms other than locks for forward progress and exploit the underlying cache-coherence mechanisms to detect conflict among threads.

Lock elision[14] was another proposal to expose concurrency in lock-based programs by executing them in a lockless fast path. It uses the hardware capability of modern processors and the underlying cache-coherence protocol to execute critical sections optimistically, without acquiring a lock. The lock is acquired only when actually required to resolve a data conflict.

In spite of the numerous proposals, high-performance synchronization remains difficult: programmers must use information known in advance to determine when to serialize, and scalable locking is complex, leading to conservative lock placement.

Recently, commercial processors from Intel Corporation and IBM have introduced hardware support to improve synchronization.[6,7,8,16] This presents a unique opportunity for software developers to improve the scalability of their software.

This article focuses on improving the existing lock-based programming model, and thus, looks at lock elision as the primary usage model.

HARDWARE SUPPORT FOR LOCK ELISION

Programmers must decide at development time how to control access to shared data—whether

by using coarse-grained locks, where a few locks protect all data, or by using fine-grained locks to protect different data. The dynamic behavior eventually determines sharing patterns. Finding errors with incorrectly used synchronization is quite difficult.

What is needed is a mechanism that has the usability of coarse-grained locks with the scalability of fine-grained locks. This is exactly what lock elision provides. The programmer must still use locks to protect shared data but can adopt a more coarse-grained approach. The hardware determines dynamically whether threads need to serialize in a critical section and executes the lock-based programs in a lock-free manner, where possible.

The processor executes lock-protected critical sections (called transactional regions) transactionally. Such an execution only reads the lock; it does not acquire or write to it, thus exposing concurrency. Because the lock is elided and the execution optimistic, the hardware buffers any updates and checks for conflicts with other threads. During the execution, the hardware does not make any updates visible to other threads.

On a successful transactional execution, the hardware atomically commits the updates to ensure all memory operations appear to occur instantaneously when viewed from other processors. This atomic commit allows the execution to occur optimistically without acquiring the lock; the execution, instead of relying on the lock, now relies on hardware to ensure correct updates. If synchronization is unnecessary, the execution can commit without any cross-threaded serialization.

A transactional execution may be unsuccessful because of abort conditions such as data conflicts with other threads. When this happens, the processor will do a transactional abort. This means the processor discards all updates performed in the region, restores the architectural state to appear as if the optimistic execution never occurred, and resumes execution nontransactionally. Depending on the policy in place, the execution may retry lock elision or skip it and acquire the lock.

To ensure an atomic commit, the hardware must be able to detect violations of atomicity. It does this by maintaining a read and a write set for the transactional region. The read set consists of addresses read from within the transactional region, and the write set consists of addresses written to, also from within the transactional region.

A conflicting data access occurs if another logical processor reads a location that is part of the transactional region's write set or writes a location that is a part of the read or write set of the transactional region. This is referred to as a *data conflict*.

Transactional aborts may also occur as a result of limited transactional resources. For example, the amount of data accessed in the region may exceed the buffering capacity. Some operations that cannot be executed transactionally, such as I/O, always cause aborts.

INTEL TSX

Intel TSX (Transactional Synchronization Extensions) provides two instruction-set interfaces to define transaction regions. The first is HLE (Hardware Lock Elision), which uses legacy-compatible instruction prefixes to enable lock elision for existing atomic-lock instructions. The other is RTM (Restricted Transactional Memory), which provides new XBEGIN and XEND instructions to control transactional execution and supports an explicit fallback handler.

Programmers who want to run Intel TSX-enabled software on legacy hardware could use the HLE interface to implement lock elision. On the other hand, with more complex locking primitives or when more flexibility is needed, the RTM interface can be used to implement lock elision.

This article focuses on the RTM interface in TSX. IBM systems provide instructions roughly similar to RTM.[6,8,16]

FALLBACK HANDLERS

To use the RTM interface, programmers add a lock-elision wrapper to the synchronization routines. Instead of acquiring the lock, the wrapper uses the XBEGIN instruction and falls back to the lock if the transaction aborts and retries don't succeed.

The Intel TSX architecture (and others, except the IBM zSeries, which has limited support for guaranteed small transactions[8]) does not guarantee that a transactional execution will ever succeed. Software must have a fallback nontransactional path to execute on a transactional abort. Transactional execution does not directly enable new algorithms but improves the performance of existing ones. The fallback code must have the capability of ensuring eventual forward progress; it must not simply keep retrying the transactional execution forever.

To implement lock elision and improve existing lock-based programming models, the programmer would test the lock within the transactional region and acquire the lock in the nontransactional fallback path, and then reexecute the critical region. This enables a classic lock-based programming model.

FIGURE 1

Lock Elision

```
Listing:
/* Start transactional region. On abort we come back here. */
if (_xbegin() == _XBEGIN_STARTED) {
       /* Put lock into read-set and abort if lock is busy */
       if (lock variable is not free)
              _xabort(_XABORT_LOCK_BUSY);
} else {
       /* Fallback path */
       /* Come here when abort or lock not free */
       lock lock;
}
/* Execute critical region either transaction or with lock */

Elided lock

Listing:
/* Critical region ends */
/* Was this lock elided? */
if (lock is free)
       _xend();
else
       unlock lock
Elided unlock
```

Alternatively, the programmer may use the transactional support to improve the performance and implementation of existing nonblocking heavyweight algorithms, using a fast and simple path to execute transactionally, but a slower and more complex nonblocking path if the transactional execution aborts. The programmer must ensure proper interaction between the two paths.

The programmer may also use the transactional support for implementing new programming models such as transactional memory. One approach uses the transactional support in hardware for a fast path, with an STM (software transactional memory) implementation for the fallback path[2] if the large overhead of STM on fallback is acceptable.[1]

IMPLEMENTING LOCK ELISION

Lock elision uses a simple transactional wrapper around existing lock code, as shown in figure 1.

`_xbegin()` tries to execute the critical section transactionally. If the execution does not commit, then `_xbegin` returns a status word different from `_XBEGIN_STARTED`, indicating the abort cause. After doing some retries, the program can then acquire the lock.

On unlock, the code assumes that if the lock is free, then the critical region was elided, and it commits with `_xend()`. (This assumes that the program does not unlock free locks.) Otherwise, the lock is unlocked normally. This is a simplified (but functional) example. A practical implementation would likely implement some more optimizations to improve transaction success. GCC (GNU Compiler Collection) provides detailed documentation for the intrinsics used here.[4] GitHub has a reference implementation of the TSX intrinsics.[9]

A thread that keeps aborting and goes to the fallback handler eventually acquires the lock. All threads speculating on the same lock have the lock in their read sets and abort their transactional execution when they detect a write conflict on the lock variable. If that happens, then eventually they will all take the fallback path. This synchronization mechanism between fallback path and speculative execution is important to avoid races and preserve the locking semantics.

This also requires that the lock variable is visible to the elision wrapper, as shown in figure 2.

FIGURE 2

Example Synchronization of Transactions through the Lock Variable

```
Thread 1                    Thread 2
Transaction A
Read Lock X
                            Transaction B
                            Read Lock X
                            … abort for some reason …
                            Fallback handler
                            Writes to lock X to acquire
                    * DATA CONFLICT ON LOCK *
Abort
Waits on lock to become free
```

ENABLING LOCK ELISION IN EXISTING PROGRAMS

Most existing applications use synchronization libraries to implement locks. Adding elision support to only these libraries is often sufficient to enable lock elision for these applications. This may be as simple as adding an elision wrapper in the existing library code. The application doesn't need changes for this step, as lock elision maintains the lock-based programming model.

We implemented lock elision for the POSIX standard pthread mutexes using Intel TSX and integrated it into Linux glibc 2.18. An implementation to elide pthread read/write locks has not yet been integrated into glibc.

The glibc mutex interface is binary compatible. This allows existing programs, using pthread mutexes for locking, to benefit from elision. Similarly, other lock libraries can be enabled for elision.

The next step is to evaluate performance and analyze transactional execution. Such an analysis may suggest changes to applications to make them more elision-friendly. Such changes also typically improve performance without elision, by avoiding unnecessary communication among cores.

ANALYZING TRANSACTIONAL EXECUTION

The behavior of explicit speculation is different from existing programming models. An effective way of analyzing transactional execution is with a transaction-aware profiler using hardware performance-monitoring counters.

Intel TSX provides extensive support to monitor performance, including sampling, abort rates, abort-cause profiling, and cycle-level analysis for successful and unsuccessful transactional executions. Often, the abort rates can be significantly reduced through minor changes to application data structures or code (for example, by avoiding false sharing).

The performance-monitoring infrastructure provides information about hardware behavior that is not directly visible to programmers. This is often critical for performance tuning. Programmers can also instrument transactional regions to understand their behavior. This provides only a limited picture, however, because transactional aborts discard updates, and the instrumentation itself may contribute to aborts.

ELISION RESULTS

Figure 3 compares the average number of operations per second when reading a shared hash table protected with a global lock versus an elided global lock. The tests were run on a four-core Intel Core (Haswell) system without SMT. The elided lock provides near-perfect scaling from one to four cores, while the global lock degrades quickly. A recent paper[17] analyzes lock elision using Intel TSX for larger applications and reports compelling benefits for a wide range of critical section types.

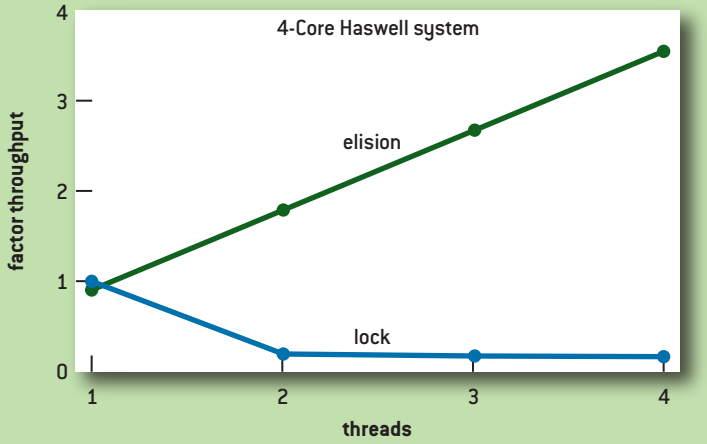
WHEN DOES ELISION HELP?

Data structures that do not encounter significant data conflicts are excellent candidates for elision. An elided lock can allow multiple readers and nonconflicting writers to execute concurrently. This is because lock elision does not result in any write operations on the lock, thus behaving as a reader-writer lock. It also eliminates any cache-line communication of the lock variable. As a result, programs with fine-grained locking may also see gains.

Not all programs benefit from lock improvements. These programs are either single-threaded, do not use contended locks, or use some other algorithm for synchronization. Data structures that repeatedly conflict do not see a concurrency benefit.

FIGURE 3

Hash Table Lookup with Elision



If the program uses a complex nonblocking or fine-grained algorithm, then a simpler transactional fast path can speed it up. Further, some programs using system calls or similar unfriendly operations inside their critical sections will see repeated aborts.

THE COST OF ABORTS

Not all transactional aborts make the program slower. The abort may occur where otherwise the thread would simply have been waiting for a lock. Such transactional regions that abort may also serve to prefetch data. Frequent, persistent aborts hurt performance, however, and developers must analyze aborts to reduce their probability.

ADAPTIVE ELISION

Synchronization libraries can adapt their elision policies according to transactional behavior. For example, the glibc implementation uses a simple adaptive algorithm to skip elision as needed. The synchronization library wrapper detects transactional aborts and disables lock elision on unsuccessful transactional execution. The library reenables elision for the lock after a period of time, in case the situation has changed. Developing innovative adaptive heuristics is an important area of future work.[13,15]

Adaptive elision combined with elision wrappers added to existing synchronization libraries can effectively enable lock elision seamlessly for all locks in existing programs. Developers should still use profiling to identify causes of transactional aborts, however, and address the expensive ones. That remains the most effective way to improve performance.

As an alternative to building adaptivity into the elision library, programmers may selectively identify which locks to apply elision to. This approach may work for some applications but may require developers to understand the dynamic behavior of all the locks in the program, and it may result in missed opportunities. Both approaches can be combined.

THE LEMMING EFFECT

When a transactional abort occurs, the synchronization library may either retry elision or explicitly skip it and acquire the lock. How the library retries elision is important. For example, when a thread falls back to explicitly acquiring a lock, this results in aborting other threads that elide the same lock. A naïve implementation on the other threads would immediately retry elision. These threads would find the lock held, abort, and retry elision, thus quickly reaching their retry threshold without any opportunity to have found the lock free. As a result, these threads quickly transition to an execution where they skip elision. It is easy to see how all threads end up in a situation where no thread reattempts lock elision for longer time periods. This phenomenon is the lemming effect[3] where threads enter extended periods of nonelided execution. This prevents software from taking advantage of the underlying elision hardware.

A simple and effective fix in the synchronization library is to retry elision only if the lock is free and spin/wait nontransactionally—and limit retry counts. Some lock algorithms implement a queue to enforce an order for threads if they find the lock unavailable. Queues are incompatible with parallel speculation. For such code, programmers must use the elision wrapper, including retry support, prior to the actual queuing code. Other mitigation strategies are possible.

Once programmers understand the underlying behavior, simple fixes can go a long way in improving unexpected performance behaviors.

Lock elision is a new technique in the scaling toolbox that is available on modern systems. It enables existing lock-based programs to achieve the performance benefits of nonblocking synchronization and fine-grained locking with minor software engineering effort.

ACKNOWLEDGMENTS

Thanks to Noel Arnold, Jim Cownie, Roman Dementiev, Ravi Rajwar, Arch Robison, Joanne Strickland, and Konrad Lai for their feedback and improvements to the article.

FURTHER READING

Intel. *IA Optimization Manual*, Chapter 12, TSX Recommendations, describes various tuning techniques to improve lock elision; http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.
Web Resources: http://www.intel.com/software/tsx.

REFERENCES

1. Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., Chatterjee, S. 2008. Software transactional memory: why is it only a research toy? *Communications of the ACM* 51(11): 40-46.
2. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M. L., Spear, M. F. 2011. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In the *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*: 39-52.
3. Dice, D., Herlihy, M., Lea, D., Lev, Y., Luchangco, V., Mesard, W., Moir, M., Moore, K., Nussbaum, D. 2008. Applications of the adaptive transactional memory test platform. The 3rd Annual ACM SIGPLAN Workshop on Transactional Computing.

4. GCC. 2013. X86 transaction memory intrinsics; http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html#X86-transactional-memory-intrinsics.
5. Herlihy, M., Moss, J. E. B. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA):* 289-300.
6. IBM. 2013. Power ISA; http://www.power.org/documentation/power-isa-version-2-07/.
7. Intel. 2012. *Intel 64 and IA-32 Architectures Software Developer Manuals*, Volume 1, Chapter 14; http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
8. Jacobi, C., Slegel, T., Greiner, D. 2012. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture:* 25-36.
9. Kleen, A. 2013. tsx-tools; http://github.com/andikleen/tsx-tools.
10. McKenney, P. E. 2013. Is parallel programming hard, and, if so, what can you do about it?; https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
11. McVoy, L. 1999. SMP scaling considered harmful; http://www.bitmover.com/llnl/smp.pdf.
12. Michael, M. M. 2013. The balancing act of choosing nonblocking features. *Communications of the ACM* 56(9): 46-53.
13. Pohlack, M., Diestelhorst, S. 2011. From lightweight hardware transactional memory to lightweight lock elision. In the Sixth Annual ACM SIGPLAN Workshop on Transactional Computing.
14. Rajwar, R., Goodman, J. R. 2001. Speculative lock elision. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture:* 294–305.
15. Usui, T., Behrends, R., Evans, J., Smaragdakis, Y. 2009. Adaptive locks: combining transactions and locks for efficient concurrency. In the 4th ACM SIGPLAN Workshop on Transactional Computing.
16. Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., Michael, M. 2012. Evaluation of Blue Gene/Q hardware support for transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques:* 127-136.
17. Yoo, R. M., Hughes, C. J., Rajwar, R., Lai, K. 2013. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*: article no. 19.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ANDI KLEEN is a software engineer at Intel's open source technology center. He worked on the x86-64 port of the Linux kernel and other kernel areas, including networking, performance, NUMA (nonuniform memory access), and error recovery. He currently focuses on performance tuning, scalability to many cores, and performance analysis. He started his career doing support for media artists and then spent more than nine years in SUSE Labs.

acmqueue

Eventually Consistent: Not What You Were Expecting?

Methods of quantifying consistency (or lack thereof) in eventually consistent storage systems

Wojciech Golab, University of Waterloo
Muntasir R. Rahman, University of Illinois at Urbana-Champaign
Alvin AuYoung, HP Labs, Palo Alto
Kimberly Keeton, HP Labs, Palo Alto
Xiaozhou (Steve) Li, Google

Storage systems continue to lay the foundation for modern Internet services such as Web search, e-commerce, and social networking. Pressures caused by rapidly growing user bases and data sets have driven system designs away from conventional centralized databases and toward more scalable distributed solutions, including simple NoSQL key-value storage systems, as well as more elaborate NewSQL databases that support transactions at scale.

Distributed key-value storage systems are among the simplest and most scalable specimens in the modern storage ecosystem. Such systems forgo many of the luxuries of conventional databases, including ACID (atomicity, consistency, isolation, durability) transactions, joins, and referential integrity constraints, but retain fundamental abstractions such as tables and indexes. As a result, application developers are sheltered from technicalities such as normalizing the relational schema, selecting the optimal transaction isolation level, and dealing with deadlocks.

Despite their simple interface and data model, distributed key-value storage systems are complex internally as they must replicate data on two or more servers to achieve higher read performance and greater availability in the face of node or network failures. Keeping these replicas synchronized requires a distributed replication protocol that can add substantial latency to storage operations, especially in geo-replicated systems. Specifically, write operations must update a subset of the replicas before the system acknowledges the completion of the write to the client, and reads may fetch data from a subset of the replicas, adopting the latest value observed (e.g., the one having the highest timestamp) as the response.

As it turns out, choosing which subset of replicas to contact for a storage operation profoundly impacts the behavior of distributed storage systems. For strong consistency a client may read and write a majority of replicas. Since majorities overlap, this ensures that each read "sees" the latest write. In contrast, eventually consistent systems may read and write non-overlapping subsets.[26,28] Once a write is acknowledged, the new value is propagated to the remaining replicas; thus, all replicas are eventually updated unless a failure occurs. In the meantime, readers may observe stale values if they fetch data from replicas that have not yet received the update.

Although many applications benefit from strong consistency, latency-sensitive applications such as shopping carts in e-commerce Web sites choose eventual consistency to gain lower latency.[1] This can lead to consistency anomalies such as items lost from a shopping cart or oversold. However, since a user can detect and correct the problem during checkout, such anomalies are tolerable provided they are short-lived and infrequent. The critical task for application developers and system

administrators, therefore, is to understand how consistency and latency are impacted by various storage and application configuration parameters, as well as the workload that the application places on the storage system. Only with proper insight into this subtle issue can administrators and developers make sensible decisions regarding the configuration of the storage system or the choice of consistency model for a given application.

This article looks at methods of quantifying consistency (or lack thereof) in eventually consistent storage systems. These methods are necessary for meaningful comparisons among different system configurations and workloads. First, the article defines eventual consistency more precisely and relates it to other notions of weak consistency. It then drills down into metrics, focusing on staleness, and surveys different techniques for predicting and measuring staleness. Finally, the relative merits of these techniques are evaluated, and any remaining open questions are identified.

DEFINING EVENTUAL CONSISTENCY

Eventual consistency can be defined as either a property of the underlying storage system, or a behavior observed by a client application. For example, Doug Terry et al. give the following definition of eventual consistency in the context of the Bayou storage system: "All replicas eventually receive all writes (assuming sufficient network connectivity and reasonable reconciliation schedules), and any two replicas that have received the same set of writes have identical databases."[26] This informal definition addresses both the propagation of writes to replicas (i.e., the *eventual*) and convergence onto a well-defined state—for example, through timestamp-based reconciliation—(i.e., the *consistency*).

This definition, while simple and elegant, does not say precisely what clients observe. Instead, it captures a range of behaviors encompassing both strongly consistent relational databases, and weakly consistent systems that may return stale or out-of-order data without bound. In contrast, Werner Vogels describes the consistency observed by clients in a concrete way: "The storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value."[28] Vogels's definition captures in an intuitive way the convergence of replicas to the last updated value, but only in the special case where updates are suspended while clients read an object—a scenario very different from the online environment in which many eventually consistent systems are deployed.

Since weak consistency is difficult to reason about, application developers often seek stronger properties such as a consistent prefix, monotonic reads, "read my writes," or causal consistency.[25] Indeed, some systems support such stronger-than-eventual properties (e.g., COPS [Clusters of Order-Preserving Servers[22]] and Pileus[27]), but the commercial success of systems such as Amazon's Dynamo[12] proves that eventual consistency can be good enough in practice. To understand why, researchers have sought to describe the range of behaviors of eventually consistent systems in more precise terms than abstract definitions in the style of Terry et al. and Vogels. Existing approaches in this endeavor fall into three categories: relaxed consistency metrics, system modeling and prediction, and empirical measurement tools. The next three sections survey representative works in each category.

RELAXED CONSISTENCY PROPERTIES

Abstract definitions of eventual consistency leave open a number of questions regarding system behavior in the presence of concurrent accesses, as well as in a failure-prone environment. For

example: How quickly do updates propagate to replicas in practice, and how often do replicas agree on the latest value of an object? When replicas do not agree, how does that affect the clients' view of the data? What exactly do clients observe during an extended failure that partitions the network, or in the moments shortly after the network is healed?

An emerging approach for describing these behaviors is to relate them to a strict form of consistency called *linearizability*.[18] Informally speaking, this property states that the storage system behaves as though it executes operations one at a time, in some serial order, despite actually executing some operations in parallel. As will be explained later, this serial order is further constrained in a manner that forbids stale reads. As a result, linearizability is a suitable gold standard against which eventually consistent systems can be judged. In other words, the observed behavior can be described in terms of deviations from this standard, which in this context can be regarded as consistency violations. For example, one can think of Amazon's Dynamo as violating linearizability when reads return stale values, even though the system promises only eventual consistency.

Whereas linearizability is ubiquitous in centralized systems—for example, in multithreaded data structures—Brewer's CAP principle[9] states that such a strong consistency property (*C*) is unattainable in always-available (*A*) and partition-tolerant (*P*) distributed systems. (Database experts should read *consistency* in this context as *transaction isolation*.) As a result, one cannot expect any storage system to accept updates and yet remain consistent during a network partition. Although this does not preclude strong consistency during failure-free operation, even in that case the system may be configured to sacrifice consistency for better latency.[1] For example, in the Cassandra key-value store, clients can achieve this trade-off by requesting various "consistency levels," which control the number of replicas that respond to a read or write.[20]

The side effect of weakening consistency is increased staleness, which so far has been discussed informally. More precisely, a value is considered *fresh* from the moment it is written into a storage system until it is overwritten, and *stale* thereafter. Thus, staleness describes the age of a value returned by a read relative to the last updated value, and hence quantifies how badly a system's behavior deviates from the gold standard. Two interpretations of this concept have been discussed in the literature,[2,15] arising from different ways to define "age":

- *Version-based staleness* defines age by counting versions of an object (e.g., a read returns the *k*th-latest version).
- *Time-based staleness* defines age in terms of wall-clock time (e.g., a read returns a value *t* time units older than the last updated value).

The concept of staleness, although intuitive, is fraught with technical subtleties. In the simple case where operations are executed one at a time, there is a natural order in which clients expect these operations to take effect, and so stale reads can be identified easily. When operations are executed in parallel, however, their order can be very difficult to determine from the client's perspective.

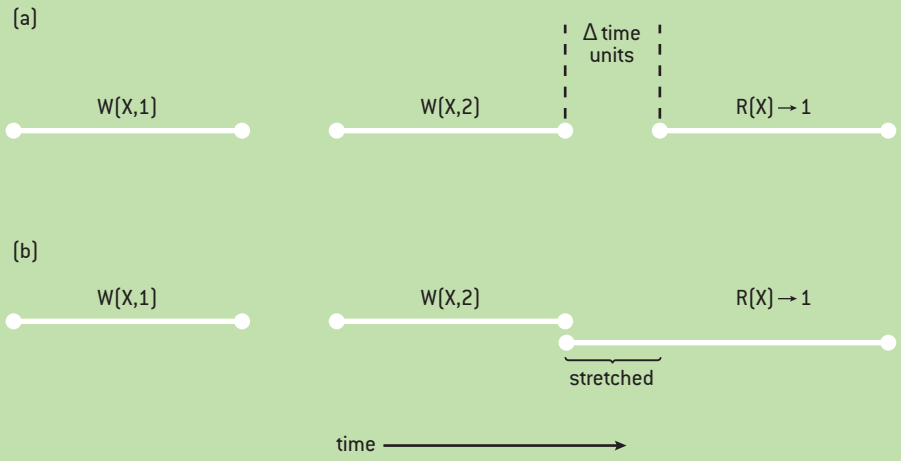
To make sense of eventual consistency, we turn to relaxed consistency properties—*k-atomicity* and *Δ-atomicity*—which give precise meaning to the notions of version-based and time-based staleness, respectively. Both of these properties are relaxed forms of linearizability,[18] but for historical reasons they refer in name to Lamport's *atomicity* property,[21] which is similar in spirit.

LINEARIZABILITY (THE "GOLD STANDARD")

Consider the trace of operations in figure 1a, showing three operations applied to an object denoted

FIGURE 1

Example Trace of Operations Illustrating Linearizability, K-Atomicity, and Δ-Atomicity



by X. First, a write operation W(X,1) assigns 1 to object X, and then this value is updated to 2 by a second write operation, W(X,2). The third operation R(X) is a read of X and begins after both writes have ended. In this case, linearizability dictates that W(X,1) should appear to take effect before W(X,2), because W(X,1) ends before W(X,2) begins. Thus, 2 is the last updated value of X when R(X) is applied, but R(X) returns 1 instead, so the trace is not linearizable. The more complex case when operations overlap in time is addressed by the formal definition of linearizability,[18] a discussion that is beyond the scope of this article.

K-ATOMICITY (VERSION-BASED STALENESS)

The *k*-atomicity property was introduced by Amitanand Aiyer et al.[2] Like linearizability, it requires that operations appear to take effect in an order that is constrained by their start and finish times; within this order, however, a read may return *any* of the *k* last updated values. For example, the trace shown in figure 1a is *k*-atomic for $k = 2$ but not $k = 1$.

Δ-ATOMICITY (TIME-BASED STALENESS)

The Δ-atomicity property was proposed by Wojciech Golab et al.[15] Similar to *k*-atomicity, it relaxes linearizability by allowing reads to return stale values. Staleness, however, is defined in terms of time: a read may return a value that is up to Δ time units stale. For example, the trace in figure 1a is Δ-atomic if Δ is defined as the width of the gap between W(X,2) and R(X). Linearizability permits R(X) to take effect before W(X,2) if the two operations overlap in time, as shown in figure 1b, whereas it requires R(X) to take effect after W(X,2) in figure 1a. If R(X) is hypothetically "stretched" to the left by Δ time units (i.e., if R(X) had started Δ time units earlier), as in figure 1b, then the trace becomes linearizable. Thus, the response of R(X) is considered only Δ time units stale in figure 1a.

PREDICTION

Another method used to characterize eventual consistency is based on a combination of system modeling and prediction. Peter Bailis et al.[6] present the PBS (Probabilistically Bounded Staleness) framework, in which a white-box system model is proposed to predict data staleness observed by client applications. The PBS model estimates the probability of <*k*,*t*> staleness—the condition that the read, which begins *t* time units after the end of the write, returns the value assigned by one of the last *k* writes. This condition is similar to *k*-atomicity but considers only a single read at a fixed distance *t* from the write. When $k = 1$, it captures the probability that the read returns the latest value (i.e., is not stale).

The PBS model makes two simplifying assumptions. First, like Vogels's definition of eventual consistency,[28] PBS does not consider workloads where writes overlap in time with reads, in which the width *t* of the gap between writes and reads is not well defined. Second, PBS does not model failures explicitly. Although storage node failures can be simulated using longer latencies, PBS does not account for network partitions. Follow-up work[11] extends the PBS model by considering node failures.

EMPIRICAL MEASUREMENT

Thus far, this article has addressed techniques for quantifying staleness in eventually consistent systems. This section discusses approaches that measure consistency empirically from the perspective of both the system and the client.

Measuring eventual consistency "in the wild" is as hard as defining it precisely. Concurrent operations make it difficult to identify the order in which operations take effect, and hence to classify reads as stale or not. To make matters worse, in the event of a network partition, clients on opposite sides of the divide may observe the last updated value differently, even if no new updates are made to an object after some point in time. Such an anomaly is possible because in an always-available system each partition will continue to accept writes.

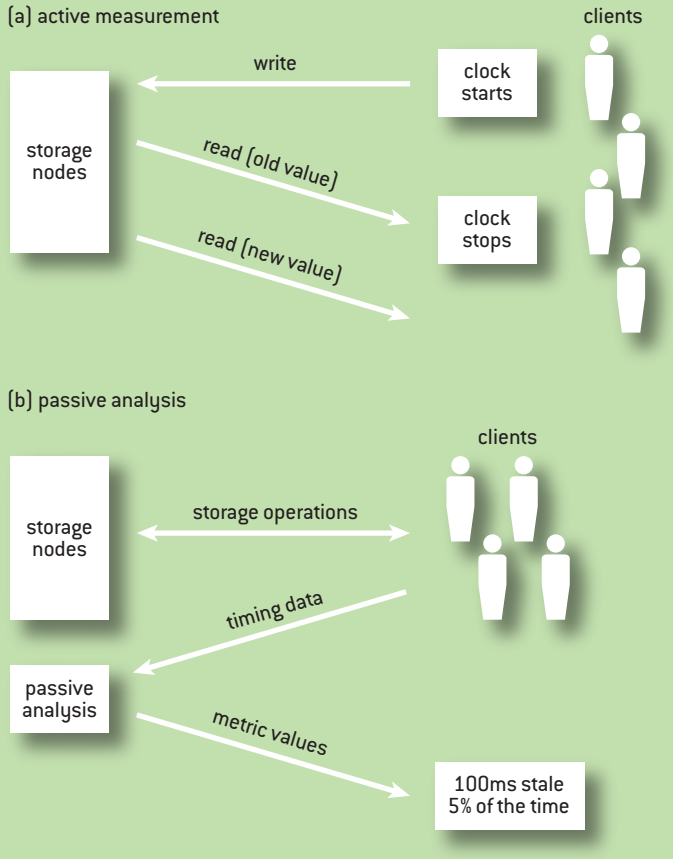
Despite these challenges, a number of techniques have been devised for measuring eventual consistency, particularly data staleness. This article looks at two fundamentally different methodologies: *active measurement*, in which the storage system is exercised in an artificial way to determine the time lag from when a new value is written to the storage system until this value becomes visible to clients; and *passive analysis,* in which a trace of operations is recorded for an arbitrary workload and analyzed mathematically to obtain a measurement of staleness.

ACTIVE MEASUREMENT

Active measurement underlies early studies of consistency in cloud storage systems. In this category of techniques, one client writes a new value to a key, and a different client then reads the same key repeatedly until the new value is returned. As in Vogels's definition of eventual consistency,[28] writes do not overlap in time with reads in this scenario. The time from the write to the last read that returns the old value—or alternatively, the first read of the new value—answers the question, "How eventual?" and can be regarded as an estimate of the *convergence time* of the replication protocol—the time needed to propagate a new value to all the replicas of an object.

At a technical level, the main challenge in active measurement is to determine the difference in time between operations executed at different client nodes—namely, a write and a read. Figure 2

FIGURE 2 **Infrastructure for Consistency Measurement**



illustrates how this difference can be computed using a collection of clients. To capture precisely the moment when the last replica receives the updated value, Hiroshi Wada et al. apply reads 50 times per second using one or more clients.[29] Bermbach et al. go one step further and use a collection of geographically distributed readers to ensure that the reads hit all possible replicas.[8] Staleness measurement in YCSB++ (Yahoo! Cloud-serving Benchmark)[23] follows a similar approach but uses a ZooKeeper producer-consumer queue[19] to synchronize writers and readers. This approach circumvents issues related to clock skew but introduces additional latencies caused by queue operations, which may limit precision.

Active measurement can be used to discover the range of update propagation times in an eventually consistent system. For example, in experiments involving Amazon's SimpleDB,[4] Wada et al. report that convergence occurred in at most one second in more than 90 percent of the runs, but took more than four seconds in a few (less than 1 percent) cases. On the other hand, active measurement does not indicate what proportion of reads in a real workload will return stale values, as this quantity depends on how the data objects are accessed. For example, the proportion of stale reads would be expected to vary with the rate at which operations are applied on a given object—

close to zero when operations are applied infrequently and the gaps between them exceed the convergence time, and larger when reads follow writes more closely. Active measurement does not separate these two cases, as it is based upon a controlled workload designed to measure convergence time.

PASSIVE ANALYSIS

In the earlier section on relaxed consistency properties, the discussion focused on ways of defining staleness in a precise and meaningful way and provided a hint of how a consistency metric can be derived from such definitions. It begins with a "black-or-white" consistency property, such as linearizability, which is either satisfied or not satisfied by a system or an execution trace; next, this property is relaxed by introducing a parameter that bounds the staleness of reads (e.g., Δ-atomicity); the last step is to determine the parameter value that most accurately describes the system's behavior (e.g., Δ-atomic for $\Delta \geq 10$ ms). Passive analysis refers to this final step and entails examining the operations recorded in an execution trace to determine the order in which they appear to take effect.

The most immediate technical challenge in passive analysis is the collection of the trace, which records for each operation its type, start and finish times, as well as its arguments and response (e.g., a read of object X, starting at time t_1 and ending at t_2, returning the value 5). This trace can be obtained at clients, as shown in figure 2.

Because the trace is obtained by merging data from multiple clients or storage nodes over a finite period of time, it is prone to two types of anomalies: *dangling reads*, which return values that lack a corresponding write (i.e., a write that assigns the value returned by the read); and *reversed operations*, whereby a read appears in the trace before its corresponding write. These anomalies must be removed if they occur; otherwise, the k-atomicity and Δ-atomicity properties are undefined and cannot be used for computing staleness.

Dangling reads indicate missing information, such as when the first access to an object in a trace is a read and the corresponding write occurs before the start of the trace. In this case the dangling read is identified easily and can be dropped from the trace with no ill effects. Reversed operations are equally easy to detect, but harder to remedy. Clock synchronization techniques such as atomic clocks and GPS[10] can eliminate reversed operations altogether. Even ordinary NTP (Network Time Protocol) is sufficient if the synchronization of clocks is tight enough. Alternatively, one can also estimate clock skew directly from reversed operations and adjust the trace accordingly.

Given a trace free from dangling reads and reversed operations, the next challenge is to compute staleness metrics. Efficient algorithms are critical in this context because the trace may be very long, which means that both the space and computation required are high. In fact, the problem of testing whether a trace is linearizable was shown by Gibbons and Korach to be intractable.[13] Testing whether a trace is k-atomic or Δ-atomic for fixed k and Δ is at least as hard since these properties are equivalent to linearizability when $k = 1$ and $\Delta = 0$. Computing k and Δ from a trace is harder still, and hence also intractable.

Fortunately, the intractability result breaks in the special case when every write on a given object assigns a unique value. This condition is straightforward to enforce (e.g., by embedding a unique token in each value written, such as a node ID and timestamp), which opens the door to efficient algorithms for computing staleness metrics from traces in practice. For Δ-atomicity, such an algorithm is known,[15] and has been used to analyze time-based staleness in traces obtained using

Cassandra.[24] For *k*-atomicity, an efficient algorithm is known only for deciding whether a trace is 2-atomic.[14]

Passive analysis in general is not limited to staleness metrics, as illustrated by the technique of *cycle analysis*,[5,30] which detects linearizability violations by analyzing a *conflict graph* representing the execution trace. The absence of cycles in such a graph indicates that the trace is linearizable, and so the number and length of such cycles can be defined as metrics for inconsistency.[5,30] The relationship of these metrics to staleness is not known precisely.

COMPARISON AND DISCUSSION

Prediction, active measurement, and passive analysis are all useful ways to study eventual consistency. Although each method has specific strengths and weaknesses, it is difficult to compare them directly, as they meet different goals. Passive analysis is *client-centric* in that it reflects the manner in which the client application interacts with the system. As a result, passive analysis can be used to compare staleness observed in two workloads applied to the same storage system. Active measurement, on the other hand, is *system-centric* in that it measures the convergence time of a system's replication protocol for a controlled workload. Thus, active measurement is best suited for comparing different systems in terms of staleness, or the same system under different software or hardware configurations. The PBS framework[6] is also designed around a controlled workload, and for that reason it is classified as system-centric.

The techniques discussed in this article also vary in terms of conceptual models of eventual consistency. Active measurement and PBS are based upon a *simplified model*, similar to Vogels's definition,[28] in which writes occur before reads. In contrast, passive analysis considers a *general model* in which writes may overlap in time with reads and with other writes. As explained in another article by Golab et al.,[16] storage systems may behave differently in these two models when clients follow the "$R + W > N$" rule,[28] which ensures that read and write operations access overlapping subsets of replicas. The latter condition has long been considered sufficient for "strong consistency,"[6,28] and in fact guarantees linearizability in the simplified model but permits linearizability violations in the general model.

Despite the somewhat high cost of analyzing a detailed trace of operations, passive analysis plays an important role in understanding eventual consistency. Whereas any perspective on consistency is ultimately affected by the state of data in replicas in a storage system, which can be thought of as the "ground truth," passive analysis is most closely tied to the actual consistency observed by client applications. Specifically, it reflects data-level anomalies only when they manifest themselves to clients—for example, as stale reads.

FUTURE WORK

For modern online service providers, small decreases in latency are known to have a measurable effect on user experience and, as a consequence, revenue.[17] As weak consistency continues to gain traction as the means for reducing latency, the ability to reason clearly about this trade-off will be an important competitive advantage for service providers.

Consistency-aware pricing schemes and guarantees are already being adopted by industry. Amazon's DynamoDB supports differentiated pricing for consistent reads and eventually consistent reads, with the former guarantee costing the application developer twice as much.[3] More recently,

Terry et al. describe a system that allows client applications to select different grades of weak consistency, such as "at most 200-ms latency and at most 5-minute staleness," through novel SLAs (service-level agreements).[27] Other read guarantees, such as monotonic reads or causal consistency, can be requested, and a "wish list" of different combinations can be specified with different utilities. This opens the door to even more fine-grained pricing schemes, making it more important than ever for users to understand the utility of different points in consistency-related trade-offs.

Whereas the understanding of eventual consistency is expected to improve with additional empirical studies involving measurement, fundamental technical problems also remain to be solved. In particular, the problem of consistency *verification* remains an open and important challenge. Storage system designers need verification techniques to test whether their implementations correctly fulfill application-specified SLAs, and, likewise, application developers could use such tools to verify the service quality actually received.

One of the open problems related to verification algorithms is to determine the computational complexity of deciding *k*-atomicity. Existing algorithms handle only the special case $k < 3$,[14] which limits their use in practice. Similar technical ideas can be applied when $k \geq 3$, but only for a restricted class of traces, and it is not known whether an efficient algorithm exists for deciding *k*-atomicity in the general case.

CONCLUSION

Eventual consistency is increasingly viewed as a spectrum of behaviors that can be quantified along various dimensions, rather than a binary property that a storage system either satisfies or fails to satisfy. Advances in characterizing and verifying these behaviors will enable service providers to offer an increasingly rich set of service levels of differentiated performance, ultimately improving the end user's experience.

ACKNOWLEDGMENTS

We are grateful to Jay J. Wylie, Indranil Gupta, and Doug Terry for their helpful feedback.

REFERENCES

1. Abadi, D. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer* 45(2): 37–42.
2. Aiyer, A., Alvisi, L., Bazzi, R. A. 2005. On the availability of non-strict quorum systems. In *Proceedings of the 19th International Symposium on Distributed Computing*: pp. 48–62.
3. Amazon Web Services. DynamoDB; http://aws.amazon.com/dynamodb/.
4. Amazon Web Services. SimpleDB; http://aws.amazon.com/simpledb/.
5. Anderson, E., Li, X., Shah, M. A., Tucek, J., Wylie, J. J. 2010. What consistency does your key-value store actually provide? In *Proceedings of the 6th Usenix Workshop on Hot Topics in System Dependability*.
6. Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., Stoica, I. 2012. Probabilistically bounded staleness for practical partial quorums. In *Proceedings of the VLDB (Very Large Data Base) Endowment 5(8)*: 776–787.
7. Bermbach, D., Zhao, L., Sakr, S. 2013. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In *Proceedings of the 5th TPC (Transaction*

Processing Performance Council) Technology Conference on Performance Evaluation and Benchmarking.

8. Bermbach, D., Tai, S. 2011. Eventual consistency: how soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service-oriented Computing*.
9. Brewer, E. A. 2000. Towards robust distributed systems (invited talk). In *Proceedings of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*.
10. Corbett, J. C., et al. 2012. Spanner: Google's globally distributed database. In *Proceedings of the 10th Usenix Conference on Operating Systems Design and Implementation*: 251–264.
11. Davidson, A., Rubinstein, A., Todi, A., Bailis, P., Venkataraman, S. 2012. Adaptive hybrid quorums in practical settings; http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F12/projects/reports/project12_report_ver2.pdf.
12. Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*.
13. Gibbons, P. B., Korach, E. 1997. Testing shared memories. *SIAM (Society for Industrial and Applied Mathematics) Journal on Computing* 26(4): 1208–1244.
14. Golab, W., Hurwitz, J., Li, X. 2013. On the *k*-atomicity-verification problem. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems*.
15. Golab, W., Li, X., Shah, M. A. 2011. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*: 197–206.
16. Golab, W., Rahman, M. R., AuYoung, A., Keeton, K., Gupta, I. 2013. Client-centric benchmarking of eventual consistency for cloud storage systems. Manuscript under submission.
17. Hamilton, J. 2009. The cost of latency; http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx.
18. Herlihy, M., Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12(3): 463–492.
19. Hunt, P., Konar, M., Junqueira, F. P., Reed, B. 2010. ZooKeeper: wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 Usenix Annual Technical Conference*: 11–25.
20. Lakshman, A., Malik, P. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44(2): 35–40.
21. Lamport, L. 1986. On interprocess communication. Part I: basic formalism; and Part II: algorithms. *Distributed Computing* 1(2): 77–101.
22. Lloyd, W., Freedman, M. J., Kaminsky, M., Andersen, D. G. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*: 401–416.
23. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B. 2011. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*: 9:1–9:14.
24. Rahman, M. R., Golab, W., AuYoung, A., Keeton, K., Wylie, J. J. 2012. Toward a principled framework for benchmarking consistency. In *Proceedings of the 8th Usenix Workshop on Hot Topics in System Dependability*.
25. Terry, D. 2013. Replicated data consistency explained through baseball. *Communications of the ACM* 56(12): 82–89.

26. Terry, D. B., Petersen, K., Spreitzer, M. J., Theimer, M. M. 1998. The case for non-transparent replication: examples from Bayou. *IEEE Data Engineering Bulletin* 21(4): 12–20.
27. Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., Abu-Libdeh, H. 2013. Consistency-based service-level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*.
28. Vogels, W. 2008. Eventually consistent. *ACM Queue* 6(6): 14–19.
29. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A. 2011. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*: 134–143.
30. Zellag, K., Kemme, B. 2012. How consistent is your cloud application? In *Proceedings of the 3rd ACM Symposium on Cloud Computing*.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

WOJCIECH GOLAB is an assistant professor in electrical and computer engineering at the University of Waterloo. His current research focuses on algorithmic problems in distributed computing with applications to storage, transaction processing, and big-data analytics. Prior to joining Waterloo, he worked as a researcher in storage systems at Hewlett-Packard Labs in Palo Alto and as a postdoctoral fellow in theory at the University of Calgary. He received a Ph.D in computer science from the University of Toronto in 2010.

MUNTASIR RAIHAN RAHMAN is a computer science Ph.D student in the Distributed Protocols Research Group at the University of Illinois at Urbana-Champaign. His current research is focused on consistency in distributed systems. He received a B.Sc. degree in computer science and engineering from Bangladesh University of Engineering and Technology in 2006, and the M. Math degree in computer science from the University of Waterloo in 2010.

ALVIN AUYOUNG is a researcher in the systems group at Hewlett-Packard Labs. His current work focuses on designing in-memory distributed systems for storage and large-scale data analysis. In the past he has also investigated problems at the intersection of economics and computer science, specifically applying ideas from markets and auctions to distributed systems. He received a Ph.D. in computer science from the University of California at San Diego.

KIMBERLY KEETON is a principal researcher at Hewlett-Packard Laboratories. Her recent research is in the areas of NoSQL databases, consistency models, and information management. She has also worked in the areas of storage management, storage dependability, workload characterization, and intelligent storage. She earned a Ph.D. in computer science from the University of California, Berkeley, and is an ACM Distinguished Scientist and a Senior Member of the IEEE.

XIAOZHOU (STEVE) LI is a software engineer at Google. In the past, he worked at Hewlett-Packard Labs as a researcher and at Microsoft as a software design engineer. His main interests are in the theory and practice of distributed computing. He received a Ph.D. in computer science from the University of Texas at Austin.

acmqueue

The API Performance Contract

How can the expected interactions between caller and implementation be guaranteed?

Robert F. Sproull and Jim Waldo

When you call functions in an API, you expect them to work correctly; sometimes this expectation is called a *contract* between the caller and the implementation. Callers also have performance expectations about these functions, and often the success of a software system depends on the API meeting these expectations. So there's a *performance contract* as well as a *correctness contract*. The performance contract is usually implicit, often vague, and sometimes breached (by caller or implementation). How can this aspect of API design and documentation be improved?

Any significant software system today depends on the work of others: sure, you write some code, but you call functions in the operating system and in a variety of software packages through APIs that cut down on the amount of code that you must write. In some cases you even outsource work to distant servers connected to you by cranky networks. You depend on these functions and services for correct operation, but you also depend on them performing well enough that the overall system performs well. There is bound to be performance variation in complex systems that involve paging, network delays, sharing resources such as disks, and so on. Even in simple setups such as a stand-alone computer with all program and data in memory, however, you can be surprised when an API or operating system doesn't meet performance expectations.

People are accustomed to speaking of the contract between an application and the implementation of an API to describe correct behavior when an API function is invoked. The caller must meet certain initial requirements, and the function is then required to perform as specified. (These dual responsibilities are like the pre- and post-conditions that appear in Floyd-Hoare logic for proving program correctness.) While today's API specifications do not make correctness criteria explicit in a way that leads to correctness proofs, the type declarations and textual documentation for an API function strive to be unambiguous about its logical behavior. If a function is described as "add the element *e* to the end of list *l*" where *e* and *l* are described by types in the calling sequence, then the caller knows what behavior to expect.

There's more to an API function than correctness, however. What resources does it consume, and how fast is it? People often make assumptions based on their own judgments of what the implementation of a function should be. Adding an element to the end of a list should be "cheap." Reversing a list might be "a linear function of list length." For many simple functions, these intuitions suffice to avoid trouble—though not always, as described later in this article. For functions of any complexity, however, such as "draw string *s* in font *f* at (*x*,*y*) in window *w*," or "find the average of the values stored on a remote file," there's room for surprise if not frustration at the performance. Further, the API documentation provides no hints about which functions are cheap and which are costly.

To complicate matters, after you've tuned your application to the performance characteristics of an API, a new version of the API implementation arrives or a new remote server stores the data

file, and rather than overall performance increasing (the eternal hope for something new), it tanks. In short, the performance contracts that accompany the composition of software components in systems deserve more attention.

A PERFORMANCE TAXONOMY

Programmers begin building intuitive API performance models early in their careers (see box 1). To be useful, the model need not be very accurate. Here's a simple taxonomy:

Always cheap (examples: `toupper`, `isdigit`, `java.util.HashMap.get`). The first two functions are always cheap, usually inlined table lookups. Lookup in a properly sized hash table is expected to be fast, but a hash collision may slow down an occasional access.

Usually cheap (examples: `fgetc`, `java.util.HashMap.put`). Many functions are designed to be fast most of the time but require occasionally invoking more complex code; `fgetc` must occasionally

Forming your own model of performance

In learning to program, you very early on acquire rules of thumb about performance. The pseudocode below is a pattern for reasonably efficient processing of a modest-size file of characters:

```
fs = fopen("~dan/weather-data.txt", "r");   //(1)
for ( i=0; i<10000; i++) {
    ch = fgetc(fs);                         //(2)
    // process character ch
}
...
```

The function call (1) is expected to take a while, but the call to get a character (2) is expected to be *cheap*. This makes intuitive sense: to process a file, a stream need be opened only once, but the "get the next character" function will be called often, perhaps thousands or millions of times.

These two stream functions are implemented by a library. The documentation for the library[2] clearly states what these functions do—an informal presentation of the correctness contract between the implementation and the application. There is no mention of performance, nor any hint to the programmer that the two functions differ substantially in performance. Therefore, programmers build models of performance based on experience, not specifications.[7]

Not all functions have obvious performance properties. For example:

```
fseek(fs, ptr, SEEK_SET); //(3)
```

This function might be cheap when the target file data is already in a buffer. In the general case, it will involve an operating-system call and perhaps I/O. In a wild case, it might require reeling off several thousand feet of magnetic tape. It's also possible that the library function isn't cheap even in the simple case: the implementation may simply store the pointer and set a flag that will cause the hard work to be done on the next stream call that reads or writes data, thus pushing the performance uncertainty onto an otherwise-cheap function.

read a new character buffer. Storing a new item in a hash table may make the table so full that the implementation enlarges it and rehashes all its entries.

The documentation for `java.util.HashMap` makes a commendable start at exposing a performance contract: "This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap..."[3]

The performance of `fgetc` depends on properties of the underlying stream. If it's a disk file, then the function will normally read from a user-memory buffer without requiring an operating-system call, but it must occasionally invoke the operating system to read a new buffer. If it's reading input from a keyboard, then the implementation probably invokes the operating system for each character read.

Predictable (examples: `qsort`, `regexec`). The performance of these functions varies with properties of their arguments (e.g., the size of an array to sort or the length of a string to search). These functions are often data-structure or common-algorithm utilities that use well-known algorithms and do not require system calls. You can usually judge performance based on expectations for the underlying algorithms (e.g., that sort will take n log n time). When using complex data structures (e.g., B-trees) or generic collections (where it may be hard to identify the underlying concrete implementation), it may be harder to estimate performance. It is important to understand that the predictability may be only probable; `regexec` is generally predictable based on its input, but there are pathological expressions that will cause exponential time explosions.

Function creep

```
SetFontSize(f, 10);
SetDrawPosition(w, 200, 20);
DrawText(w, f, "This is a passage.");
```

This example is from a fictitious window system: it sets a font size and a drawing position, then draws some text. You might expect all of these functions to be quite cheap, because rendering text windows on a screen should be fast. Indeed, in early window systems, such as QuickDraw on the Macintosh, you would be right.

The functionality of today's window systems has, rightly, crept upward. Character rasters—even those rendered on a display—are computed from geometric outlines, so preparing a font of a given size may take a while. Modern window systems speed up rendering text by generous use of caching—even saving prepared rasters on disk so that they are available to multiple applications and are preserved across system restarts. Application programmers, however, have no idea whether the fonts they are requesting have been prepared and cached, whether `SetFontSize` will do the (considerable) computation for all of a font's characters, or whether characters will be converted one at a time as needed by strings passed to DrawText.

Augmented functionality may also allow storing font data on network-attached file servers, so font access may take a while, or even fail after a long timeout if the server does not respond.

In many cases the details won't matter, but if the delays are substantial or highly variable, the application might choose to prepare during its initialization all the fonts it plans to use. Most window system APIs do not have such a function.

Unknown (examples: `fopen`, `fseek`, `pthread_create`, many "initialization" functions, and any call that traverses a network). These functions are not cheap and often have high variance in their performance. They allocate resources from pools (threads, memory, disks, operating-system objects), often requiring exclusive access to shared operating-system or I/O resources. There is usually substantial initialization required. Calling across a network is always expensive (relative to local access), but the variation in the expense can be larger still, making the formation of a reasonable performance model much more difficult.

Threads libraries are easy marks for performance issues. The Posix standard took many years to settle down, and implementations are still plagued with problems.[6] Portability of threaded applications remains dicey. Some of the reasons for difficulties with threads are: (1) the need for tight integration with operating systems, almost all of which (including Unix and Linux) were not designed originally with threads in mind; (2) interaction with other libraries, especially making functions thread-safe and dealing with performance problems thereby induced; and (3) several different design points for threads implementations, loosely characterized as lightweight and heavyweight.

SEGMENTING APIS BY PERFORMANCE

Some libraries offer more than one way of performing a function, usually because the alternatives have quite different performance.

Returning to box 1, note that most programmers are told that using a library function to fetch each character is not the fastest way (even if the code for the function is inlined to avoid the function-call overhead). The more performance-conscious will read a very large array of characters and extract each one using array or pointer operations in the programming language. In extreme cases the application can map pages of the file into memory pages to avoid copying data into an array. In return for performance, these functions put a larger burden on the caller (for example, to

When the performance contract breaks

```
SetColor(g, COLOR_RED);
DrawLine(g, 100, 200, 200, 600);
```

Graphics library functions are usually fast. It's reasonable to assume that you draw many lines of varied colors by setting the color for each line; if you're worried that the `SetColor` function is not cheap, you might call it only when the color changes. At one point, the graphics hardware offering of a workstation company required clearing the (hardware) geometry pipeline to change a line's color, so color change was a costly operation. To be fast, the application was, for example, forced to sort lines by color and draw all red lines together. This was not the intuitive way to write the program, and it forced a major change to the structure and programming of the application.

The expensive `SetColor` operation came to be understood as a mistake and was fixed in subsequent hardware. The code that had been written to overcome the mistake, however, may remain, becoming an inexplicable complexity for anyone maintaining the code who doesn't know the performance history of the hardware.

get buffer arithmetic correct and to make the implementation consistent with other library calls such as a call to `fseek` that would require adjusting buffer pointers and perhaps contents).

Programmers are always counseled to avoid premature optimization in their programs, thus delaying extreme revisions until simpler ones have proven inadequate. The only way to be sure about performance is to measure it. Programmers usually write the entire program before confronting mismatches between performance expectations (or estimates) and the reality delivered by an implementation.

PERFORMANCE VARIATION

The performance of "predictable" functions can be estimated from properties of their arguments. The "unknown" functions may also vary greatly depending on what they are asked to do. The time required to open a stream on a storage device will certainly depend on access times, and perhaps data-transfer rates, of the underlying device. Storage accessed via network protocols may be especially costly; certainly it will be variable.

Many cheap functions are cheap only *most* of the time, or they have an *expected* cheap cost. A "get a character" routine must occasionally refill a buffer using an operating-system call that always takes much longer than fetching a character from a full buffer—and might occasionally take a very long time indeed (for example, reading a file on a heavily loaded file server or from a disk that is dying and will succeed only after multiple read retries).

The long story of malloc and dynamic memory allocation

It would be nice if dynamic memory allocation with the `malloc()` function could be characterized as "usually cheap," but that would be wrong since memory allocation—and `malloc` in particular—is one of the first suspects when programmers start hunting for performance problems. As part of their education in performance intuition, programmers learn that if they are calling `malloc` tens of thousands of times, especially to allocate small fixed-size blocks, they are better off allocating a single larger chunk of memory with `malloc`, chopping it up into the fixed-size blocks, and managing their own list of free blocks.

Implementers of `malloc` have struggled over the years to make it usually cheap in the presence of wide variations in usage patterns and in the properties of the hardware/software systems on which it runs.[4] Systems that offer virtual memory, threading, and very large memories all present challenges to "cheap," and `malloc`—along with its complement, `free()`—must trade off efficiency and evils of certain usage patterns such as memory fragmentation.[8]

Some software systems, such as Lisp and Java, use automatic memory allocation along with garbage collection to manage free storage. While this is a great convenience, a programmer concerned with performance must be aware of the costs. A Java programmer, for example, should be taught early about the difference between the String object, which can be modified only by making a new copy in new memory, and a `StringBuffer` object, which contains space to accommodate lengthening the string. As garbage-collection systems improve, they make the unpredictable pauses for garbage collection less common; this can lure the programmer into complacency, believing that the automated reclamation of memory will never be a performance problem, when in fact it is only less often a performance problem.

The functions with "unknown" performance are likely to exhibit wide performance variations for a variety of reasons. One reason is function creep (see box 2), where a generic function becomes more powerful over time. I/O streams are a good example: a call to open a stream invokes very different code in libraries and operating systems depending on the type of stream being opened (local disk file, network-served file, pipe, network stream, string in memory, etc.). As the range of I/O devices and types of files expands, the variance of the performance can only increase. The common life cycle of most APIs—to add functionality incrementally over time—inevitably increases performance variations.

A large source of variation is differences between ports of libraries to different platforms. Of course, the underlying speed of the platform—both hardware and operating system—will vary, but library ports can lead to changes in the *relative* performance of functions within an API or performance across APIs. It's not uncommon for a quick-and-dirty initial port to have many performance problems, which are gradually fixed. Some libraries, such as those for handling threads, have notoriously wide porting-performance variation. Thread anomalies can surface as extreme behaviors—an impossibly slow application or even deadlock.

These variations are one reason why constructing precise performance contracts is difficult. It's usually not necessary to know performance with great precision, but extreme variations from expected behavior can cause problems.

FAILURE PERFORMANCE

The specifications for an API include details of behavior when a call fails. Returning an error code and throwing an exception are common ways of telling the caller that the function did not succeed. As with specifications for normal behavior, however, the *performance* of the failure is not specified. Here are three important cases:

- **Fail fast.** A call fails quickly—as fast or faster than its normal behavior. Calling `sqrt(-1)` fails fast. Even when a `malloc` call fails because no more memory is available, the call should return about as fast as any `malloc` call that must request more memory from the operating system. A call to open a stream for reading a nonexistent disk file is likely to return about as fast as a successful call.
- **Fail slow.** Sometimes a call fails very slowly—so slowly that the application program might have wanted to proceed in other ways. For example, a request to open a network connection to another computer may fail only after several long time-outs expire.
- **Fail forever.** Sometimes a call simply stalls and does not allow the application program to proceed at all. For example, a call whose implementation waits on a synchronization lock that is never released may never return.

Intuition about failure performance is rarely as good as that for normal performance. One reason is simply that writing, debugging, and tuning programs provides far less experience with failure events than with normal events. Another is that a function call can fail in many, many ways, some of them fatal, and not all described in the API's specs. Even exception mechanisms, which are intended to more precisely describe the handling of errors, do not make all possible exceptions visible. Moreover, as library functionality increases, so do the opportunities for failure. For example, APIs that wrap network services (ODBC, JDBC, UPnP,...) intrinsically subscribe to the vast array of network-failure mechanisms.

A diligent application programmer uses massive artillery to deal with unlikely failures. A common

technique is to surround rather large parts of a program with `try...catch` blocks that can retry whole sections that fail. Interactive programs can try to save a user's work using a giant `try...catch` around the entire program, the effect of which is to mitigate failure of the main program by saving, in a disk file, key logs or data structures that record the effects of the work done before the failure.

The only way to deal with stalls or deadlocks is to set up a watchdog thread, which expects a properly running application program to check in periodically with the watchdog, saying, in effect, "I'm still running properly." If too much time elapses between check-ins, the watchdog takes action—for example, saving state, aborting the main thread, and restarting the entire application. If an interactive program responds to a user's command by calling functions that may fail slowly, it may use a watchdog to abort the entire command and return to a known state that allows the user to proceed with other commands. This gives rise to a defensive programming style that plans for the possible abortion of every command.

WHY DOES THE PERFORMANCE CONTRACT MATTER?

Why must an API adhere to a performance contract? *Because major structures of an application program may depend on the API's adherence to such a contract.* A programmer chooses APIs, data structures, and overall program structures based in part on expectations of API performance. If the expectations or the performance are grossly wrong, the programmer cannot recover merely by tuning API calls but must rewrite large, possibly major, portions of the program. The defensive structure of the interactive program previously mentioned is another example.

In effect, a serious violation of the performance contract leads to a composition failure: a program written to the contract cannot be mated to (composed with) an implementation that fails to uphold the contract.

Of course, there are many programs whose structure and performance are influenced very little by library performance (scientific computations and large simulations are often in this category). However, much of today's "routine IT," especially the software that pervades Web-based services, makes extensive use of libraries whose performance is vital to overall performance.

Even small variations in performance can cause major changes in the perception of a program by its users. This is especially true in programs dealing with various kinds of media. Dropping occasional frames of a video stream might be acceptable (indeed, more acceptable than allowing the frame rate to lag other media), but humans have evolved to detect even slight dropouts in audio, so minor changes in the performance of audio media may have major impacts on the acceptability of the overall program. Such worries have led to considerable interest in notions of quality of service, which in many ways is the attempt to ensure performance at a high level.

HOW CAN CONTRACT VIOLATIONS BE AVOIDED?

Although performance contract violations are rare and rarely catastrophic, paying attention to performance when using a software library can help lead to more robust software. Here are some precautions and strategies to use:

1. Choose APIs and program structures carefully. If you have the luxury of writing a program from scratch, ponder performance contract implications as you begin. If the program starts out as a prototype and then remains in service for a while, it will undoubtedly be rewritten at least once; a rewrite is an opportunity to rethink API and structure choices.

2. API implementers have an obligation to present a consistent performance contract as new versions and ports are released. Even a new experimental API will garner users who will begin to derive a performance model of the API. Thereafter, changing the performance contract will surely irritate developers and may cause them to rewrite their programs.

Once an API is mature, it is essential that the performance contract not change. In fact, the most universal APIs (e.g., libc) arguably have become so in part because their performance contracts have been stable during the evolution of the API. The same goes for ports of APIs.

One could hope that API implementers might routinely test new versions to verify that they have introduced no performance quirks. Unfortunately, such testing is rarely done. This doesn't mean, however, that you can't have your own tests for the parts of an API that you depend on. With a profiler, a program can often be found to rely on a small number of APIs. Writing a performance test suite that will compare new versions of a library against the recorded performance of earlier versions can give programmers an early warning that the performance of their own code is going to change with the release of the new library.

Many programmers expect computers and their software to get faster with time—*uniformly.* That is, they expect each new release of a library or a computer system to scale up performance of all API functions equally, especially the "cheap" ones. This is in fact very difficult for vendors to guarantee, but it so closely describes actual practice that customers believe it. Many workstation customers expected newer versions of graphics libraries, drivers, and hardware to increase the performance of *all* graphics applications, but they were equally keen on functionality improvements of many kinds, which usually reduce the performance of older functions, even if only slightly.

One could also hope that API specifications would make the performance contract explicit, so that people using, modifying, or porting the code would honor the contract. Note that a function's use of dynamic memory allocation, whether implicit or automatic, should be part of this documentation.

3. Defensive programming can help. A programmer can use special care when invoking API functions with unknown or highly variable performance; this is especially true for considerations of failure performance. You can move initializations outside performance-critical areas and try to warm up any cached data an API may use (e.g., fonts). APIs that exhibit large performance variance or have a lot of internally cached data could help by providing functions for passing hints from the application to the API about how to allocate or initialize these structures. Occasional pings to servers that a program is known to contact can establish a list of those that might be unavailable, allowing some long failure pauses to be avoided.

One technique sometimes used in graphics applications is to do a dry run, rendering a window of graphics into an off-screen (not visible) window, merely to warm up caches of fonts and other graphics data structures.

4. Tune parameters exposed by the API. Some libraries offer explicit ways of influencing performance (e.g., controlling the size of buffers allocated for files, the initial sizes of tables, or the size of a cache). Operating systems also provide tuning options. Adjusting these parameters can improve performance *within* the confines of a performance contract; tuning does not remedy gross problems but can mitigate otherwise fixed choices embedded in libraries that heavily influence performance.

Some libraries provide alternative implementations of functions with identical semantics, usually in the form of concrete implementations of generic APIs. Tuning by choosing the best concrete implementation is usually very easy. The Java Collections package is a good example of this structure.

Increasingly, APIs are designed to adapt to usage on the fly, freeing the programmer from choosing the best parameter settings. If a hash table gets too full, it is automatically expanded and rehashed (a virtue balanced by, alas, the performance hit of the occasional expansions). If a file is being read sequentially, then more buffers can be allocated so that it is read in larger chunks.

5. Measure performance to verify assumptions. Common advice to programmers is to instrument key data structures to determine whether each structure is being used correctly. For example, you might measure how full a hash table is or how often hash collisions occur. Or you might verify that a structure designed to be fast for reading at the expense of write performance is actually being read more than written.

Adding sufficient instrumentation to measure the performance of many API calls accurately is difficult, a large amount of work, and probably not worth the information it yields. Adding instrumentation on those API calls that are central to the performance of an application (assuming that you can identify them and that your identification is correct), however, can save a lot of time when problems arise. Note that much of this code can be reused as part of the performance monitor for the next release of the library, as mentioned earlier.

None of this is meant to discourage dreamers from working on tools that would automate such instrumentation and measurement, or on ways to specify performance contracts so that performance measurements can establish compliance with the contract. These are not easy goals, and the return may not be huge.

It is often possible to make performance measurements without having instrumented the software in advance (e.g., by using profilers or tools such as DTrace[5]). These have the advantage of not requiring any work until there's a problem to track down. They can also help diagnose problems that creep in when modifications to your code or to libraries upset performance. As *Programming Pearls* author Jon Bentley recommends, "Profile routinely; measure performance drift from a trusted base."[1]

6. Use logs: detect and record anomalies. Increasingly, violations of performance contracts appear in the field when distributed services are composed to form a complex system. (Note that major services offered via network interfaces sometimes have SLAs [service-level agreements] that specify acceptable performance. In many configurations, a measurement process occasionally issues service requests to check that the SLA is met.) Since these services are invoked over a network connection using methods akin to API function calls (e.g., remote procedure call or its variants such as XML-RPC, SOAP, or REST), the expectation of a performance contract applies. Applications detect failure of these services and often adapt gracefully. Slow response, however, especially when there are dozens of such services that depend on each other, can destroy system performance very quickly. Professionally managed services environments, such as those that provide Web services at major Internet sites, have elaborate instrumentation and tooling to monitor Web-service performance and to tackle problems that arise. The far more modest computer collections in homes also depend on such services—many that are part of the operating system on each laptop computer and some that are embedded in appliances on the network (e.g., a printer, network file system, or file backup service)—but there are no aids to help detect and deal with performance problems.

It would be helpful if clients of these services made note of the performance they expect and produced log entries that help diagnose problems (this is what syslog is for). When your file backup seems unreasonably slow (one hour to back up 200 MB), is it slower than yesterday? Slower than it was before the latest operating-system software update? Given that several computers may be sharing

the backup device, is it slower than you should expect? Or is there some logical explanation (e.g., the backup system finds a damaged data structure and embarks on a long procedure to rebuild it)?

Diagnosing performance problems in compositions of opaque software (where no source code is available, and there are no details of the modules and APIs that form the composition) requires that the software play a role in reporting performance and detecting problems. While you cannot address performance problems within the software itself (it's opaque), you can make adjustments or repairs to operating systems and the network. If the backup device is slow because its disk is almost full, then you can surely add more disk space. Good logs and associated tools would help; sadly, logs are an undervalued and neglected area of computer system evolution.

MAKING COMPOSITION WORK

Today's software systems depend on composing independently developed components in such a way that they work—meaning they perform the desired computations at acceptable speed. The dream of statically checking a composition to guarantee that the composition will be correct ("correct by composition") remains elusive. Instead, software-engineering practice has developed methods for testing components and compositions that work pretty well. Every time an application binds to a dynamic library or launches atop an operating-system interface, the correctness of the composition is required.

Despite its importance, the performance of a composition—whether the client and the provider of an interface adhere to the performance contract between them—has been given short shrift. True, this contract is not as important as the correctness contract, but the full power of composition depends on it.

ACKNOWLEDGMENTS

The ideas in this paper are not new; casts of thousands have preceded us. Special mention is due to Butler Lampson, who coined the term *cheap* to describe functions whose performance is fast enough to dispel all attempts to optimize them away; it has a wonderful ring of "economical yet serviceable." Thanks also to Eric Allman, who made helpful comments on a rather too glib early draft, and Jon Bentley, an inveterate performance debugger.

REFERENCES

1. Bentley, J. Personal communication.
2. GNU C Library; http://www.gnu.org/software/libc/manual/html_node/index.html.
3. Java Platform, Standard Edition 7. API Specification; http://docs.oracle.com/javase/7/docs/api/index.html.
4. Korn, D. G., Vo, K.-P. 1985. In search of a better malloc. In *Proceedings of the Summer '85 Usenix Conference:* 489-506.
5. Oracle. Solaris Dynamic Tracing Guide; http://docs.oracle.com/cd/E19253-01/817-6223/.
6. Pthreads(7) manual page; http://man7.org/linux/man-pages/man7/pthreads.7.html.
7. Saltzer, J. H., Kaashoek, M. F. 2009. Principle of least astonishment. In *Principles of Computer System Design*. Morgan Kaufmann: 85.
8. Vo, K.-P. 1996. Vmalloc: a general and efficient memory allocator. *Software Practice and Experience* 26(3): 357-374; http://www2.research.att.com/~astopen/download/ref/vmalloc/vmalloc-spe.pdf.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ROBERT F. SPROULL is an adjunct faculty member of the computer science department at the University of Massachusetts (Amherst), a position he assumed after retiring as director of Sun Microsystems Laboratories.

JIM WALDO is a professor of the practice of computer science at Harvard University, where he is also the chief technology officer, a position he assumed after leaving Sun Microsystems Laboratories.