DOI:10.1145/3015146

Microsecond-scale I/O means tension between performance and productivity that will need new latency-mitigating ideas, including in hardware.

BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN

Attack of the Killer Microseconds

THE COMPUTER SYSTEMS we use today make it easy for programmers to mitigate event latencies in the nanosecond and millisecond time scales (such as DRAM accesses at tens or hundreds of nanoseconds and disk I/Os at a few milliseconds) but significantly lack support for microsecond (µs)-scale events. This oversight is quickly becoming a serious problem for programming warehouse-scale computers, where efficient handling of microsecond-scale events is becoming paramount for a new breed of low-latency I/O devices ranging from datacenter networking to emerging memories (see the first sidebar "Is the Microsecond Getting Enough Respect?").

Processor designers have developed multiple techniques to facilitate a deep memory hierarchy that works at the nanosecond scale by providing a simple synchronous programming interface to the memory system. A load operation will logically block a thread's execution, with the program appearing to resume after the load completes. A host of complex microarchitectural techniques make high performance possible while supporting this intuitive programming model. Techniques include prefetching, out-of-order execution, and branch prediction. Since nanosecond-scale devices are so fast, low-level interactions are performed primarily by hardware.

At the other end of the latency-mitigating spectrum, computer scientists have worked on a number of techniques—typically software based—to deal with the millisecond time scale. Operating system context switching is a notable example. For instance, when a `read()` system call to a disk is made, the operating system kicks off the low-level I/O operation but also performs a software context switch to a different thread to make use of the processor during the disk operation. The original thread resumes execution sometime after the I/O completes. The long overhead of making a disk access (milliseconds) easily outweighs the cost of two context switches (microseconds). Millisecond-scale devices are slow enough that the cost of these software-based mechanisms can be amortized (see Table 1).

These synchronous models for interacting with nanosecond- and millisecond-scale devices are easier than the alternative of asynchronous models. In an asynchronous programming model, the program sends a request to a device and continue processing other work

» key insights

- **A new breed of low-latency I/O devices, ranging from faster datacenter networking to emerging non-volatile memories and accelerators, motivates greater interest in microsecond-scale latencies.**
- **Existing system optimizations targeting nanosecond- and millisecond-scale events are inadequate for events in the microsecond range.**
- **New techniques are needed to enable simple programs to achieve high performance when microsecond-scale latencies are involved, including new microarchitecture support.**

ILLUSTRATION BY PETER CROWTHER ASSOCIATES

until the request has finished. To detect when the request has finished, the program must either periodically poll the status of the request or use an interrupt mechanism. Our experience at Google leads us to strongly prefer a synchronous programming model. Synchronous code is a lot simpler, hence easier to write, tune, and debug (see the second sidebar "Synchronous/Asynchronous Programming Models").

The benefits of synchronous programming models are further amplified at scale, when a typical end-to-end application can span multiple small closely interacting systems, often written across several languages—C, C++, Java, JavaScript, Go, and the like—where the languages all have their own differing and ever-changing idioms for asynchronous programming.[2,11] Having simple and consistent APIs and idioms across the codebase significantly improves software-development productivity. As one illustrative example of the overall benefits of a synchronous programming model in Google, a rewrite of the Colossus[6] client library to use a synchronous I/O model with lightweight threads gained a significant improvement in performance while making the code more compact and easier to understand. More important, however, was that shifting the burden of managing asynchronous events away from the programmer to the operating system or the thread library makes the code significantly simpler. This transfer is very important in warehouse-scale environments where the codebase is touched by thousands of developers, with significant software releases multiple times per week.

Recent trends point to a new breed of low-latency I/O devices that will work in neither the millisecond nor the nanosecond time scales. Consider datacenter networking. The transmission time for a full-size packet at 40Gbps is less than one microsecond (300ns). At the speed of light, the time to traverse the length of a typical datacenter (say, 200 meters to 300 meters) is approximately one microsecond. Fast datacenter networks are thus likely to have latencies of the order of microseconds. Likewise, raw flash device latency is on the order of tens of microseconds. The latencies of emerging new non-volatile memory technologies (such as the Intel-Micron Xpoint 3D memory[9] and Moneta[3]) are expected to be at the lower end of the microsecond regime as well. In-memory systems (such as the Stanford RAMCloud[14] project) have also estimated latencies on the order of microseconds. Fine-grain GPU offloads (and other accelerators) have similar microsecond-scale latencies.

Not only are microsecond-scale hardware devices becoming available, we also see a growing need to exploit lower latency storage and communication. One key reason is the demise of Dennard scaling and the slowing of Moore's Law in 2003 that ended the rapid improvement in microprocessor performance; today's processors are approximately 20 times slower than if they had continued to double in performance every 18 months.[8] In response, to enhance online services, cloud companies have increased the number of computers they use in a customer query. For instance, single-user search query already turns into thousands of remote procedure calls (RPCs), a number that will increase in the future.

Techniques optimized for nanosecond or millisecond time scales do not scale well for this microsecond regime. Superscalar out-of-order execution, branch prediction, prefetching, simultaneous multithreading, and other techniques for nanosecond time scales do not scale well to the microsecond regime; system designers do not have enough instruction-level parallelism or hardware-managed thread contexts to hide the longer latencies. Likewise, software techniques to tolerate millisecond-scale latencies (such as software-directed context switching) scale poorly down to microseconds; the overheads in these techniques often equal or exceed the latency of the I/O device

Table 1. Events and their latencies showing emergence of a new breed of microsecond events.

nanosecond events	microsecond events	millisecond events
register file: 1ns–5ns	datacenter networking: O(1μs)	disk: O(10ms)
cache accesses: 4ns–30ns	new NVM memories: O(1μs)	low-end flash: O(1ms)
memory access: 100ns	high-end flash: O(10μs)	wide-area networking: O(10ms)
	GPU/accelerator: O(10μs)	

Figure 1. Cumulative software overheads, all in the range of microseconds can degrade performance a few orders of magnitude.

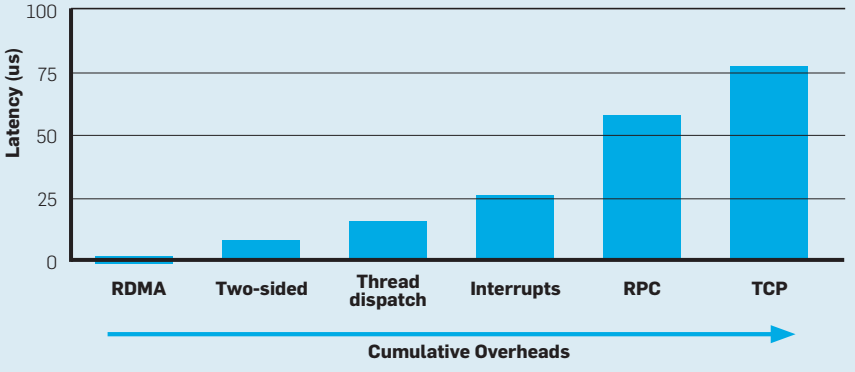


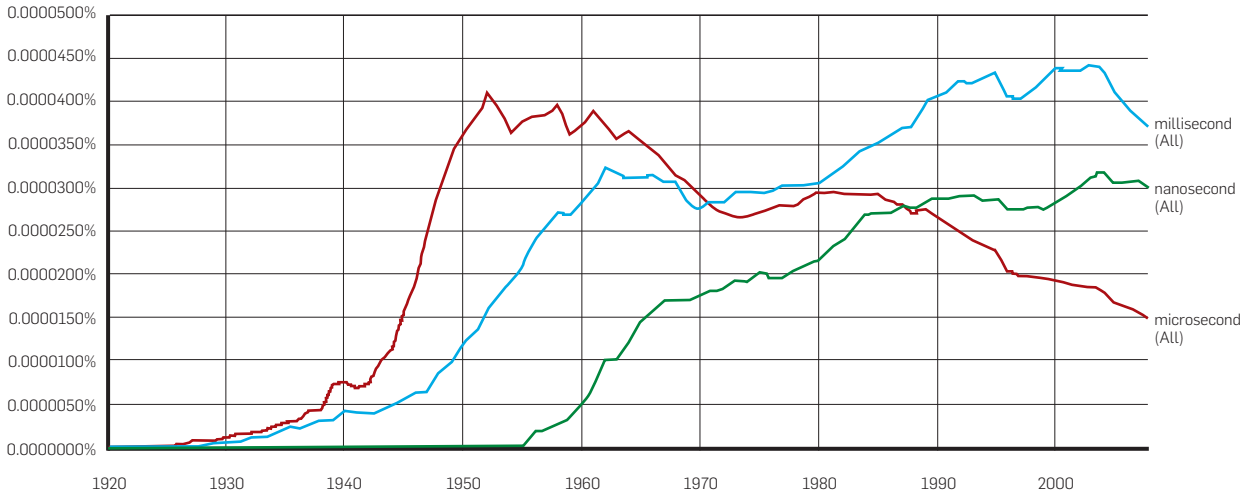
Table 2. Service times measuring number of instructions between I/O events for a production-quality-tuned web search workload. The flash and DRAM data are measured for real production use; the bolded data for new memory/storage tiers is based on detailed trace analysis.

Task: data-intensive workload	Service time (task length between I/Os)
Flash	225K instructions = O(100μs)
Fast flash	***~20K instructions = O(10μs)***
New NVM memory	***~2K instructions = O(1μs)***
DRAM	500 instructions = O(100ns–1μs)

Is the Microsecond Getting Enough Respect?

A simple comparison, as in the figure here, of the occurrence of the terms "millisecond," "microsecond," and "nanosecond" in Google's n-gram viewer (a tool that charts frequencies of words in a large corpus of books printed from 1800 to 2012, https://books.google.com/ngrams) points to the lack of adequate attention to the microsecond-level time scale. Microprocessors moved out of the microsecond scale toward nanoseconds, while networking and storage latencies have remained in the milliseconds. With the rise of a new breed of I/O devices in the datacenter, it is time for system designers to refocus on how to achieve high performance and ease of programming at the microsecond-scale.

n-gram viewer of ms, μs, and ns.



itself. As we will see, it is quite easy to take fast hardware and throw away its performance with software designed for millisecond-scale devices.

How to Waste a Fast Datacenter Network

To better understand how optimizations can target the microsecond regime, consider a high-performance network. Figure 1 is an illustrative example of how a 2μs fabric can, through a cumulative set of software overheads, turn into a nearly 100μs datacenter fabric. Each measurement reflects the median round-trip latency (from the application), with no queueing delays or unloaded latencies.

A very basic remote direct memory access (RDMA) operation in a fast datacenter network takes approximately 2μs. An RDMA operation offloads the mechanisms of operation handling and transport reliability to a specialized hardware device. Making it a "two-sided" primitive (involving remote software rather than just remote hardware) adds several more microseconds. Dispatching overhead from a network thread to an operation thread (on a different processor) further increases latency due to processor-wake-up and kernel-scheduler activity. Using interrupt-based notification rather than spin polling adds many more microseconds. Adding a feature-filled RPC stack incurs significant software overhead in excess of tens of microseconds. Finally, using a full-fledged TCP/IP stack rather than the RDMA-based transport adds to the final overhead that exceeds 75μs in this particular experiment.

In addition, there are other more unpredictable, and more non-intuitive, sources of overhead. For example, when an RPC reaches a server where the core is in a sleep state, additional latencies—often tens to hundreds of microseconds—might be incurred to come out of that sleep state (and potentially warm up processor caches). Likewise, various mechanisms (such as interprocessor interrupts, data copies, context switches, and core hops) all add overheads, again in the microsecond range. We have also measured standard Google debugging features degrading latency by up to tens of microseconds. Finally, queueing overheads—in the host, application, and network fabric—can all incur additional latencies, often on the order of tens to hundreds of microseconds. Some of these sources of overhead have a more severe effect on tail latency than on median latency, which can be especially problematic in distributed computations.[4]

Similar observations can be made about overheads for new non-volatile storage. For example, the Moneta project[3] at the University of California, San Diego, discusses how the latency of access for a non-volatile memory with baseline raw access latency of a few microseconds can increase by almost a factor of five due to different overheads across the kernel, interrupt handling, and data copying.

System designers need to rethink

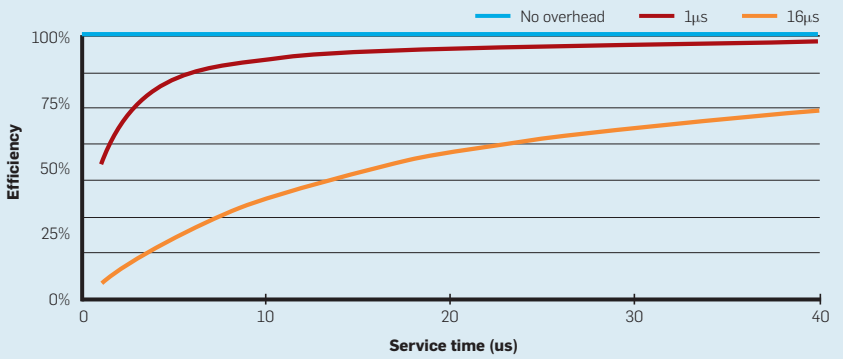
the hardware and software stack in the context of the microsecond challenge. System design decisions like operating system-managed threading and interrupt-based notification that were in the noise with millisecond-scale designs now have to be redesigned more carefully, and system optimizations (such as storage I/O schedulers targeted explicitly at millisecond scales) have to be rethought for the microsecond scale.

How to Waste a Fast Datacenter Processor

The other significant negative effect of microsecond-scale events is on processor efficiency. If we measure processor resource efficiency in terms of throughput for a stream of requests (requests per second), a simple queuing theory model tells us that as service time increases throughput decreases. In the microsecond regime, when service time is composed mostly of "overhead" rather than useful computation, throughput declines precipitously.

Illustrating the effect, Figure 2 shows efficiency (fraction of achieved vs. ideal throughput) on the y-axis, and service time on the x-axis. The different curves show the effect of changing "microsecond overhead" values; that is, amounts of time spent on each overhead event, with the line marked "No overhead" representing a hypothetical ideal system with zero overhead. The model assumes a simple closed queuing model with deterministic arrivals and service times, where service times represent the time between overhead events.

Figure 2. Efficiency degrades significantly at low service times due to microsecond-scale overheads.



As expected, for short service times, overhead of just a single microsecond leads to dramatic reduction in overall throughput efficiency. How likely are such small service times in the real world? Table 2 lists the service times for a production web-search workload measuring the number of instructions between I/O events when the workload is tuned appropriately. As we move to systems that use fast flash or new nonvolatile memories, service times in the range of 0.5µs to 10µs are to be expected. Microsecond overheads can significantly degrade performance in this regime.

At longer service times, sub-microsecond overheads are tolerable and throughput is close to the ideal. Higher overheads in the tens of microseconds, possibly from the software overheads detailed earlier, can lead to degraded performance, so system designers still need to optimize for killer microseconds.

Other overheads go beyond the basic mechanics of accessing microsecond-scale devices. A 2015 paper summarizing a multiyear longitudinal study at Google[10] showed that 20%–25% of fleet-wide processor cycles are spent on low-level overheads we call the "datacenter tax." Examples include serialization and deserialization of data, memory allocation and de-allocation, network stack costs, compression, and encryption. The datacenter tax adds to the killer microsecond challenge. A logical question is whether system designers can address reduced processor efficiency by offloading some of the overheads to a separate core or accelerator. Unfortunately, at single-digit microsecond I/O latencies, I/O operations tend to be closely coupled with the main work on the processor.

It is this frequent and closely coupled nature of these processor overheads that is even more significant, as in "death by 1,000 cuts." For example, if microsecond-scale operations are made infrequently, then conservation of processor performance may not be a concern. Application threads could just busy-poll to wait for the microsecond operation to complete. Alternatively, if these operations are not coupled closely to the main computation on the processor, traditional offload

Table 3. High-performance computing and warehouse-scale computing systems compared. Though high-performance computing systems are often optimized for low-latency networking, their designs and techniques are not directly applicable to warehouse-scale computers.

	High-Performance Computing	Warehouse-Scale Computing
Workloads	Supercomputing workloads that often model the physical world; simpler, static data structures.	Large-scale online data-intensive workloads; operate on big data and complex dynamic data structures; response latency critical.
Programming environment	Code touched by a fewer programmers; slower-changing workloads.	Codebase touched by thousands of developers; significant software releases 100 times per year.
	Hardware concurrency visible to programmers at compile time.	Automatic scale-out of queries per second.
System constraints	Focus on highest performance; recent emphasis on performance per Watt.	Focus on highest performance per dollar.
	Stranding of resources (such as underutilized processors) acceptable.	Significant effort to avoid stranding of resources (such as processor, memory, and power).
Reliability/ security	Reliability in hardware; often no-long-lived mutable data; no encryption.	Commodity hardware; reliability across the stack. Encryption/authentication requirements.

Synchronous/Asynchronous Programming Models

Synchronous APIs maintain the model of sequential execution, and a thread's state can be stored on the stack without explicit management from the application. This leads to code that is shorter, easier-to-understand, more maintainable, and potentially even more efficient. In contrast, with an asynchronous model, whenever an operation is triggered, the code must typically be split up and moved into different functions. The control flow becomes obfuscated, and it becomes the application's responsibility to manage the state between asynchronous event-handling functions that run to completion.

Consider a simple example where a function returns the number of words that appears in a given document identifier where the implementation may store the cached documents on a microsecond-scale device. The example represented in the first code portion makes two synchronous accesses to microsecond-scale devices: the first retrieves an index location for the document, and the second uses that location to retrieve the actual document content for counting words. With a synchronous programming model, it is not only straightforward but the model also can completely abstract away device accesses altogether by providing a natural, synchronous `CountWords` function with familiar syntax.

```
// Returns the number of words found in the document specified by 'doc_id'.
int CountWords(const string& doc_id) {
  Index index;
  bool status = ReadDocumentIndex(doc_id, &index);
  if (!status) return -1;
  string doc;
  status = ReadDocument(index.location, &doc);
  if (!status) return -1;
  return CountWordsInString(doc);
}
```

The C++11 example in the second code portion shows an asynchronous model where a "callback" (commonly known as a "continuation") is used. When the asynchronous operation completes, the event-handling loop invokes the callback to continue the computation. Since two asynchronous operations are made, two callbacks are needed. Moreover the `CountWords` API now must be asynchronous itself. And unlike the synchronous example, state between asynchronous operations must be explicitly managed and tracked on the heap rather than on a thread's stack.

```
// Heap-allocated state tracked between asynchronous operations.
struct AsyncCountWordsState {
  bool status;
  std::function<void(int)> done_callback;
  Index index;
  string doc;
};

// Invokes the 'done' callback, passing the number of words found in the
// document specified by 'doc_id'.
void AsyncCountWords(const string& doc_id, std::function<void(int)> done) {
  // Kick off the first asynchronous operation, and invoke the
  // ReadDocumentIndexDone when it finishes. State between asynchronous
  // operations is tracked in a heap-allocated 'state' object.
  auto state = new AsyncCountWordsState();
  state->done_callback = done;
  AsyncReadDocumentIndex(doc_id, &state->status, &state->index,
                         std::bind(&ReadDocumentIndexDone, state));
}

// First callback function.
void ReadDocumentIndexDone(AsyncCountWordsState* state) {
  if (!state->status) {
    state->done_callback(-1);
    delete state;
  } else {
    // Kick off the second asynchronous operation, and invoke the
    // ReadDocumentDone function when it finishes. The 'state' object
    // is passed to the second callback for final cleanup.
    AsyncReadDocument(state->index.location, &state->status,
                      &state->doc, std::bind(&ReadDocumentDone, state));
  }
}

// Second callback function.
void ReadDocumentDone(AsyncCountWordsState* state) {
  if (!state->status) {
    state->done_callback(-1);
  } else {
    state->done_callback(CountWordsInString(state->doc));
  }
  delete state;
}
```

Callbacks make the flow of control explicit rather than just invoking a function and waiting for it to complete. While languages and libraries can make it somewhat easier to use callbacks and continuations (such as support for async/await, lambdas, tasks, and futures), the result remains code that is arguably messier and more difficult to understand than a simple synchronous function-calling model.

In this asynchronous example, wrapping the code inside a synchronous `CountWords()` function—in order to abstract away the presence of an underlying asynchronous microsecond-scale device—requires support for a `wait` primitive. The existing approaches for waiting on a condition invoke operating system support for thread scheduling, thereby incurring significant overhead when wait times are at the microsecond scale.

engines can be used. However, given fine-grain *and* closely coupled overheads, new hardware optimizations are needed at the processor microarchitectural level to rethink the implementation and scheduling of such core functions that comprise future microsecond-scale I/O events.

Solution Directions

This evidence indicates several major classes of opportunities ahead in the upcoming "era of the killer microsecond." First, and more near-term, it is relatively easy to squander all the benefits from microsecond devices by progressively adding suboptimal supporting software not tuned for such small latencies. Computer scientists thus need to design "microsecond-aware" systems stacks. They also need to build on related work from the past five years in this area (such as Caulfield et al.,[3] Nanavati et al.,[12] and Ousterhout et al.[14]) to continue redesigning traditional low-level system optimizations—reduced lock contention and synchronization, lower-overhead interrupt handling, efficient resource utilization during spin-polling, improved job scheduling, and hardware offloading *but for the microsecond regime*.

The high-performance computing industry has long dealt with low-latency networking. Nevertheless, the techniques used in supercomputers are not directly applicable to warehouse-scale computers (see Table 3). For one, high-performance systems have slower-changing workloads, and fewer programmers need to touch the code. Code simplicity and greatest programmer productivity are thus not as critical as they are in deployments like Amazon or Google where key software products are released multiple times per week. High-performance workloads also tend to have simpler and static data structures that lend themselves to simpler, faster networking. Second, the emphasis is primarily on performance

(vs. performance-per-total-cost-of-ownership in large-scale Web deployments). Consequently, they can keep processors highly underutilized when, say, blocking for MPI-style rendezvous messages. In contrast, a key emphasis in warehouse-scale computing systems is the need to optimize for low latencies while achieving greater utilizations.

As discussed, traditional processor optimizations to hide latency run out of instruction-level pipeline parallelism to tolerate microsecond latencies. System designers need new hardware optimizations to extend the use of synchronous blocking mechanisms and thread-level parallelism to the microsecond range.

Context switching can help, albeit at the cost of increased power and latency. Prior approaches for fast context switching (such as Denelcor HEP[15] and Tera MTA computers[1]) traded off single-threaded performance, giving up on latency advantages from locality and private high-level caches and, consequently, have limited appeal in a broader warehouse-scale computing environment where programmers want to tolerate microsecond events with low overhead and ease of programmability. Some languages and runtimes (such as Go and Erlang) feature lightweight threads[5,7] to reduce memory and context-switch overheads associated with operating system threads. But these systems fall back to heavier-weight mechanisms when dealing with I/O. For example, the Grappa platform[13] builds an efficient task scheduler and communication layer for small messages but trades off a more restricted programming environment and less-efficient performance and also optimizes for throughput. New hardware ideas are needed to enable context switching across a large number of threads (tens to hundreds per processor, though finding the sweet spot is an open question) at extremely fast latencies (tens of nanoseconds).

Hardware innovation is also needed to help orchestrate communication with pending I/O, efficient queue management and task scheduling/dispatch, and better processor state (such as cache) management across several contexts. Ideally, future schedulers will have rich support for I/O (such as being able to park a thread based on the readiness of multiple I/O operations). For instance, facilities similar to the x86 `monitor/mwait` instructions[a] could allow a thread to yield until an I/O operation completes with very low overhead. Meanwhile, the hardware could seamlessly schedule a different thread. To reduce overhead further, a potential hardware implementation could cache the thread's context in either the existing L1/L2/L3 hierarchy or a special-purpose context cache. Improved resource isolation and quality-of-service control in hardware will also help. Hardware support to build new instrumentation to track microsecond overheads will also be useful.

Finally, techniques to enable microsecond-scale devices should not necessarily seek to keep processor pipelines busy. One promising solution might instead be to enable a processor to stop consuming power while a microsecond-scale access is outstanding and shift that power to other cores not blocked on accesses.

Conclusion

System designers can no longer ignore efficient support for microsecond-scale I/O, as the most useful new warehouse-scale computing technologies start running at that time scale. Today's hardware and system software make an inadequate platform, particularly given support for synchronous programming models is deemed critical for software productivity. Novel microsecond-optimized system stacks are needed, reexamining questions around appropriate layering and abstraction, control and data plane separation, and hardware/software boundaries. Such optimized designs at the microsecond scale, and corresponding faster I/O, can in turn enable a virtuous cycle of new applications and programming models that leverage low-latency communication, dramatically increasing the effective computing capabilities of warehouse-scale computers.

Acknowledgments

We would like to thank Al Borchers, Robert Cypher, Lawrence Greenfield, Mark Hill, Urs Hölzle, Christos Kozyrakis, Noah Levine, Milo Martin, Jeff Mogul, John Ousterhout, Amin Vahdat, Sean Quinlan, and Tom Wenisch for detailed comments that improved this article. We also thank the teams at Google that build, manage, and maintain the systems that contributed to the insights we have explored here.

a The x86 `monitor/mwait` instructions allow privileged software to wait on a single memory word.

References

1. Alverson, R. et al. The Tera computer system. In *Proceedings of the Fourth International Conference on Supercomputing* (Amsterdam, The Netherlands, June 11–15). ACM Press, New York, 1990, 1–6.
2. Boost C++ Libraries. Boost asio library; http://www.boost.org/doc/libs/1_59_0/doc/html/boost_asio.html
3. Caulfield, A. et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 IEEE/ACM International Symposium on Microarchitecture* (Atlanta, GA, Dec. 4–8). IEEE Computer Society Press, 2010.
4. Dean, J. and Barroso, L.A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.
5. Erlang. *Erlang User's Guide Version 8.0. Processes*; http://erlang.org/doc/efficiency_guide/processes.html
6. Fikes, F. Storage architecture and challenges. In *Proceedings of the 2010 Google Faculty Summit* (Mountain View, CA, July 29, 2010); http://www.systutorials.com/3306/storage-architecture-and-challenges/
7. Golang.org. Effective Go. Goroutines; https://golang.org/doc/effective_go.html#goroutines
8. Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach, Sixth Edition.* Elsevier, Cambridge, MA, 2017.
9. Intel Newsroom. Intel and Micron produce breakthrough memory technology, July 28, 2015; http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology
10. Kanev, S. et al. Profiling a warehouse-scale computer. In *Proceedings of the 42nd International Symposium on Computer Architecture* (Portland, OR, June 13–17). ACM Press, New York, 2015.
11. Microsoft. *Asynchronous Programming with Async and Await (C# and Visual Basic)*; https://msdn.microsoft.com/en-us/library/hh191443.aspx
12. Nanavati, M. et al. Non-volatile storage: Implications of the datacenter's shifting center. *Commun. ACM 50*, 1 (Jan. 2016), 58–63.
13. Nelson, J. et al. Latency-tolerant software distributed shared memory. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, July 8–10). Usenix Association, Berkeley, CA, 2015.
14. Ousterhout, J. et al. The RAMCloud storage system. *ACM Transactions on Computer Systems 33*, 3 (Sept. 2015), 7:1–7:55.
15. Smith, B. A pipelined shared-resource MIMD computer. Chapter in *Advanced Computer Architecture.* D.P. Agrawal, Ed. IEEE Computer Society Press, Los Alamitos, CA, 1986, 39–41.
16. Wikipedia.org. Google n-gram viewer; https://en.wikipedia.org/wiki/Google_Ngram_Viewer

Luiz André Barroso (luiz@google.com) is a Google Fellow and Vice President of Engineering at Google Inc., Mountain View, CA.

Michael R. Marty (mikemarty@google.com) is a senior staff software engineer and manager at Google Inc., Madison, WI.

David Patterson (davidpatterson@gmail.com) is an emeritus professor at the University of California, Berkeley, and a distinguished engineer at Google Inc., Mountain View, CA.

Parthasarathy Ranganathan (partha.ranganathan@google.com) is a principal engineer at Google Inc., Mountain View, CA.

Watch the authors discuss their work in this exclusive *Communications* video. http://cacm.acm.org/videos/the-attack-of-the-killer-microseconds