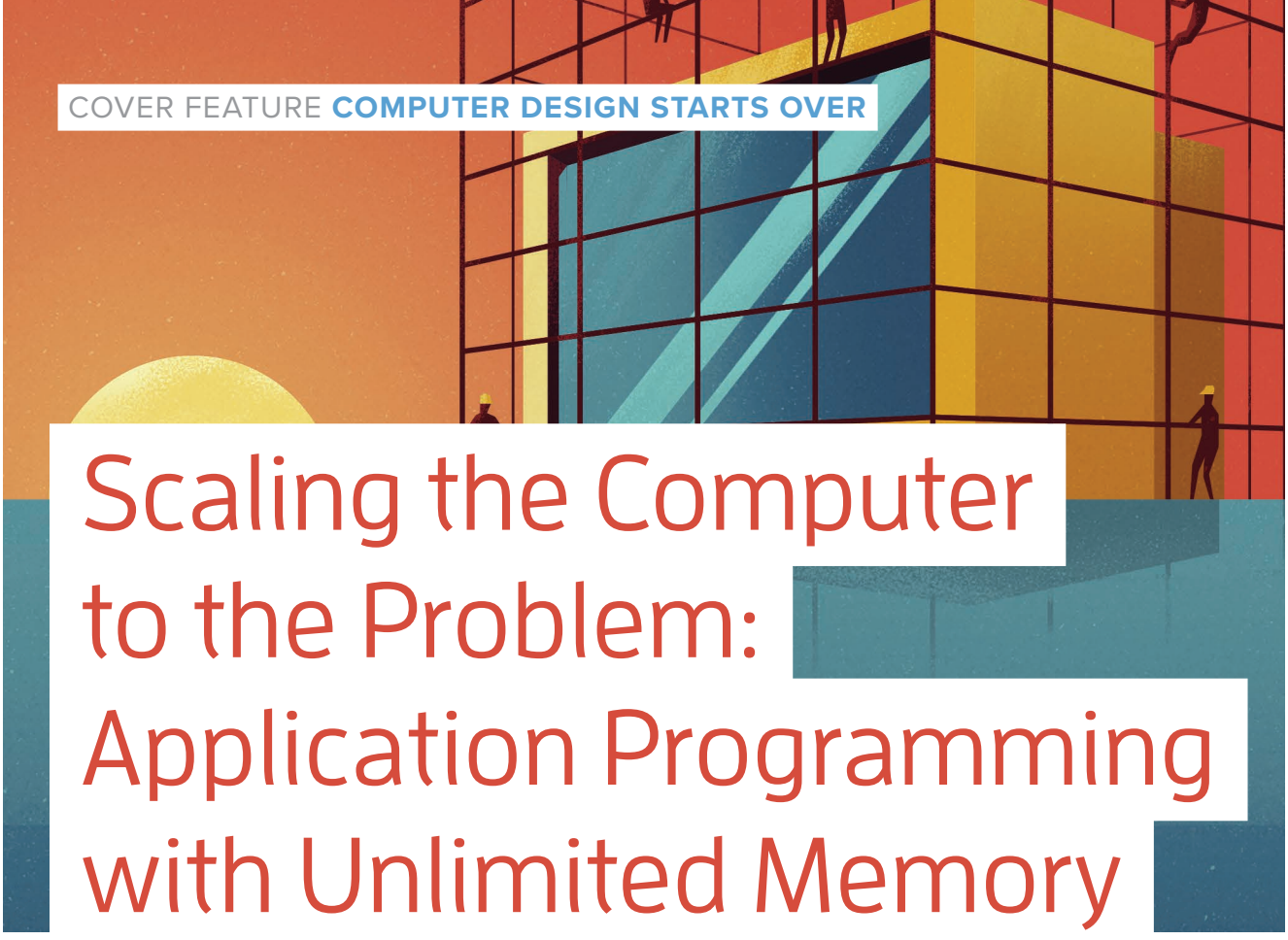

Scaling the Computer to the Problem: Application Programming with Unlimited Memory

Ike Nassi, TidalScale and University of California, Santa Cruz

Instead of scaling an application and data around the computer, programmers can use a software-defined server—an inverse hypervisor—in which multiple physical machines run a single virtual machine. Memory can be expanded as needed without modifying the application or limiting its data.

In computing's early days, memory and address space were significant limitations. Virtual memory evolved to broaden those limits, greatly expanding the amount of memory available to an application. So, OSs could now automatically manage the mapping of the application's virtual memory to the system's physical memory. This allowed software to become more ambitious because virtual memory could be much larger than the physical memory.

However, this expansion gave rise to a disparity between virtual-memory size and physical-memory size, which required introducing paging to backing (secondary) storage to accommodate information not currently in main memory. Paging, in turn, created application performance overhead. To improve performance, the computing industry turned to hardware dynamic address translation to convert virtual-memory addresses to physical-memory addresses.

But the foundational problem remained: applications are frequently insatiable memory consumers, particularly in the current big data era. To avoid this dilemma, industry and academia have been exploring scale-out alternatives.[1,2] Some experts point to the combination of cloud computing and these scale-out alternatives as the solution to the ever-expanding demand for memory. But far from being a panacea, the cloud makes the entire scale-out process more difficult because now the information mapping of data to physical machines must occur across a massive cloud. MapReduce software, such as Apache Hadoop (hadoop.apache.org) and Apache Spark (spark.apache.org), was developed to help manage the additional complexity, but software must be written

 0018-9162/17/$33.00 © 2017 IEEE

or rewritten to align with the broader distribution of information across the cloud. Virtualization is a well-established technology for multiplexing a set of virtual machines (VMs) onto a single physical server, using a software hypervisor.

At TidalScale, we began asking questions about how to use virtualization to avoid scale-out's extra work:

- What if all of a cluster's nodes could be combined to form a single virtual computer that contained all the memory, all the processors, all the networks, and all the disks?
- What if that virtual computer could automatically optimize itself? Could it be better and faster than humans at adjusting its behavior and operations?
- What if the computer could get bigger and better without needing new silicon generations—for example, use more main memory, cores, Ethernets, or disks and better aggregate memory and PCI bandwidth? Would it be possible to opportunistically reduce the computer's size without changing the application, so as to increase the datacenters' energy efficiency or enable them to use servers more effectively?
- How can application software best exploit such a computer?

Although the first two questions are intriguing, if not controversial, we focused on the last two. To investigate answers to these questions, TidalScale developed a *hyperkernel*, which is essentially an inverse hypervisor. In a traditional virtual environment, multiple VMs share a single physical machine. In an inverse hypervisor, multiple physical machines run a single VM, and each hyperkernel instance runs on a physical server. By connecting individual physical servers on a standard private interconnect, a set of tightly coupled hardware servers each running this hyperkernel can cooperatively form a single large VM running a single standard OS. The OS runs on what it views as a single hardware server but what is actually a virtual server. We call this VM a *software-defined server*.

A software-defined server exploits the idea that physical memory as seen by a guest OS can differ from real physical memory. Guest physical memory is the sum of all the physical memory of all the physical servers. So, it can be significantly larger than previous memory limits have allowed—sizable enough that physical memory might no longer be the limitation it has been, and paging could be greatly reduced or even eliminated. The hyperkernel sees the OS's view of the physical memory it is managing as a second level of virtual memory and the OS's view of the physical processors it is managing as virtual processors. A similar situation exists with networks and storage.

HOW MEMORY DEMAND GREW

Memory started out as a costly resource that needed to be conserved. In the 1970s, for example, the PDP-11 had a 16-bit physical address space, in which all effective addresses had to fit. Addressable memory locations were thus limited to 2^{16} (65,536) addressable locations of 8-bit bytes.

Virtual memory

Over time, memory became less expensive, but expanding software ambitions motivated the quest for even larger address spaces. Virtual memory was introduced, which gave programmers the memory address space they sought. Programmers had the illusion of dealing with physical memory, although it was actually an artificial representation of memory. The association between virtual-memory addresses and physical-memory addresses was, and still is, invisible to most applications. From a programmer's viewpoint, reading and writing appear to be in physical memory, not virtual memory. The location of the physical memory represented by the virtual memory can vary over time, as the program's execution progresses.

Virtual memory might be partly or entirely contained in physical memory, or it might be paged out and actually reside in backing storage (partly or entirely). The OS manages the mapping between virtual and physical memories with hardware support in the form of dynamic address translation. When presented with a virtual-memory address, the hardware for dynamic address translation converts the virtual-memory address to a physical-memory address.

Virtualization

Hardware support for virtualization—not to be confused with virtual memory—was introduced to accommodate the need for different VMs to share a physical computer.[3] Before virtualization, the OS was solely responsible for mapping the application's virtual memory to physical memory. With hardware virtualization support, the OS now manages the mapping between guest virtual memory and guest physical memory. However, the virtualization software becomes responsible for mapping guest physical memory to real physical memory.

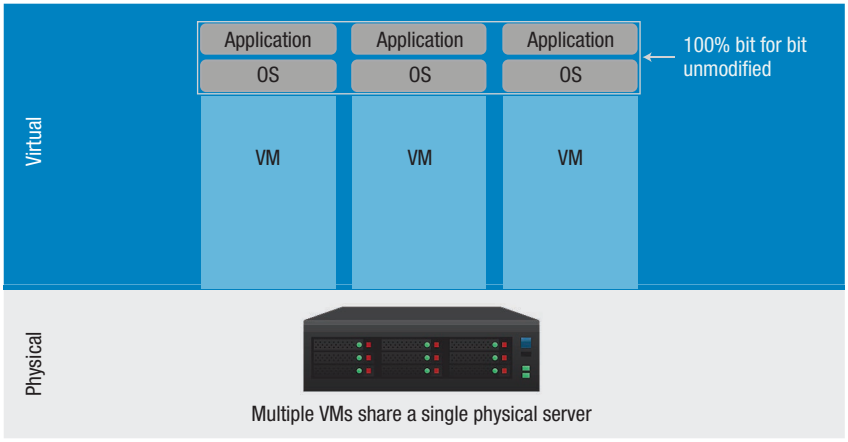

FIGURE 1. Traditional virtualization. In a traditional virtual environment, a set of virtual machines (VMs) share a single physical machine. No modifications are needed for the OS or for applications. However, for modern big data applications, more memory is better, not less.

Because the OS doesn't need to know this is happening and because this hardware virtualization support is invisible to the OS, a virtualization system can potentially run an unmodified guest OS (see Figure 1). The OS continues to be responsible for managing any insufficiencies in guest physical-memory size, assisted by paging and first-level dynamic address translation, which continues to move memory blocks to and from backing storage to compensate. The result is that virtualization meets the application's memory needs. However, the virtualization system now becomes responsible for managing second-level dynamic address translation.

Application programming models that exploit virtualization have remain largely unchanged, which was virtualization's goal. For the most part, virtualization still works well, but for applications there is a catch. The maximum amount of real physical memory that a system can support is not up to either the application or the OS. It is defined by the number of physical pins on the microprocessor chip that are used for reading, writing, and addressing physical memory, interprocessor communication, and communication with external devices. Quite simply, mechanical limits on pin count and interprocessor protocols limit the amount of physical addressable memory on a single system.

Shared-memory multiprocessing

Shared-memory multiprocessors were created to increase the number of processors available for use. However, simply increasing the number of processors does not remove the obstacle of having insufficient memory available for applications.[4] Shared-memory multiprocessors failed in that part of the mission because adding processors did not change the amount of physically addressable shared memory. Unfortunately, the same memory limitation applies to modern multicore processors.

Scale-out computing

Big data, particularly in-memory computing, is the latest stimulant to whet the memory appetite. Virtual memory cannot replace the need for increased real memory.[5] Cloud computing was introduced to increase the available computation and memory resources for applications. However, that expansion comes at the cost of more complexity in mapping memory and computation in a distributed environment. Incorporating MapReduce software is a hassle. In theory, scale-out computing is effective if programmers can live with rewriting software and managing data placement to accommodate it.

THE SOFTWARE-DEFINED SERVER

But is there a better way? We think a valid alternative is our software-defined server, which adopts a single-system-image approach. Figure 2 shows how several such servers can be organized.

Our goal at TidalScale was to achieve the simplicity of scale-up with the linear-cost economics of scale-out. We wanted users to be able to scale their computer to their problem instead of having to scale their problem to their computer. In short, we wanted a software-defined server that provided the best of both the scale-out and scale-up worlds.[6]

Why not just scale-out?

The combination of scale-up and scale-out can yield many additional benefits over scale-out alone. As with scale-out computing, an organization can start with a small number of less expensive networked computers rather than buying or renting a single, expensive supercomputer. As the organization's needs evolve, it can add physical computers, resulting in more efficient use of capital. So, hardware costs grow linearly in cost and dynamically over time. Using software-defined servers can also be more energy efficient, and servers can easily be repurposed to increase a datacenter's server utilization.

No modification. Scale-up has the major advantage that applications and OSs can run with no modifications—a tremendous plus for legacy applications—and with unmodified hardware. In addition, there is less need for new hardware. The hardware features required for software-defined servers, principally virtualization support that delivers two levels of automatic dynamic address translation, are already in production and widely used in modern microprocessors.

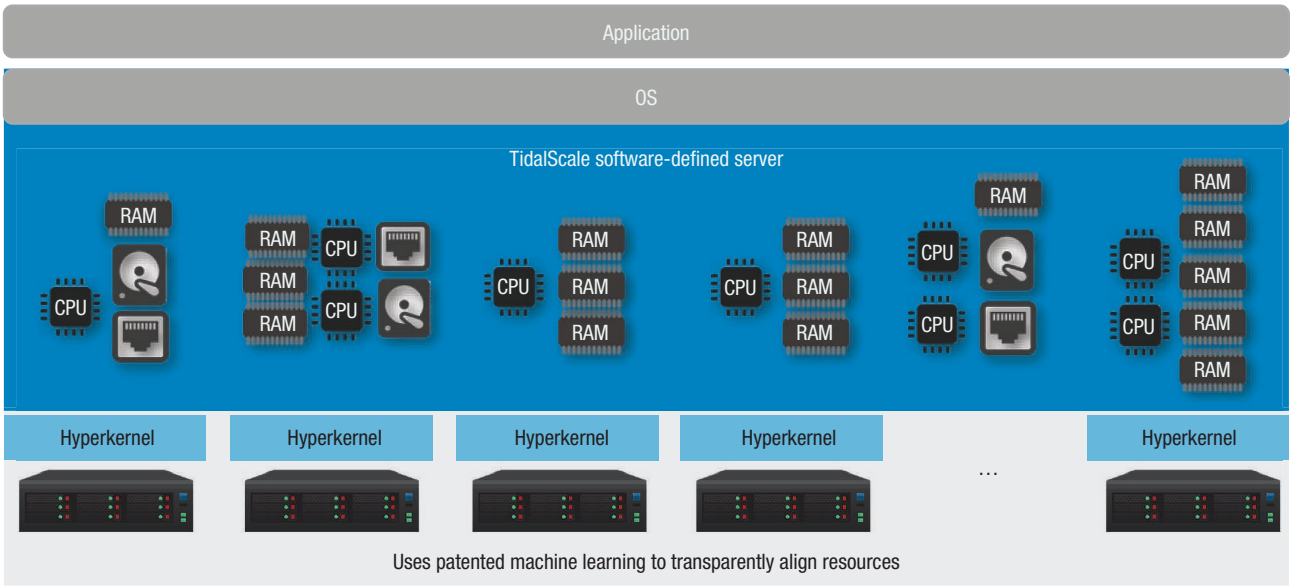

FIGURE 2. A software-defined server. A collection of physical servers, connected by a private high-speed network and running a cooperating set of hyperkernels, can form a VM running a standard unmodified OS that supports unmodified applications. Virtual memory and virtual processors can automatically migrate among physical servers on demand to satisfy the needs of applications running on a guest OS.

Virtualization indirection. Software-defined servers make it possible to introduce a level of virtualization indirection between the OS and the physical hardware and provide a place where the virtualization system could optimize system performance without any modifications to the application or the OS. Such virtualization is not possible with scale-out alone. Virtualization indirection could also dynamically modify system reliability, making the system more resilient to certain failure types. For example, an OS might have difficulty dealing with hardware resource failures because it assumes that it is dealing with reliable, physical hardware. However, if the same resources, as seen by the VM, are mobile and migratable, the hyperkernel could, in many cases, replace those failing resources with alternatives without the guest OS's knowledge or participation. In other words, the VM's reliability can exceed that of the physical machine.

Fewer cores per processor. As in scale-out, adding computers can increase not only the total number of processors but also the available hardware bandwidth to memory and the total network and I/O bandwidth. For example, given 10 servers, each with a certain amount of memory bandwidth, adding a server increases the raw memory bandwidth by 10 percent and increases the number of processors and the I/O and networking bandwidth.

This approach contrasts sharply with the strategy of adding cores to each processor, which, because of power dissipation constraints, decreases processor frequencies and thus degrades single-stream computing's performance. With fewer cores per processor, manufacturers can offer processors at higher operating frequencies, which improves single-stream performance. However, an additional problem is that memory densities have not kept pace with processor-core densities, so the ratio of available memory to available cores has actually been declining. An alternative to reducing the number of available cores is to use software-defined servers to combine the cores in ways that avoid compromising core-count requirements.

Using unlimited memory

One of TidalScale's goals was to never require software modification, and we achieved that goal. But another question is how a greatly relaxed memory limitation might change the nature of an application written for limited memory, giving it novel possibilities that were impossible or impractical before. Thus, we sharpened the last original question (how can application software best exploit such a computer?) to, how might application developers reconsider their software assumptions?

Recall that when an application uses more virtual memory than physical memory, paging occurs, which can degrade performance. Access to main memory is about three orders of magnitude faster than the fastest solid-state disk drives currently available. Because of paging, applications that regularly use more virtual memory than physical memory often hit a memory cliff—the point at which performance as measured in total elapsed time drops dramatically.[7] Although it is highly desirable to have sufficient memory to support in-memory computing, the required amount of memory can often be very expensive and might not even be achievable.

From an application performance view, moving large amounts of data

costs time and energy, regardless of where it is moved, be it within a single server, between a motherboard and storage, or across a network. The cost is due to not only transmission time but also the reality that interconnect resources (memory buses, networks, and PCI buses) are always time and energy constrained. If hardware resources are used to move data from memory to backing storage, they cannot be used at the same time for any other purpose. Faster interconnect technologies are not the answer. Regardless of interconnect speed, not having to move data is always better than having to move it.

> WITH GREATLY RELAXED MEMORY LIMITATIONS, APPLICATION DEVELOPERS CAN CONSIDER IMPLEMENTATIONS THAT WERE ONCE DEEMED IMPRACTICAL.

APPLYING EXPANDED MEMORY

A large, uniform, and strongly coherent memory space has a several advantages that algorithms can exploit. Consider data stored in tables. Instead of moving the rows of the tables stored in memory to achieve some specified effect, we can use pointers to rows rather than the rows themselves, and not have to move any memory at all. However, this is possible only if every processor can directly address every byte of memory. Manipulating pointers to data can be much, much less expensive than manipulating the data itself. An interesting problem in the financial-technology domain illustrates how cost can be reduced.

Problem definition

Suppose that each of 3,000 securities is represented by two tables of historical data, labeled Left and Right. Each table has a column of time stamps and a column containing the name of the security it represents (for example, AAPL and GM) as well as some additional data. Suppose also that Left has 150 additional data columns, Right has 100 additional data columns, and each table has 10,000 rows. All these constants are arbitrary. The numbers can be much larger, or smaller, depending on specific needs.

The information in the tables could be initially read from disk or a networked data repository or streamed directly into memory in real time. In this example, we assume that the tables are artificially generated and that the data, because it is historical, is rarely (if ever) updated.

All rows of data then need to be merged into a single master table, sorted by time stamp. For good performance, the sorted in-memory data must be ready for many different queries without paging, and the query order is unpredictable.

Increasing the number of securities, the number of rows, or the number of columns of data per row can greatly enlarge the amount of data being managed. The cost of a single computer to manage this much in-memory data might be extremely expensive, possibly even impractical, with current technology. The network traffic for a scale-out implementation can be high and exhibit an unpredictable amount of network congestion. Bugs might also be introduced into a complex distributed application.

Possibly, no single affordable physical server exists that can be deployed for this problem type.

The software-defined server as a solution

With a software-defined server, however, the problem is manageable. By combining a set of commodity servers into a single software-defined server that the application views as a single server large enough to hold all the in-memory data, the application can now support fast query response. In fact, TidalScale has already developed a prototype solution and made it freely available at tidalscale.com/how_to_use_large_memory.

Ingesting data. Tables can be created in parallel using subthreads, one for each security. Each subthread generates (or loads from a local or remote database) the data for each security into a table stored in a region of shared memory that the parent thread can directly access. Because all the subthreads operate in parallel, ingestion has the potential to be fast. When a subthread finishes, it notifies the parent that it has completed its job and then terminates, leaving its tables in shared memory. Ultimately, when all subthreads have finished, only the parent thread remains. All the data

remains in main memory, which the parent thread can directly address.

The data and the ingesting thread for a security are most likely on the same physical hardware server for two reasons. First, securities were loaded in parallel by their own separate threads into the memory of the node on which the thread was running. Second, the software-defined server understands affinity and locality, and tends to keep the thread and its data together. Data ingestion is embarrassingly parallel. Because the threads themselves are spread out across nodes through strategies employed by the hyperkernel, the data will be also spread out over as many nodes as it finds necessary or available.

Merging and sorting data. Now that all the data is in memory, the parent thread can merge the rows virtually. To do so, it first builds a table of pointers to all the rows of all the tables of all the securities. The total number of rows is the number of securities × number of rows × 2. The table of pointers is in essence a faithful representation of a single combined table, which the parent thread then sorts in order of the time stamps of the rows to which it points. One sorted, the application is then ready to respond to arbitrary queries.

The merge step is quite simple and fast. In the sample program, pointers to the data from each security are copied in arbitrary order along with some metadata into a single merged array. A recursive merge sort, running in $O(n\log n)$ time, results in a view of all the data sorted by time stamp. Thus, it fully exploits the combined aggregated processors, combined memory, and combined memory bandwidth. Moreover, once the tables for each security are read (or generated), they need never move and are, in this example, never updated. Consequently, no data is ever lost. If desired for archiving or checkpointing, the (virtually) merged table can be written to backing storage as a physical table, in time-stamp order.

The point is not that this algorithm is unique. In fact, it's the obvious algorithm to use. The key point is that unless the address space is sufficiently large, the obvious algorithm will not be able to be run at all!

The software-defined server is a viable scale-up alternative to scale-out, removing many of the current boundaries on an application's memory use. The server also aggregates processors, enabling designs to use many fast single-stream processors rather than the more expensive and slower high-density multicore processors. Finally, once real, physical memory is no longer the limitation of the past, developers can rethink how best to use it, which could motivate the creation of software application architectures with significantly less complexity and significantly better performance.

ABOUT THE AUTHOR

IKE NASSI is the founder, chairman, and chief technical officer of TidalScale; an adjunct professor of computer science at the University of California, Santa Cruz; and a founding trustee of the Computer History Museum. His research interests include programming languages, OSs, system architecture, and the history of computer science and mathematics. Nassi received a PhD in computer science from Stony Brook University. He is a Senior Member of IEEE. Contact him at ike.nassi@tidalscale.com.

REFERENCES

1. R. White, "The Single System Image Feature Delivers Greater Flexibility and Resilience," *IBM Systems Magazine*, May 2013; www.ibmsystemsmag.com/mainframe/administrator/Virtualization/ssi_feature_zvm.
2. R. Buyya, "Architecture Alternatives for Scalable Single System Image Clusters," *Proc. 1999 Conf. High Performance Computing on Hewlett-Packard Systems* (HiPer 99), 1999; buyya.com/papers/ssiArch.html.
3. M. Rosenblum, "The Reincarnation of Virtual Machines," *ACMQueue*, vol. 2, no. 5, 2004; doi:10.1145/1016998.1017000.
4. G. Bell et al., "The Encore Continuum: A Complete Distributed Workstation-Multiprocessor Computing Environment," *Proc. Nat'l Computer Conf.* (NCC 85), 1985, pp. 147–155.
5. I. Nassi, "A Preliminary Report on the UltraMax," *Proc. DARPA Conf. Mathematical and Scientific Computing*, 1987.
6. M. Sevilla et al., "A Framework for an In-Depth Comparison of Scale-Out and Scale-Up," *Proc. 2nd Int'l Workshop Data-Intensive Scalable Computing Systems* (DISCS 13), 2013; issdm.soe.ucsc.edu/sites/default/files/sevilla-discs13.pdf.
7. P.-H. Kamp, "You're Doing It Wrong," *ACMQueue*, vol. 8, no. 6, 2010, pp. 20–27.