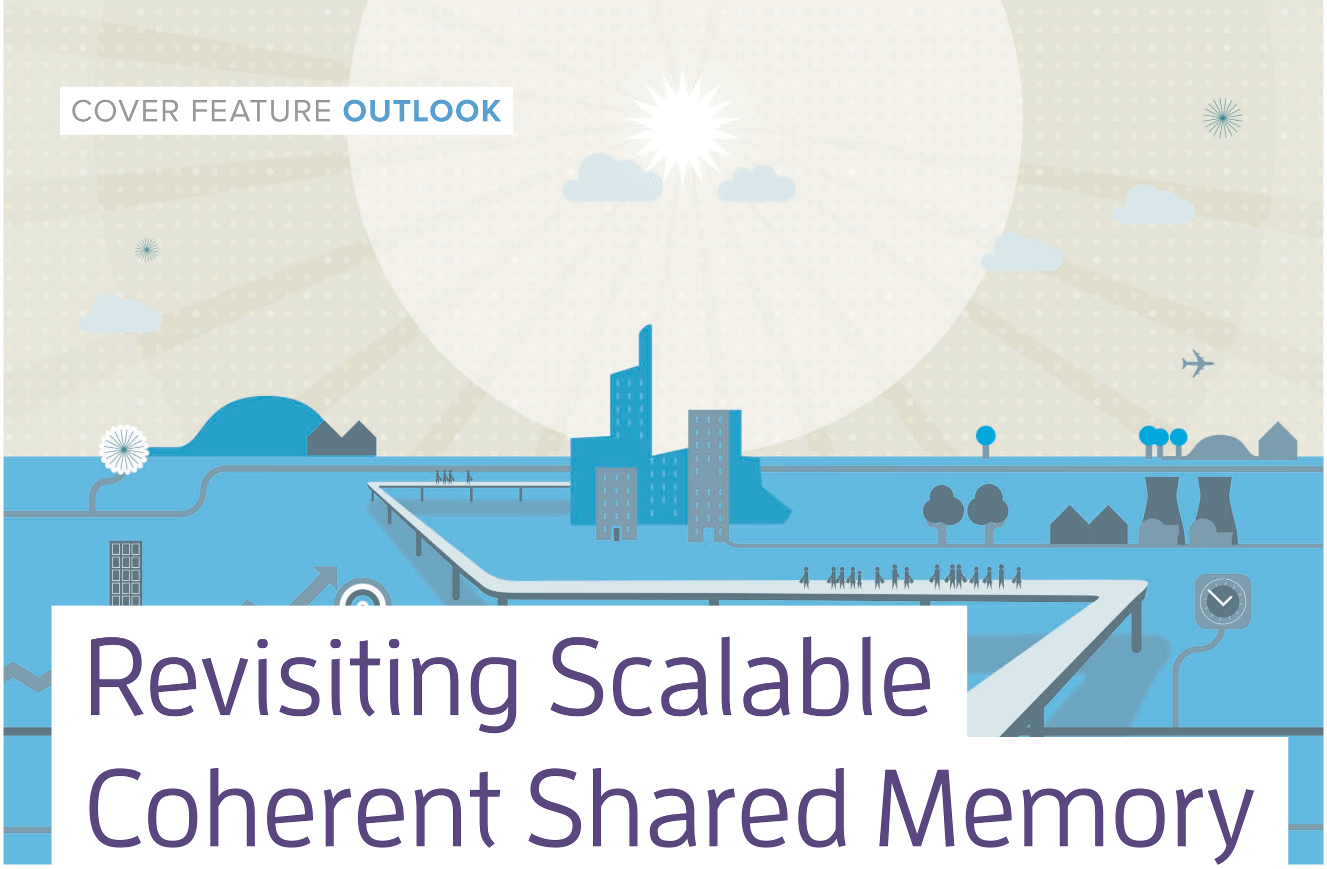

Revisiting Scalable Coherent Shared Memory

C. Gordon Bell, Microsoft Research (retired)
Ike Nassi, TidalScale and University of California, Santa Cruz

Scalable and coherent shared memory has been a long-sought-after but elusive goal. In contrast to today's popular distributed-computing models, the authors present a software-defined server architecture that is a scale-up shared-memory multiprocessor, yet uses ubiquitous commodity scale-out clusters.

When our exploration of architectures began, high-performance systems were few and expensive and access was limited. Today we have inexpensive, elastic, computation services that, on-demand, provide a multiprocessor, multithread computing platform, perhaps creating the illusion that the underpinning hardware system just works as expected. However, some experiments on Amazon Web Services (AWS) gave surprising results.[1] Running a CPU- and memory-intensive data generation application on a four-processor AWS instance showed normalized CPU utilization of 89 percent on a program utilizing 32 parallel threads. The application took 4,832 CPU seconds (see Table 1). Looking to explore this a bit further, the degree of parallelism was reduced to 16 cores, and it actually sped the program up. Repeating this process, the best trial exhibited a 15.79× performance improvement using only 10 percent of the threads. Since I/O was not a factor, this suggests that memory contention by the physical processors is a significant issue. Can we do something about this?

BACKGROUND

The high-performance computer (HPC; also known as supercomputing) market transition from monomemory computers to multicomputers began in 1987 when a Sandia National Laboratories team won the first Gordon Bell Prize for parallelism using 1,024 individual computers (referred to as nodes) organized as a single Ncube computer. In 1993, a 1,024-node Connection Machine from Thinking Machines outperformed all the traditional supercomputers, such as the Cray YMPs that had reached the limit of being able to scale-up. The 1994 MPI and Beowulf standards established the beginning of the transition to clusters of computers for HPC. In 2018, all HPC apps run across a collection of a few hundred to up to 10 million computers tightly connected into a network.

The situation in commercial transaction processing was considerably different because application programs operated across a system that assumed parallelization at the application level. New programming languages were developed that had explicit and implicit concurrency support. In 1979, a team from Caltech formed Teradata and built a highly scalable, parallel relational database built on "shared nothing" access to disk storage. Similarly, today's cloud web applications can be thought of as a large collection of parallel and pipelined processes, but again such configurations are static with explicit process-to-process communication.

In the mid-2000s with the introduction of MapReduce[2] and Hadoop,[3] a transition in programs occurred that required access to substantially larger and more diverse resources. These techniques all required large memories. A few years ago, machine learning began to achieve production status. These applications began to look a lot like HPC, and vice versa. Convergence was occurring. Taken together, these techniques necessitate advances in distributed architecture to simplify programming, because of the need to manage and analyze ever increasing amounts of data. These hardware and software architectures are being called on to provide more storage, more connectivity, and more computing power, including franken-architectures with diverse collections of cpus, gpus, field-programmable gate arrays, and TPUx. The cost of managing these distributed architectures is growing as a result of the complexity of managing the infrastructure to deal with it and the software modifications needed.

In current architectures, 48-bit memory address limitations stand in the way of very large memories. Processors need physical paths to memory for addressing and data transfer, and if the memory is shared, there needs to be arbitration mechanisms among the processors to coordinate access to shared memory. As the number of processors (n) increases, the amount of coordination goes up by n^2. Therefore, we trade hardware complexity for software complexity. We build distributed systems that allow data and computation to spread out over a large number of servers that partition data and computations as well as manage server operations. But whereas data partitioning might be straightforward, and the computation partitioning might be straightforward, doing both at the same time is far more difficult.

TABLE 1. Results of experiment in which CPU- and memory-intensive data generation application was run on four-processor Amazon Web Services instance.

Degree of parallelism	CPU seconds	Normalized CPU utilization (%)	Total elapsed time (s)
1	259.0	3	257.7
2	335.0	6	165.0
2	333.0	6	164.2
3	306.0	9	102.4
3	375.0	10	122.5
4	484.0	13	118.6
4	482.0	13	117.7
4	475.0	13	116.1
4	356.0	13	86.4
5	563.0	16	110.4
5	590.0	16	115.6
6	670.0	19	109.2
8	993.0	26	121.1
16	2,200.0	50	137.1
32	4,832.0	89	170.1

Further, if we embed these decisions in our software, then when the landscape changes, software has to be revised. Often, data has to be repartitioned when the amount or "shape" of it changes. We build increasing layers of abstraction to address this, which often has unintended negative performance consequences. Data center operators and software engineers face these complexity challenges daily.

Can we rethink the problem and build a different kind of computer that is much easier to deal with than the situation we now face with the pervasive, inexpensive, ubiquitous clusters that industry provides? In contrast to these scale-out systems, we define a new scale-up system. Scale-up has the advantage of having a much simpler programming model but at a cost of expensive, less flexible hardware; scale-out has the advantage of more flexible and cost-efficient hardware, but incurs a higher cost of software complexity and data partitioning. Fortunately, these two models need not be mutually exclusive. We can use scale-out hardware to build scale-up computers which, in turn, can run scale-out software.

Early attempts at solving the scale-up problem were not altogether successful. Two examples immediately come to mind: the Encore Ultramax[4] built for DARPA and the KSR-1.[5] Both provided an easy-to-use programming model: multiple processors sharing strongly cache-coherent memory. Many have pointed out that earlier advances in single-stream performance of microprocessors made these higher-complexity projects less desirable than the alternative of simply utilizing faster uniprocessors. But microprocessors hit a wall; single-stream performance stopped increasing, constrained by limitations on power consumption and heat dissipation. Single-stream processors evolved into multicore processors. Even that was not enough to satisfy emerging needs. Hyperthreads, which give the illusion of being processors but in fact contend for common hardware, began to emerge for greater system utilization. The levels of hierarchy grew to encompass multiple hyperthreads per core, multiple cores per processor, multiple processors per server, and multiple servers per rack. This in turn resulted in racks of multiprocessor multicore servers, rows of racks of servers, and networked datacenters each consisting of the rows of racks of servers at major cloud providers. Although some of this cannot be avoided, it is probably worth asking whether we could simplify at least some of it. Could we also reduce the number of OS images under management?

WHY REVISIT THIS TOPIC NOW?

In 1984, we submitted a research proposal to DARPA to develop a distributed approach to managing coherent shared memory using recently introduced hardware multiprocessors connected together in a bus topology.[4] The proposal was subsequently funded, and the resulting machine was demonstrated to DARPA in early 1989.

Bell writes:[6]

> *The most important part of virtual memory is locality as embodied in the concept of the working sets and hardware managed caches. The aspects of virtual memory and caches are what the all-cache architecture uses to "cache" the active portion of a program and automatically exploit temporal and spatial locality.*

He also writes:

> *Cache only, a natural extension of virtual memory and multiprocessor caching, first permits a single data item to exist in more than one location at a time. Once a memory page is brought from secondary memory to one of the nodes, hardware and software automatically move, replicate and control data flow with other nodes on an elemental basis.*

However, in 1999, Bell published a paper[7] that questioned whether a more costly and complex distributed approach to maintaining coherent shared memory would ever find widespread adoption and be competitive with simple and straightforward clusters. Hence, large programs were doomed to intensive reprogramming using MPI.

In 2012, we were discussing the topic again, and decided it was worth taking another look at the problem of single-systems image systems,[8,9] and distributed shared memory.[10] The company TidalScale was formed to build a software-defined, scale-up server composed of multiple scale-out computers. TidalScale's goals were the following:

- Use off-the-shelf, relatively inexpensive commodity servers.
- Take advantage of hardware advances that support virtualization as a first-class part of modern computing architectures.
- Build a virtual machine that would run across a tightly coupled set of networked servers. The virtual machine would be built on a set of cooperating hyperkernels, each running on

a single discrete server. We call this a *software-defined server* (SDS; Figure 1). This is the inverse of what we think of as virtual machines,[11] in which a number of virtual machines run simultaneously on a single server.

- Create a resulting architecture that could scale linearly in cost and dynamically over time.
- Run any one of a set of guest OSs from their original distributions in the virtual machine with no changes at all (that is, they are bit-for-bit compatible).
- Continue to run without modification any application that already runs on one of those OSs.
- Enable the virtual machine to optimize its own behavior through introspection and machine learning without any human intervention.
- Allow the virtual machine to inherit, as much as possible, future hardware innovations.

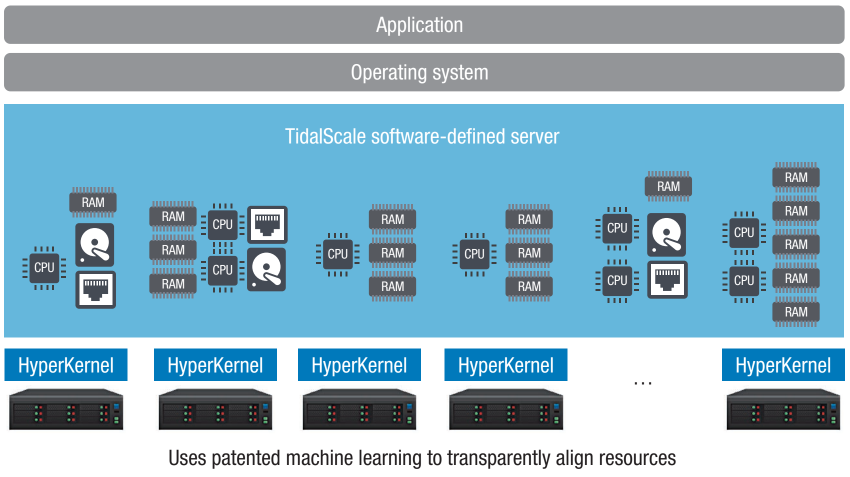
Application
Operating system
TidalScale software-defined server
RAM
CPU
HyperKernel
...
Uses patented machine learning to transparently align resources

FIGURE 1. TidalScale software-defined server.

ACHIEVING CHAMPAGNE SCALABILITY ON A BEER BUDGET

A recent paper[12] discusses an interesting financial application using historical stock information that uses approximately 6 Tbytes of an in-memory data in a table containing 6M rows and sorts the rows by one of the columns. It should have been simple, but because of memory limitations, it was not. Rather than moving rows of data around in memory, we created an array of pointers to rows and sorted the array of pointers. While this is the obvious algorithmic solution, it is problematic. One can buy machines today with large amounts of memory, but unlike clusters, the cost of the machine is not linear in the size of memory. The availability of large coherent distributed shared memory enabled the more straightforward solution. It was also affordable, because it enabled a system that can evolve over time and has a strictly linear cost profile (that is, adding a node to a 10-node system increases the cost by only 10 percent). Moreover, it increases the aggregate available memory and PCI bandwidth by the same 10 percent.

Many hardware and software innovations have emerged in the last 30 years that can be exploited, including but not limited to

- increasing memory density;
- emergence of larger caches with more levels in the cache hierarchy;
- convergence on the X86 hardware instruction set;
- emergence of high-volume, cost-optimized computer systems;
- emergence of multicore processors with increasingly higher core density;
- broad use of commodity OSs and applications that support symmetric multiprocessing;
- emergence of multiple multicore processors that can now coexist on a single motherboard;
- hardware support for and broad use of high-performance, binary-accurate virtualization software; and
- lower latency networks based on cost-effective VLSI switching capable of high bandwidth utilization.

One way to exploit these advances would be to build large, scalable coherent memories that can be easily utilized by a large number of processors. However, this is easier said than done. Modern OSs are written to use processors having a view of memory that is symmetric and strongly coherent and with uniform access latencies. While it is certainly true that OSs have interfaces that allow applications to exploit nonuniform latencies, they are difficult to use effectively. Further, it is

TABLE 2. Typical access ratios for three Intel Xeon processor cache levels.

Source	Latency (ns)
Registers	0
L1	4
L2	10
L3 near, unshared, unmodified	40
L3 near, shared, unmodified	65
L3 near, shared, modified	75
L3 far min	100
L3 far max	300

sometimes difficult to maintain the necessary changes as the data and compute landscape changes. Similarly, applications have similar views of virtual memory, in that main memory access is assumed to have the same uniform latencies.

A basic tenet of computing is that caching improves performance. Larger caches are generally better than smaller ones. More levels of cache hierarchy work. Intel Xeon processors provide three cache levels, called L1, L2, and L3. Access to primary memory without caches is not particularly fast. Typical access ratios are shown in Table 2.

Multiple levels of caches help satisfy the illusion of low latency. It works well, up to a point. Hardware-based caches have limited, fixed sizes defined by physical silicon-layout constraints and cost. In addition, coherency algorithms are defined in hardware. In contrast, software is far more flexible.

Extending the model, the TidalScale hyperkernel models all the primary memory DRAM on a motherboard as an L4 cache of the virtual machine. In a sense, we have replaced all physical memory with a distributed L4 cache. The guest OS, running in a virtual machine, does not know this, in the same way that a guest application does not know when it is accessing data out of an L3 cache instead of going directly to DRAM. Some people refer to this as an "all-cache" design. No changes to the guest OS are required. From the above ratios, we now see the "magic" as to why software caching works—several hundred instructions can be executed in the time it takes to move data from L4 to a register.

Certain applications might behave poorly given the limitations of L1, L2, and L3. Experienced users can write programs that exhibit very poor performance on today's "bare metal" servers. But this does not mean that we should get rid of caches, because in practice we know the benefits of caches over a wide range of applications and have broadly concluded that caches are helpful. Do certain applications behave better by carefully exploiting nonuniform memory access latencies? Yes.

It is beyond the scope of this article to fully describe all the ways virtualization is now supported in hardware, but it is perhaps sufficient to review the process of an application accessing memory. An application typically references memory in a virtual address space. When a virtual address is referenced by a processor running the application, the processor translates that virtual address to a physical address by consulting a page table that maps application virtual addresses to physical addresses. Page tables are maintained by an OS. We call an OS running on a virtual machine a *guest OS*, and the page tables that the guest manages become first-level page tables.

In a virtual machine, there can be multiple levels of virtual to physical address translation. Just as an OS takes an application virtual address and translates it to what it thinks is a physical address, a virtualization system takes that guest physical address and converts it to a real physical address. Today, two levels of address translation are widely supported, but the key concept is extendable in that one could easily envision stacked virtual machines using the similar algorithms.

Today, a software construct called *containers* largely obviates the need for additional levels of virtual machines. Containers let a user package programs and data, instantiate, transmit, and run them with consistent results and similar performance on other systems. In the world of big data, larger containers are preferred. Because containers run above an OS, they work without modification in an SDS.

However, there is a much less obvious advantage in having the machinery to support these virtual machines. Due to the greater flexibility of software over hardware, the machinery managing the interface between virtualized machines and physical machines is an excellent place to implement many optimizations and enhancements that cannot easily be implemented in hardware. With this machinery, we do not have to modify an OS, which might be proprietary, or in the interest of standardization, tightly controlled. Virtualization software can implement introspection, intelligence, machine learning, resource tracking, telemetry statistics, mobilization, working set tracking, and I/O virtualization. Through message passing, it can also provide an enhanced global view of the behavior of an SDS without having to introduce any shared hardware state.

We have found it desirable to introduce that concept of a virtual motherboard. At TidalScale, we provide a virtual motherboard as part of an SDS. A virtual motherboard can span many individual hardware servers (Figure 1). Unlike a physical motherboard, it can grow and shrink either explicitly on a user-driven basis, or automatically as needed.

Resources like virtual general-purpose processors, virtual memory, virtual networks, and virtual disks can migrate. Virtual interrupts can be remotely delivered. As long as the basic hardware abstractions expected by the OS are not violated, a virtual machine can look to the OS just like a physical machine. Because the virtual machine looks like hardware from the OS's point of view, compatibility tests are run as if the virtual machine were in fact a physical machine. Today, multiple OSs are supported (Centos, Red Hat, Ubuntu, and FreeBSD). Windows Server runs but is not released at this time.

There is an identical instance of the hyperkernel running on each node of the virtual machine. Due to physical hardware boundaries, a physical processor cannot directly address every guest physical address. When a guest physical address needs to be read or written, it must be translated into a physical address that the processor can access.

This translation is handled through the processor's second-level page tables. When software makes a reference to a guest physical address, if the page of memory containing that address is resident on the node that has the processor that generated that address, the address is represented in the second-level page table. Automatic address translation hardware will then translate that address to a guest physical address and then to a real physical address as it normally does by using the first- and second-level page tables, with no performance degradation. But, if the memory address is not present in the second-level page table, the hardware cannot completely translate that guest address to a real physical address, the processor generates an interrupt. The hyperkernel fields that interrupt, and analyzes the request, similar to what an OS might do when it needs to copy a page that is not memory-resident, but only resident on backing store. That analysis might result in a request for that page to be sent from a different node, or it might result in a decision to migrate that virtual processor to the node that has that page of memory. Page reads and page writes are handled differently. Readable pages can be replicated,[6] but a writable page requires additional overhead to remove that page from the L4 cache of other nodes that might have a copy (invalidation). (The actual set of steps is far more complex than what we have outlined.)

To migrate a virtual processor, the hyperkernel uses a standard mechanism to take a snapshot of the state of the processor (at this writing, approximately 6,400 bytes of data) and sends it in a message over the dedicated Ethernet to the chosen destination, where it can be restored onto another physical processor. Saving and restoring processor state is now standard for processors supporting virtualization. The program counter has not advanced, so the instruction is then restarted. Because the page and the virtual processor are now co-resident, the processor continues running. It is possible that the instruction generates additional interrupts to access different nonresident pages, but the mechanism is the same. When the virtual processor migrates, its updated location is recorded. However, for reliability, we never assume perfect location knowledge, because the processor might have subsequently remigrated.

[RESOURCES LIKE VIRTUAL GENERAL-PURPOSE PROCESSORS, VIRTUAL MEMORY, VIRTUAL NETWORKS, AND VIRTUAL DISKS CAN MIGRATE.]

WHY DOES A SOFTWARE-DEFINED, SCALE-UP COMPUTER RUNNING ON A COMPUTER CLUSTER JUST WORK?

In his seminal paper on working sets, Peter Denning[13] asserted that processors needing pages can often arrange to have those pages in memory rather than backing store. We have significantly generalized this notion of working sets to include not only memory but processors, I/O, interrupts, storage, and so on. If we could do a perfect job, we would make sure that all processors were co-located with all the memory they reference. Of course, we cannot guarantee that in general, but on a statistical basis, the system works well. It is also important that once a working set is established, it is not unnecessarily destroyed. Further, because each hyperkernel independently learns about the state of the computation as it proceeds, it continues learning about the pattern of memory accesses for virtual processors and can factor that into discriminating between migration versus page copy or page move.

The hyperkernel can also rate the goodness of the decisions it is making and provide strategic feedback to itself. For example, it is very straightforward to track guest execution time and the number of stalls. If the ratio between them is high, the computation is "good"; if not, it can use improvement. The hyperkernels, taken together as a virtual machine, begin to automatically adapt to the pattern of memory access for each virtual processor.

To minimize overhead, the hyperkernel needs to minimize

*number of stalls * the average stall time* .

Each hyperkernel instance maintains a model of its own behavior and, through introspection, can share it among its peers and adjust its behavior. These algorithms are run very frequently, and this allows for the virtual machine to quickly adapt to the observed pattern of accesses.

This is where the machine learning comes in. The hyperkernels, taken together, watch the progress of the computation, and through coordination, learn good ways of dealing with locality and nonlocality. They then remember the decisions they make and use those decisions to help make subsequent decisions.

WE DESIGNED A VERY GENERAL SYSTEM THAT WORKS OVER MANY CLASSES OF APPLICATIONS.

PERFORMANCE CONSIDERATIONS

Our goal is to achieve 100 percent binary compatibility, both in OSs and applications, and thereby substantially simplify the computer system landscape while at the same time providing good performance and high reliability. To do that, we designed a very general system that works over many classes of applications. Our goal is to do as good a job as we can on every workload running on every popular OS.

This has been largely achieved. All common applications we have tested work (MySQL, Oracle, SAP/Hana, applications in R, Python, and so on).

We also need to consider the performance of these systems. It might not be possible to broadly characterize classes of applications as good candidates for SDSs. It is important to realize that the execution profile of an individual application is dependent only on the program and its data. This access pattern is fixed for single-threaded applications. The actual performance might vary according to processor speed, cache size, levels of cache hierarchy, amount of memory, memory bandwidth, onboard communication contention, paging activity, interference from other processes or processors, and so on, but the pattern of access is generally deterministic (more so for single-threaded programs than multithreaded programs). Unfortunately, there is no expectation that programs in the same application class will share the same access patterns.

Specific access patterns are not generally factored into processor design. Designers test against existing representative workloads. There was, and still is, no guarantee that today's architecture will be appropriate for tomorrow's workloads. This has nothing at all to do with the concept of a virtual server; rather, it is fixed by the requirements of sample benchmarks. At TidalScale, we have adopted a similar approach.

To get good performance, we need to minimize L1–L4 cache misses. With the hyperkernel managing L4, if a page of memory is not on the node that is running some core that is requesting that page, we incur overhead. If it were L1–L3, hardware might trigger a cache invalidation, a TLB shootdown, and a new copy might have to be fetched from memory. The situation is the same with L4, except that the hyperkernel does not fetch it from local main memory but, rather, from remote memory. This results in a network transaction between nodes.

This is analogous to what a processor needs to do when accessing memory over Intel's QPI, SCI, or AMD's HyperTransport. L1–L3 latencies vary by, for example, memory contention, number of sockets, number of memory banks, speed of memory, and available bandwidth to memory. The same is true with the hyperkernel's distributed L4 cache. As we suggested earlier, memory access times might be surprising.

The questions about performance of this sort of software virtualization reduce to the question of how often a page access pattern causes cache "breakage." The simple answer is that cache misses occur in the case of L1–L3, when the cache is not smart enough to predict future access patterns. This is generally unpredictable. Larger cache sizes for L1–L3 reduce the probability of cache misses. The same is true for L4. But the L4 cache size of an SDS is enormous relative to the sizes of L1–L3. The L4 cache on the hyperkernel consists of all the memory on the motherboard. Therefore, the L4 cache size might be 256 Gbytes, 500 Gbytes, 1 Tbyte, or more. Also, the hyperkernel has the advantage of much more sophisticated cache-coherency and cache-management algorithms than can be implemented in silicon.

There might be concern that this sort of system would put an enormous amount of pressure on the virtual backplane and require enormous bandwidth. This is not the case. If the SDS can co-locate processors and memory, there is no overhead at all. Smarter algorithms coupled with machine learning show a measured bandwidth utilization of 5 percent for a class of simulation applications on a dedicated virtual backplane (for example, 10 Gigabits Ethernet).

APPLICATIONS: THE PROOF IS IN THE …

Earlier we raised the question about which classes of applications are particularly well suited for SDSs, and which are not. Unfortunately, this is very difficult to determine. Application classes do not display sufficient uniformity in their usage of different layers of L1–L4 cache hierarchy. If past access patterns are not a good predictor of future access patterns, the probability of cache misses can be high.

L4 cache hit statistics dominate performance. Two examples of poor L4 cache performance have been observed:

1. In a specific implementation of computational genomic sequencing, when run on a software-defined server whose nodes had 256 Gbytes of real memory, a large array (250 Gbytes) was being rehashed on every update, making prediction difficult. As a result, there was a high probability of not having that part of the array local to a node when needed. When the memory size was increased from 256 to 320 Gbytes, the problem was resolved. The initial program had been trying to conserve memory, which is not a good idea when the amount of memory is adjustable. Once realized, a new version of the program resolved the problem.
2. The second instance also has to do something that is difficult to predict: in a heavily multithreaded (multiprocessor) application, we observed a lot of contention on a central shared lock in a Python runtime library, causing the containing page to frequently migrate between nodes. However, the same contention pattern would happen among multiple cores on different processors on a single motherboard when accessing the same lock. The L1–L3 caches of those processors would have a difficult time figuring out which cache is the best one to contain the lock, and as a result, the lock would constantly migrate between caches, resulting in poor performance. When this application was modified to reduce unnecessary sharing, performance increased by an order of magnitude.

The system performs particularly well when we find a use case that hits the "memory cliff"—a program and its data that could benefit from being totally memory resident with sufficient available memory. The desired memory footprint need not be very much larger than the available memory[14] to cause thrashing, which then causes an OS to start paging. This is entirely normal and frequently happens. The application will continue to work, but it will underperform. Unfortunately, no matter how fast a backing store is, it is several orders of magnitude slower than DRAM. Even the fastest SSDs on the market are often three orders of magnitude slower than DRAM. Faster backing store latencies have never been able to compete with DRAM latencies.

Consider the following case study.[15] An application ran a MySQL job of three queries over a database with 100,000,000 rows and 100 columns, taking seven hours to complete on a server with 128 Gbytes of memory. The memory required was greater than the memory capacity of the server, resulting in frequent paging. When replaced with two 96-Gbyte nodes for a total of 192 Gbytes of physical memory running an SDS, the exact same set of queries on exactly the same data took about seven minutes to complete, resulting in a 60× performance speedup. Paging was entirely eliminated.

In another study, we worked with a large financial institution to run an analytics simulation model of customer behavior. This was not a performance experiment in the traditional sense: the customer had never been able to run a simulation this large before, so there was no before-and-after comparison. The model was run on an SDS with 3.5 Tbytes of memory, running transparently across five physical servers. Because the customer could perform the simulation in memory, it was able to run the simulation at a 1:1 granularity level and perform a three-part analysis including granularity analysis, sensitivity analysis, and model optimization. In essence, this experiment removed traditional limitations and enabled exploratory analysis without changing the models, tools, or operating environment.

COMPATIBILITY AND RELIABILITY

Compatibility is very clear. Because the hyperkernel looks like hardware to the OS, compatibility should be, and is, 100 percent.

Additionally, the guest has its own reliability features, for example, RAID.

Although the general topic might be beyond the scope of this article, the SDS provides new ways to improve reliability due to the new virtualization level mentioned earlier.

When the following common conditions are satisfied, we gain in reliability:

- Early indicators signal a possible impending hardware failure.
- Mechanisms exist that allow continued correct operation prior to an unrecoverable failure.
- There is sufficient time between early indicators and the unrecoverable failure to adjust behavior.

Considering that resources are all mobile, previously configured hot standby machines can be utilized. When failure is suspected to occur in the near future due, for example, to soft error-correcting code errors, rising server temperature, or higher than normal network anomalies, it can be dealt with by dynamically adding an additional hot standby node to the cluster, informing all nodes about the pending node failure so that they do not migrate any virtual processors to it, and having the failing node evict virtual processors at the earliest possible time, and pages of memory in active use. The mechanisms to do this all exist as a byproduct of resource mobility. For example, if a processor fails on access to a page on a remote node, we would normally migrate the processor to that node, or migrate the page to the requesting processor. If the remote node is failing, we will migrate the page to some other node that has a copy. In other words, rather than trying to come close to the reliability of a single server, our goal is to have the SDS exceed it, because we can replace nodes containing resources in real time without rebooting the guest system.

[APPLICATIONS CAN BE WRITTEN IN A MORE CONVENTIONAL AND SIMPLER WAY WHEN HOSTED BY SDSS.]

We have presented the case for an innovative, modern distributed, coherent, shared memory system hosted by virtually any cluster of computers. The advantages of the SDS architecture are clear. A single system image provides a much simpler programming model, and a much simpler datacenter server management model, especially for large containers. Scale-out systems are more complex from a software and operational support perspective, but less expensive than traditional HPC scale-up computers. Scale-up is simpler for applications and operations. The advantageous hardware cost and flexibility efficiencies of scale-out can be achieved with the inherent simplicity of scale-up using the same cost-efficient hardware.

We have shown here

- that both scale-up and scale-out systems can effectively host an SDS;

- that by introducing a layer of software above the hardware but below the OS, we can increase the set of automatic optimization possibilities and create a place to introduce machine-learning into computer systems without having to create new hardware;
- that, in the aggregate, memory and I/O bandwidth can be increased without resorting to new hardware designs; and
- that by mobilizing both virtualized processor and memory resources we minimize interconnect bandwidth requirements. Applications can be written in a more conventional and simpler way when hosted by SDSs. SDSs often reduce the need for explicit data partitioning and partitioning management. This translates directly to the ability to build large, diverse application programs reliably and rapidly. **C**

ABOUT THE AUTHORS

C. GORDON BELL is a Microsoft researcher emeritus (retired); former head of R&D at Digital Equipment Corp.; a Fellow and member of ACM, AMACAD, IEEE, NAE, and NAS; and a Computer History Museum founding trustee. He has received several honors including CMU and WPI doctorates (hon.), the IEEE von Neumann medal, and the 1991 National Medal of Technology for work on computers, lifelogging, and new ventures. Contact him via gordonbell .azurewebsites.net.

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.

ACKNOWLEDGMENTS

We thank Peter Christy, Charles Levine, and the referees for their comments on earlier versions of this article.

REFERENCES

1. C. Levine, personal communication.
2. J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Proc. 6th Symp. Operating Systems Design & Implementation* (OSDI 04), 2004, p. 10.
3. D. Cutting, “ Welcome to Apache Hadoop!,” Apache Hadoop, 18 Nov. 2017; adoop.apache.org.
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. *KSR-1 Technical Summary*, Kendall Square Research, 1992.
6. G. Bell, “Scalable Parallel Computers: Alternatives, Issues, and Challenges,” *Int’l J. Parallel Programming*, vol. 22, no. 1, 1994, pp. 3–46.
7. G. Bell and C. Van Ingen, “DSM Perspective: Another Point of View,” *Proceedings of the IEEE*, vol. 87, no 3., 1999, pp. 412–417.
8. R. White, “The Single System Image Feature Delivers Greater Flexibility and Resilience,” *IBM Systems Mag.*, May 2013; www.ibmsystemsmag .com/mainframe/administrator /Virtualization/ssi_feature_zvm.
9. R. Buyya, “Architecture Alternatives for Scalable Single System Image Clusters,” *Proc. Conf. High Performance Computing on Hewlett-Packard Systems* (HiPer 99), 1999; www .buyya.com/papers/ssiArch.html.
10. J.K. Ousterhout et al, “The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM,” *SigOPS, Operating System Rev.*, vol. 43, no. 4, 2009, pp. 92–105.
11. M. Rosenblum, “The Reincarnation of Virtual Machines” *ACMQueue*, vol. 2, no. 5, 2004, pp. 34–40.
12. I. Nassi, “Scaling the Computer to the Problem: Application Programming with Unlimited Memory,” *Computer*, vol. 50, no. 8, 2017, pp 46–53.13.
13. P.J. Denning, “The Working Set Model for Program Behavior,” *Comm. ACM*, vol. 11, no. 5, 1968, pp. 323–333.
14. A. Aho. P.J. Denning, and G. Ullman, “Principles of Optimal Page Replacement,” *J. ACM*, vol. 18, no. 1, 1971, pp. 80–93.
15. I. Nassi, “Advances in Virtualization in Support of In-Memory Big Data Applications,” *Proc. Int’l Workshop High Performance Transaction Systems* (HPTS 15), 2015; www.hpts.ws /papers/2015/tidalscale.pdf.