A Parallel Page Cache: IOPS and Caching for Multicore Systems

Da Zheng, Randal Burns
Department of Computer Science
Johns Hopkins University

Alexander S. Szalay
Department of Physics and Astronomy
Johns Hopkins University

Abstract

We present a set-associative page cache for scalable parallelism of IOPS in multicore systems. The design eliminates lock contention and hardware cache misses by partitioning the global cache into many independent page sets, each requiring a small amount of metadata that fits in few processor cache lines. We extend this design with message passing among processors in a non-uniform memory architecture (NUMA). We evaluate the set-associative cache on 12-core processors and a 48-core NUMA to show that it realizes the scalable IOPS of direct I/O (no caching) and matches the cache hits rates of Linux's page cache. Set-associative caching maintains IOPS at scale in contrast to Linux for which IOPS crash beyond eight parallel threads.

1 Introduction

Recent hardware advances have produced multicore systems that exceed one million IOPS with an extreme amount of parallelism in both the threads initiating I/Os and the devices servicing requests. Such systems transform cloud data services, specifically, key/value stores and NoSQL databases that require a large number of index lookups that fetch small amounts of data. Random I/O becomes the critical performance factor [17]. Whereas index lookups have locality—most NoSQL systems support range queries [5, 16]—the dominant workload consists of scaling concurrent key lookups to an arbitrarily large number of users. Amazon CTO Voegels promotes SSD integration as the differentiating factor for the DynamoDB NoSQL platform [20].

However, operating system page caches bottleneck IOPS well below 1M and performance degrades with multicore parallelism. This reflects that page caches were designed for magnetic disk; an array of 20 disks provides only 4000 IOPS. Performance problems stem from CPU cache misses to shared data structures and lock contention that arises when implementing the global notion of recency common to modern page caching algorithms [1, 10, 11, 15]. One possible solution uses direct I/O to eliminate cache management overhead. Doing so provides scalable parallel throughput, but gives up on the benefits of caching in their entirety.

	random IOPS	latency	granularity
ioDrive Octal [7]	1,300,000	45μs	512B
OCZ Vertex 4 [18]	120,000	20μs	512B
DDR3-1333	7,300,000	15ns	128B

Table 1: The performance of SSDs and SDRAM. IOPS are measured with 512-byte random accesses.

We believe that memory caching can benefit a system with very high IOPS. Even though high-end NAND flash memory delivers millions of IOPS, it is slower than DRAM in both throughput and latency (Table 1). SSDs also require larger accesses than DRAM. The smallest read on SSDs is 512 bytes to 4096 bytes and it is the CPU cache line size on DRAM, usually 128 bytes. The large performance gap between SSDs and DRAM mandates memory caching.

We redesign the page cache for parallel access, eliminating all list and tree structures and declustering locks to reduce contention. The goal of this redesign is to provide performance equivalent to direct I/O for random read workloads and to preserve the performance benefits of caching when workloads exhibit page reuse.

We implement a set-associative cache based on hashing page addresses to small page sets that are managed independently. Threads on different cores accessing different sets have no lock contention or memory interference. Set associative caches have long been used in processors as a compromise between the amount of hardware needed to implement a fully associative cache and the conflicts that arise from direct mapped caches [9]. Using them in a page cache creates a different tradeoff, we improve parallel performance at the cost of a small amount of cache hit rate: associative caches approximate global recency, but the approximation is imperfect [19].

We extend our design to non-uniform memory architectures (NUMA) by partitioning the cache by processor and using a message passing protocol between partitions. Treating processors as the nodes of a distributed system [3] avoids the contention and delays associated with remote memory accesses.

On random I/O benchmarks, our set-associative page cache realizes 95% of the IOPS of direct I/O, saturates memory bandwidth, and scales up to the 48 cores on our test machine. In contrast, buffered I/O in the Linux page cache shows a IOPS collapse beyond 6 cores and never achieves more than half the performance of direct I/O.

Our work is preliminary in that we do not (yet) have an in-kernel implementation of the set-associative cache.

2 Related Work

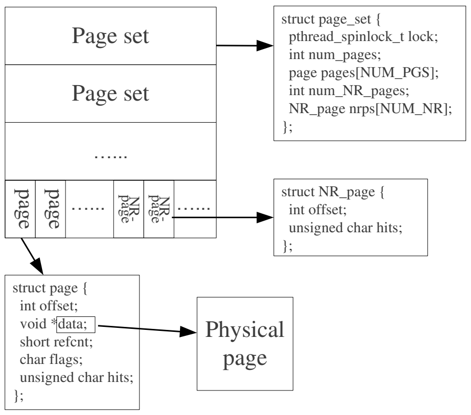
The scalability of the Linux kernel to multicore has garnered a lot of academic and open-source interest recently. Gough et al. [8] showed that respecting memory and interrupt locality improves many basic kernel operations. Many of these ideas were adopted upstream, including the scalable direct I/O to which we compare. A team at MIT [4], conducted a comprehensive survey of Linux kernel scalability issues for multicore, concluding that traditional OS designs have the potential for good parallel performance. Related to our work, but more specific, they found a locking bottleneck in the use of `lseek()` by PostGres.

Most closely related to our work is Nb-GCLOCK [13], which uses a lock-free hash table as an index and manages pages in a global page buffer in a lock-free fashion. Nb-GCLOCK removes locking overheads, but preserves a global cache that will incur processor cache faults during parallel accesses. Our design may benefit from a lock-free implementation within page sets and we will explore this design in future work.

Read-copy-update (RCU) [14] optimizations allow parallel reads with low overhead to data in caches. RCU has been adopted in the Linux page cache and dcache. RCU improves the number of page reads from cache contents. In contrast, we focus on the scalability of updating contents of the cache. The RCU work removed a bottleneck and revealed scalability problems with cache IOPS.

Our work was inspired by theoretical results that showed that caches with restricted associativity can approximate LRU [19] and set-associativity has long been used in processor caches [9].

Multicore processors partition L2 and L3 caches, allocating cache space to thread groups so the threads do not interfere. Lin et al. [12] summarize many different techniques and instrument performance benefits. This work is superficially similar: it prevents interference among parallel tasks. However, partitioning memory for applications is very different than our goal of a global shared cache accessible to all cores.



```
struct page_set {
  pthread_spinlock_t lock;
  int num_pages;
  page pages[NUM_PGS];
  int num_NR_pages;
  NR_page nrps[NUM_NR];
};
```

```
struct NR_page {
  int offset;
  unsigned char hits;
};
```

```
struct page {
  int offset;
  void *data;
  short refcnt;
  char flags;
  unsigned char hits;
};
```

Figure 1: The organization of the set-associative cache

3 A Set-Associative Page Cache

The emergence of SSDs has introduced a new performance bottleneck into page caching: managing the high churn or page turnover associated with the large number of IOPS supported by these devices. Previous efforts to parallelize the Linux page cache focused on parallel read throughput from pages already in the cache. For example, read-copy-update (RCU) [14] provides low-overhead wait free reads from multiple threads. This supports high-throughput to in-memory pages, but does not help address high page turnover.

Cache management overheads associated with adding and evicting pages in the cache limit the number of IOPS that Linux can perform. The problem lies not just in lock contention, but delays from the L1-L3 cache misses to locking data structures. This limits scalability for as few as six concurrent threads.

We redesign the page cache to eliminate lock and memory contention among parallel threads by using set-associativity. The page cache consists of many small sets of pages (Figure 1); empirically, we found that eight pages per set performs best. A hash function maps each logical page to a set in which it can occupy any physical page frame.

We manage each set of pages independently using a single lock and no lists. For each page set, we retain a small amount of metadata to describe the page locations. We also keep one byte of frequency information per page. If a set is not full, a new page is added to the first unoccupied position. Otherwise the least frequently used clean page is evicted. When a frequency counter overflows, we halve all counters in the set using a bit-shift. We capture some recency information by keeping frequency counts for the most recently seen non-resident (NR) pages. Thus, pages that have been evicted can be reinstated with higher priority.

Small page sets keeps metadata (counters and occupancy) in one or few cache lines and minimizes CPU cache misses. The total metadata for eight pages is 208 bytes without non-resident pages enabled. This and the metadata for 37 non-resident pages fits within 4 128-byte cache lines.

Currently, the treatment of page writes is unmodified from Linux. Dirty pages are scheduled for background writing. Eliminating lock contention on the write path is a separate optimization challenge. When no clean pages are available in a set, synchronous writes are needed to free physical pages. This problem exists already in Linux when the system hits dirty page limits [2].

We extend the set associative design to non-uniform memory (NUMA-SA) with a message passing protocol. Message passing follows recent operating system redesigns that treats multiple processors (or cores) as a distributed system [3]. We split that global cache into partitions, assigning one partition to each processor die with 6 cores. We use a hash function to distribute requests to dies and use message passing to access page sets managed on remote partitions. Message passing is implemented with a memory copy to a message queue. Message queues are the only data structures shared among processors. Our design uses a worker thread on each core to serve remote requests. We bundle multiple requests into a message to amortize overheads. Each request contains the pointer to the memory to which requested data should be written. When requested data is ready, it will be written to the destination memory directly and a reply is sent back to notify the initiator thread. We also bundle multiple replies in the same fashion as requests.

The programming interface of the NUMA-SA design is similar to AIO in Linux (Figure 2). It accepts an array of requests. When the maximal number of requests are sent, the application has to wait for replies. Once replies arrive, the callback function specified by the user application will be called. This interface allows us to potentially reduce the number of system calls as user applications can send hundreds of requests in one call. They are encouraged to do so, because more requests can be bundled in a message.

```
typedef ssize_t (*reply_callback_t)(io_reply *);
struct io_request
{
    char *buf;
    off_t offset;
    ssize_t size: 32;
    int access_method: 1;
    int thread_id: 31;
};

struct io_reply
{
    char *buf;
    off_t offset;
    ssize_t size: 32;
    int success: 1;
    int status: 16;
    int access_method: 1;
 };

ssize_t access(io_request *requests, int num);
void wait_replies(int max_replies, reply_callback_t↩
      func);
```

Figure 2: Programming interface to NUMA-SA.

4 Evaluation

We evaluate the effectiveness of set associative caching in several dimensions: matching the IOPS of direct I/O (no caching); approximating the cache hit rate of the ClockPRO approximation [10] implemented in the Linux page cache; and, overall performance on a trace derived from the YCSB [6] cloud benchmark. IOPS always refers to IO accesses to storage devices, not to the number of requests to the page cache.

The preliminary nature of this work constrains our experimental methods. We perform IOPS experiments using a RAMDISK so that memory throughput and page cache overheads limit performance. Experiments on a system using SSDs would be preferable. Also, we have only implemented set-associative caching in user space. Thus, set associative caching does not incur system call overheads, such as context switching and parameter checks, for cache hits as does the Linux page cache. Again, this is imperfect, but system call overhead never exceed 10% in our experiments.

We conduct all experiments on a non-uniform memory architecture (NUMA) machine with 4 AMD Opteron 6172 CPUs and 512GB memory. Each AMD Opteron 6172 has 2 dies and each die integrates 6 cores. We choose Linux kernel v3.2.0 because this version makes direct I/O reads scalable. We also use the XFS file system for its superior parallel performance to ext3 and ext4. Direct I/O writes on XFS or any other file systems do not scale with multicore. Thus, our experiments study read performance. We also disable the kernel filesystem notification because it needs to access the directory-entry cache (dcache) and grab spin locks on dentries, which slows down read operations by more than 20% in 8 threads. We read data from a 40GB file on a RAMDISK. When accessing a page, we read the first 128 bytes. This loads the page into cache, but copies a small amount of data into user space to minimize overhead.

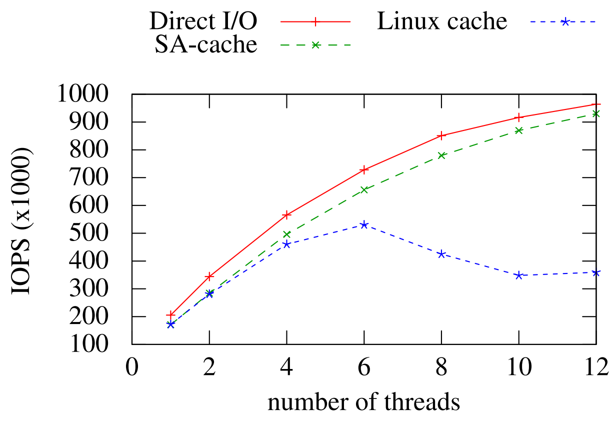


Figure 3: Random reads in a single processor.

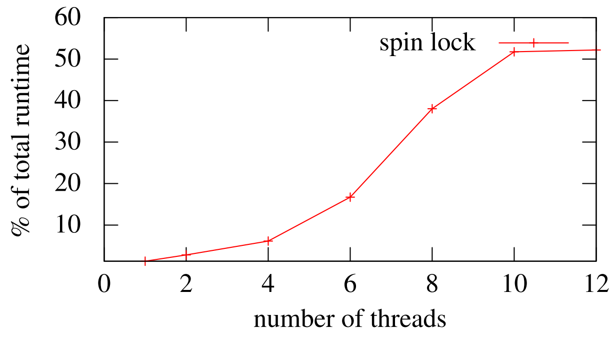


Figure 4: Percentage of runtime spent waiting on the spin lock in the Linux page cache.

4.1 Random I/O

We conduct a random I/O microbenchmark in order to evaluate the overhead of cache management. We read the entire RAMDISK (40 GB) uniformly at random and measure the read throughput. Random I/O means that there are essentially no cache hits and throughput is governed by IOPS alone.

The set-associative cache (SA-cache) and direct I/O provide good scalability. Figure 3 shows the results for all cores in a single AMD 6217 processor with two dies and twelve cores. Limiting the experiment to a single processor avoids second-order affects from non-uniform memory accesses. At 6 cores, the worst case, the set-associative cache management incurs 10% overhead and it realizes only 4% overhead at 12 cores.

In contrast, buffered reads in the Linux page cache display a throughput collapse beyond 6 threads. When using the whole processor, they realize less that 40% of the throughput from direct I/O. Lock contention accounts for much of the loss. Figure 4 show that 50% of the runtime is spent waiting on the spin lock that protects the cache search tree.

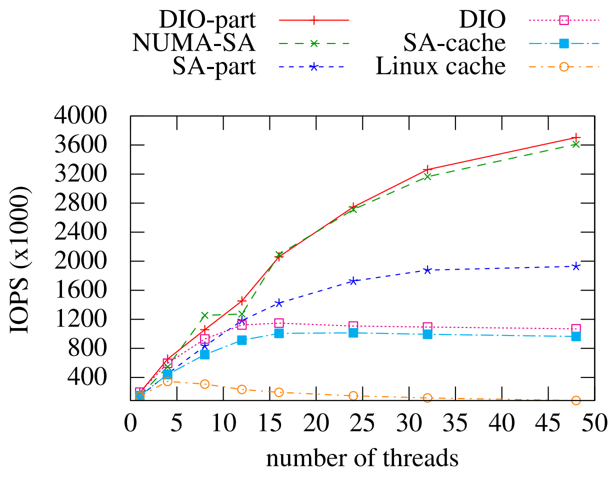


Figure 5: Random reads on a 48-core NUMA.

Our next experiment extends this microbenchmark to reading 40GB of data uniformly at random across all 48 cores in order to explore page caching in NUMA (Figure 5). NUMA experiments shows further limitations of the Linux page cache. IOPS starts to collapse at just 4 threads for a peak performance of 346K IOPS. When using the whole machine, it drops to only 82K IOPS. Both direct I/O and the set-associative cache (SA-cache) maintain about 1M IOPS up to 48 cores. NUMA introduces a slight decrease in IOPS with scale because the RAMDISK is allocated in the memory of a single CPU. As the number of threads increases, there are remote memory references.

To reduce the effect of remote memory references, we partition the RAMDISK evenly across all 4 CPUs into 8 partitions, one for each die. It emulates the parallel I/O channels from each processor to storage. In our target hardware, a processor accesses a device directly, and no inter-processor communication is needed. Direct I/O on the partition RAMDISK (DIO-part) relieves the memory bottleneck and more than triples the aggregate IOPS at 48 cores to 3703K IOPS. The set-associative cache on the partitioned RAMDISK (SA-part) does not scale well beyond a single processor. Remote memory references limit the throughput to 1929K IOPS. However, our NUMA-SA design, which partitions the cache and uses message passing among partitions, improves locality and tracks the the scalable performance of direct IO to all 48 cores for 3610K IOPS.

4.2 Cache Effectiveness

The next experiments examine how well a set-associative cache preserves the hit rate of global caching. We compare set-associative cache against the ClockPro approximation used in the Linux kernel using workload from

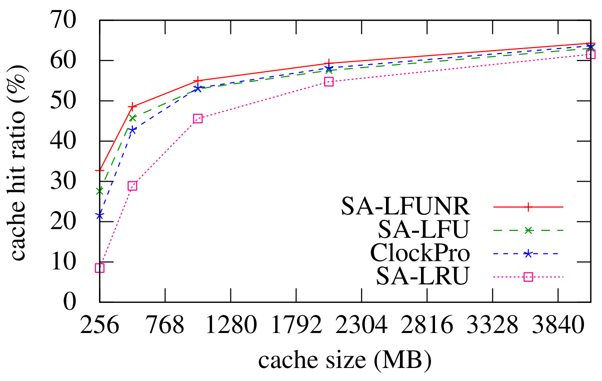


Figure 6: Cache hit rate as a function of cache size.

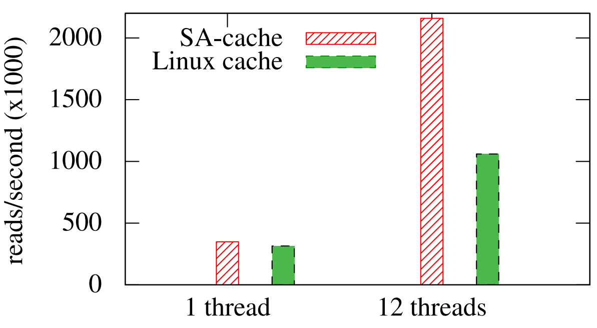


Figure 7: Aggregate performance for a MemcacheDB-derived YCSB workload.

the Yahoo! Cloud Serving Benchmark (YCSB) [6]. We extract a workload by inserting 30 millions items into MemcacheDB and performing 30 million lookups according to YCSB's user-profile cache workload, a read-only Zipfian generator. The workload has 39,188,480 reads from 5,748,822 pages. We reverse-engineered the Linux kernel and re-implemented its ClockPro approximation (without non-resident pages) in user space so that we could control exactly the cache space available to the algorithm. The Linux kernel implementation dynamically allocates cache and can only be controlled at a coarse granularity.

Figure 6 shows that the set-associative cache managing each set with LFU (SA-LFU) realizes a cache hit rate similar to ClockPro. We find that managing page sets with frequency outperforms recency (SA-LRU), because the page sets are small and susceptible to being flushed. Adding non-resident pages (SA-NRLFU) captures more history information and provides the best performance. Outperforming ClockPro was quite surprising and, likely, reflects that YCSB does not create workload shifts so that frequency information never gets stale. Specific eviction policies require more detailed study.

YCSB and the key/value workloads that it generates have less locality than traditional file systems. All techniques converge on about a 60% hit rate. Lower cache hit rates underscore the importance of IOPS in cloud workloads.

4.3 Cloud Benchmarks

We execute the same MemcacheDB workload on our user-space set associative cache and the Linux page cache to characterize overall performance (figure 6). Unlike previous experiments, this measures the application throughput, rather than the IOPS into the cache. We restrict cache sizes to 2GB; we can only constrain Linux approximately by occupying memory with other data, but the hit rate matched the previous experiment. Consistent with Figure 6, the Linux page cache and the set-associative cache get about a 60% cache hit rate.

For a single thread, set-associative only performs a little bit better. This mostly reflects the simple design of the set-associative cache, with small data structures that incur few cache misses. Some of the benefit comes from our user space implementation, which avoids the kernel overhead on cache hits.

At 12 threads, the set associative cache more than doubles the throughput of the Linux page cache, because it has very low lock contention and fewer processor cache misses.

5 Discussion and Future Work

Hardware developments have driven parallelism that mandates operating system modifications to achieve scalable performance [3, 4]. The problem is particularly acute in the page cache: in addition to parallel I/O from multiple cores to the cache, SSDs have increased the number of IOPS from the cache to storage by orders of magnitude. We address scalability for multicore with a set-associative cache that minimizes lock contention during parallel access and uses compact metadata with no lists or trees to reduce memory overhead. We extend this design to NUMA by partitioning the cache by processor and using message passing to avoid remote memory access. The outcome is a system that tracks the scalable performance of direct I/O (no caching) for up to 48 cores, while preserving the hit rates of the Linux page cache.

Several aspects of this work require refinement. A kernel implementation (in progress) will allow for a direct comparison of the set associative cache with the Linux page cache. Also, an evaluation on a scalable I/O subsystem using an array of SSDs would better demonstrate the importance of page cache IOPS.

Many challenges persist in system design. The current implementation has static limits on both the number of page sets and the number of nodes in the message passing protocol. Memory management and fault tolerance require dynamic sizing of both, perhaps with extendible or linear hashing or peer-to-peer data structures. Similarly, dynamic load balancing among processors will be necessary to deal with heterogeneous processor load and memory availability. Finally, scalable writes loom as a different, but equally important, performance concern.

Acknowledgements

We would like to thank Alainna White of the Institute for Data-Intensive Engineering and Science for her heroic support of our experimental systems. We would also like to thank Michael Bender and Garth Gibson who provided feedback on early versions of this work and our shepherd Xiaodong Zhang.

This material is based upon work supported by the National Science Foundation under grants AST-0939767, OCI-1040114, and CCF-0937810.

References

[1] G. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *Conference on File and Storage Technologies*, 2004.

[2] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. AWOL: An adaptive write optimization layer. In *Conference on File and Storage Technologies*, 2008.

[3] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Symposium on Operating Systems Principles*, 2009.

[4] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Conference on Operating systems design and implementation*, 2010.

[5] Cassandra. `cassandra.apache.org`, Accessed 3/6/2012.

[6] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Symposium on Cloud computing*, 2010.

[7] Fusion-IO ioDrive Octal. `http://www.fusionio.com/platforms/iodrive-octal/`, Accessed 4/25/2012.

[8] C. Gough, S. Siddha, and K. Chen. Kernel scalability expanding the horizon beyond fine grain locks. In *Linux Symposium*, 2007.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. Morgan Kaufmann, 2007.

[10] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conference*, 2005.

[11] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS*, 2002.

[12] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High-Performance Computer Architecture*, 2008.

[13] S. Uemura H. Yamana M. Yui, J. Miyazaki. Nb-GCLOCK: A non-blocking buffer management based on the generalized clock. In *International Conference on Data Engineering*, 2010.

[14] P. McKenney, D. Sarma, A. Arcangeli, A. Kleen, and O. Krieger. Read-copy update. In *Linux Symposium*, 2002.

[15] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Conference on File and Storage Technologies*, 2003.

[16] MongoDB. `www.mongodb.org`, Accessed 3/6/2012.

[17] K. Muthukkaruppan. Storage infrastructure behind facebook messages. In *High-Performance Transaction Systems*, 2011.

[18] OCZ VERTEX 4. `http://www.ocztechnology.com/ocz-vertex-4-sata-iii-2-5-ssd.html`, Accessed 4/25/2012.

[19] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6), 2002.

[20] W. Voegels. Amazon DynamoDB—a fast and scalable NoSQL database service designed for Internet-scale applications. `http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html`, January 18, 2012.