RC25239 (W1111-175) November 21, 2011
Computer Science

IBM Research Report

Evaluating the Performance of Two Disk-Offload Architectures for Application-Caching Middleware

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA

Tom Gissel, Ben Parees
N111/B503
4205 S. Miami Blvd.
Durham, NC 27703-9141 USA

Evaluating the Performance of Two Disk-Offload Architectures for Application-Caching Middleware

Avraham Leff, James T. Rayfield
IBM T.J. Watson Research Center
POB 704
Yorktown Heights, NY, USA
{avraham, jtray}@us.ibm.com

Tom Gissel, Ben Parees
N111/B503
4205 S Miami Blvd
Durham NC 27703-9141
{gissel, bparees}@us.ibm.com

Abstract

Main-memory caching is a well-known technique for improving application performance. However, the caching requirements of certain workloads exceed even modern main-memory capacity. Disk-offload can be an effective technique for such caching scenarios. In this paper, we present two disk-offload caching architectures, evaluate their performance, and draw conclusions as to their relative effectiveness.

1. Introduction

Caching data from slower to faster media is a well-known technique for improving performance. In *application-caching*, the caching is handled outside of the standard database middleware path. The primary reason for this is peformance: by giving up some of the functionality and guarantees (ACID properties [3]) provided by traditional database software, applications may be able to achieve very significant speed improvements [12].

For example, application data often does not need to be durable, either because it is derived from durable sources (e.g., a customer's total balance across all their accounts), or because the consequences of data loss are not that severe (e.g., loss of web session data requiring the customer to log-in again). Other performance benefits may result from using a non-relational data model, if appropriate for the application, and from giving up the ability to execute complex dynamic queries (e.g., using SQL) against the data.

Finally, many non-traditional caching systems give up strict consistency (atomicity, consistency, and isolation) in order to achieve greater performance. For example, data in a main-memory cache is not usually kept completely consistent with the data in the back-end database. When the application needs to read data, it typically checks first to see if the data has recently been cached. If it has, the cache data is used by the application even though it may be (slightly) stale. On write, the cache and database are updated at approximately the same time, but since two-phase commit is usually not used, the cache and database may temporarily return different values for the data.

Note that although the caching is handled outside the normal database-access path, it still may be managed by middleware (e.g., a servlet engine communicating with a shared cache), or directly by the application code.

A class of middleware has been developed to support application-caching, including Memcached [7] and WebSphere eXtreme Scale [14].

1.1. Motivation

Although it seems clear that a RAM-based cache of almost any size may be physically constructed, configurations using only RAM may not be the most efficient. For example, a mix of RAM and solid-state disk (SSD) may provide better price/performance, smaller floor-space, and/or lower power requirements. Similarly, virtual memory was invented because a mix of RAM and disk generally provides a more cost-efffective solution for storing process data than an all-RAM configuration. A cache that uses a mix of RAM and disk is even more reasonable when the data includes important but non-critical temporary data (e.g., web session data). In such cases, overall application performance is improved by using a cache that supports more data that fit in RAM, even if access times are worse than RAM-only performance.

When main-memory is too small to accommodate the data-sets of an enterprise's applications, application-caching is faced with the following challenge. In traditional database-caching, the database will transparently offload data from main-memory to disk, making room for incoming data using management policies such as LRU. With application-caching, although the applications have populated the main-memory cache "by hand", it is unreasonable to place the offload responsibility on the application. Application-caching middleware must therefore be augmented with disk-offload capability. The increased availability of flash memory, especially SSD, makes a disk-offload approach attractive because it reduces the performance gap between the main-memory and disk layers of the system.

At first glance, it is tempting to apply well-known and mature database techniques to augment application-caching with a disk-offload capability. A closer look shows that database techniques are tightly coupled to providing ACID properties to the client. Naive coupling of application-caching middleware to a DBMS unacceptably degrades the performance that is the main reason for the use of application-caching. When providing disk-offload, application-caching middleware should therefore exploit the fact that its cached data have more relaxed requirements regarding transactions and durability. That is, disk-offload is used to increase the size of the data-sets available to such applications, rather than providing such applications with a "database". Note that commercial products exist are similar to the application-caching disk-offload work that we discuss in this paper. These include Oracle Coherence [2], and Schooner NoSQL / Memcached Appliance [10]. However, these products are designed to provide durability and some transactional capabilities. In our paper we propose alternative disk-offload architectures for application-caching middleware (Section 2) and evaluate their performance (Section 3). We present some conclusions about disk-offload architecture design in Section 4.

2. Disk Offload

Disk-offload systems can be characterized in terms of several important architecture decisions:

- **Fixed versus Variable Sized Pool Elements**: Whether to internally manage main-memory as a pool of fixed-sized or variable-sized elements. (Traditionally, fixed-sized elements have been termed *buffers* or *frames*. Variable-sized elements are called "slabs" [13, 1].) If, for example, buffers are sized at 4K each, then a 13K data element will be partitioned into four buffers. In contrast, if the system manages a pool of variable-sized elements, the 13K data element will be managed as a single unit. The variable-sized elements may range in complexity from a set of byte arrays or a set of objects. Cache replacement algorithms such as LRU or Clock-Pro [5] are examples of pool-management issues whose implementation varies depending on whether the system uses fixed-sized or variable-sized pool elements.

- **Fixed versus Variable Disk Addressing**: Whether data are stored on disk in fixed-sized pages or in variable-sized chunks.

- **Aggressive versus Lazy Disk Mapping**: Whether or not to defer the work needed to map main-memory data to disk until absolutely necessary (i.e., when the data must be offloaded to disk).

In this paper, we consider these issues by contrasting two disk-offload architectures: a *synchronous* architecture *versus* a *spill-over* architecture. These terms relate to how and when the system does the mapping from main-memory to disk. In the *synchronous* design, even if movement between main-memory and disk does not maintain ACID properties, main-memory and disk versions of the data are kept as synchronous as possible. This design resembles a traditional database design albeit with relaxed consistency requirements that are more suited to the application-caching usage scenarios (Section 1). This design does agressive disk mapping because the data are viewed as ultimately residing on disk even if it currently resides in main-memory. Since this design is more disk-centric than *spill-over*, we included the use of fixed-sized pool elements to manage main-memory and the use of fixed disk addressing in this architecture.

In the *spill-over* design, data are primarily viewed as residing in main-memory; data are spilled-over to disk only as main-memory is filled with other data. This design decouples the main-memory and disk versions of data to a greater extent than the *synchronous* design; this is possible only because application-caching doesn't provide traditional durability guarantees. Since this design is less disk-centric than *synchronous*, we included the use of variable-sized pool elements to manage main-memory and the use of variable disk addressing in this architecture.

2.1. The *synchronous* Architecture

Figure 1 shows the *synchronous* architecture studied in this paper. It is based on an Aries-like design [8],

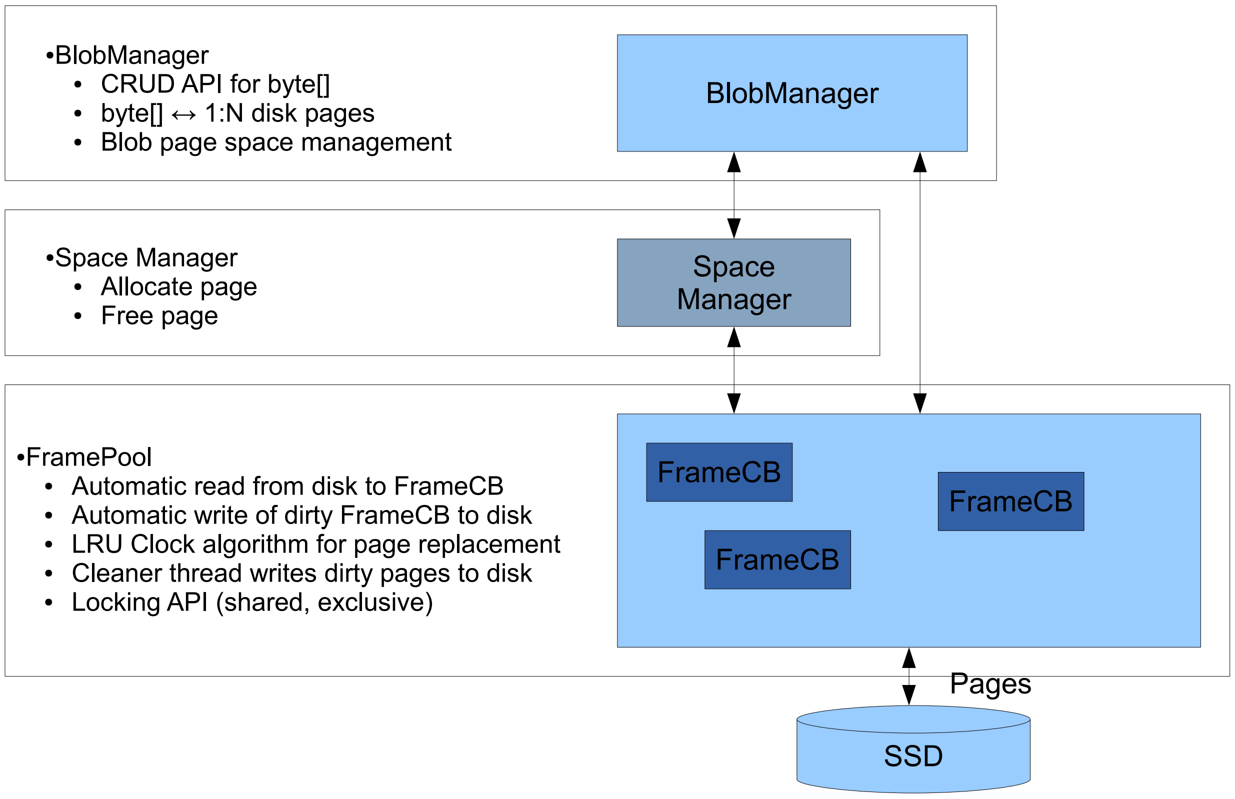
•BlobManager
• CRUD API for byte[]
• byte[] ↔ 1:N disk pages
• Blob page space management
BlobManager
•Space Manager
• Allocate page
• Free page
Space Manager
•FramePool
• Automatic read from disk to FrameCB
• Automatic write of dirty FrameCB to disk
• LRU Clock algorithm for page replacement
• Cleaner thread writes dirty pages to disk
• Locking API (shared, exclusive)
FrameCB
FrameCB
FrameCB
Pages
SSD

Figure 1. The *synchronous* Architecture

except that our *synchronous* design is simpler because (as discussed in Section 1) it does not require logging, transactions, or durability.

The architecture divides the SSD into fixed-sized pages, of which a certain percentage may be cached in main-memory. This is done by configuring the FRAME-POOL layer (termed a buffer-pool in [3] (Chapter 13.4)) to hold a specified number of *frame control-blocks* in main-memory. A frame control-block instance represent a single (fixed-size) disk page. When clients need to read data from a specific disk page, the FRAME-POOL reads that data into a frame control-block, using a LRU Clock algorithm to page another frame control-block to disk if no room remains in main-memory. Conversely, when a disk client has modified the data, the FRAME-POOL is responsible for writing the changed data to disk. A cleaner daemon is used to periodically write such dirty frame control-blocks to disk. (It is this behavior that makes the architecture "synchronous".). The SPACE-MANAGER interacts with the FRAME-POOL layer to allocate pages on disk so that new data can be stored, and to free previously allocated disk pages when data are deleted.

Application-caching clients interact with the disk-offload component *via* the BLOB-MANAGER API. This presents a C/R/U/D interface for byte arrays ("blobs"). A BLOB is stored on or more pages (if the BLOB size exceeds the system's page size); one or more BLOBS are stored on a single page (if the BLOB sizes are less than a single page).

From the perspective of achieving good performance, the most important issues with implementing the *synchronous* architecture were:

1. Determining the optimal page-size. In this architecture, all pages have a fixed-size, yet client work-loads may have BLOB sizes that differ from the system's page size. We found that the optimal page size is one that matches the average BLOB size; given that the latter may not be known in advance, this implies a weakness in this architecture.

2. Efficiently allocating space for new or updated data. In this architecture, the size of the largest free space in each page is stored in a BPLPage (Blob Page List Page). The algorithm used was a linear search through BPLPage entries until a large enough space was found. This necessitated

a potentially lengthy search, especially in the case where the BPLPages were mostly full. It would be difficult to come up with a more efficient algorithm than linear search, because the requests for storage are all different sizes, and the space remaining on each (fixed-size) page is random.

2.2. The *spill-over* Architecture

Figure 2 shows the *spill-over* architecture studied in this paper. This architecture was designed specifically to perform well in situations where the application's data-set "mostly" fits in main-memory, but "some" disk-offload is required hold the entire data-set. From that perspective, it is natural to do lazy disk mapping because there is a relatively low probability of actually paging data to disk.

Application-caching clients interact with the disk-offload component *via* the PLACEMENT-MANAGER and STORAGE-ENTRY APIs. All cache items – even if their values are stored on disk – have their keys and other meta-data stored in main-memory. When the PLACEMENT-MANAGER inserts an object into the cache, it returns a STORAGE-ENTRY to the client. This is a proxy to either a cache entry (while the object is resident in main-memory) or to a Slab instance (while the object is resident on disk). In contrast to the *synchronous* architecture, the PLACEMENT-MANAGER will not write a STORAGE-ENTRY's data to disk until the assigned main-memory has been exhausted. The *spill-over* architecture is "lazy" because only at that point are STORAGE-ENTRY instances spill-over to disk.

3. Performance Evaluation

In this section we compare the effectiveness of the two disk-offload architectures. We implemented the *synchronous* and *spill-over* architectures in Java rather than C to ease interoperability of the disk-offload component with an ORB-based server environment.

3.1. Benchmark Methodology

Our benchmark methodology is to measure the throughput (operations per second) that results when the CPU of the server running the caching service is approximately 100% utilized. This is done by injecting an application-caching style work-load from multiple clients. The work-loads used were not network or disk constrained.

In order to control for (1) the effect of using Java as the implementation language, and (2) the impact of non-cache components in application-caching middleware, we used *jmemcached* [6] (v0.9.1) to provide a base-line of main-memory performance. *jmemcached* "is a Java implementation of the daemon (server) side of the memcached protocol". We compare the performance of the vanilla *jmemcached* with versions that differs only in that they:

- replace the use of the *CacheImpl* class with a subclass that delegates the C/U/D operations to the disk-offload component.

- replace the use of the *LocalCacheElement* class whose elements are stored in the *ConcurrentLinkedHashMap* with a subclass that delegates the *getData()* implementation to the disk-offload component.

All other portions of the middleware (e.g., the complete Memcached protocol support and the use of JBoss Netty [9] for non-blocking, scalable network I/O) use the same code as the vanilla *jmemcached*.

Performance scenarios are characterized by the:

- Average size of the data being cached; in our tests this varies from 1 KiB to 16 KiB. Our experiments showed that (for our system under test (Section 3.2) data-sizes larger than 16 KiB caused the network to be saturated. The size of the Data values are generated using an exponential distribution with a parameter equal to the specified average data size.

- A fixed amount of main-memory allocated to the caching component; we set this to 500 MiB in our experiments.

- A PWrite value that specifies the ratio of "write" to "read" operations; in our experiments, we use the values of 0.2 (i.e., 80% of the operations on the cached data were reads rather than writes).

- The size of the data-set accessed by the benchmark during a given experiment. A uniform probability distribution is used to select which datum is accessed during a given operation.

  We specify the data-set size in terms of a weighting factor of the amount of main-memory allocated to the caching component. Since 500MiB are allocated to main-memory, a factor of 0.8 means that all of the accessed data can fit into main-memory so that no data needs to be offloaded to disk. When the factor is set to 12.0, the data-set occupies 6GiB, and more than 90% of the data must be offloaded to disk.

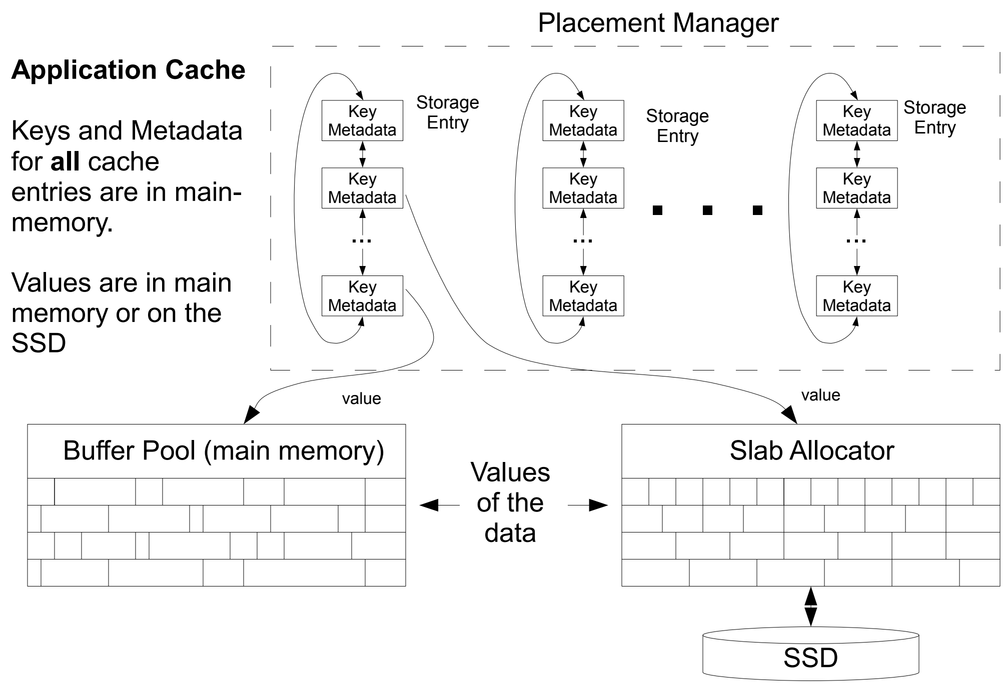
Placement Manager
Application Cache
Keys and Metadata for **all** cache entries are in main-memory.
Values are in main memory or on the SSD
Key Metadata
Storage Entry
value
Buffer Pool (main memory)
Values of the data
Slab Allocator
SSD

Figure 2. The *spill-over* Architecture

The work-load driver uses the *spymemcached* client [11] to drive invocations of the Memcached's client API on the server. First, the driver creates (i.e., caches) a data-set with the specified number of elements, whose elements have the average specified size. Then, the driver invokes a sequence of read and write operations that are mapped to the RETRIEVE and UPDATE API. After warming up the system for 30 minutes, we report the maximum number of such operations per second measured over 30 minutes.

3.2. System Under Test

The system-under-test consists of two clients connected to a server *via* two full-duplex, 1GB Ethernet networks. The server is an Intel X5550, running at 2.66 GHz, with one core and one thread. (Given the network and client characteristics, we had to turn off seven of the eight cores in order to saturate the server.)

The server contains a 160GB SSD PCIe Adapter [4]. The reported performance features of the SSD are:

- High performance – 230 times better IOPS. That means, for example, 97,014 IOPS at 4K block random reads versus 420 IOPS for a 15K RPM 146GB disk drive.
- Low latency of 50 μ-seconds – 1% of the latency of a 15K RPM 146GB disk drive.
- 600 MB/sec random writes sustained.

3.3. Results and Analysis

Figure 3 and Figure 4 show the performance of our implementations of the *synchronous* and *spill-over* disk-offload architectures. The x-axis shows a range of data-set sizes, starting with 0.8 of the available main-memory, and increasing till 12.0 times the available main-memory. The y-axis shows the throughput achieved by a given disk-offload architecture, expressed in terms of operations per second. Figure 3 shows the performance for data-values that average 1KiB; Figure 4 for data-values that average 16KiB. Each Figure

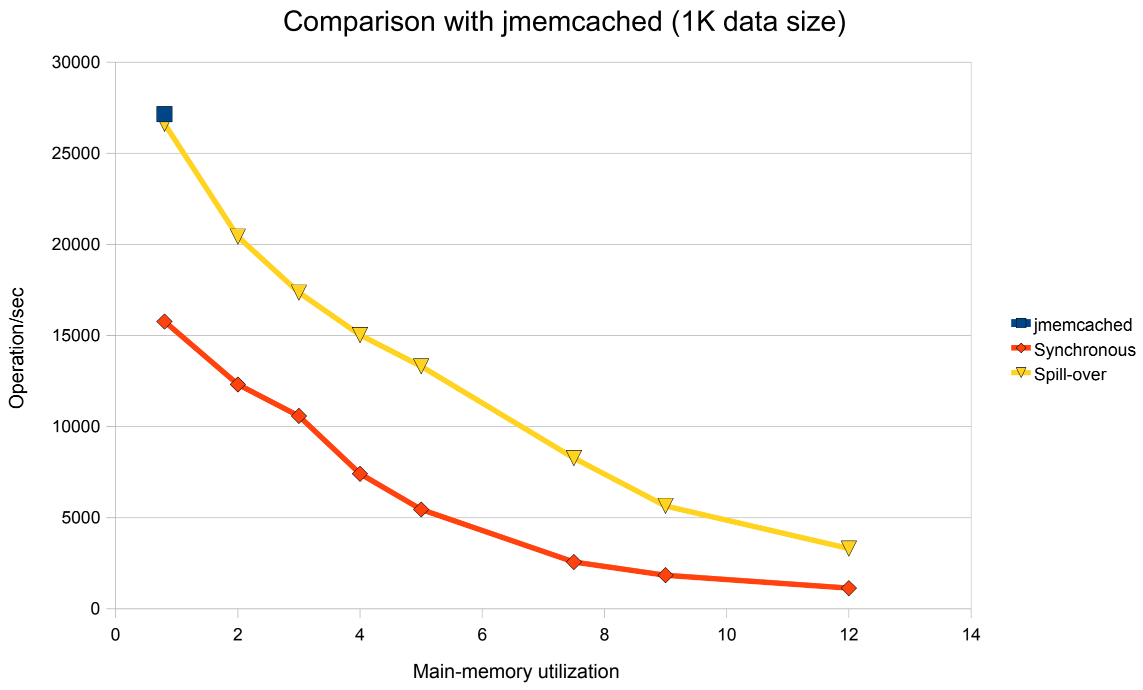

Figure 3. Disk-Offload *versus* In-Memory Performance: Average Data-Size == 1K

also shows the base-line performance of *jmemcached*, representing the performance of a main-memory-only application-caching middleware. Because our implementations of *synchronous* and *spill-over* are embedded inside the *jmemcached* middleware, their performance cannot do better *jmemcached* for the "in-memory" (0.8) case. *jmemcached* cannot do disk-offload and, when its main-memory capacity is exhausted, simply throws out cache data to make room for incoming data.

First, note that for the in-memory case, *spill-over* performs about as well as *jmemcached*: its performance is 98% of *jmemcached* for the 1KiB scenario, and 97% of *jmemcached* for the 16KiB scenario. In contrast, *synchronous*'s performance is 58% and 38%, respectively, for the in-memory case. In other words, *spill-over* shows that enhancing application-caching middleware with disk-offload capability imposes negligible overhead for environments where this capability is not needed. Note that *synchronous* could have used the same methodogy: offloading to disk only when memory is (nearly) full. This most likely would have resulted in a graph that started at 97% or 98% of *jmemcached*, and declined to the *synchronous* results as main-memory utilization increased to large values.

Second, note how much better *spill-over* performs than *synchronous* for disk-offload scenarios. In fact, the performance of *spill-over* increases relative to *synchronous* as the disk-offload factor increases. This is most likely due to the Slab allocator response-time being roughly constant, as opposed to the linear search required by *synchronous*.

On the 1KiB scenario (Figure 3), the relative performance of *spill-over* versus *synchronous* peaks at a disk-offload factor of 7.5 (3.23x). The reason is perhaps because, as the disk-offload factor increases, more offload operations are necessary, and so the advantages of *spill-over* over *synchronous* become less significant.

4. Conclusions

This paper shows that the *spill-over* disk-offload architecture can provide performance that imposes almost no penalty on existing main-memory application caching middleware for the in-memory scenario. It also shows that the *spill-over* architecture is superior to the *synchronous* architecture for disk-offload scenarios. There are two reasons for this. First, *synchronous* is unnecessarily aggressive in moving data from main-memory to disk, periodically moving "dirty" data to

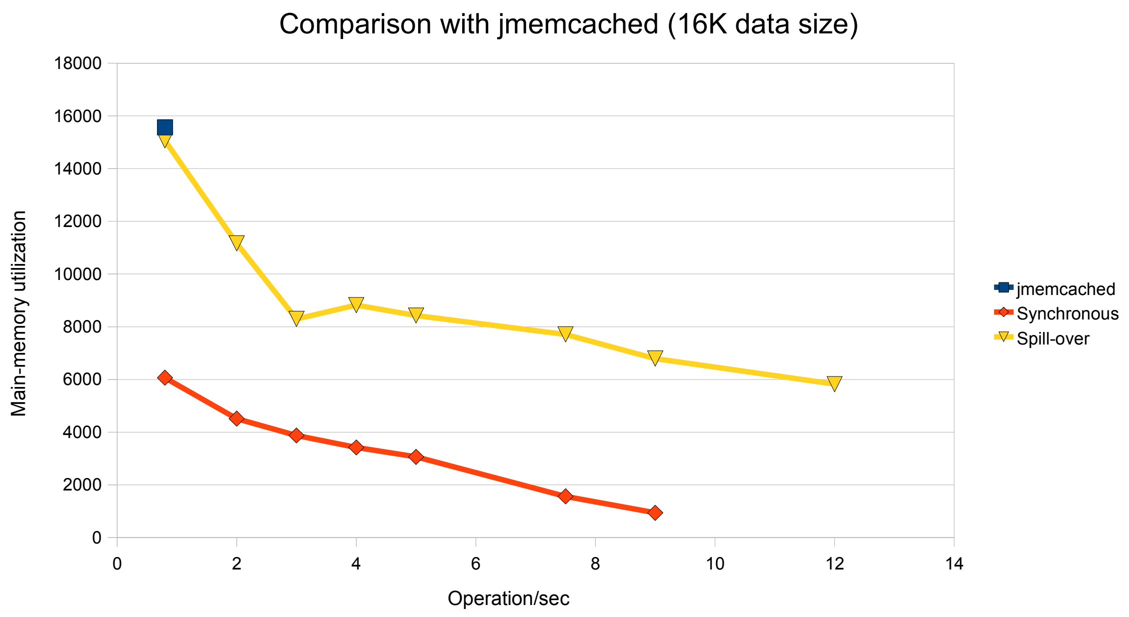

Figure 4. Disk-Offload *versus* In-Memory Performance: Average Data-Size == 16K

disk so as to keep main-memory and disk synchronous. In contrast, *spill-over* postpones this expensive operation until the system actually needs main-memory so that other data must be evicted to disk. Second, *spill-over* is more efficient than *synchronous* in the way it manages space allocation on disk, using constant-time operations rather tha linear search of the disk freelist. This is because *spill-over* allocates fixed-size blocks on disk, rather than variable-size blocks like *synchronous*.

Having designed and implemented these alternative disk-offload architectures, it seems obvious in hindsight that *spill-over* will perform better than *synchronous*. However, it is important to note that *spill-over* achieves this performance by being a special-purpose middleware, suitable only for an environment for which a cold-restart is an adequate recovery strategy. Our original design (*synchronous*) could be extended to implement Durability. As soon as these requirements change – e.g., because data are too important to be corrupted; because of atomic (transactions), or atomicity requirements; or because a cold-restart takes too much time – some degree of synchronicity becomes important. Furthermore, *spill-over* relies on storing all the keys in RAM. While this architecture could be modified to fetch the keys from SSD if necessary, it would result in slower performance for *spill-over*.

On the one hand, this would be more difficult with a *spill-over* design. On the other hand, *synchronous* does not go all the way towards implementing Durability. For *synchronous* to go farther towards durability would make it even slower.

References

[1] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[2] Oracle coherence. `http://www.oracle.com/technetwork/middleware/coherence/overview/index.html`, 2011.

[3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

[4] IBM. `http://www.redbooks.ibm.com/abstracts/tips0729.html`, 2010.

[5] S. Jiang, F. Chen, and X. Zhang. Clock-pro: an effective improvement of the clock replacement. In *Proceedings of the annual conference on USENIX Annual*

Technical Conference, ATEC '05, pages 35–35, Berkeley, CA, USA, 2005. USENIX Association.

[6] Jmemcached. `http://code.google.com/p/jmemcache-daemon/`, 2010.

[7] Memcached. `http://memcached.org/`, 2010.

[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17:94–162, March 1992.

[9] Netty project. `http://www.jboss.org/netty`, 2011.

[10] Schooner nosql / memcached appliance. `http://www.schoonerinfotech.com/products/memcached_nosql_cache_appliance`, 2011.

[11] spymemcached: Java client for memcached. `http://code.google.com/p/spymemcached/`, 2011.

[12] M. Stonebraker and J. Hong. Saying good-bye to dbmss, designing effective interfaces. *Commun. ACM*, 52(9):12–13, 2009.

[13] C. B. Weinstock and W. A. Wulf. An efficient algorithm for heap storage allocation. *SIGPLAN Not.*, 23:141–148, October 1988.

[14] Websphere extreme scale. `http://www-01.ibm.com/software/webservers/appserv/extremescale/`, 2010.