The Pennsylvania State University
The Graduate School
College of Engineering

IMPROVING PERFORMANCE OF IN-MEMORY KEY-VALUE STORES USING A 3D-STACKED ARCHITECTURE

A Thesis in
Computer Science and Engineering
by
Ivan D. Stalev

© 2015 Ivan D. Stalev

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2015

The thesis of Ivan D. Stalev was reviewed and approved* by the following:

John M. Sampson
Professor of Computer Science and Engineering
Thesis Advisor

Mary Jane Irwin
Professor of Computer Science and Engineering

Lee Coraor
Professor of Computer Science and Engineering
Director of Academic Affairs

*Signatures are on file in the Graduate School.

Abstract

Web services and cloud computing are rapidly growing as more users get online around the world and utilize the internet for a growing number of purposes. This puts more demand on in-memory key-value stores as web servers must handle a massive influx of user requests. Data centers will thus find it more challenging to meet their SLAs (Service Level Agreements), as the latency of the 90th percentile of requests may become quite unpredictable. To alleviate this growing concern, we utilize a stacked DRAM architecture as a LLC (last-level cache) that is modified to exploit some common power-law access patterns in user requests. More specifically, we observe that the majority of the memory traffic generated by a key-value store is due to requests for large values, even though large values account for a very small portion (typically around 5%) of overall requests. Thus, we choose to prioritize the cachelines that belong to large values in the stacked DRAM cache by allowing priority cachelines to only be evicted by other priority cachelines. Using this priority scheme, we are able to improve the 90th percentile request latency by as much as 42.4% over a standard stacked DRAM cache architecture.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Data centers are heavily relied upon as web services and cloud computing continue their massive growth. For example, in the first quarter of 2015, Facebook reported having 936 million daily active users [1]. A detailed request trace from Facebook's servers in 2012 reported an average of 54 thousand requests per second. These numbers are quite staggering and will only continue to grow. Throughout this growth, data centers must strive to meet minimum performance requirements, known as SLAs (Service Level Agreements). SLAs assure a customer that his/her content will be available nearly 100% of the time and requests for the content will be served within a certain time frame. In other words, an SLA promises a certain QoS (Quality of Service) level. Meeting the SLA requirements becomes more challenging as demand grows, especially during peak times of day. To improve performance, many web services attempt to serve most of their requests directly from memory, without disk operations being on the critical path. Memcached [2] and Redis [3] are two of the most popular in-memory key-value stores used by popular web-service companies such as Google, Facebook, Craigslist, Snapchat, etc [4]. More explanation on such key-value stores can be found in Appendix A. These key-value stores are commonly used as a caching layer between a front-end web server and a backend database and typically have a hit-rate of 95% or higher [2] [5] [6]. The goal is to keep the most frequently accessed data in this cache so that it can be served directly from main memory. However, as demand continues to rise, meeting a target QoS will become increasingly difficult [7]. More specifically, the 90th percentile of requests (the slowest 10% of requests) may not satisfy the expected (and typically paid-for) QoS by a customer. Simply adding more memory to servers would be costly not only in terms of the physical hardware, but also due to the extra space (in terms of area) consumed in the real estate of the data center. This will quickly lead to the need to build more data centers, the cost of which exceeds 1 billion USD [8].

A very promising solution to the issues explained above is 3D DRAM stacking. Stacked DRAM provides a much higher bandwidth and lower latency than the standard 2D DRAM [9] [10] [11]. Naturally, it also leads to a much higher density in terms of area since chips are now stacked vertically as opposed to a horizontal arrangement. One possible such architecture is having layers of DRAM stacked on top of one other, with the core layer being the bottom-most layer, all in the same package. Stacked DRAM is typically used as a LLC (Last-Level Cache) [12] [13] [14] [15]

but it can also be used as a portion of main memory [16] [17]. The ultra-high bandwidth can be achieved by the use of TSVs (Through-Silicon Vias). A TSV array consisting of one million TSVs running at 1GHz can provide as much as 1Tb/sec of bandwidth [18]. As of 2013, state-of-the-art manufacturing allows for tens of millions of TSVs in just one square centimeter [19]. The vastly increased bandwidth also leads to new implications as to how it should be utilized. Previous issues such as the trailing-edge effect will virtually cease to exist [18]. Consequently, fetching larger cache lines no longer "wastes" precious bandwidth since bandwidth is abundant. In fact, a new issue arises that not all the potential bandwidth can even be fully utilized. DRAM stacking is already out in commercial products [20] [21] and provides a significant increase in performance. One of the key drivers for 3D DRAM stacking has been the widening gap between CPU and memory performance, referred to as the Memory Wall [22] [23] [24].

Due to its high bandwidth, low latency, and high density, stacked DRAM is an ideal architecture for memory intensive workloads such as key-value stores. It would relax the need to build new data centers and would also provide a performance boost compared to current server architectures.

After analyzing publicly available data based on production key-value stores, we make some important observations. Requests tend to follow power-law access patterns referred to as Zipf's law [25]. More specifically, the popularity of documents as a function of size, the distribution of user requests for documents, and the number of references to documents as a function of their overall rank in popularity can be modeled using power-law distributions. For example, the majority of document sizes are small (on the order of tens and hundreds of bytes) and the majority of requests are for small documents. However, the majority of memory traffic is not for small documents. For example, 100 requests for 100B values results in 10KB of memory traffic, whereas just a single request for a 1MB value triggers 1MB of memory traffic. The infrequent 1MB request can therefore slow down all the other requests, especially for single-threaded key-value stores such as Redis whose request queue is served by a simple round-robin mechanism.

Based on these key observations, we design a stacked DRAM cache architecture where we prioritize the cachelines that belong to large values and priority cachelines can only be evicted by other priority cachelines. The results show a significant improvement in the 90th percentile latency, which often accounts for the requests that fail to meet SLAs. Aside from QoS, an improvement in tail latency also provides more opportunity for other optimizations, such as DVFS (Dynamic Voltage and Frequency Scaling), more/better co-scheduling, etc.

Chapter 2
Related Work

This section goes over similar works related to either key-value stores, stacked DRAM, or both.

EuroCloud is a European joint project between industry and academia led by ARM but also includes Cancer Research UK, IMEC (Interuniversity Microelectronics Centre), EPFL (Ecole Polytechnique Federale de Lausanne), UCY (University of Cyprus), and Nokia. Its primary goal was to build a 3D server-on-chip concept integrating many ARM processor cores with 3D DRAM for very dense and low-power data centers. After its completion in 2013, the project claims to have a significant impact on society, environment, and the European IT sector [26]. More specifically, the authors/implementors were able to realize a 20% performance improvement as well as a reduction in power consumption by using stacked DRAM as a last-level cache. They have created a suite of benchmarks, called CloudSuite [27], which is representative of the typical workloads being processed in data centers. CloudSuite includes Memcached and uses a real-world dataset from Twitter. However, EuroCloud focuses on improving data center workloads as a whole, and does not focus on specific key-value store optimizations as we do in our work.

Gutierrez et al. simulate a 3D-stacked DRAM architecture using the same architectural simulator that we use, GEM5 (see Chapter 4). By adding stacked DRAM to data centers, the authors project a 2.9x improvement in density (i.e. more hardware per unit area) [28]. They focus on optimizing for density due to the high cost of data center real estate. When running Memcached on the simulated system, they realize a 10x improvement in throughput measured in TPS (transactions per second) by tightly coupling NICs (Network Interface Cards), DRAM, and the processors. It is not surprising that a memory bound key-value store would benefit from a stacked DRAM cache which has a higher bandwidth and lower latency than main memory. The authors do not make any optimizations based on the workload characteristics of a key-value store, such as the known power law patterns in the frequency of keys and value sizes.

The design/implementation of the DRAM cache architecture we use is called Alloy Cache and comes from Qureshi et al. [12]. Alloy Cache stores both the tag and data together as opposed to having separate tag and data arrays as is typically done in SRAM caches. Consequently, there is no serialization delay for having to first look up the tag and then the data, i.e. only a single access is needed. The authors make a point that latency is a more important design constraint for DRAM caches than hit-rate and therefore make Alloy Cache a direct-mapped cache. To avoid

the relatively expensive DRAM cache miss penalty, the authors create an access predictor with a performance that is only 2% worse than a perfect predictor. Please refer to Appendix B for a more in-depth explanation on Alloy Cache, along with other DRAM cache architectures. Qureshi et al. only evaluate Alloy Cache using SPEC2006 benchmarks, whereas we are using it for an in-memory key-value store.

At the time of this writing, the newest/best DRAM cache implementation that builds onto Alloy Cache (explained in the previous paragraph) and Footprint Cache [13] is called Unison Cache by Jevdjic et al. [14]. Unison Cache takes two of the main advantages of Alloy Cache: 1) storing the tags in DRAM, making the cache capacity scalable since they do not need to be stored in precious SRAM 2) avoiding the tag-then-data serialization by fetching them at the same time. It also takes two of the main advantages of Footprint Cache: 1) Having tags be page-based instead of block-based, thus leading to a small total tag storage 2) High hit-rate due to having an associativity and a very accurate way predictor which fetches the correct way 95% of the time. Footprint cache is evaluated on server workloads, but again, it does not focus on specifically optimizing for key-value stores. To compensate for the fact that Alloy Cache is slightly outdated, we partially implement associativity in it in order to predict the effect. Regardless of the specific DRAM cache architecture, we can conclude that our optimization policy will still be effective due to the overall DRAM cache characteristics (i.e. a stacked DRAM cache LLC having lower latency and higher bandwidth than main memory).

Atikoglu et al. analyze a real request trace from Facebook's Memcached servers [6]. They observe a consistent pattern in terms of the frequency of the keys accessed as well as the value sizes. For example, 40% of the keys have corresponding values of 400B or less, and 90% of the keys have values of 50KB or less. Approximately 5% of the keys have large values of 1MB or larger. In other words, most keys have relatively small values. The authors also observe that the frequency of the accessed keys follows a power law. Approximately 50% of the keys appear in only 1% of all requests, whereas 10% of the keys appear in 90% of the total requests. In other words, a relatively small set of keys are considered hot since they account for the majority of the requests. One way these trends can be exploited is to focus optimization efforts for the hottest keys. For example, if those keys are always fetched from the hardware cache instead of main memory, the result will likely lead to a performance benefit.

Basu et al. observe that memory addresses suffering from cache misses typically exhibit repetitive patterns due to the temporal locality inherent in the access stream. The authors identify the set of memory addresses that account for a significant portion of LLC misses (and hence performance degradation). They create a novel LLC architecture, called Scavenger [29], which divides the LLC by carving out a section to store the blocks which are identified to miss most frequently. In other words, the LLC in the Scavenger architecture prioritizes the cachelines that have recently caused a large number of misses and retains them in a victim cache. Our work also prioritizes cache lines, but it does so in a DRAM (as opposed to SRAM) cache and uses different priority metrics.

Chapter 3
Optimization

In this chapter, we describe our optimization approach in detail in order to improve the performance of in-memory key-value stores using the priority-based scheme introduced in Chapter 1. We explain the modifications that have to be applied to Redis and the micro-architecture for the priority policy to work in a real system.

3.1 Key-Value Store Access Pattern

In-memory key-value stores are heavily used by web services in order to improve the performance of most requests by serving them from memory instead of disk [30]. As the name suggests, each entry in the key-value store is stored as a key-value pair. In terms of Facebook, the key could be a unique post-id and the value would be the corresponding Facebook post written by a user. Powerful relational databases such as MySQL are unnecessary since features like merging and joining tables are not needed. The goal is to simply obtain a value/document and return it to the user in as little time as possible. For more explanation on what key-value stores are and how they work, please refer to Appendix A. Here we go over the typical access patterns seen in key-value stores across industry. The observations are based on Atikoglu et al. [6] and are significant to understanding our study. As we briefly discussed in Chapter 2, the authors analyze a real request trace from Facebook's Memcached servers over the course of several days which amounts to over 284 billion requests. The key takeaway from the paper is that Facebook's traces follow Zipf's law [25] which states that many characteristics of Web use can be modeled using power-law distributions, including the distribution of document sizes, the popularity of documents as a function of size, the distribution of user requests for documents, and the number of references to documents as a function of their overall rank in popularity. In this case, the documents are the values in the key-value stores and each document has a unique identifier to serve as the key. Atikoglu et al. show that a significant portion of requests are for relatively small values (on the order of a few tens of bytes), while only about 5% of requests are for large values of around 1MB. Similarly, half of the key space accounts for only 1% of all requests. This means that half of the total keys repeat very few times (if at all), while a small portion of the keys account for a large portion of all requests.

3.2 Performance Optimization

As we mentioned in the previous section, the majority of values in a key-value store are small, i.e. a few tens to hundreds of bytes. However, although requests for large values (1MB or more) are rare, they do account for a majority of the memory and network traffic. Since Redis is single-threaded, all requests (regardless of their size) end up in the same queue waiting to be serviced. When a request for a large value is in the queue, most requests that follow it will likely be for small values. However, these small-valued requests must now wait longer in order to be serviced since a large-valued request has "plugged-up" the queue. This inevitably results in an overall upward shift in the QoS (Quality of Service) curve. In other words, the latency of all requests is now higher due to the few large-valued requests taking a relatively long time to complete.

Realizing the effect of large-valued requests on overall performance, we consider the architecture of the server system we are using. Stacked DRAM provides a low-latency, high-bandwidth memory whose overall performance sits between SRAM and main memory. When used as an on-chip LLC (last-level cache) stacked on top of the CPUs (as we do in this study), capacities can range between hundreds of megabytes to a few gigabytes. This is extremely large compared to modern on-chip SRAM LLCs which are typically less than 100MB. With this in mind, we attempt to improve the overall QoS by treating memory requests for large values with a higher priority than the rest. We do this by modifying our DRAM cache to treat cachelines corresponding to large requests as priority cachelines. A priority cacheline can only be evicted by another priority cacheline, whereas a non-priority cacheline can be evicted by any cacheline. The goal is to serve memory requests for large values quicker by having them be in the DRAM cache. This optimization proves to be effective, as can be seen in the Results section.

3.2.1 Modifications to Redis

We modify Redis in order to identify the address ranges that correspond to large values. When Redis allocates a megabyte or more for a value, then the corresponding address range becomes a priority one. Once the memory is de-allocated (the pointer is freed), the range is removed from the priority set. These addresses are stored in a piece of hardware explained in the next section. Please refer to Appendix B for more details on our modifications.

3.2.2 Modifications to Micro-Architecture

Redis needs a way to communicate to the hardware the addresses that it has marked as priority ones. This is accomplished by adding two extra instructions to the ISA (Instruction Set Architecture), one for adding an address range and the other for removing. The custom instructions take the virtual address and size as arguments. The virtual address is then translated by the TLB (Translation Lookaside Buffer) or page table and then the corresponding physical address is stored in a special structure within the DRAM cache memory controller. In practice, this special structure would be finite in size and can, for example, hold the set of most recent address ranges. The size of this structure can be derived empirically by considering power, area, and

performance trade-offs.

3.2.3 Alternatives to Modifying Redis

Instead of modifying Redis, another approach is to modify Linux's malloc system call. Whenever malloc requests more than a certain amount of memory (1MB in this case), Linux can instead call the custom instruction. However, this would affect other applications as well, which is an undesired effect. One could potentially attempt to detect priority addresses at the memory controller level as well by looking at which cachelines are always accessed together. However, this approach would be quite complicated for it to work accurately.

Chapter 4
Workloads and Results

4.1 Workloads

Here we describe the workloads we ran on our simulated system for this study.

4.1.1 SPEC Benchmarks

We start out by running a few SPEC2006 [31] benchmarks to see how their performance improves on our 3D-stacked architecture. In general, SPEC2006 benchmarks have working sets on the order of tens of megabytes [32] which is relatively small compared to typical server workloads. We pick the benchmarks with larger working sets in order for the DRAM cache (having a capacity of 128MB or more) to have an effect. Table 4.1 shows the workloads we chose and their corresponding L3 cache MPKI (Misses Per Kilo Instruction) that we measured on our simulated system. The results can be seen in Section 4.3.1. We intend to achieve an even higher performance improvement when running a server/datacenter workload which has a much larger working set and should therefore benefit more from a stacked DRAM cache. As a result, we chose Redis – an in-memory key-value store which should intuitively benefit from a high-bandwidth and low-latency memory. We initially chose Memcached (another popular in-memory key-value store) but quickly switched to Redis due to Redis being much better documented.

Workload	**Benchmarks**	**MPKI**
WL1	lbm, libquantum, mcf, milc	20.37
WL2	gems, libquantum, milc, soplex	15.25

Table 4.1: Workloads containing a mix of SPEC2006 benchmarks

4.1.2 Redis

The primary application we evaluate for our study is Redis: an open-source, in-memory key-value store. Redis is heavily used in industry by popular web services such as Snapchat, Github, Twiter, Pinterest, among many others. A key-value store such as Redis usually sits between a front-end

web server and a persistent database. It serves as a cache to hold the hottest (i.e. currently the most frequently accessed) data in order to provide better performance by having requests be served from memory instead of disk. Key-value stores are usually optimized to achieve hit-rates of 95% or more; thus, most requests are served by them. A miss in the key-value cache results in a request to the back-end database which then updates the cache.

4.1.2.1 Redis Benchmark

Redis provides a benchmarking utility which simulates multiple clients sending requests. We heavily modify this utility in order for it to support our needs. For example, the utility generates integer keys at random given a user-supplied upper bound and then converts them to string keys. Our modification allows the utility to take a set of input keys and then generate requests off of those keys based on the power law frequency pattern observed in the work of Atikoglu et al. [6]. In addition, the utility is able to read a request trace file as the one described in the next paragraph and generate requests based off of it.

4.1.2.2 Wikipedia Traces

We create a synthetic workload using the publicly available Wikipedia access traces [33]. The traces include five months of user requests to Wikipedia: from September 2007 to January 2008. Each request entry includes four fields: 1) a unique value from a monotonically increasing global request counter 2) a Unix timestamp with milli-second precision 3) the requested URL 4) whether or not the request caused a database update. The traces do not contain any PII (Personally Identifiable Information) such as locations and IP addresses. The requested URLs represent the keys in the key-value store. The synthetic workload that we generate uses the same key sequence and frequency as the Wikipedia traces. However, we simplify and generalize the values to have only three unique sizes: 10 bytes, 300 bytes, and 1 megabyte. These numbers represent the general data points we gathered from Atikoglu et al. [6]. More specifically, 50% of the requests are for the 10B values, 45% are for the 300B values, and the remaining 5% are for the 1MB values. We use this discrete approximation since we do not know the size for all the values from the Wikipedia trace. Naturally, each URL in the trace can be visited and the value size can be derived. However, most value sizes would have changed or even disappeared since the time of the trace (8 years ago from the time of this writing). Also, modifying the Redis benchmark utility to dynamically generate values of a wide range of sizes would require more development time which we were unable to afford.

Figure 4.1 shows the frequency of accesses to unique keys derived from the Wikipedia trace. Over a period of 2.85 seconds, there are 5000 requests to 2710 unique keys. One might assume that the keys are accessed uniformly, meaning that each key is accessed approximately two times (5000 / 2710). Interestingly, the key frequency follows a power law pattern where most keys are accessed only once, but a small fraction are accessed very frequently (as much as 76 times in this instance). A very similar pattern was observed in Facebook's request traces analyzed in the work of Atikoglu et al. [6]. Their timescale, however, was on the order of minutes and hours

but from analyzing the Wikipedia trace we see that the pattern exists for second intervals as well. This is significant due to the fact that we are dealing with hardware-level optimizations where events occur on the order of microseconds or even nanoseconds. Figure 4.2 shows how the key-frequency distribution in the Wikipedia trace varies between the time intervals of 1 second, 1 minute, and 1 hour. The graph is a CDF (Cumulative Distribution Function) where the X-axis represents the cumulative unique keys and the Y-axis is a log-scale representing the portion of total requests. For example, looking at the 1-second graph, it can be observed that 50% of the unique keys are accessed by only 30% of requests and 95% of the unique keys are accessed by 67% of the requests. Consequently, that leaves the remaining 5% of keys which are accessed by 33% of the requests, thus making them the hottest keys in the database.

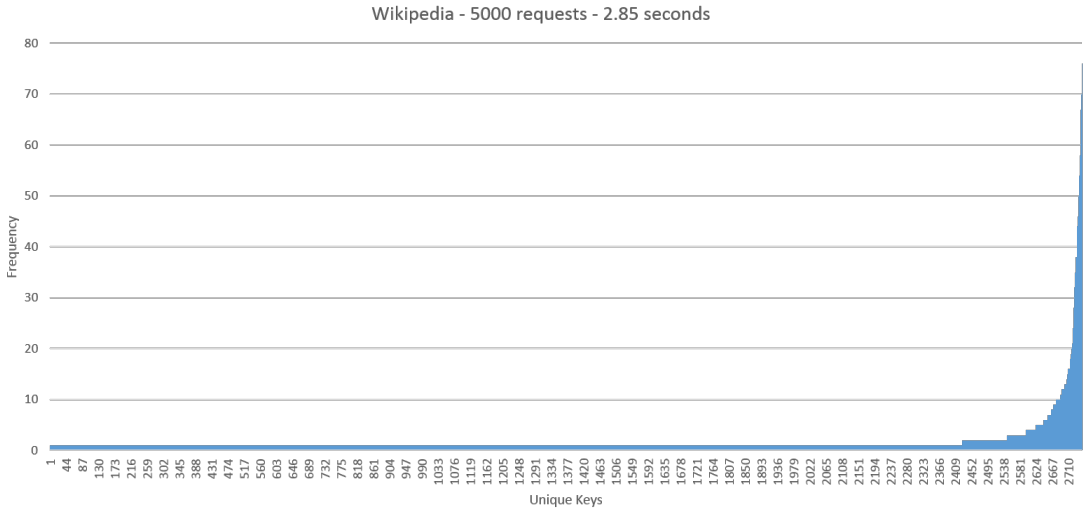

Figure 4.1: Key frequency

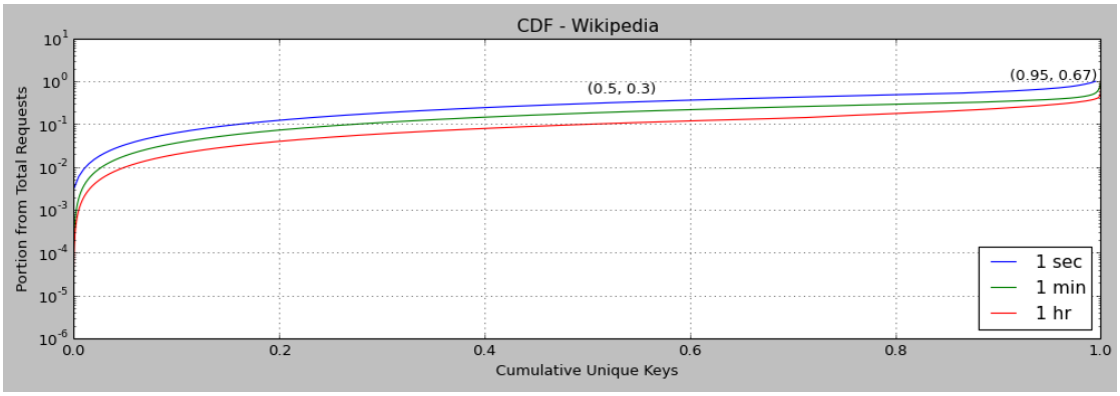

Figure 4.2: Key frequency CDF

4.2 Methodology

The following sections describe how we set up and ran Redis on our simulated system.

4.2.1 Simulation Infrastructure

We use the GEM5 [34] open-source architectural simulator to perform full-system simulation. The simulator supports multiple ISAs (Instruction Set Architectures) and we chose ARMv8 64-bit due to it being the most active and updated by the GEM5 developer community. We cross-compiled Linux 3.16 with PCIe Ethernet support since we are running a client-server setup connected over Ethernet. To simulate the stacked DRAM cache as well as the main memory, we patch the NVMain [35] simulator to run with GEM5. NVMain is a cycle accurate memory simulator designed to simulate emerging volatile and non-volatile memory technologies at the architectural level. We have modified the source code for both GEM5 and NVMain in order to support our optimizations previously described.

4.2.2 Redis Setup

We cross-compiled Redis 2.8.19 (latest stable version at the time) for the ARMv8 64-bit ISA. Redis offers several options for persistence, i.e. saving the volatile memory contents to disk so that the content persist in case of a power outage. Because we only care about the in-memory performance of the key-value store, we disable persistence. We set the Redis server verbosity level to debug (the highest setting) so that we can monitor the server in detail in real-time. Each Redis server instance runs standalone without any replication or partitioning enabled. Redis' 'maxmemory' setting is disabled; in our experiments we make sure not to exceed the system's main memory capacity. We enable Redis' slowlog feature to log all the requests the server handles so that we can obtain fine-grain latency stats on each simulated request.

4.2.3 Experiment Setup

First, we boot the client and server systems and take a checkpoint. After restoring from the checkpoint, we launch four independent Redis server instances in parallel on separate ports. We then use Redis' mass insertion feature in order to populate each instance with 20,000 key-value pairs. Mass insertion is described in more detail in Section A.1.4. At this point, the server system still uses atomic CPUs to take advantage of quicker simulation speed since stats are not yet being gathered. The client system always runs with atomic CPUs since the focus is purely on the server system performance. We also have NVMain's cache warmup feature enabled so that the caches are warmed up for the detailed simulation that is to follow. GEM5 does not currently support checkpointing of caches. However, since the SRAM caches are relatively small and we simulate for a long enough time, we consider the cold cache misses to be negligible. Once the mass insertion is complete, we checkpoint the GEM5 system as well as the NVMain DRAM cache memory contents. We then restore the system using GEM5's detailed out-of-order CPUs and reset both the GEM5 and Redis server stats. The server signals to the client to begin

executing our modified Redis benchmark utility. At the end of the simulation, we query the Redis server to obtain the slowlog as well as the server 'INFO' command to obtain useful stats such as transactions per second, hit-rate, peak memory usage, etc. Detailed specifications of our simulated server and client systems are listed in Table 4.2 and Table 4.3, respectively.

Server System Specs	
Processor	AVMv8 64-bit, 4-core, 2GHz, 4-way
Private L1 Cache	2-way, 32KB I cache, 64KB D cache
Private L2 Cache	8-way, 2MB per core
Shared L3 Cache	16-way, 16MB
Stacked DRAM Cache	128MB - 512MB Alloy Cache (Direct Mapped)
Off-chip DRAM	16GB
TLB size	64 entries per core

Table 4.2: The simulated server system

Client System Specs	
Processor	ARMv8 64-bit, 8-core, atomic CPUs
Off-chip DRAM	16GB

Table 4.3: The simulated client system

4.3 Results

4.3.1 SPEC2006

Figure 4.3 shows the performance measured in IPC (Instructions Per Cycle) for SPEC2006 each workload after we simulated them with varying DRAM cache capacities. We see a noticeable increase in performance simply by adding a stacked DRAM cache to the system. Performance then improves by adding more DRAM cache capacity, but only slightly due to the fact that the working set mostly fits within the cache. We say 'mostly' since our Alloy Cache [12] implementation is direct mapped, so conflict misses occur more frequently even if the overall capacity is greater than the working set. The performance improvement in terms of IPC peaks at 23% for Workload 1 and 19% for Workload 2.

4.3.2 Redis

4.3.2.1 Standard vs Priority DRAM cache

In this section we show and discuss the results of adding a stacked DRAM cache to a system running Redis. As was the case with SPEC2006, we expect to see a performance improvement when using a stacked DRAM cache with Redis. We picked a key-value store workload due to it having a much larger working set than SPEC2006, and thus expect to see a higher performance improvement than what is shown in Figure 4.3 for SPEC2006. As can be seen in Figure 4.4, we

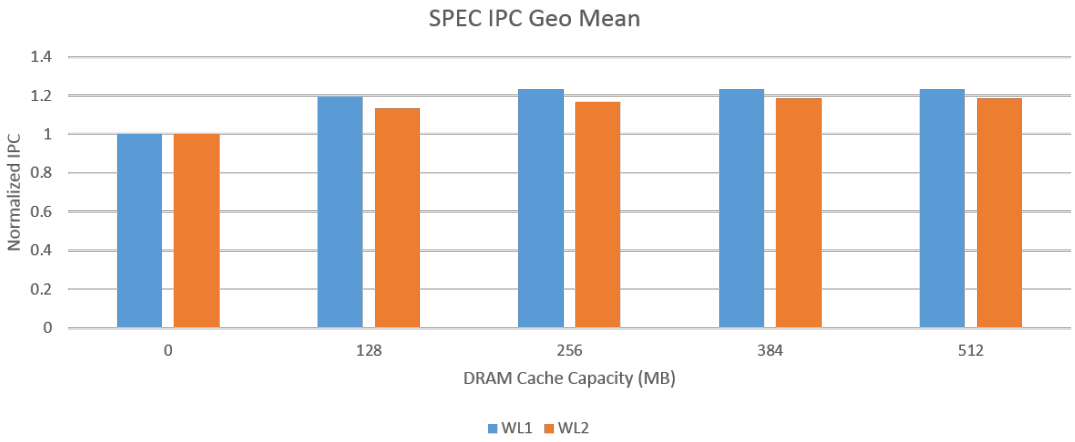

Figure 4.3: IPC (Instructions Per Cycle) of SPEC2006 workloads for different DRAM cache capacities normalized to a system with no DRAM cache

were indeed correct with our assumption. The figure shows the performance measured in RPS (Requests Per Second) for three distinct real Wikipedia traces that range over approximately one second of real time (on our simulated system these traces took between one and three seconds to complete). The chart is normalized to a system which does not have any stacked DRAM cache (i.e. 0MB capacity). Simply adding a stacked DRAM cache provides a minimum benefit of 10%. From there on, performance continues to improve as we add more capacity and peaks with a 32% improvement at 512MB.

Figure 4.5 presents the results in the same manner, but for a priority-based Alloy cache that was explained in Section 3.2. The priority cachelines are the ones that belong to large values, which in our case are of size 1MB or larger. We see a similar trend as with the standard Alloy cache, with performance peaking at 36% for a 512MB capacity. Our goal was to see a performance improvement when using the priority-based cache over the standard Alloy cache. The comparison is shown in Figure 4.6. The priority-based cache consistently outperforms the standard Alloy cache, with the largest benefit being 9% at 128MB. The benefit of the priority-based cache tends to be higher at the smaller DRAM cache capacities. We speculate this is largely attributed to the fact that there is more interference at smaller capacities. Since there are four Redis server instances running simultaneously, there will be more conflict misses at lower capacities especially since the Alloy cache is direct-mapped. Having a priority-based cache reduces the conflict misses that result from non-priority lines.

Although it appears that there is not a spectacular benefit in RPS from the priority-based cache over the standard Alloy cache, the QoS (Quality of Service) curves show us otherwise. The graphs in Figure 4.7 show all the requests sorted by latency when running Redis on the standard Alloy cache. Figure 4.8 is formatted the same way but contains the data points for the priority-based Alloy cache. The interesting part of the graph is the 95th percentile, i.e. the 5% of requests that had the highest latency. By comparing the graphs for the standard and priority-based Alloy cache, we see that the QoS curves are shifted more to the right for the priority

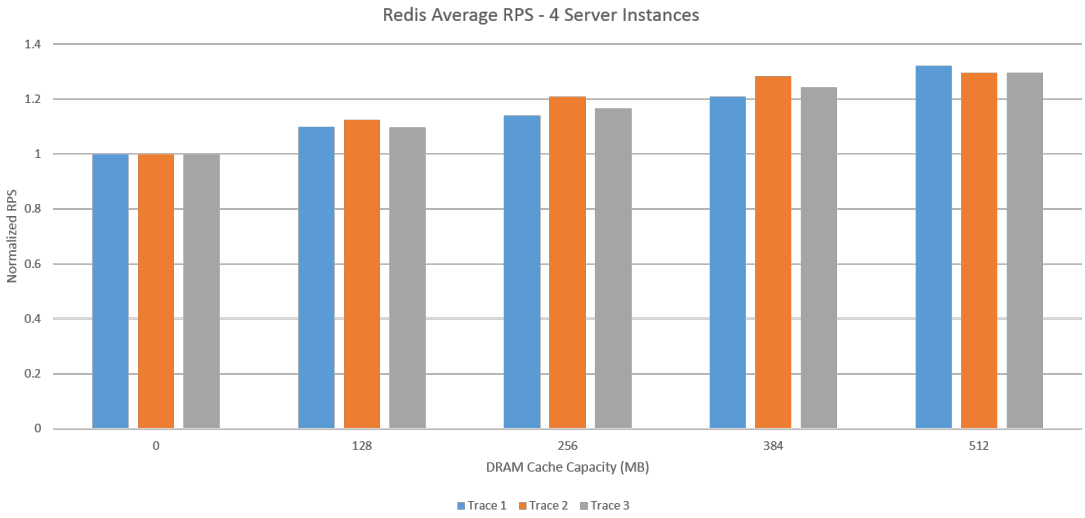

Figure 4.4: Redis performance of 3 Wikipedia traces on the standard direct-mapped Alloy cache with different capacities

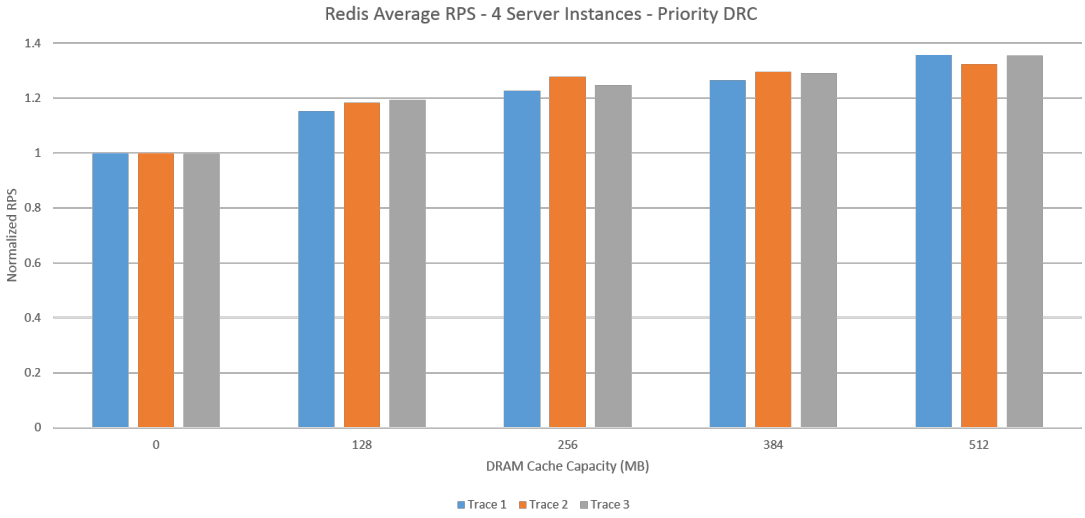

Figure 4.5: Redis performance of 3 Wikipedia traces on the priority-based Alloy cache with different capacities

cache, i.e. there is a benefit. Figures 4.9, 4.10, 4.11, and 4.12 zoom in on the 90th percentile for the 128MB, 256MB, 384MB, and 512MB DRAM cache capacities and separate the latencies by value size. The large values are abbreviated as lg, medium as md, and small as sm. The -p suffix refers to the priority-based cache. The takeaway from these results is quite significant due to the large decrease in the tail (90th percentile) latency when using the priority-based Alloy cache over the standard Alloy cache. Table 4.4 quantifies these results in terms of percentage drop for the average tail latency. The priority-based cache actually improves the tail latency for all value

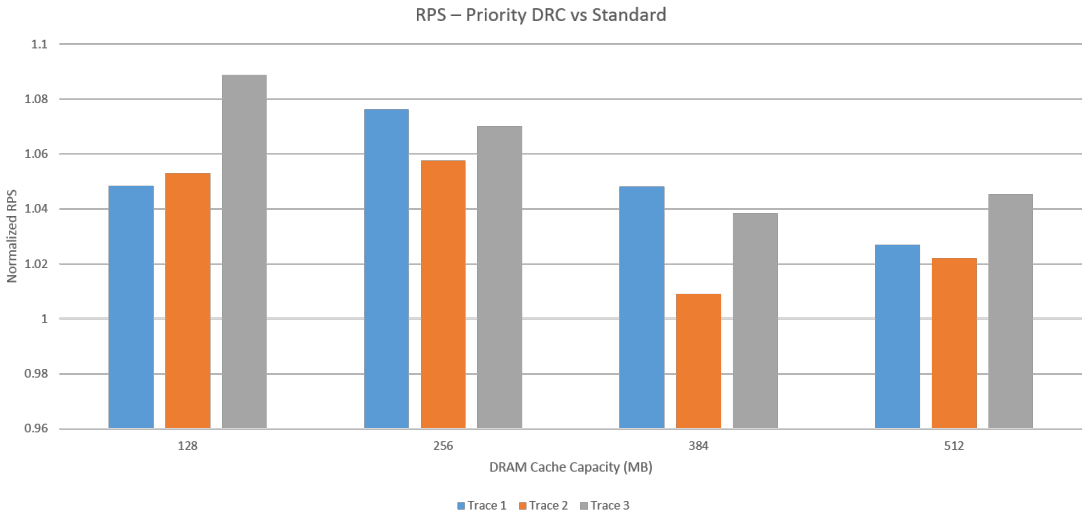

Figure 4.6: Comparison between Figure 4.4 and Figure 4.5 normalized to a system without DRAM cache

sizes, not just the large ones. This is attributed to Redis being single-threaded, so the overall queueing latency is improved since large-valued requests are served quicker (because they are in the DRAM cache). Although the performance improvement measured in RPS is not significant, the fact that the average tail latency decreases as much as 42.4% is a big win for QoS.

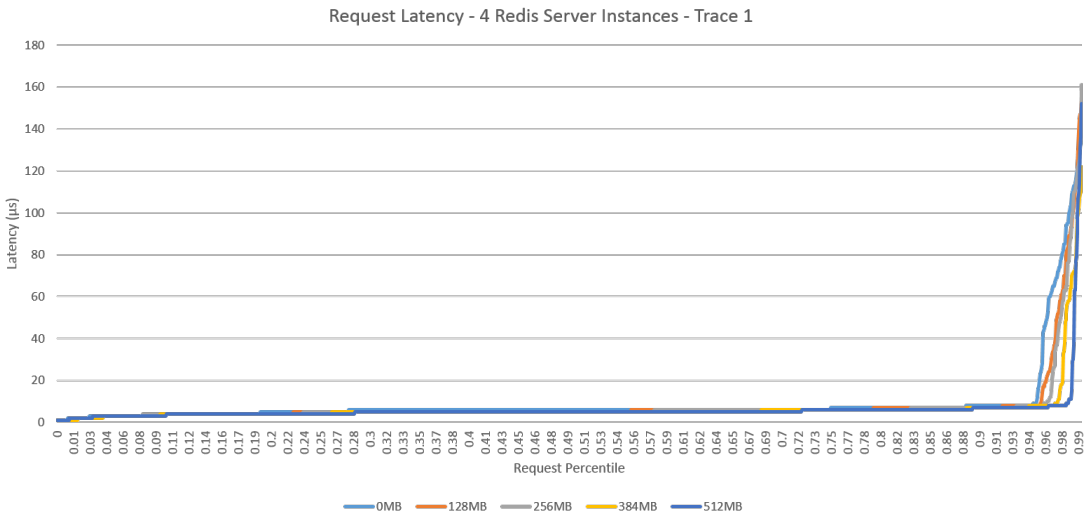

Figure 4.7: QoS curves using different capacities of the standard direct-mapped Alloy cache

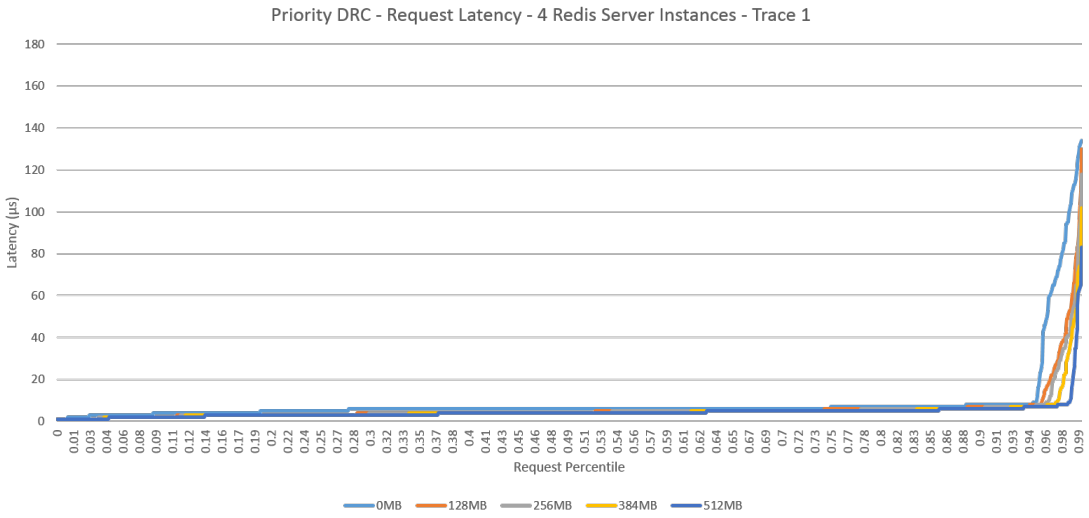

Figure 4.8: QoS curves using different capacities of the priority-based Alloy cache

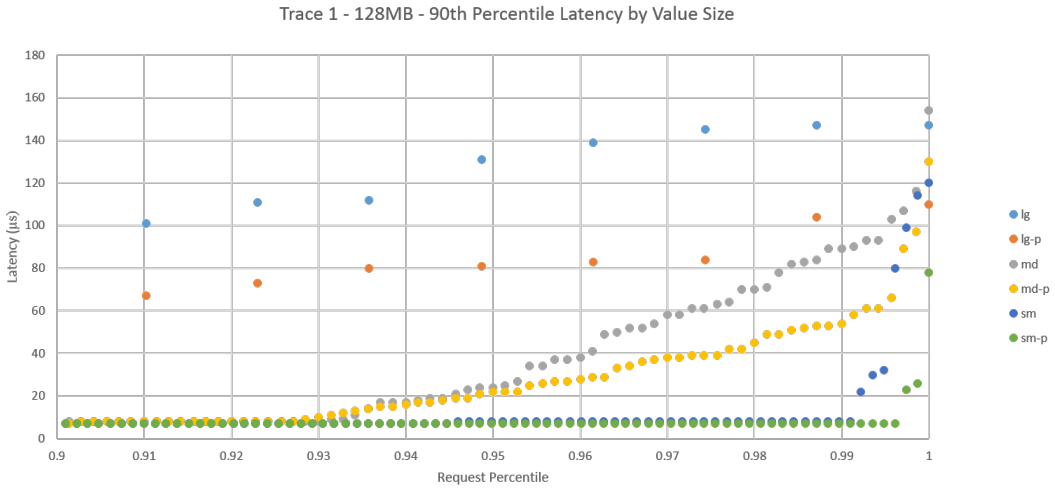

Figure 4.9: Zoom in on the 90th percentile of Trace 1 from Figure 4.7 and Figure 4.8 for a 128MB cache capacity

4.3.2.2 Adding Associativity to the DRAM cache

Naturally, one might wonder if an associative DRAM cache would provide a similar, if not better, benefit in performance. In the Alloy cache paper [12], the authors emphasize that DRAM caches should have latency as the first priority optimization. This leads them to design Alloy cache to be direct-mapped, thus sacrificing hit-rate for a decrease in hit latency. To alleviate frequently paying the miss-penalty due to a lower hit-rate, the authors design a near-perfect predictor which predicts whether or not the access will result in a hit or a miss in the DRAM cache. If the access would result in a miss, the DRAM cache is bypassed and the access goes directly to main

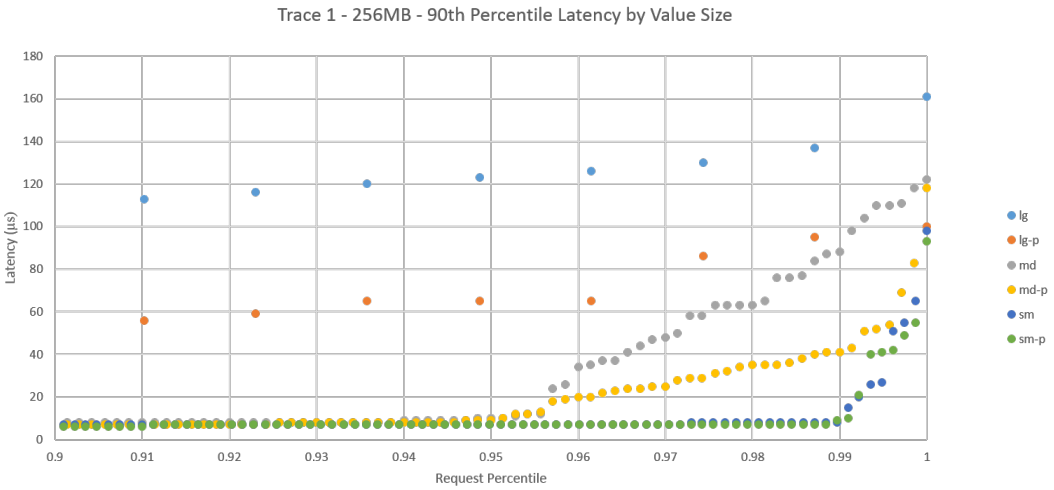

Figure 4.10: Zoom in on the 90th percentile of Trace 1 from Figure 4.7 and Figure 4.8 for a 256MB cache capacity

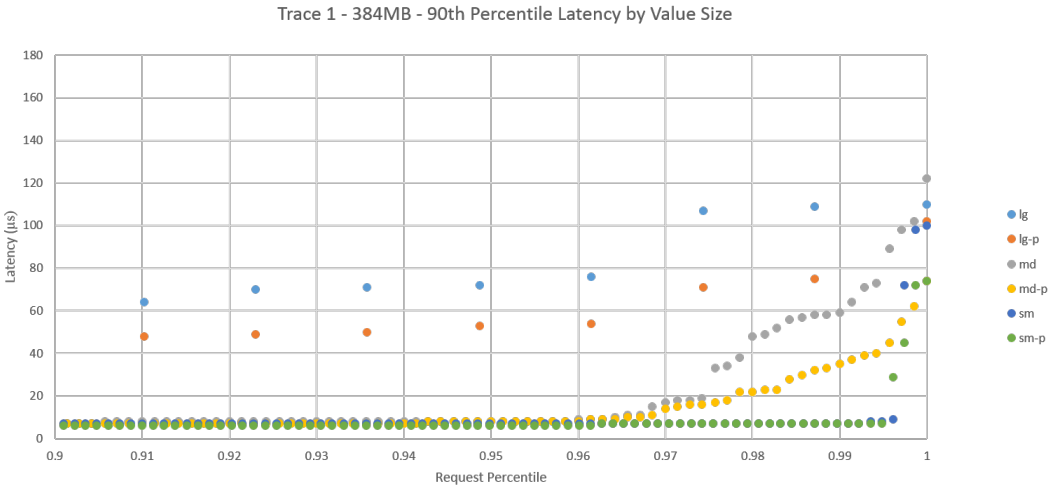

Figure 4.11: Zoom in on the 90th percentile of Trace 1 from Figure 4.7 and Figure 4.8 for a 384MB cache capacity

memory. The authors experiment with an associative Alloy cache, but conclude that overall the performance impact of degraded hit latency outweighs the marginal improvement in hit-rate due to associativity. Adding associativity approximately doubles the burst length and bandwidth consumption and reduces the row buffer hit-rate. As a result, they stick with Alloy cache being direct-mapped.

For curiosity, we also wanted to see how an associative Alloy cache would impact the performance of our workload. More specifically, we are interested in comparing the benefit of only

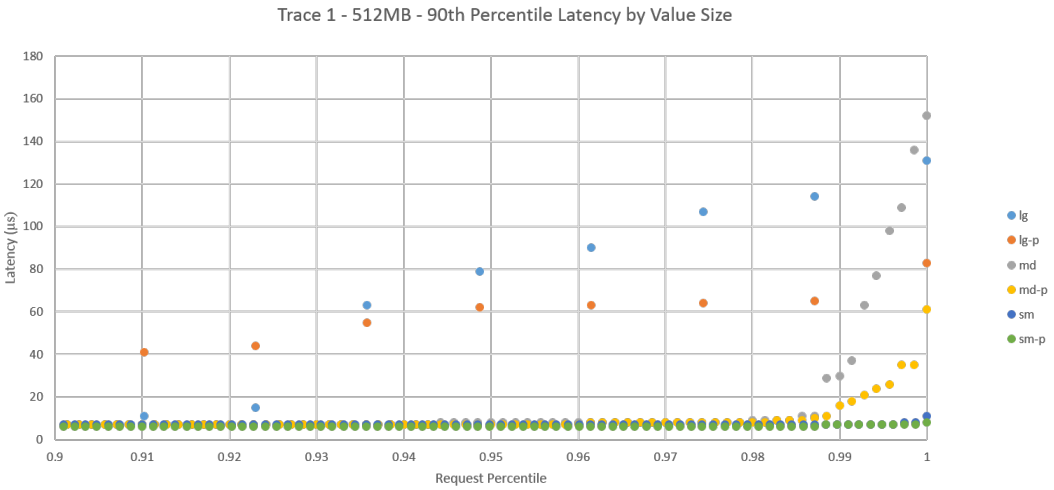

Figure 4.12: Zoom in on the 90th percentile of Trace 1 from Figure 4.7 and Figure 4.8 for a 512MB cache capacity

DRAM Cache Capacity (MB)	**128**	**256**	**384**	**512**
Small Values (10B)	36.6%	3%	13.6%	13.2%
Medium Values (300B)	27.3%	39.4%	35.0%	41.8%
Large Values (1MB)	34.0%	42.4%	26.1%	21.8%

Table 4.4: Percent drop in average tail (90th percentile) latency from standard to priority-based Alloy cache for each value size for different DRAM cache capacities

adding associativity vs the benefit of our priority-based (direct-mapped) Alloy cache. We do not fully implement an associative Alloy cache, but instead keep track of some metadata within NVMain to estimate the benefit. Aside from keeping track of hit-rates, one of our associativity counters records the number of hits to priority cachelines in the DRAM cache that would have otherwise been misses in the standard direct-mapped Alloy cache. Similarly, we keep track of a priority counter that counts the number of hits on priority cachelines in the priority-based Alloy cache that would have otherwise been misses in the standard direct-mapped Alloy cache. The pseudo-code for the priority counter logic is shown in Figure 4.13. The code is executed for every memory access that reaches the DRAM cache controller. The pBenefit variable for each cache set is initially set to false. When a non-priority access results in a miss due to a priority cacheline currently occupying the set (thus a fill request is not generated), the set's corresponding pBenefit value is set to true. If there is one or more hits to that priority cacheline before it is evicted, the priority counter is incremented by one. In other words, we are seeing if there was a benefit in keeping the cacheline in the cache instead of evicting it for a non-priority line. Figure 4.14 shows the normalized results of these counters, while Figure 4.15 compares the hit-rate of the standard Alloy cache to the hit-rates of a 2-way, 4-way, and 8-way associative cache along with the priority-based Alloy cache. Naturally, the hit-rate goes up slightly when increasing the associativity but

with diminishing returns. However, the hit-rate for our priority-based cache is lower than the associative. While hit-rate is not the only performance metric (RPS would be better), it is still a gauge that suggests that associativity is better. Consequently, we add priority to the associative cache to see if we can still extract gains with our priority-based caching. Figure 4.16 shows that there is indeed an improvement in hit-rate when using priority with associativity. Thus, we can conclude that our priority-based caching scheme is indeed beneficial. Priority is more beneficial at lower capacities mainly because there is more demand/contention. The benefit is also more prevalent at lower associativities due to priority-based caching eliminating some conflict misses from non-priority lines.

```
1  // runs on every access to the DRAM Cache
2  if ( hit )
3      if ( access.isPriority && pBenefit[set] )
4          pBenefit[set] = false
5          count++
6  else  // miss
7      if ( !access.isPriority && cache[set].isPriority )  // no resulting eviction
8          pBenefit[set] = true
9      else
10         pBenefit[set] = false
```

Figure 4.13: Priority Pseudo Code

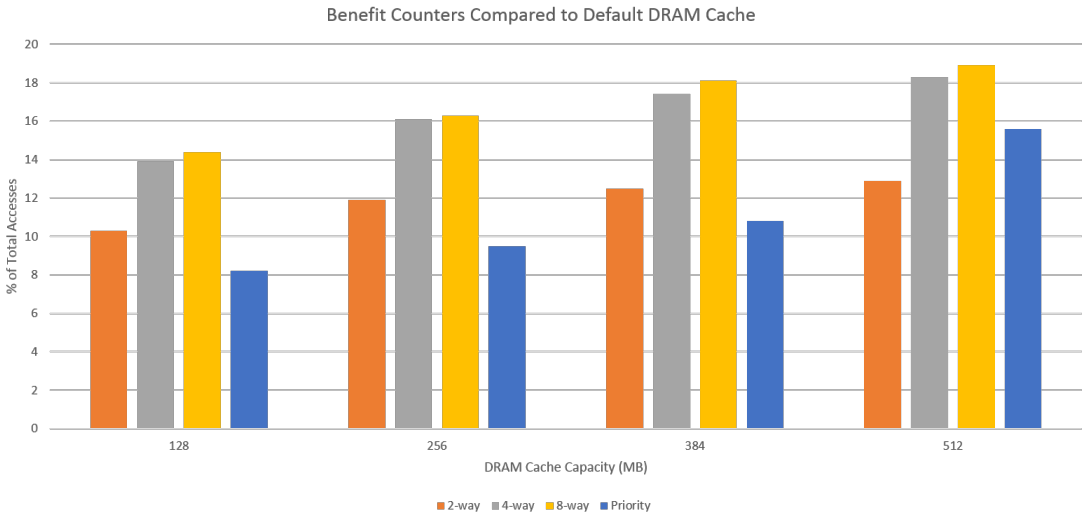

Figure 4.14: DRAM cache benefit counters as explained in Section 4.3.2.2

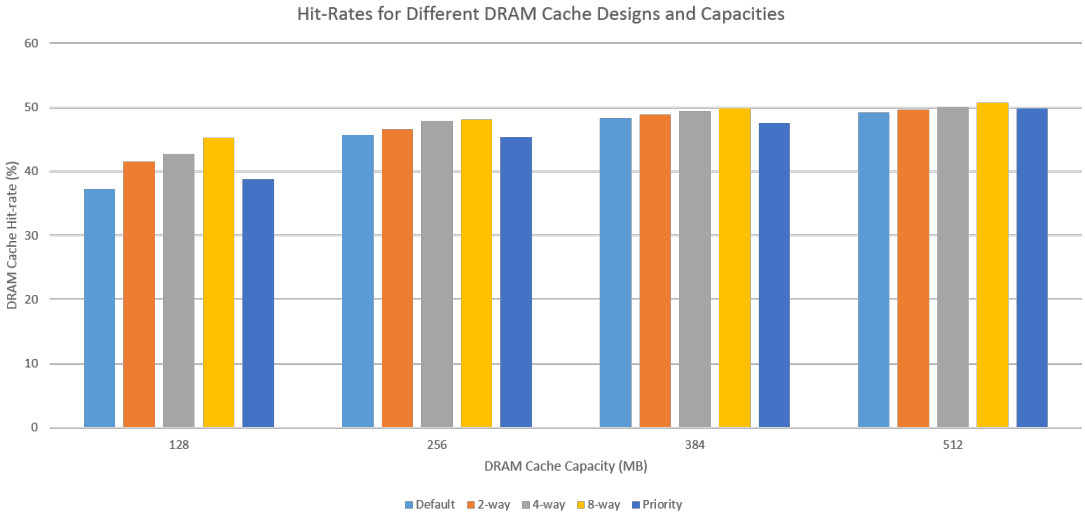

Figure 4.15: DRAM cache hit-rates for various configurations

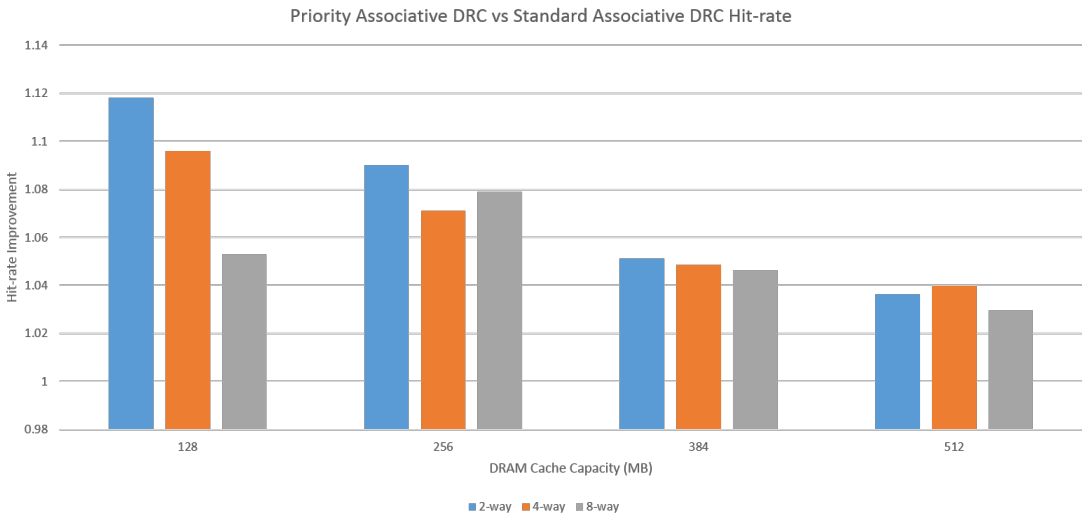

Figure 4.16: DRAM cache hit-rates - Priority Associative vs Standard Associative

Chapter 5
Conclusion

In this work, we described the importance of in-memory key-value stores and how crucial they are to meeting strict performance requirements outlined in SLAs. As demand for web services increases, keeping a strict QoS level will become increasingly difficult. To cope with this problem, we propose running the key-value stores on a modified stacked DRAM architecture. We start out with the standard Alloy cache implementation and show that it provides as much as a 32% improvement in RPS for Redis. Next, we introduce our priority-based Alloy cache which is able to provide as much as a 9% improvement in RPS over the standard Alloy cache. While the improvement in RPS is not significant, the decrease in tail latency is. Our priority-based Alloy cache decreases the 95th percentile request latency by as much as 42.4%. It also improves the tail latency for all value sizes, not just the large ones. We then compare the hit-rate of our priority-based Alloy cache to one that has an associativity and realize that simply adding associativty provides just as much, if not more, improvement in hit-rate. Consequently, we add our priority scheme to an associative Alloy cache and realize that there is still a benefit in using our scheme. The key contribution of our work, however, is our significant improvement in latency for the slowest 10% of requests.

Appendix A
Key-Value Stores

A.1 Introduction

Key-value stores are widely used throughout industry in order to improve performance of many web applications. For example, popular websites such as Facebook and Twitter heavily rely on key-value stores to provide an acceptable user experience.

A.1.1 Overview

The basic functionality of a key-value store system is based on the hash table data structure. Entries are stored in a key-value manner where keys are mapped to values via a hash function. The goal of the hash function is to compute a unique index based on the key and then the computed index is used to retrieve the value. A collision may occur when distinct keys are mapped to the same value. There are various methods to handle collisions, but their particulars are beyond the scope of this paper.

We use a simple example to explain basic key-value store operations. When a Twitter user posts a tweet, a unique ID is generated which corresponds to his/her tweet. The tweet is then saved in a key-value database. The unique ID serves as the key and the tweet itself is the value.

To improve the latency of data retrieval, key-value stores are designed to operate in main memory as opposed to disk. Main memory provides access speeds which are magnitudes faster than disk when performing random accesses [36]. Due to the nature of hash tables (explained above), it is expected that hash table accesses will not perform many sequential memory reads/writes, especially when the values are small. For example, when accessing keys at random whose values are just a few tens of bytes (which is often the case; see next section), a separate cache line (memory block) will likely have to be fetched for each. As a result, using disk as the only storage medium for a key-value database would have unacceptable access latency.

Ideally, we would like to store our entire database in main memory. Unfortunately, it would be virtually impossible to do so due to the massive sizes of these databases. As a simple example, as of March 31, 2015 Facebook reported having 1.44 billion monthly active users [1]. In 2013, Facebook reported an average of 300 million photo uploads per day [37]. If we roughly estimate

each photo to be about 1MB in size, that amounts to 300 terabytes per day–just for photos. Having enough main memory to support such volumes of data would be extremely expensive not only due to purchasing the hardware itself, but also keeping it all powered on simultaneously. Main memory is primarily composed of DRAM which is a volatile memory and consequently requires a constant power supply to retain its contents. Consequently, a nonvolatile disk storage must still be used to serve as a persistent backing store.

Due to the reasons explained in the previous paragraph, key-value stores are primarily used as a caching layer above a database stored on disks. These caches are optimized for high hit-rates and low latency. Hit-rates are typically above 90% [2] [6]. The replacement policy used by the caches tends to be Least Recently Used (LRU). In other words, when the cache fills up to capacity, the oldest blocks are evicted first.

A.1.2 Implementations

Two of the most popular key-value store implementations in industry are Memcached and Redis [4]. Since we chose to use Redis in this study, we will now describe its main functionality and features. Redis is an open-source key-value cache and store and is currently used by companies such as Twitter, GitHub, Pinterest, Snapchat, and many others. Unlike Memcached, the datatypes are not limited to just strings; they also include hashes, lists, sets, sorted sets, bitmaps, and hyperloglogs. The maximum allowed key or value size is 512MB which is more than enough for most applications. Redis is single threaded and therefore a single instance does not benefit from multiple cores. It does, however, benefit from fast CPUs and large on-chip caches since all instructions are executed by a single thread. In general, however, the CPU is not the bottleneck since key-value stores spend most of the time reading/writing to/from memory and sending/receiving data over the network.

A.1.2.1 Partitioning

Redis stores all entries in memory, so therefore the maximum capacity a Redis instance can store is limited by the main memory capacity of the machine it is running on. As explained earlier, the database sizes demanded by popular applications such as Facebook are enormous. In order for a cache to hold a large enough subset of this data and still incur a high hit-rate, the cache must be able to scale. To solve this, Redis supports partitioning the cached data among separate Redis instances. In other words, each instance contains a subset of the data and the memories of multiple computers are essentially shared.

Partitioning can be implemented at different levels of the software stack. At the client level, the clients themselves can keep track of which keys are stored in which Redis partitions/nodes. Partitioning can also be done further down the stack by using a proxy to route the client's requests to the correct node. In this case, the proxy keeps track of the client-to-node mapping instead of the clients themselves. Finally, the clients can naively send their requests to a random Redis node which will then forward the request to the correct node, unless it luckily ended up being the right one. The Redis developers recommend using Redis Cluster in order to manage

partitioning. With Redis Cluster, the clients send a small query to a random node to see if it is the correct one. If not, the node responds with the address of the correct node as opposed to forwarding the request. The client then sends the actual queries once it knows the right node.

A.1.2.2 Eviction Policies

As described earlier, when a key-value store is used as a cache, it typically employs the LRU eviction policy. Redis supports LRU eviction, along with a few other eviction policies. However, the implementation of the LRU policy is actually an approximation of a true LRU policy for performance reasons. Instead of checking the last accessed time of every single entry in the cache, Redis randomly picks a subset of the entries. The size of this sampled subset is configurable. Simulations conducted by the Redis community show that the difference between true LRU and Redis approximation are minimal or non-existent [3]. Other eviction policies employ techniques such as setting a time to live (TTL) for each entry and evicting it once expired, randomly selecting an entry to evict, or not evicting any entries and instead returning errors when the cache is full.

A.1.3 Communication Protocol

The Redis developers have created a simple-to-use communication protocol named Redis Serialization Protocol (RESP). In order for a client to communicate with a Redis server, it must send commands that are formatted according to RESP. Each command is formatted as a RESP Array. The carriage return and line feed (CRLF) sequence serves as the delimiter that separates each element (and element metadata, such as length of the element) in the array. Figure A.1 shows an example Redis command and its corresponding RESP representation. The first byte of a RESP Array is always the star (*) character which is then followed by the length (number of elements) of the array. In the case of the SET command illustrated in Figure A.1, there are three elements: set, key, and value. Each element is also preceded by its length in terms of number of bytes (using the $ character). Going back to Figure A.1, set and key have a length of three, while value has a length of five. The reply from the server can be in the form of RESP Array again, but it can also be of type error, integer, or simple string.

SET key value → *3\r\n$3\r\n**SET**\r\n$3\r\n**key**\r\n$5\r\n**value**\r\n

Figure A.1: Example of the Redis Serialization Protocol (RESP)

A.1.4 Mass Insertion

As described in Chapter 4, we attempt to model a realistic workload based on publicly available metrics from large-scale production key-value stores. We therefore aim at a GET/SET request ratio of 30:1 as was observed in Facebook's Memcached trace [6]. Consequently, when we run our simulations and gather statistics on how well our system performs, we have to assume that the Redis database is already populated with data. Otherwise the majority of GET requests will result in misses which would not test the performance of the system Redis is running on, and

especially not the memory hierarchy. However, it would take a significant amount of simulation time to load a large amount of data via the standard way of client/server communication. While we do checkpoint the database state after it is populated, applying additional changes to the GEM5 simulator and/or Redis would often require redoing the pre-load process.

To help alleviate this issue, we utilize Redis' mass insertion feature. Instead of the server sending a reply to every (SET) operation requested by the client, the server waits until it has received all the pre-load requests and then sends a single reply acknowledging the number of requests it received successfully. To speed up insertion even more, we perform it locally using TCP loopback and thus avoid extra networking delay.

Another issue we faced with mass insertion was when performing large valued sets, i.e. inserting values of 1MB. As outlined in Chapter 4, we make a total of 20,000 insertions per Redis instance. Of those 20,000 requests, 10,000 are 10B, 9,000 are 300B, and 1,000 are 1MB.

Appendix B
Key-Value Store Simulation

B.1 Introduction

Computer architecture researchers need a way to test the performance of new system-level and micro-architecture level designs without incurring much cost in terms of time and money. In other words, it would be too expensive and time-consuming to build a physical prototype every time a new design is proposed. An efficient and common solution is to use an architectural simulator created purely in software. The behavior of all the system components is imitated, such as CPU/GPU cores, caches, buses, devices, etc. For example, if an architect wants to change a CPU's instruction fetch width, he/she can simply do so by tweaking a few simulator options. The way a hardware cache manages and allocates new cache blocks can be tweaked as well by modifying the code that imitates those components. Benchmark applications can then be executed on the simulator to analyze and compare the performance of the new/tweaked design. Inevitably, a downside to using software simulation to evaluate new designs is the execution speed. In real hardware, the speed of light is the limiting factor in terms of how fast things can happen, whereas in software everything is much slower. To be more concrete, simulating a few seconds of real-time execution may take tens of hours to simulate depending on the level of accuracy we imitate the system at. Simulators usually allow the architect/user to trade off accuracy and simulation speed across the various components.

B.2 The GEM5 Simulator

This project uses GEM5 [34], which is an open-source architectural simulator developed by researchers across industry and academia. More specifically, we use GEM5's Full System simulation mode which allows Linux to be booted and benchmark applications to be executed on it. The Full System mode supports three Instruction Set Architectures (ISAs): ARM, ALPHA, and x86. We chose the ARMv8 64-bit implementation since it is currently one of the most actively developed ones in the GEM5 community. This resulted in being able to resolve issues with a faster response time from other users and developers.

GEM5 was the simulator of choice mainly due to its popularity among other architecture researchers at Penn State, hence making the learning curve slightly easier. However, while GEM5 does have an active community and supports many features, there are a number of other alternatives such as Sniper, Mars, Flexus, etc. Many of them are much faster in terms of simulation time, while providing a similar level of accuracy. Unfortunately, once an architect invests time to learn a simulator and develop his/her infrastructure on it, it can be costly to switch.

B.2.1 Pseudo Instructions

Simulators often allow a user to create his/her own magic instructions, which in GEM5 are referred to as pseudo instructions. The pseudo instruction has its own opcode just like any other instruction in the ISA and can take upto two arguments. The instruction can then be compiled into the kernel and/or a benchmark application and be used to directly communicate with the simulator. For example, we use a pseudo instruction in order to figure out which process is currently running on a particular CPU. We modified the Linux kernel source code by having it call the GEM5 pseudo instruction whenever a context switch occurs. More specifically, if the context switch involves a process we are interested in (i.e. our benchmark), we call the pseudo instruction with a parameter that includes the process name, whether or not it was put on/off the CPU, and which CPU it occurred on. Inside the simulator, we maintain a state of the current mapping of our benchmarks to CPUs. This can provide useful information, such as how many instructions are executed per application as opposed to per CPU (the default GEM5 statistic).

We also use a pseudo instruction to communicate the addresses of the large values stored in Redis (the reason for keeping track of larges values is outlined in Section 3.2; here we describe the low-level implementation details). To accomplish this, we first modify Redis and compile the pseudo instruction into Redis. The pseudo instruction is named m5addAddr and takes a 64-bit address as an argument. When Redis stores a large value (i.e. 1MB or larger), we save the address returned by malloc and call the m5addAddr instruction with that address. The CPU that Redis is executing on then decodes that instruction and calls our function that marks that address as special.

The address that is returned by malloc and then communicated to GEM5 is a virtual address. In other words, that address is only accurate in the context of that particular Redis process/instance. From the memory's point of view, however, it only sees and deals with physical addresses that were returned by the page table. The mapping of virtual to physical addresses can change during runtime which means that a virtual address will not necessarily correspond to a constant physical address. In addition, memory allocated in the virtual address space is contiguous. For example, if we allocate memory for a 1MB block, the subsequent 999,999 addresses (byte granularity) following the address returned by malloc will correspond to the 1MB block. This is not the case in physical memory. Due to these reasons, we must store virtual addresses along with the particular process they belong to when keeping track of the addresses of large values. We explain how we efficiently store and keep track of these addresses in the following paragraph.

A naive implementation of keeping track of large values is to store each address that corre-

sponds to the value in a hash table (key: address, value: present or not). That way, when a load instruction contains an address corresponding to any byte, we can directly verify if it is within the address range of a 1MB value. While that is a possible solution, it does not scale well if we are keeping track of many large values. More specifically, the space complexity is O(M*N) where M corresponds to the size of the value (i.e. how many addresses we are storing per value, such as 1 million for a 1MB value) and N is the number of values we are keeping track of. Instead, we implement a much more space efficient approach. We still use a hash table in the same way, but instead use 1MB aligned addresses as the key and a list containing the precise starting addresses of 1MB blocks as the value. When Redis calls the m5addAddr pseudo instruction, the address will most likely not be 1MB aligned. We therefore calculate the starting address of the 1MB region that corresponds to the address supplied by Redis. The simplified pseudo code is shown in B.1.

```
1  // VAddr: The virtual address coming from malloc in Redis
2  // specAddr: A list of hash tables, one hash table per Redis instance
3  //           Each hash table is of type <address, list of addresses>
4  // PID: Unique process ID of Redis instance
5  //      mapped to range(0, total \# of Redis instances)
6
7  offset = VAddr mod 1000000
8  key = VAddr - offset
9  specAddr[PID][key].push(VAddr)
10 if ( offset > 0 )  // if VAddr is not 1MB aligned
11     key2 = key + 1000000
12     specAddr[PID][key2].push(VAddr)
```

Figure B.1: Pseudo code for keeping track of addresses.

As can be seen in the pseudo code, we append the virtual address communicated from Redis to the list that corresponds to the computed key. If the 1MB value is not aligned (the common case), we also do the same for key+1000000. This concept is visualized in B.2. Addr0, Addr1, and Addr2 represent 1MB aligned addresses in the virtual address space and AddrX and AddrY represent the starting addresses of two 1MB values supplied by Redis. In this case, we would append AddrX to the list of addresses corresponding to the hash tables of Addr0 and Addr1. We would then do the same for AddrY, but to the hash tables for Addr1 and Addr2.

When a load instruction is eventually encountered, we would compute the key as in the pseudo code above and then simply check if the address in the load instruction is present. A 1MB block of virtual memory can contain at most (parts of) two distinct 1MB values. Consequently, we can say that the lookup time is constant since we are computing an index using a hash function and then traversing a list of length two or less. The space complexity is now O(2*N) since we will at most store two addresses per 1MB block.

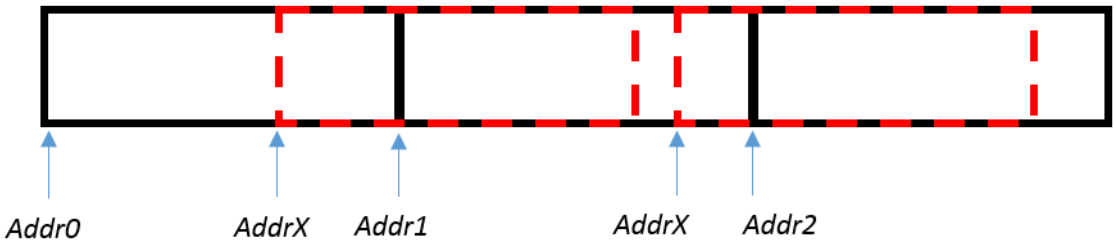

Figure B.2: Virtual Address Space. Addr0, Addr1, and Addr2 represent 1MB aligned addresses in the virtual address space and AddrX and AddrY represent the starting addresses of two 1MB values supplied by Redis.

Bibliography

[1] "Facebook Company Info," http://newsroom.fb.com/company-info/, accessed: 2015-06-17.

[2] Fitzpatrick, B. (2004) "Distributed Caching with Memcached," *Linux J.*, **2004**(124), pp. 5–.
URL http://dl.acm.org/citation.cfm?id=1012889.1012894

[3] "Redis," .
URL http://redis.io/

[4] "Techstacks," http://www.techstacks.io, accessed: 2015-07-19.

[5] "Netflix Tech Blog,"
http://techblog.netflix.com/2012/01/ephemeral-volatile-caching-in-cloud.html, accessed: 2015-06-23.

[6] Atikoglu, B., Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny (2012) "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, ACM, New York, NY, USA, pp. 53–64.
URL http://doi.acm.org/10.1145/2254756.2254766

[7] Intel (2013), "Configuration and Deployment Guide For Memcached on Intel Architecture,"
https://software.intel.com/sites/default/files/article/402153/configuration-and-deployment-guide-for-memcached-on-intel.pdf.

[8] "Facebook's next-gen data center network debuts in Altoona, Iowa,"
http://www.techrepublic.com/article/facebooks-next-gen-data-center-network-debuts-in-altoona-iowa/, accessed: 2015-07-19.

[9] Loh, G. H. (2008) "3D-Stacked Memory Architectures for Multi-core Processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, IEEE Computer Society, Washington, DC, USA, pp. 453–464.
URL http://dx.doi.org/10.1109/ISCA.2008.15

[10] Black, B., M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb (2006) "Die Stacking (3D) Microarchitecture," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, IEEE Computer Society, Washington, DC, USA, pp. 469–479.
URL http://dx.doi.org/10.1109/MICRO.2006.18

[11] Kgil, T., S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner (2006) "PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor," *ACM SIGARCH Computer Architecture News*, **34**(5), pp. 117–128.

[12] Qureshi, M. K. and G. H. Loh (2012) "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, IEEE Computer Society, Washington, DC, USA, pp. 235–246.
URL http://dx.doi.org/10.1109/MICRO.2012.30

[13] Jevdjic, D., S. Volos, and B. Falsafi (2013) "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, New York, NY, USA, pp. 404–415.
URL http://doi.acm.org/10.1145/2485922.2485957

[14] Jevdjic, D., G. H. Loh, C. Kaynak, and B. Falsafi (2014) "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, IEEE Computer Society, Washington, DC, USA, pp. 25–37.
URL http://dx.doi.org/10.1109/MICRO.2014.51

[15] Hameed, F., L. Bauer, and J. Henkel (2014) "Reducing Latency in an SRAM/DRAM Cache Hierarchy via a Novel Tag-Cache Architecture," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, ACM, New York, NY, USA, pp. 37:1–37:6.
URL http://doi.acm.org/10.1145/2593069.2593197

[16] Loh, G. H., N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung (2012) "Challenges in heterogeneous die-stacked and off-chip memory systems," in *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*.

[17] Chou, C., A. Jaleel, and M. K. Qureshi (2014) "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, IEEE Computer Society, Washington, DC, USA, pp. 1–12.
URL http://dx.doi.org/10.1109/MICRO.2014.63

[18] Woo, D. H., N. H. Seong, D. Lewis, and H.-H. Lee (2010) "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12.

[19] "International Technology Roadmap for Semiconductors, 2013," http://www.itrs.net/.

[20] "Hybrid Memory Cube," http://www.hybridmemorycube.org/technology.html, accessed: 2015-06-23.

[21] "AMD Press Release,"
http://www.amd.com/en-us/press-releases/Pages/new-era-pc-gaming-2015jun16.aspx, accessed: 2015-06-23.

[22] Wulf, W. A. and S. A. McKee (1995) "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, **23**(1), pp. 20–24.
URL http://doi.acm.org/10.1145/216585.216588

[23] McKee, S. A. (2004) "Reflections on the Memory Wall," in *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, ACM, New York, NY, USA, pp. 162–.
URL http://doi.acm.org/10.1145/977091.977115

[24] Nowatzyk, A., F. Pong, and A. Saulsbury (1996) "Missing the memory wall: The case for processor/memory integration," in *Computer Architecture, 1996 23rd Annual International Symposium on*, IEEE, pp. 90–90.

[25] Cunha, C., A. Bestavros, and M. Crovella (1995) *Characteristics of WWW Client-based Traces*, *Tech. rep.*, Boston, MA, USA.

[26] Özer, E., K. Flautner, S. Idgunji, A. Saidi, Y. Sazeides, B. Ahsan, C. Nicopoulos, I. Sideris, A. Adileh, M. Ferdman, et al. (2010) "EuroCloud: energy-conscious 3D server-on-chip for green cloud services," in *Workshop on Architectural Concerns in Large Datacenters in conjunction with ISCA*, vol. 10.

[27] Ferdman, M., A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi (2012) "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, ACM, New York, NY, USA, pp. 37–48.
URL http://doi.acm.org/10.1145/2150976.2150982

[28] Gutierrez, A., M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge (2014) "Integrated 3D-stacked server designs for increasing physical density of key-value stores," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, ACM, pp. 485–498.

[29] Basu, A., N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez (2007) "Scavenger: A New Last Level Cache Architecture with Global Block Priority," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, IEEE Computer Society, Washington, DC, USA, pp. 421–432.
URL http://dx.doi.org/10.1109/MICRO.2007.36

[30] Nishtala, R., H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani (2013) "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, USENIX Association, Berkeley, CA, USA, pp. 385–398.
URL http://dl.acm.org/citation.cfm?id=2482626.2482663

[31] "SPEC CPU2006," .
URL https://www.spec.org/cpu2006/

[32] Jaleel, A. (2007) "Memory characterization of workloads using instrumentation-driven simulation–a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites," *Intel Corporation, VSSAD*.

[33] Urdaneta, G., G. Pierre, and M. van Steen (2009) "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, **53**(11), pp. 1830–1845.
URL http://www.globule.org/publi/WWADH_comnet2009.html

[34] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib,

N. Vaish, M. D. Hill, and D. A. Wood (2011) "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, **39**(2), pp. 1–7.
URL http://doi.acm.org/10.1145/2024716.2024718

[35] Poremba, M. and Y. Xie (2012) "NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories," in *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pp. 392–397.

[36] Jacob, B., S. Ng, and D. Wang (2010) *Memory systems: cache, DRAM, disk*, Morgan Kaufmann.

[37] "The Top 20 Valuable Facebook Statistics,"
https://zephoria.com/top-15-valuable-facebook-statistics/, accessed: 2015-06-17.