Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached

Kevin Lim
HP Labs
kevin.lim@hp.com

David Meisner
Facebook
meisner@fb.com

Ali G. Saidi
ARM R&D
ali.saidi@arm.com

Parthasarathy Ranganathan
HP Labs
partha.ranganathan@hp.com

Thomas F. Wenisch
EECS, Univ. of Michigan
twenisch@eecs.umich.edu

ABSTRACT

Distributed in-memory key-value stores, such as `memcached`, are central to the scalability of modern internet services. Current deployments use commodity servers with high-end processors. However, given the cost-sensitivity of internet services and the recent proliferation of volume low-power System-on-Chip (SoC) designs, we see an opportunity for alternative architectures. We undertake a detailed characterization of `memcached` to reveal performance and power inefficiencies. Our study considers both high-performance and low-power CPUs and NICs across a variety of carefully-designed benchmarks that exercise the range of `memcached` behavior. We discover that, regardless of CPU microarchitecture, `memcached` execution is remarkably inefficient, saturating neither network links nor available memory bandwidth. Instead, we find performance is typically limited by the per-packet processing overheads in the NIC and OS kernel—long code paths limit CPU performance due to poor branch predictability and instruction fetch bottlenecks.

Our insights suggest that neither high-performance nor low-power cores provide a satisfactory power-performance trade-off, and point to a need for tighter integration of the network interface. Hence, we argue for an alternate architecture—Thin Servers with Smart Pipes (TSSP)—for cost-effective high-performance `memcached` deployment. TSSP couples an embedded-class low-power core to a `memcached` accelerator that can process GET requests entirely in hardware, offloading both network handling and data look up. We demonstrate the potential benefits of our TSSP architecture through an FPGA prototyping platform, and show the potential for a 6X-16X power-performance improvement over conventional server baselines.

1. INTRODUCTION

Internet services are increasingly relying on software architectures that enable rapid scale-out over clusters of thousands of servers to manage rapid growth. As the volume of data that must be processed at interactive speeds increases, only such scale-out architectures can maintain performance and availability with sustainable costs in light of hardware failures. Due to their large scale, efficiency is of particular concern for numerous services (e.g., web search, social media, video sharing, web email, collaborative editing, and social games).

Distributed in-memory key-value stores, such as `memcached`, have become a central piece of infrastructure to allow online services to scale, with some services relying on thousands of `memcached` servers (e.g., Facebook [27], Zynga, Twitter, YouTube). Today, operators use the same commodity high-end servers for their `memcached` clusters as for other aspects of their software infrastructure. However, in light of recent trends enabling cost-effective volume low-power System-on-Chip (SoC) designs and several prior studies advocating embedded-class CPUs in the data center [5, 24, 31], we perceive an opportunity to consider new architectures for `memcached`.

To discover performance- and power-efficiency bottlenecks, we undertake a detailed architectural characterization of `memcached`. Our study considers both high-performance and low-power CPUs and network interfaces (NICs), and measures a variety of carefully-designed benchmarks that explore the range of `memcached` behavior. We develop a load-testing methodology and infrastructure to allow us to reproduce precisely-controlled object size, popularity, and load distributions to mimic the traffic a `memcached` server receives from a large client cluster.

Our characterization paints a frustrating picture—neither high-performance (Xeon-class) nor current low-power (Atom-class) multi-core systems provide appealing cost-performance scaling trends. We discover that, regardless of CPU microarchitecture, `memcached` execution is remarkably inefficient, saturating neither network links nor available memory bandwidth (we estimate that more than 64 Xeon-class cores are needed to saturate a 10Gb Ethernet link assuming perfect multicore software scalability). Instead, we find performance is typically limited by the per-packet processing overheads in the NIC and OS kernel. Front-end (branch prediction and fetch) stalls are a key performance bottleneck across CPU microarchitectures. Despite the small codebase of `memcached` itself, frequent trips into the TCP/IP stack, kernel, and library code result in poor instruction supply due to ICache misses, virtual memory (VM) translation overheads and poor branch predictability. ICache and ITLB performance are often an order of magnitude worse relative to benchmarks commonly used for microarchitecture design (e.g., SPEC). Furthermore, large last-level caches seem to provide little benefit. Conversely, we find that advanced NIC features that optimize packet hand-off from

NIC to CPU, such as segment offload and receiver-side scaling, are crucial to achieving high throughput.

Our measurements suggest that Xeon CPUs do not provide a sufficient performance advantage to justify their cost-premium, and yet, Atom cores cannot achieve high throughput or low latency under load. The results also point to a need for tighter integration of the network interface. Hence, we argue for a new architecture—Thin Servers with Smart Pipes (TSSP)—for power-efficient high-performance memcached deployment. TSSP seeks to maintain the energy-efficiency advantage of low-power cores while mitigating their poor performance when handling networking operations under high load. To this end, TSSP couples an embedded-class low-power core to an accelerator for networking operations that can process memcached GET requests entirely in hardware in an integrated SoC.

We evaluate the feasibility and potential of such a GET accelerator using an FPGA prototyping platform [14]. Our evaluation using this prototype demonstrates that a TSSP implementation built using a low-power core and integrated reconfigurable logic (e.g., as in the Xilinx Zynq platform) can exceed the performance-per-watt of conventional baseline servers by 6X-16X; implementation in a custom ASIC will likely enable additional efficiency gains.

The rest of this paper is organized as follows. In Section 2, we provide brief background on memcached. We describe our load testing framework in Section 3. We delve into individual performance bottlenecks in Section 4. Based on our insights, in Section 5, we propose an SoC memcached accelerator design and evaluate its performance and energy-efficiency advantages. In Section 6 we cover related work, and in Section 7 we conclude.

2. BACKGROUND

We first briefly describe the operation of memcached clusters and their importance in large-scale internet services. Many internet workloads leverage memcached as a caching layer between a web server tier and back-end databases. Memcached servers are high-performance, distributed key-value stores, with objects entirely stored in DRAM. Under normal usage, each memcached server is a best-effort store, and will discard objects under an LRU replacement scheme as main memory capacity is exhausted. Keys are unique strings and may be at most 250 bytes in length, while the stored values are opaque data objects and must be 1 MB or smaller. Memcached is implemented using a hash table, with the key used to index into the table, and the value (along with other bookkeeping information) stored as an object in the table.

The memcached interface comprises a small number of simple key-value operations, the most important being GET and SET. Memcached is attractive because the server interface is general—data is simply stored and retrieved according to a single key, a model which can be applied to nearly any workload. For example, responses to database queries can be cached by using the SQL query string as a key and the response as the value. Clients access data stored in the memcached cluster over the network, maintaining an independent connection to each server. Typically, clients use *consistent hashing* (similar to the method in [15]), providing a key-to-node hash that load-balances and can adapt to node insertions and deletions while ensuring that each key maps to only a single server at any time.

Moreover, memcached is easy to scale from a cluster perspective. Memcached servers themselves do not directly interact, nor do they require centralized coordination. If either throughput or capacity demands grow, a pool can be scaled simply by directing clients to connect to additional servers; consistent hashing mechanisms immediately map a fraction of keys to the new servers. Operationally, it is useful that key-value throughput can be scaled independent of the back-end storage system (database servers are typically considerably more expensive than memcached servers).

As the demands of internet service workloads continue to grow, so too do the performance requirements of memcached caching layers. For companies such as Facebook, not only has the number of users grown exponentially, but so too has the amount of data required to process each user request. This increasing processing requirement is inexorably linked to the energy efficiency of the underlying hardware. Data centers are typically limited by peak power delivery (i.e, once installed, a data center can only deliver a fixed number of mega-watts). Therefore, the throughput per watt of the caching layer is a fundamental limitation of the memcached installation and a large improvement in energy efficiency translates to an opportunity to improve performance through greater scale-out within the data center power budget.

3. METHODOLOGY

Our goal is to perform an in-depth architectural characterization of the performance of existing servers to identify any inefficiencies and bottlenecks when running memcached. In this section we discuss our methodology for evaluating server hardware by load-testing memcached. Although the API is simple, from a performance standpoint the memcached workload is more complex than it may appear. Previous studies of memcached have used simple load testing tools, such as memblaster or memslap, which do not attempt to reproduce the statistical characteristics of live traffic. As our results will demonstrate, the manner in which memcached is loaded drastically alters its behavior and microarchitectural bottlenecks. In particular, we find that the object size distribution has a large impact on system behavior. The existing load testing tools use either a fixed or uniform object size distribution and a uniform popularity distribution, which yield misleading conclusions.

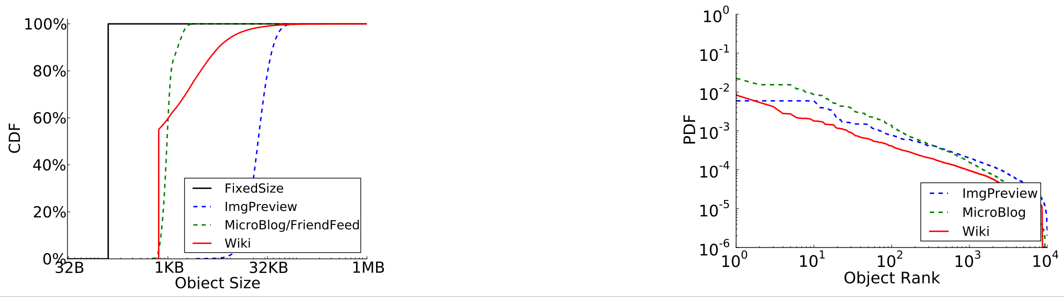
To this end, we develop a stress-testing framework that emulates the traffic that a large cluster of client machines offer to a memcached server, using publicly available data and popularity statistics that approximate typical memcached use cases. Our workloads exhibit similar statistical properties as recently reported for several production clusters at Facebook [6].

3.1 Workloads

Memcached behavior can vary considerably based on the size and access frequency of the objects it stores (the actual values are opaque byte strings and do not affect behavior). Hence, we have designed a suite of five memcached workloads that capture a wide range of behavior. Our goal is to create a set of easy-to-understand yet realistic micro-benchmarks that expose memcached performance sensitivity. We refer interested readers to [6] for statistical analyses of live memcached traffic.

Each of our workloads is defined by an object size distribution, popularity distribution, fraction of set requests, and whether or not MULTI-GET requests are used. (MULTI-GETs aggregate numerous GET requests into a single request-and-response exchange to reduce networking overheads.) Summary statistics for each of our workloads are provided in Figure 1. The measured popularity distributions are shown in Figure 1 (probability as a function of rank). The zipf-like distribution is consistent with a previous study by Cha et al [13].

FixedSize. Our simplest workload uses a fixed object size of 128 B and uniform popularity distribution. We include this workload



Name	Single/Multi-get	% Writes	Object Size				Type
			Avg.	Std. Dev.	Min	Max	
FixedSize	Single-get	0%	128 B	0 B	128 B	128 B	Plain Text
MicroBlog	Single-get	20%	1 KB	0.26 KB	0.56 KB	2.7 KB	Plain Text
Wiki	Single-get	1%	2.8 KB	10.4 KB	0.30 KB	1017 KB	HTML
ImgPreview	Single-get	0%	25 KB	12.4 KB	4 KB	908 KB	JPEG Images
FriendFeed	Multi-get	5%	1 KB	0.26 KB	0.56 KB	2.7 KB	Plain Text

Figure 1: **Workload Characteristics.** We construct a set of workloads to expose the broad range of memcached behavior. The object size distributions of the workloads differ substantially. Our measurements show that popularity for most web objects follows a zipf-like distribution, although the exact slope may vary.

because small objects place the greatest stress on memcached performance; anecdotal evidence suggests production clusters frequently cache numerous small objects [1].

MicroBlog. This workload represents queries for short snippets of text (e.g., user status updates). We base the object size and popularity distribution on a sample of "tweets" (brief messages shared between Twitter users) collected from Twitter. The text of a tweet is restricted to 140 characters, however, associated meta-data brings the average object size to 1KB with little variance. Even the largest tweet objects are under 2.5KB in size.

Wiki. This workload comprises a snapshot of articles from Wikipedia.org. We use the entire Wikipedia database, which has over 10 million entries. Each object represents an individual article in HTML format. Articles are relatively small, 2.8 KB on average, but have a notable variance because some articles have significantly more text. This workload exhibits the highest normalized object size variance (relative to the mean) of any of our workloads. Object popularity is derived from the page view count for each article.

ImgPreview. This workload represents photo objects used in photo-sharing sites. We collect a sample of 873,000 photos and associated view counts from Flickr. In particular, Flickr provides a set of "interesting" photos every day; we collect these photos because they are likely to be accessed frequently. Photo sharing sites often offer the same images in multiple resolutions. In a typical web interaction, a user will view many low-resolution thumbnails before accessing a high-resolution image. We collect a sample of these thumbnails, which are on average 25 KB in size. We focus on thumbnails because larger files (e.g., high-resolution images) are typically served from data stores other than memcached (e.g., Facebook's Haystack system [8]).

FriendFeed. We construct our final workload to understand the implications of heavy use of MULTI-GET requests. Facebook has disclosed that MULTI-GETs play a central role in its use of memcached [16]. This workload seeks to emulate the requests required to construct a user's Facebook Wall, a list of a user's friends' most recent posts and activity. We reuse the distributions from our MicroBlog workload, as Facebook status updates and tweets have similar size and popularity characteristics [6].

	Xeon	**Atom**
Processor	1x 2.25 GHz 6-core Xeon Westmere L5640 12 MB L3 Cache	1.6 Ghz Atom D510 Dual-core 2x SMT 1 MB L2 Cache
DRAM	3x 4GB DDR3-1066	2x 2GB DDR2-800 SDRAM
Commodity	Realtek RTL8111D Gigabit	–
Enterprise	Broadcom NetXtreme II Gigabit	Intel 82574L Gigabit
10GbE	Intel X520-T2 10GbE NIC	Intel X520-T2 10GbE NIC

Table 1: **Systems Under Test (SUT).**

3.2 Load Testing Framework

We next describe the load-testing infrastructure we developed to emulate load from a large client cluster. To fully saturate the target server, we employ several load generator clients that each emulate 100 memcached clients via separate TCP/IP connections. Throughout our experiments, we explore the tradeoff between server throughput and response time by varying the request injection rate of our load generator clients; the clients automatically tune their offered load as high as possible while still meeting a specified latency constraint.

The load-testing infrastructure first populates and then accesses the memcached server according to the specific workload's object size and popularity distributions. To maximize efficiency, we use memcached's binary network protocol. Furthermore, we disable Nagle's algorithm, which causes packet buffering, to minimize network latency [26].

The primary objective of our investigation is to determine which aspects of system architecture impact memcached performance. Accordingly, we intentionally decouple our study from network switch saturation, since many other studies have investigated efficient techniques to avoid switch over-subscription in data center networks [3, 4, 17]. All experiments use dedicated switches or cross-over links between the load generators and test servers.

3.3 Systems under test

We study two drastically different server systems, a high-end Xeon-based server and a low-power Atom-based server, and three different classes of network interface cards (NIC),

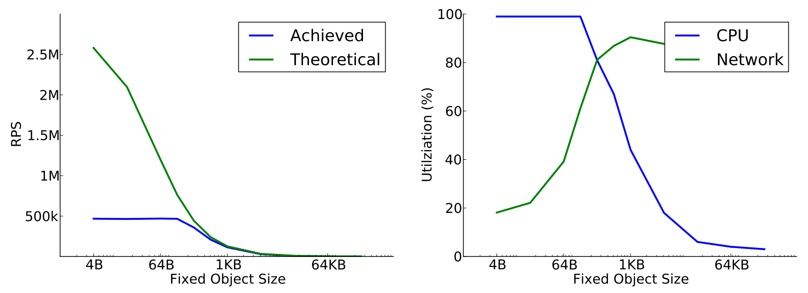


Figure 2: **Fixed object size microbenchmark.** `memcached`'s behavior varies greatly as a function of object size. For small objects, the workload is far from its theoretical performance (see left) because it is processing constrained (see right). Conversely, large objects are primarily network bandwidth bound and do not require much processing at all. Realistic workloads, however, have variation in object size and operate between these two extremes.

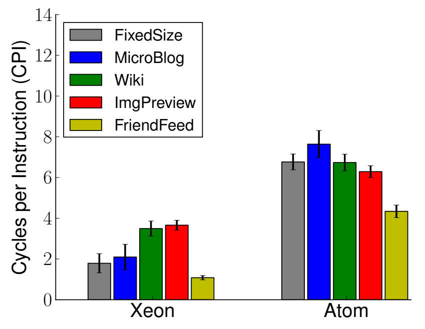


Figure 3: **Microarchitectural inefficiency with `Memcached`.** Modern processors exhibit unusually high CPI for Xeon and Atom-based servers. Xeon-class systems operate at less than one eighth of their theoretical instruction throughput. Atom-class systems fare even worse. Compared to other workloads (e.g., SPEC CPU), these CPIs are high and demonstrate that current microarchitectures are a poor match for `memcached`. We identify specific microarchitectural hurdles in 4.1.

spanning consumer-grade, manufacturer-installed, and high-end 10GbE NICs. The details of each system are shown in Table 1. By exchanging NICs and selectively disabling cores, we evaluate 21 different system configurations. Our goal is to cover a wide range of possible server `memcached` configurations to gain greater insight into the importance of the different microarchitectural bottlenecks.

The Xeon-class system is typical of the `memcached` servers described in media reports. Though our test system includes only 12GB of RAM (lower than is typically reported), we have confirmed that memory capacity has no direct effect on latency or throughput. (Note that memory capacity per node indirectly affects performance because the share of a cluster's overall load directed to a particular server is proportional to the server's share of the overall cluster memory capacity.) The Atom system represents a low-power alternative to improve energy efficiency and enable greater scale-out within a fixed data center power budget, as suggested in several recent studies [24, 31]. We study several NICs to demonstrate that the feature set of the NIC and driver, rather than the theoretical peak network bandwidth, primarily affect `memcached` performance, particularly for small object sizes.

All systems run Ubuntu 11.04 (2.6.38 kernel) with `memcached` 1.4.5. We gather utilization, response time, bandwidth, and microarchitectural statistics using our load generators, `sysstat`, and `perf`.

4. MEMCACHED BOTTLENECKS

To understand how to design optimized `memcached` systems, we begin with a simple microbenchmark to determine when `memcached` is network-bandwidth-limited and when CPU performance begins to matter. Figure 2 depicts this result. In the left sub-figure, we report the actual and theoretical (based on network bandwidth) requests-per-second for GET-requests to fixed-size objects. Below a critical size of about 1KB, an enormous gulf opens between theoretical and actual performance. In the right sub-figure, we illustrate the cause: the performance bottleneck shifts from the network to the CPU. In Figure 3, we see that the CPI for `memcached` is unusually high (relative to e.g., SPEC) and far away from the theoretical peak instruction throughput of either microarchitecture.

In the rest of this section, we explore the causes and implications of `memcached` microarchitectural bottlenecks. First, in Section 4.1, we explore why `memcached` exhibits the exceedingly poor CPIs seen in Figure 3. In Section 4.2, we explore the impact of the server's networking stack. Then, we examine the overall picture of power efficiency of Atom and Xeon systems to consider the hypothesis that low-power cores might enable better peak-power efficiency than servers built with high-performance processors in Section 4.3. Finally, we comment on the broader implications for `memcached` system architecture in Section 4.4.

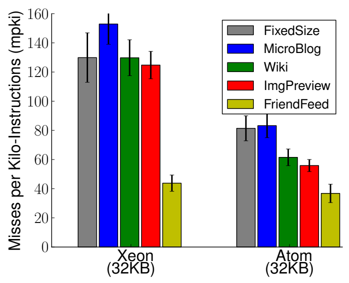
4.1 Microarchitecural Inefficiency

We begin by investigating what microarchitectural bottlenecks cause the poor CPI we observe in `memcached`. We gather performance counter data for a variety of microarchitectural structures on both Atom and Xeon cores. We report results with error bars indicating one standard deviation from the mean.

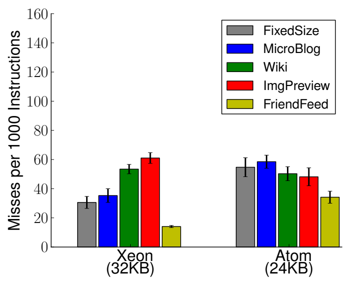
Broadly, our results suggest that current processors (whether Xeon-class or Atom-class) do not run `memcached` efficiently: Xeons achieve only one eighth and Atoms only one sixteenth of their theoretical peak instruction throughput. Prior to undertaking this microarchitectural study, our expectation was that `memcached` might be memory bandwidth bound, with performance limited primarily by the speed of copying data to outgoing network packets. In fact, measuring bandwidth to main memory, we find it is massively underutilized, always falling under 15% of max throughput and often much less (e.g., 5% for MicroBlog).

Surprisingly, the most significant bottlenecks lie in the processor front-end, with poor instruction cache and branch predictor performance. Neither increased memory bandwidth, nor larger data caches are likely to improve performance. Future architectures must address these bottlenecks to achieve near-wire-speed processing rates. We address caching, address translation, and branch prediction behavior in greater detail.

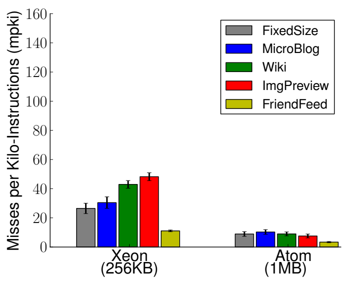
Caching bottlenecks. Figure 4 presents cache performance metrics. Our most surprising finding is that the instruction cache performance of `memcached` is drastically worse than typical workloads. A typical SPEC CPU 2006 integer benchmark incurs at most ten misses per thousand instructions. In contrast, Figure 4 (a) reveals rates up to 15x worse. Our result is surprising because `memcached` itself comprises little code, fewer than 10,000 source lines. The poor instruction behavior is due to the massive footprint of the Linux kernel and networking stack.



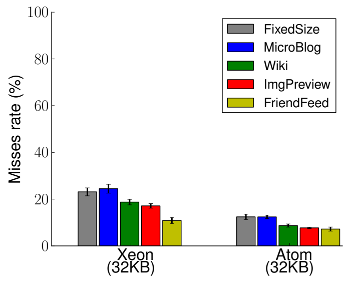
(a) L1 ICache MPKI



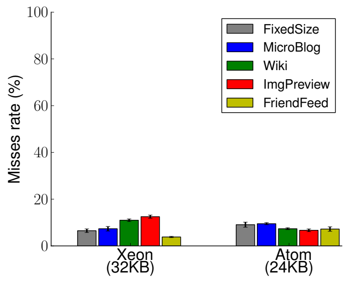
(b) L1 DCache MPKI



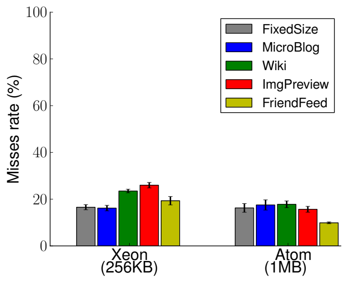
(c) L2 MPKI



(d) L1 ICache miss rate



(e) L1 DCache miss rate



(f) L2 miss rate

Figure 4: **Caching Behavior.** `Memcached` performance suffers from an inordinate number of ICache misses (over an order of magnitude worse than SPEC benchmarks). L1 data caching performs significantly better. L2 caching performs moderately well, although many of the hits are to service instructions. The Xeon also has a 12MB L3 (data not shown), but it provides little benefit, exhibiting miss rates as high as 95%.

L1 data cache behavior is more typical of other workload classes (e.g., SPEC). There are numerous compact but hot data structures accessed as a packet traverses the networking stack that can be effectively cached in L1. L2 caches are moderately effective, missing between 15-25% of the time as seen in Figure 4 (c). However, the majority of L2 accesses are for instructions that do not fit in the L1 instruction cache; if these are removed, the L2 data cache miss rates are substantially higher, in excess of 50%. Unlike the Atom, the Xeon processor also has a large L3 cache. We find the L3 to be highly inefficient, with miss rates from 60% to nearly 95%, a finding that suggests that increasing data cache sizes will yield little gain.

Overall, our analysis suggests that `memcached` requires far more instruction cache capacity than cores currently provide (primarily to hold the OS networking stack), but will not gain from larger data caches.

Virtual memory translation bottlenecks. In most applications, TLB misses are rare events and the overhead of virtual memory is hidden. A recent characterization of the Parsec Benchmark suite [10] finds that ITLB miss rates typically occur at a rate of 3x10^{-4} MPKI to 1 MPKI. DTLB miss rates fall generally in the range of 1x10^{-2} to 10 MPKI, but can be as high as 140 MPKI. In Figure 5, we see comparable TLB behavior for Xeon, but find that Atom provides an insufficient ITLB (16 entries vs. 128 for Xeon), which contributes significantly to its instruction fetch stalls. DTLB pressure is not a problem for either class of core.

Branch prediction bottlenecks. Figure 6 demonstrates that the front-end bottlenecks seen in the instruction caches also extend to the branch predictor. Despite numerous bulk memory copies with tight, predictable loops, branch misprediction rates are significant, particularly on the Atom, which has a considerably less capable branch predictor than Xeon.

The table in Figure 6 lists the top functions by execution time that also have misprediction rate of $> 10\%$. Many of the memory copies that dominate the total execution time are not present in the list. While they have many branches, they are highly predictable with mispredict rates of $< 1\%$.

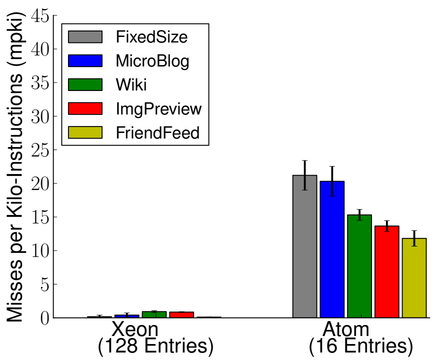
The entire networking stack is well represented in the table from the entry into the kernel from userspace `sys_sendmsg`, through the tcp stack `tcp_*`, the device layer `dev_*`, and finally the NIC driver `e1000_*`. The networking stack has significant irregular control flow, due to the complexity of the protocols and numerous performance optimizations. The kernel memory management functions are invoked primarily to allocate and free buffers for network packets.

Usercode also contains highly unpredictable branches. Several synchronization functions (e.g., `_pthread_mutex_lock`) figure prominently in the time breakdown, and have poor branch predictability because branch outcomes depend on synchronization races for contended locks. Other `memcached` functions contain case statements that change behavior based on the current connection state or request, which are difficult to predict.

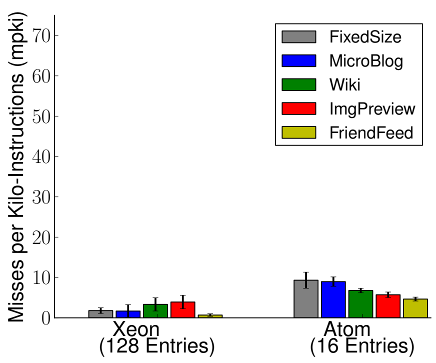
4.2 Impact of NIC quality

While it may seem obvious that `memcached` performance depends on networking performance, it is somewhat surprising that the quality of a NIC (i.e., its efficiency on a per-packet basis) is more important than raw bandwidth. We find that the choice of NIC is critical—not all NICs are created equal. We now explain the features that differentiate NIC performance and quantify their benefit.

There are a variety of features that advanced NICs can support, listed in Figure 7. *Segment offloading* allows the driver to provide the NIC a payload that is larger than the protocol maximum transfer unit; the NIC handles the task of splitting the packet, saving the CPU time of generating additional packet headers. *Software batching*, also called Generic Receive Offload (GRO), batches multiple smaller requests between the driver and the TCP stack, which ultimately reduces TCP stack invocations, reducing

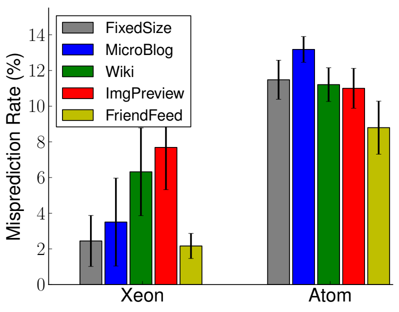


(a) Instruction TLB MPKI



(b) Data TLB MPKI

Figure 5: **Virtual memory behavior.** The Atom microarchitecture suffer from numerous translation misses due to its ITLB mere 16 entries. DTLB misses fall within a nominal range for both processor classes.



Function		%Time	Misprediction Rate
tcp_sendmsg	(k)	2.12%	10.74%
copy_user_generic_string	(k)	1.81%	10.97%
pthread_mutex_lock		1.04%	37.56%
event_handler		0.82%	13.68%
memcached main		0.77%	18.68%
tcp_clean_rtx_queue	(k)	0.77%	15.23%
kmem_cache_alloc_node	(k)	0.72%	11.43%
dev_queue_xmit	(k)	0.71%	12.19%
e1000_clean_rx_irq	(k)	0.71%	18.27%
sys_sendmsg	(k)	0.63%	12.96%

Figure 6: **Branch Prediction.** The table lists the top ten functions with misprediction $> 10\%$ out of fifty that consume the most execution time for the Microblog workload running on the Atom system.

processing time. These features coupled with the more optimized interface between the driver and device (many fewer programmed I/O requests) in the Enterprise NIC result in a 70% - 110% performance increase in all but one workload.

Multiple-queue (MQ) support allows the NIC to communicate with a driver running on more than one core. The receive portion of this, Receiver-Side-Scaling (RSS), hashes packet header fields in the NIC to choose among the available receive queues, each assigned to a different core. The hashing ensures that a single core processes all packets received on a particular flow, improving locality. Transmit scaling allows multiple CPUs to enqueue packets for transmission simultaneously without the need for locks. For the FixedSize workload, an impressive performance improvement of 47% is achieved with multiple queues. The gains are so pronounced in this workload because its average packet size is the smallest. Most of the other workloads see little gain at 1Gb/s line speeds because a single CPU core is able to keep up with the resulting request rate due to the larger average packet size.

The 10GbE and 10GbE+MultQueue bars in each cluster show the effectiveness of using a 10Gb/s NIC (with and without multiple-queues, respectively). The impact of greater network bandwidth (10GbE bar) varies drastically. For workloads with large average object size (Wiki and ImgPreview), gains are up to 22x. (Both ports of the 10GbE NIC are utilized for testing and thus the maximum bandwidth from the 10Gb NIC is actually 20Gb.) Unlike the Enterprise NIC, the 10GbE NIC supports *hardware batching*, or Receiver Side Coalescing, which performs similar batching in hardware. Enabling multi-queue support further increases these gains. In contrast, the workloads with small average packet sizes (Microblog and FriendFeed) are able to utilize 1.4Gbps, 2.2Gbps respectively, only a small fraction of the available bandwidth.

4.3 Power-Performance Across Architectures

Efficiency relative to peak server power is a crucial concern in designing `memcached` clusters. Since peak power consumption is often the limiting factor in the capacity of a data center, using machines with a better throughput per peak watt may allow the deployment of additional capacity. If Atom-based servers provide better throughput efficiency with respect to peak power, then, in the aggregate, a `memcached` cluster with a fixed power-budget constraint could support higher overall throughput with Atom-based systems, even though individually the servers support fewer requests each.

The sustainable throughput of a server depends on the latency constraint imposed on each individual request. As we offer more load to a server, throughput increases until we reach saturation, but per-request latency grows even faster due to queueing effects. Hence, we consider the power efficiency of each type of server as a function of a target 95th-percentile request latency. Figure 8 shows the power efficiency in kOps per peak watt for both Atom and Xeon for our FixedSize and ImgPreview workload, given a specific latency constraint (defined by the x-axis). As latency requirements may vary per deployment, we show a continuum of latency targets. We define “peak watt” as the highest full-system power draw we ever observed for each server across our experiments, measured by a Yokogawa power meter connected to the AC side of the power supply. We assume that a data center is provisioned based on this peak power (the implications of power capping schemes are beyond the scope of this study). Note that power efficiency is a measure of data center provisioning capacity, and is not a measure of energy

	Commodity	Enterprise	10GbE
Approximate Price	$10	$145	$850
Segment Offload[a]	X	✓	✓
Multi-Queue[a]	X	✓	✓
Software Batch	X	✓	✓
Hardware Batch	X	X	✓

[a]While the Realtek NIC hardware supports multiple queues and segment offload, the former isn't supported by the Linux driver and the latter is disabled by the driver by default.

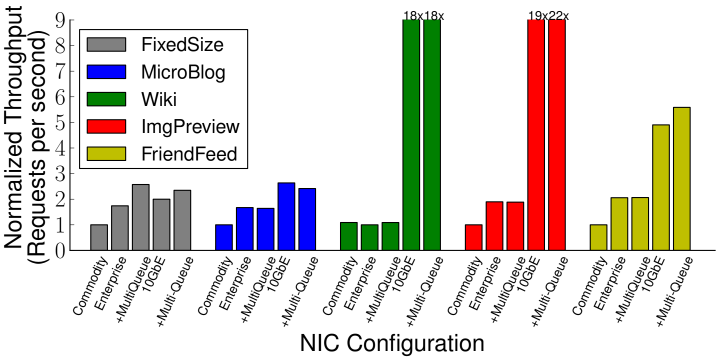


Figure 7: **NIC Features.** NICs have a number of key features that distinguish their performance beyond pure bandwidth. The data presented is the maximum achievable throughput with the various NICs and a latency constraint of 5ms with all 6-cores of the Xeon enabled. We demonstrate the performance of each of two classes of 1GbE NIC (Commodity and Enterprise) and a 10GbE model. We also selectively disable and enable multiple-queues(MQ). Note that adding these features can be expensive as is apparent with the switch between commodity and enterprise. However, these features can be quite powerful, for example with FixedSize Enterprise and MQ provide over a 2.5x performance boost. Wiki and ImgPreview suffer from being network bandwidth bound and achieve large gains from the switch to 10GbE alone.

efficiency.

Surprisingly, even though the Atom is a low-power processor with a significantly lower peak power than the Xeon, its power efficiency is considerably worse. A cluster provisioned with Xeons outperforms one provisioned with Atom servers by as much as a factor of 4 for a given fixed power budget.

4.4 Implications

Our analysis has revealed significant microarchitectural bottlenecks that cause both Atom- and Xeon-based servers to achieve surprisingly low instruction throughput for `memcached`. While the notion of using low-power servers to improve scalability under a power constraint initially appears appealing, our results in Section 4.3 indicate that this strategy falls short on two fronts: (1) current Atom-based systems fail to match the low per-packet processing latency that current Xeons can achieve at moderate loads, which may preclude their use from a latency perspective, and (2) at almost any load, the worst-case power draw of the current Atom remains too high to justify the loss in per-server throughput—Xeon systems are simply more efficient from a provisioning perspective.

What, then, might architects do to improve the capacity of a `memcached` cluster without resorting to raising the power budget and deploying additional Xeons? One alternative might be to leverage multicore scaling and deploy servers (Atom or Xeon) with a larger number of cores. However, our throughout measurements indicate that, even if throughput scales linearly with the number of cores (which is difficult to achieve from a software perspective) at least 6 Xeon cores are needed to saturate a 1Gb ethernet link and over 64 Xeon cores are required to saturate a 10Gb link when serving small (128-byte) values. Simply scaling the number of cores per server is not an appealing avenue to improve `memcached` provisioning efficiency.

However, our microarchitectural analysis also reveals a number of opportunities to eliminate processing bottlenecks that plague both the Atom and Xeon. Foremost, we note that the long execution paths in the networking stack are one of the central causes of poor microarchitectural performance. The large code footprint of the kernel, networking stack and NIC driver thrashes instruction supply mechanisms. Secondly, our analysis of the impact of NIC improvements shows that tighter integration between NIC and CPU (e.g., segment offload and multi-queue support) substantially improves performance. These observations point us towards a design where we leverage hardware acceleration of the networking stack to accelerate the common-case paths through the `memcached` code. This approach can simultaneously accelerate packet processing and remove the pressure that the networking stack places on the microarchitectural structures for instruction supply. In the next section, we propose *Thin Servers with Smart Pipes*, a new architecture for `memcached` based on this concept.

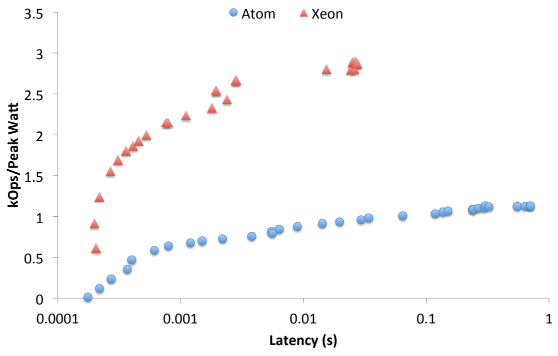
5. OVERCOMING BOTTLENECKS: THIN SERVERS WITH SMART PIPES

Our microarchitectural analyses motivate an architecture that addresses networking bottlenecks and their associated kernel burden. We leverage the opportunity from the industry trend towards increasing ease of integrating processors with accelerators and networking components in an SoC design. We propose an SoC architecture that implements the most latency- and throughput-critical `memcached` task, GET operations, in hardware. Our design pairs a hardware GET processing engine and networking stack near the NIC, integrating both with a conventional CPU to handle less latency- and throughput-sensitive operations.

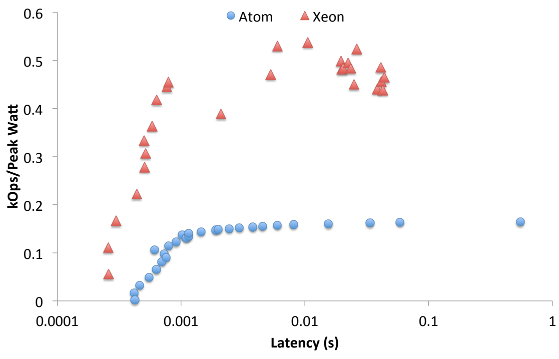
As recent studies of large-scale `memcached` deployments indicate an up to 30:1 GET/SET ratio [6], shifting GET processing to hardware allows approximately 97% of operations to be offloaded from the core enabling drastic improvements in both performance and performance/watt. However, by reserving more complex functionality to software, we avoid the programmability difficulties of a hardware-only implementation for non-critical operations (e.g., logging). Because the vast majority of processing shifts from the CPU to a hardware pipeline, our approach enables an embedded-class core to handle the remaining load. Hence, we can gain the power reductions of an embedded-class core without sacrificing latency and throughput, as in the Atom-based system. We therefore call our design Thin Servers (embedded class cores) with Smart Pipes (integrated, on-die NIC with nearby hardware to handle `memcached` GET requests).

5.1 TSSP Architecture

The overall TSSP architecture is shown in Figure 9. The SoC comprises one or more CPUs, a `memcached` hardware unit, and a NIC, which communicate to handle incoming requests. A system MMU translates virtual addresses that can be shared between software and hardware. The SoC has two memory controllers and a shared interconnect that includes both the processors and the I/O devices. The hardware accelerator can respond to a GET request



(a) FixedSize.



(b) ImgPreview.

Figure 8: **Power Efficiency vs. Latency.** These figures show power efficiency (throughput per peak Watt) as a function of 95th-percentile latency constraint for Atom and Xeon for the FixedSize and ImgPreview workloads.

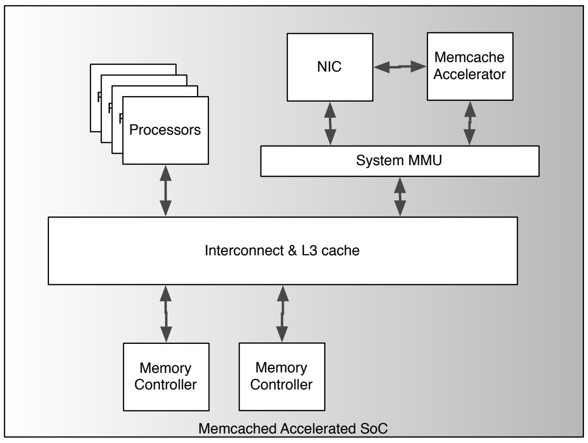


Figure 9: **Thin Server with Smart Pipe architecture.** Our TSSP design includes low-power cores, an integrated NIC, and a tightly coupled `memcached` accelerator in an SoC. The `memcached` accelerator is used to process GET requests entirely in hardware.

without any software interaction, but all other request types and memory management are handled by software.

Our design leverages several common characteristics of `memcached` workloads. First, we optimize for GETs, as they vastly outnumber SETs and other request types in most `memcached` deployments [6]. Second, as `memcached` is a best-effort cache, we use UDP, a lighter-weight protocol than TCP that provides more relaxed packet delivery guarantees, for requests that will be handled by the hardware GET accelerator. Requests that must be reliable (e.g., SETs), are transmitted over TCP and will be executed by software. Our decision to split traffic between TCP and UDP matches optimizations reported for some `memcached` deployments to improve software performance [9]. We verify in Section 5.5 that simply switching all traffic from TCP to UDP is insufficient to obtain the power-efficiency that TSSP can achieve. Lastly, we split `memcached`'s lookup structure (hash table) from the key and value storage (slab-allocated memory) to allow the hardware to efficiently perform look-ups, while software handles the more complex memory allocation and management.

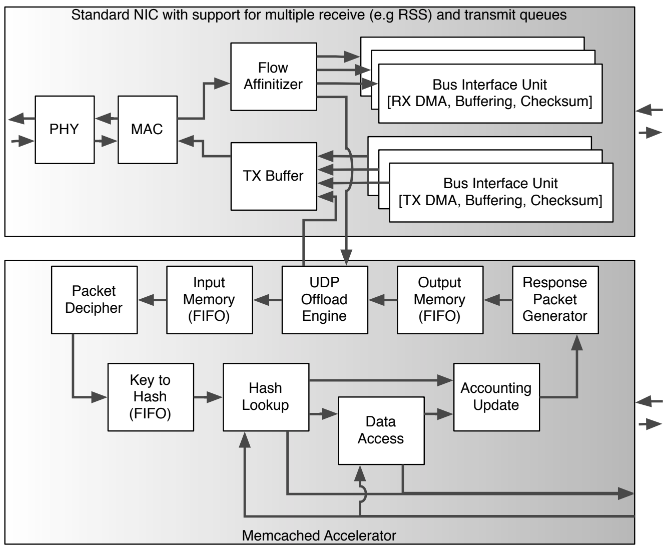
Figure 10 shows in detail the NIC and the `memcached` hardware unit. The flow affinitizer, which normally routes between several hardware queues destined for different cores based on IP address, port, and protocol, has been modified to allow the `memcached` accelerator to be a target. Similarly, on the transmit path, the NIC has been modified to allow packets to both be transmitted through the normal DMA descriptor rings as well as from the `memcached` hardware. After a packet is routed to the `memcached` hardware, it is passed to a UDP Offload Engine which decodes the packet and places the `memcached` payload into a buffer for processing. This design requires few NIC modifications and leverages flow affinity features already present in Gigabit NICs to route traffic across the accelerator (UDP on `memcached` port) and software (TCP on `memcached` port).

After the traffic is routed, our `memcached` hardware deciphers the request and passes control signals along with the key to a hardware hash table implementation. Since one of our design goals is to allow the hardware to respond to GETs without software involvement, the hardware must be able to perform hash-table lookups. This hash table must be hardware-traversable, so we choose a design in which the hardware manages all accesses to the hash table (including on behalf of cores) to avoid expensive synchronization between the hardware and software. Figure 11 illustrates the hash table and slab memory management scheme.

A critical concern with hardware hash tables is the requirement to bound the number of memory accesses per lookup, which makes some space-efficient software designs (e.g., cuckoo-hashing) infeasible. Kirsch and Mitzenmacher [22] demonstrate how to use key relocation and a small O(100) CAM structure to make hash conflicts extremely rare. For a given hash there are four possible slots in which a key may reside; if a conflict occurs and none of the four locations is available, the hardware attempts to relocate a victim key to a secondary location. Should this relocation fail, the hardware places the offending key in the CAM. In the extremely rare case the CAM is full, a key is discarded (in violation of the design objective to replace keys in LRU order).

We design each hash table entry to contain a fixed-size identifier ($\texttt{key}'$), a pointer to the memory location of the object (key, value, bookkeeping information), and a bit to indicate if it is free. `Memcached` keys are of variable size, ranging from 1 to 255 bytes; storing the keys directly in the hash table would make it difficult to manage space and traverse the hash table in hardware. Therefore we only store a 64-bit identifier, $\texttt{key}'$, based on the actual key in the hash table. We can calculate $\texttt{key}'$ in several ways, for example, using the low-order bits from each byte of the actual key. Thus when a hash table look up is performed, each entry has its $\texttt{key}'$ compared with the $\texttt{key}'$ of the requested key.

Upon a match, the full key (stored in memory) is compared. The



Standard NIC with support for multiple receive (e.g RSS) and transmit queues
PHY
MAC
Flow Affinitizer
Bus Interface Unit [RX DMA, Buffering, Checksum]
TX Buffer
Bus Interface Unit [TX DMA, Buffering, Checksum]
Packet Decipher
Input Memory (FIFO)
UDP Offload Engine
Output Memory (FIFO)
Response Packet Generator
Key to Hash (FIFO)
Hash Lookup
Data Access
Accounting Update
Memcached Accelerator

Figure 10: **NIC and GET accelerator details.** The major device blocks for the GET accelerator are illustrated in this figure along with how the accelerator is integrated into a typical NIC. The PHY, in the left side of the figure, would be connected to the data center network, while the connections on the right side of the figure are connected to the SoC fabric as shown in Figure 9.

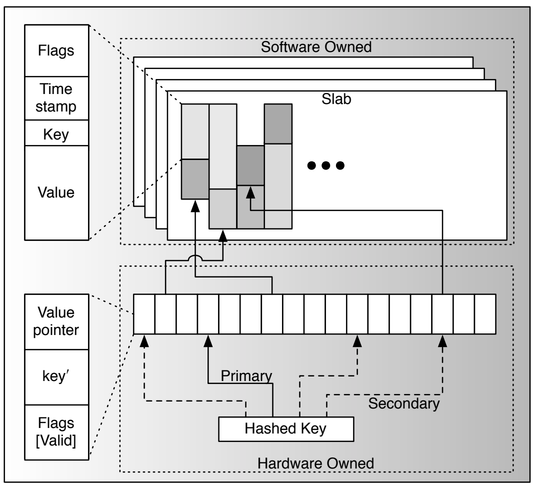


Figure 11: **TSSP - Hardware and Software data structures.** The ownership of the data structures is split between the hardware and software, allowing software to handle complex operations such as memory management or logging. The hardware owns the primary hash table to enable quick lookups and processing.

hardware then updates the key's last-access timestamp (for LRU replacement) and builds a `memcached` response packet. If no match is found, the hardware sends a miss response packet. The response packet is passed to the UDP offload engine, which adds UDP and IP headers and forwards the complete packet to the NIC hardware for transmission.

5.2 Software Support

Whereas TSSP manages the `memcached` hash table in hardware, other aspects, such as memory allocation, key-value pair eviction and replacement, logging, and error handling, are implemented in software. Leaving these less frequent, yet more complex, operations to software enables software updates and improves the feasibility of the hardware design (in particular the slab memory allocation is difficult to implement in hardware).

To insert new entries into the cache (e.g., on a SET), the software slab allocator reserves memory and copies the object (the key, value, and some accounting information) into place. The software then instructs the hardware accelerator (via programmed I/O) to probe the hash table with the desired key. The hardware responds indicating that space is available, or returns a pointer to an object that must be evicted. If an eviction is necessary (due to hash table overflow), software then frees the corresponding object's storage. The software then instructs the hardware to place the new object in the hash table, and the hardware calculates and stores the object's `key`′ and address in the hash table.

To implement LRU replacement for stale objects, the software maintains a list of the oldest objects in each slab. If this list reaches a pre-determined low-water-mark, software scans the slab linearly and enters the oldest `N` objects to the replacement candidate list. Object timestamps are updated upon access by the hardware. When an object is selected for replacement, its address is sent to the hardware to remove it from the hash table and its storage is recycled.

5.3 Evaluation platform

TSSP performance is determined primarily by the speed of GET operations. Therefore to evaluate our proposal, we seek to understand the implications of accelerating GETs in hardware. The full system interactions and multi-gigabyte workloads of `memcached` make it difficult to use simulation to model our design with sufficient accuracy. Instead, we leverage a recent hardware prototype [14] that implements `memcached` entirely as an FPGA appliance. This FPGA appliance implements SETs and GETs entirely in hardware, and includes numerous design compromises to implement memory management and key replacement fully in hardware. In contrast, we employ hardware acceleration only for GETs, relying on software for remaining functionality in the context of an SoC design coupling a general-purpose embedded core with a `memcached`-optimized accelerator. However, the existing FPGA design still serves as a useful platform to make performance projections for our TSSP GET accelerator.

The existing FPGA implementation targets an Altera DE4 development board with a Stratix IV 530 FPGA. The board has two 4GB DDR2 memory modules and four 1GbE ports. The design comprises a custom-built `memcached` implementation, a UDP offload engine, `memcached` packet parsing, hash calculation, hash table lookups, and slab memory management. Additional Altera IP blocks are used to implement the Ethernet IP and DDR2 memory controller.

5.4 Area and Power

We estimate the power and area requirements of TSSP based on a detailed breakdown of the FPGA design. The hardware components reused in TSSP require a total of approximately 8,000 Look Up Tables (LUTs), barely 2% of the FPGA.

Based on the existing hardware breakdown, we estimate the implementation costs of TSSP in the Xilinx Zynq platform [2], an SoC platform with a dual-core ARM A9 processor and on-die configurable logic. The Zynq can operate at frequencies up to 1GHz and provides 78,000 configurable LUTs—more than sufficient for our design—in a 6W power envelope. We estimate TSSP's full-system power requirements by adding the Zynq's power requirements to that of similar non-CPU components of the Atom-based server system studied previously. As TSSP is an SoC that integrates additional components, such as the NIC, it can

System	Power	Perf.	Perf./W	Norm. Perf./W
	(W)	(KOps/Sec)	(KOps/Sec/W)	
TSSP	16	282	17.63	16.1x
Xeon-TCP	143	410	2.87	2.6x
Xeon-UDP	143	372	2.60	2.4x
Atom-UDP	35	58	1.66	1.5x
Atom-TCP	35	38	1.09	1.0x

Table 2: TSSP energy efficiency comparison for FixedSize workload. Results for Xeon and Atom systems are shown for both TCP and UDP to isolate benefits from using UDP connections.

achieve lower power than the Atom platform (the Zynq platform includes an integrated NIC). Combined, we arrive at an overall full-system power estimate of 16W. A Zynq-based platform implements the TSSP GET accelerator in programmable logic; a custom ASIC could further reduce power requirements at the cost of ease of upgradability as `memcached` software implementations evolve.

5.5 Performance and Efficiency

We estimate the performance of our TSSP architecture based on measured processing time of the FPGA implementation of portions of the `memcached` application. (In some cases, we had to limit the key sizes and value sizes of the workload to fit the constraints of the FPGA platform, but we do not expect this to change our results qualitatively.) We calculate total GET throughput by measuring end-to-end latency for GET operations on the FPGA as a function of key and value size. We then calculate total processing time for the stream of GET operations seen in our workloads from these per-request measurements. Since GET operations do not require software intervention or hardware communication between the TSSP accelerator and core, the end-to-end latency measured on the FPGA accounts for the entire GET turnaround time. This calculation allows us to determine the total time required to complete a certain number of operations, and we thus translate the result to overall throughput. We focus specifically on those workloads that are almost exclusively GET operations, which include the FixedSize, Wiki, and ImgPreview workloads. Based on each request's value size, we determined the latency for the prototype to process the GET request.

Our results show that, based on an FPGA prototype running at 125MHz with a 1Gigabit Ethernet NIC, our proposed TSSP design achieves throughputs of 282 KOps/s, 68 KOps/s, and 80 KOps/s for the FixedSize, ImgPreview, and Wiki workloads, respectively.

Combining our power and throughput numbers, we calculate the overall efficiency of our TSSP architecture. The results, based on the FixedSize workload, are shown in Table 2. We see that compared to the baseline Atom and Xeon systems using TCP, the TSSP design provides 16X improvement over the Atom design, and 6X improvement over the Xeon design. We expect greater improvements if the accelerator were implemented as an ASIC rather than in an FPGA.

Based on the bottlenecks identified for the Atom-based system, one reasonable hypothesis is that a low-power core with an improved front end could perform as well as TSSP. To match the efficiency of TSSP, however, an improved Atom system requires a 10X performance improvement within its existing power envelope. We note that the Xeon system, which dedicates significantly more resources to the front-end, still achieves only 7X better performance than Atom with a far higher power draw. Hence, even with a front end as capable as Xeon, we do not expect an Atom system to match the performance or power-performance efficiency of TSSP. Similarly, though TCP offload might improve high-end system performance, TSSP provides a 6X performance/watt gain over the Xeon system and high-end NIC we analyze; we do not expect TCP offload alone to provide a 6X performance boost and instead expect that TSSP will maintain its efficiency advantage.

To ensure our use of UDP in the TSSP architecture was not the only source of efficiency improvement, we also tested our existing servers using UDP. While moving to UDP does improve Atom performance significantly (by 53%), its power efficiency remains well below that of the Xeon, and our TSSP design still provides greater than 10X improvement. On the other hand, UDP only minimally impacts Xeon performance, incurring a small degradation as some packets are dropped. Comparing our baseline TCP results to our UDP results indicates that our TSSP benefits come primarily from shifting software burden into hardware. Our findings regarding the impact of UDP have been corroborated by Facebook engineers [1], who indicate that for a large-scale deployment, switching to UDP provides at most a 10% performance improvement.

6. RELATED WORK

Other memcached-optimized designs. `Memcached` has been studied in the context of the Tilera many-core processor architecture [9]. Although that study demonstrates substantial performance gains, it does not enumerate the scalability bottlenecks in `memcached` that we identify (e.g., the NIC and TCP/IP stack), and still focuses on a high-power, high-performance solution. Other work [18] has looked at using a GPU to accelerate the hash table look up portion of `memcached`. In contrast, our work designs a solution targeting key energy-efficiency bottlenecks across the entire system stack, using a low-power SoC design with `memcached`-optimized accelerators.

The work in [14] discusses our experiences developing a `memcached` appliance on an FPGA platform, implementing the entire algorithm in hardware. TSSP's objective differs from [14], improving in its functionality rather than in performance. The prior design is an FPGA-only implementation that supports only GET and SET operations, neglecting other important operations, such as INCR, ADD, or CAS, and makes numerous design compromises (e.g., restrictive memory management, no support for software upgrades, logging or debugging). TSSP on the other hand avoids these restrictions by relying on hardware only to accelerate the common case (GETs), allowing software to provide a full-featured `memcached` system. Based on recent industry trends, our proposed SoC TSSP design is likely to have more market acceptance than the FPGA-only design.

Improving networking. Other studies have also identified the network stack as one of the key bottlenecks for `memcached` performance. One work has proposed using non-commodity networking hardware (e.g., Infiniband) and RDMA interfaces [20] to improve performance. Other work proposes to improve performance using a software-based RDMA interface over commodity networking [30]. Meanwhile, Kapoor and co-authors have implemented user-level networking and request partitioning to remove network bottlenecks [21]. In comparison, we focus on commodity Ethernet-based systems, as Ethernet currently leads to more cost-effective clusters. Unlike RDMA-based solutions, which can push the performance burden to the client machines, our design supports the standard `memcached` interface with no modifications.

Improving software scalability. In earlier versions of `memcached`, heavy contention on a few key locks caused poor software scalability. In response, some works [25, 32] propose modified designs that use partitioned structures to achieve better scalability. Additionally, memcached developers have also tried to improve scalability in recent releases by adding finer-grained locking. These developments are orthogonal to our work, as they do not address the significant networking overheads we identified, and do not greatly improve the energy efficiency of a single system.

Other work. Similar to our examination of Xeon- and Atom-class servers, the power-performance trade-offs between "wimpy" or "beefy" architectures have been considered for other workloads [24, 31], but have not been studied for memcached. Others have looked at addressing the scalability of TCP/IP stacks and NICs and ways to reduce packet processing overheads. There is significant interest in designing ultra-low latency networked systems, but most work has focused on removing software overheads [7, 29]. Studies considering hardware modification include TCP onloading[23, 28], Direct Cache Access (an architectural optimization to deliver packets into the CPU cache) [19], and tighter coupling between the NIC and CPU to provide "zero-copy" packet delivery [11, 12]. We believe these networking optimizations could also be applied in our TSSP architecture.

7. CONCLUSIONS

Memcached has rapidly become one of the central tools in the design of scalable web services with several important commercial deployments including Facebook, Twitter, Wikipedia, etc. Based on a detailed power/performance characterization across both high-performance and low-power CPUs, we identify several interesting insights about current system bottlenecks. Notably, processing bottlenecks associated with the networking stack are one of the central causes for poor performance. Based on the insights from our characterization, we propose a new design—Thin Servers with Smart Pipes (TSSP)—that integrates low-power embedded-class cores with a memcached-optimized accelerator in a custom, yet volume, SoC for future memcached servers. Our TSSP architecture, carefully splits responsibilities between the hardware accelerator and software running on the core, and can scale across multiple memcached versions and scenarios. We assess the feasibility and evaluation of our design using a prototype memcached FPGA implementation. Our results (conservatively) show the potential for a factor of 6-16X improvement in energy efficiency over existing servers. We also believe that our design will have applicability to other emerging data- and network-centric workloads, such as distributed file systems, other key-value stores, and in-memory databases.

8. ACKNOWLEDGMENTS

The authors would like to thank Sai Rahul Chalamalasetti and Alvin AuYoung for their suggestions and contributions to the paper. This work was partially supported by NSF CCF-0815457.

References

[1] Private communication with Facebook engineers, 2012.

[2] Zynq-7000 All Programmable SoC, 2012.

[3] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. In *Proceedings of the International Symposium on Computer Architecture*, 2010.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera : Dynamic flow scheduling for data center networks. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2010.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the Symposium on Operating Systems Principles*, 2009.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2012.

[7] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proceedings of the International Symposium on Computer Architecture*, 2011.

[8] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. In *Proceedings of the Symposium on Operating System Design and Implementation*, 2010.

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the International Green Computing Conference*, 2011.

[10] A. Bhattacharjee and M. Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[11] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Performance Analysis of System Overheads in TCP / IP Workloads. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[12] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[13] M. Cha, A. Mislove, and K. P. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *Proceedings of the International Conference on World Wide Web*, 2009.

[14] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2013.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the Symposium on Operating Systems Principles*, 2007.

[16] Facebook. Memcached Tech Talk with M. Zuckerberg, 2010.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the Conference on Data Communication*, 2009.

[18] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2012.

[19] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[20] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the International Conference on Parallel Processing*, 2011.

[21] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Symposium on Cloud Computing*, 2012.

[22] A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware. *IEEE/ACM Transactions on Networking*, 18(6):1752 –1765, dec. 2010.

[23] G. Liao and L. Bhuyan. Performance measurement of an integrated nic architecture with 10gbe. In *Proceedings of the Symposium on High Performance Interconnects*, 2009.

[24] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[25] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: a cache-partitioned hash table. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. ACM, 2012.

[26] G. Minshall, Y. Saito, J. C. Mogul, and B. Verghese. Application performance pitfalls and TCP's Nagle algorithm. *SIGMETRICS Performance Evaluation Review*, 27(4), 2000.

[27] R. Nishtala, 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. Scaling memcache at facebook. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2013.

[28] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *Computer*, 37(11), nov. 2004.

[29] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. In *Proceedings of the Conference on Hot Topics in Operating Systems*, 2011.

[30] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[31] V. Janapa Reddi, Benjamin Lee, Trishul Chilimbi, and Kushagra Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proceedings of the International Symposium on Computer Architecture*, 2010.

[32] A. Wiggins and J. Langston. Enhancing the Scalability of Memcached, 2012.