Software-Defined Caching

[Technical Report CSRG-626]

Ioan Stefanovici*, Eno Thereska, Greg O'Shea, Bianca Schroeder*, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, Tom Talpey†

University of Toronto*, Microsoft Research, Microsoft†

Abstract

In data centers, caches work both to provide low IO latencies and to reduce the load on the back-end network and storage. But they are not designed for multi-tenancy; system level caches today cannot be configured to match tenant or provider objectives. Exacerbating the problem is the increasing number of un-coordinated caches on the IO data plane. The lack of global visibility on the control plane to coordinate this distributed set of caches leads to inefficiencies, increasing cloud provider cost.

We present Moirai, a tenant and workload aware system that allows data center providers to control their distributed caching infrastructure. Moirai can help ease the management of the cache infrastructure and achieve various objectives, such as improving overall resource utilization or providing tenant isolation and QoS guarantees, as we show through several use cases. A key benefit of Moirai is that it is transparent to applications or VMs deployed in data centers. Our prototype runs unmodified OSes and databases providing immediate benefit to existing applications.

1. Introduction

An increasing number of enterprise applications have migrated to hosted platforms in private enterprise and public cloud data centers. Such platforms are typically virtualized, i.e., tenants deploy applications in virtual machines (VMs) whose access to the underlying resources (memory, storage, network) is shared with other tenants, and mediated by hypervisors such as Hyper-V, VMware ESX, or Xen. Uninhibited sharing of such resources in a multi-tenant environment leads to poor and variable application performance. While recent efforts give providers control over how resources like network [1, 17, 22, 30, 33] and storage [2, 15, 16, 34, 38] are shared, there is no coordinated end-to-end control of the distributed caching infrastructure, made up of storage caches at multiple places along the IO stack (inside VMs, hypervisors, storage servers; see Figure 1). Today, storage caches along the IO stack are transparent to both applications and cloud providers, lack workload-aware mechanisms, and are each managed in isolation, leading to multiple problems:

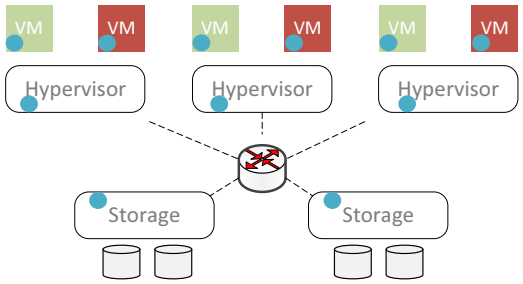
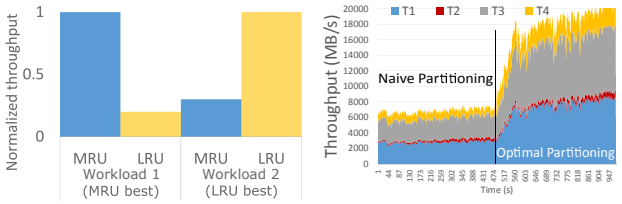


Figure 1: Simplified IO stack in a multi-tenant data center. Two tenants, a green and red one are shown, with 3 VMs each spread over 3 hypervisors. The circles represent typical caches on the IO stack.

• **Lack of performance isolation** Since caches are not tenant- or workload-aware, applications with different IO patterns and request rates sharing the same cache will impact each other's cache performance. For example, depending on the cache eviction policy, one application's large sequential reads can blast away another workload's working set. Even with scan-resistant cache management policies, such as ARC [27], aggressive clients with higher request rates will still be allocated larger portions of the cache.

• **Lack of customization** Since caches are not tenant aware, the entire cache is treated as a single pool with one cache write policy (write-through, write-back, etc), despite different durability requirements of different applications, and one eviction policy, despite the fact that different workloads benefit from different cache eviction policies. For example, Figure 2(a) shows two IOMeter workloads under two different eviction policies, LRU and MRU [9] respectively. The workload on the left performs at its peak with an MRU policy, while the one on the right performs best with LRU. Today, if both workloads were running atop the same hypervisor, they would have to follow the same eviction policy, leading to performance penalties on the order of 4-5x.

• **Lack of coordination** Each cache in the IO stack makes its decisions locally, agnostic to the state of other caches in the stack, leading to inefficiencies, such as double caching, as was also noted by Wong and Wilkes [46].



(a) The effect of eviction policy (b) The effect of cache size

Figure 2: Performance depends on the cache policy (a) and allocation (b).

• **Lack of adaptability** Currently, the organization and configuration of caches is fixed. Caches cannot be added, removed, or resized on the fly to adapt to changes in the workload or in provider objectives.

• **Waste of system resources** Simple solutions for partitioning caches along the IO stack are not sufficient. For example, Figure 2(b) shows that the observed performance triples when cache space is optimally allocated according to workload characteristics (the workload consists of 4 tenants using 120 VMs in total), compared to the case when caches are naively allocated across tenants. We will describe the details of this experiment in Section 5, but note that all workloads' throughputs benefit when the right cache size is chosen. This is true even for tenants that receive less total cache, as the contention at the storage device is reduced.

While some of these problems have been tackled in isolation, there is no comprehensive framework for the end-to-end management of caches that allows providers to address the major issues they are facing today. We present Moirai[1], a tenant- and workload-aware system that allows data center providers to control their distributed caching infrastructure to achieve provider objectives, such as improving resource utilization and request latency, achieving tenant isolation and QoS guarantees. Moirai does not require changes to the IO stack architecture, is transparent to applications and VMs, and does not change cache consistency semantics.

2. Design

Figure 3 shows the architecture of Moirai, which comprises three key components. At the core is a logically-centralized *controller* that uses information on workload characteristics maintained by the *metrics engine* to configure the *programmable caches* to achieve provider objectives. Details on each of the three components are provided next.

2.1 The Metrics Engine

The Metrics Engine is a hypervisor-based module that maintains key characteristics for each workload running on the system, such as throughput, number of reads vs. writes, etc., but also *hit ratio curves*, which describe the percentage of requests serviced from cache as a function of the cache size. We use *phantom caches*, which inspect IO headers (with fields such as accessed file name, offset, length, etc.) and exploit techniques from recent work [32, 44, 45] to generate hit ratio curves efficiently at runtime. The Metrics Engine periodically sends these performance metrics to the centralized controller.

[1] Moirai (Ancient Greek for "apportioner") in Greek mythology are the three personifications of fate, who control the thread of life of every mortal from birth to death, analogously to the end-to-end control of caches by the three components that comprise Moirai.

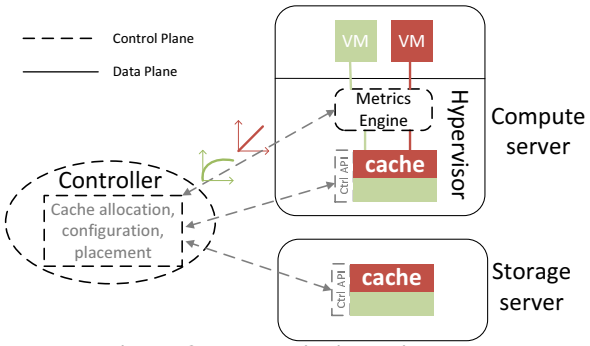


Figure 3: The Moirai architecture.

2.2 Programmable Caches

Caches along the IO stack are programmable through a simple API shown in Table 1. Caches are created at the desired position in the IO stack by sending a `createCache` call to the appropriate level in the stack (more details in Section 4). A cache c is made workload-aware using the `createRule` call, which installs a rule to specify the IOs that should be cached in c. If the header of an incoming I/O matches one of c's rules, the IO (header+data) is sent through the cache. The controller can also configure cache properties (`configureCache`) to set the size, eviction, and write policies. Similarly, cache performance metrics are obtained via the `getCacheStats` call.

Care must be taken to maintain consistency semantics when the location of a cache changes. For example, the controller could decide to cache at the storage server rather than at the hypervisor. In order to maintain consistency, Moirai first removes the caches on the old path, which automatically triggers the eviction of all cached state, including writing any dirty blocks to the back-end storage, and then installs caches on the new path. We considered other options, such as keeping the old caches until all accesses eventually move to the new caches, but they add complexity and require maintaining extra metadata.

2.3 Controller

The centralized controller uses the API described in Section 2.2 and information provided by the Metrics Engine to create and configure caches in order to implement a set of objectives specified by the provider, as illustrated in the next section.

createCache (<size,eviction pol,write pol>)	
	returns a reference to the newly created cache *c*
removeCache (Cache *c*)	
createRule (IO Header *h*, Cache *c*)	
	creates cache rule <src,op,file,range>→ *c*
removeRule (IO Header *h*, Cache *c*)	
configureCache (<size,eviction pol,write pol>, Cache *c*)	
getCacheStats (Cache *c*)	
	returns cache statistics

Table 1: Moirai's API for a configurable cache.

3. Data Plane Transformations

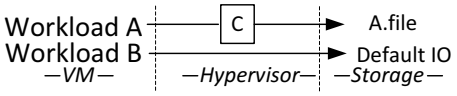
In this section, we explore Moirai's ability to program and transform the data plane to implement various cloud provider objectives and improve workload performance. For each goal, we illustrate how the controller effects the necessary changes on the data plane.

3.1 Prioritizing a Workload

It's often desirable to be able to isolate the performance of a particular (high-priority) application *A* from that of another application *B* sharing a cache in the same VM. The controller can achieve this by configuring a dedicated cache *C* (a 50GB LRU write-through cache in this particular example) inside the hypervisor, which is exclusive to workload *A*:

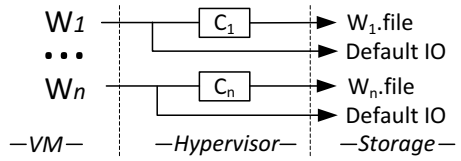
1: `C = createCache` (< 50*GB*, LRU, write-through>)
2: `createRule` (< *VM*, *, A.file, *>, `C`)

The `createRule` call configures the cache to accept all R/W IOs originating from the VM, that access any part of A.file. The figure below shows the resulting data plane. Workload A flows through its own cache *C* in the hypervisor, while workload B continues along its previous path, bypassing that cache, effectively isolating A's traffic from it.



3.2 Providing Per-Workload Bandwidth Guarantees

Next we extend the objectives beyond simple priorities, and examine how Moirai allocates cache space to several arbitrary workloads $W_1, W_2, \ldots, W_n$, all running on the the same system, in order to guarantee each workload W_i a particular bandwidth B_i. Similar to Section 3.1, the controller passes each workload's traffic through its own dedicated cache C_i at the hypervisor (see figure on the following column), but the question now becomes what the size each of the caches needs to be. In this section, we focus on hypervisor level caches only, but the techniques can be expanded to include simultaneous allocation of hypervisor and storage level cache space, as we explain in Section 3.5.



To answer this question, the controller uses information from the Metrics Engine to first determine the hit ratio Hit_i^{cache} required for workload W_i to meet a certain bandwidth guarantee, and then allocates the workload W_i cache space a_i, such that $U(a_i) = Hit_i^{cache}$, where U is the workload's hit ratio function (provided by the Metrics Engine).

More precisely, note that if the total bandwidth achievable from the storage back-end [2] is $BW_i^{storage}$ and main memory bandwidth is BW^{memory}, a workload's bandwidth depends on its hit ratio Hit_i^{cache} as follows:

$$SLA_i^{BW} \leq Hit_i^{cache} \times BW^{memory} + (1 - Hit_i^{cache}) \times BW_i^{storage} \quad (1)$$

That means the cache hit ratio in order to achieve a bandwidth SLA_i^{BW} needs to be at least:

$$Hit_i^{cache} \geq \frac{SLA_i^{BW} - BW_i^{storage}}{BW^{memory} - BW_i^{storage}} \quad (2)$$

After the min. data bandwidth guarantees $SLA_1^{BW}, \ldots, SLA_n^{BW}$ are met for all n workloads, the leftover cache space can be allocated based on priorities or using approaches highlighted in Section 3.3 to optimize for global utility.

3.3 Maximizing Global Workload Utility

Rather than per-workload guarantees, a provider might strive to maximize the global workload utility, i.e., the sum of the utilities across all workloads in the system. Utility of a workload could be measured by hit ratio, or be defined more generally in terms of bytes per second (Bps) satisfied by the cache, or by extending the notion of hit ratio by introducing weights to account for the type of IO (i.e. reads vs. writes), or even to account for the impact of a workload on the storage device (e.g. sequential vs. random access). The choice of definition for utility will be dictated by the optimization goals of the cloud provider.

Using the example of hit ratios as the utility function, the controller can create a separate cache for each workload (similar to Section 3.2) and then use a classic result [37] to determine the cache allocations $a_1, ..., a_n$. The algorithm, shown in Algorithm 3.1, uses a water-filling approach, i.e, it allocates the cache to workloads in small increments. The basic idea at each step is to allocate the next increment of cache to the workload that will achieve the highest hit rate out of the allocation. When the hit rate curves of workloads are *concave* functions, this algorithm will achieve an allocation that maximizes the total hit rate, i.e., total hit rate at the

[2] If the storage back-end is remote, $BW_i^{storage}$ is the minimum of the network, and the back-end storage array's bandwidth.

cache. We are currently investigating meta-heuristics to deal with non-concave hit ratio curves. E.g. Soundararajan [36] proposed hill-climbing search, although we find that their particular algorithm and implementation is too slow for our system to react dynamically.

Algorithm 3.1 Utility-maximizing cache allocation

Require: n workloads sharing a cache of capacity C; U_1,...,U_n: hit rate curves for workloads.

Ensure: Assign cache allocation a_i to workload i s.t. $\sum a_i = C$, and $\max \sum U(a_i)$

1: $\forall i, a_i = 0$ //Initialize allocations
2: $leftC$ =C //Cache left to distribute
3: $\varepsilon = 0.001 \times$C //Water-filling constant (as fraction of C)
4: **while** ($leftC > 0$) **do**
5: $cacheAlloc = \min(\varepsilon, leftC)$
6: $j = \arg\max_i (U_i(a_i + cacheAlloc) - U_i(a_i))$ //workload with the most utility gained from extra cache
7: $a_j += cacheAlloc$
8: $leftC -= cacheAlloc$

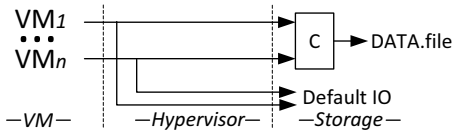
One might argue that a standard, workload-agnostic system that manages the entire cache as a single pool and applies its favourite replacement policy to it is also designed to achieve the same goal of maximizing overall hit ratio. However, Moirai can provide generalizations of this goal (e.g. a weighted sum of the hit ratios across workloads) and simultaneously provide other goals, such as isolation (e.g. protecting one workload from the effects of workload spikes in another workload), which a standard system cannot.

3.4 Consolidating Memory Over Fast Networks

As systems are increasingly making use of fast networks with speeds in excess of 40-100Gbps , and RDMA capabilities [10], use of remote resources is becoming increasingly feasible and can improve overall utilization of resources. Consider as an example a read-only dataset DATA.file accessed by N VMs across N hypervisors. Placing one consolidated cache at the storage server can result in an Nx reduction in total cache space used, with potentially only small increases in latency. The controller can accomplish this as follows (using the example of a 100GB MRU write-back cache C as the consolidated cache):

1: `C = createCache` (<100GB , MRU, write-back>)
2: `createRule` (< $VM1-N$, *, DATA.file, *>, `C`)

The resulting data plane is shown in the figure below:



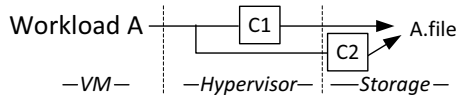
3.5 Scaling Out Caches

In addition to fully-remote caching, caching capacity per workload can be split across the compute and storage server, while appearing to the VM and applications as one single aggregate cache. Note that today, workloads do flow through both caches (at the hypervisor, and at the storage server), but this occurs in an uncontrolled fashion, leading to wasted memory capacity by double-caching of data in both places.

In situations where the hypervisor is hosting several applications and memory is limited, the controller has several choices for how to split the cache for a workload A and configure it at the hypervisor(1) and storage server(2). If the workload access is uniform across the file, one choice is to cache half the file in each of the respective caches:

1: `createRule` (< VM, *, A.file, 0, size/2>, C1)
2: `createRule` (< VM, *, A.file, size/2+1, size>, C2)

The resulting data plane is shown in the figure below:



The controller can also match workload access patterns to the way the cache is split based on hot or cold blocks or files.

Another option is to treat both caches as a global LRU cache. To do that, the controller programs C1 to cache the IOs from A.file, and C2 to only cache IOs that were evicted (or "demoted") from C1. To provide per-workload bandwidth guarantees, Moirai extends the cache allocation method presented in Section 3.2. The controller now needs to determine two things:

1. How much cache space a_i to allocate for the global LRU cache made up of both C1 and C2, such that:

$$U(a_i) = Hit_i^{cache} \tag{3}$$

2. The individual cache space allocations a_i^1 and a_i^2, for C1 and C2 respectively. Thus, for some α:

$$\begin{aligned} a_i^1 &= \alpha \times a_i \\ a_i^2 &= (1-\alpha) \times a_i \end{aligned} \tag{4}$$

The relationship between these variables is illustrated using a simple, example hit rate utility function in Figure 4.

A workload's bandwidth SLA_i^{BW} now depends on the hit ratio Hit_i^{cache} of the global LRU cache as follows:

$$SLA_i^{BW} \leq Hit_i^{cache} \times BW_i^{GlobalLRU} + (1 - Hit_i^{cache}) \times BW_i^{storage} \tag{5}$$

Similar to Equation (2), the cache hit ratio Hit_i^{cache} needs to be at least:

$$Hit_i^{cache} \geq \frac{SLA_i^{BW} - BW_i^{storage}}{BW_i^{GlobalLRU} - BW_i^{storage}} \tag{6}$$

Here, $BW_i^{GlobalLRU}$ refers to the total bandwidth achievable from the global LRU cache.

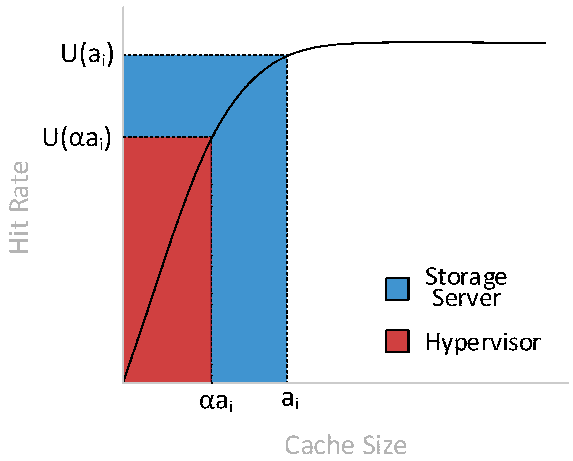


Figure 4: Example of a cache hit rate function U, and associated parameters a_i and α, used to compute cache allocations for scaled-out LRU caches with bandwidth guarantees.

Since C1 and C2 form the global LRU cache, the fraction α of cache space allocated to C1 will result in $U(\alpha a_i)\%$ of the hits, while the rest of the hits, $[U(a_i) - U(\alpha a_i)]\%$, will be served from C2. Since C1 is a hypervisor cache, its achievable bandwidth is BW^{memory}, while C2's achievable bandwidth is constrained by the bandwidth of the network $BW_i^{network}$. Thus:

$$BW_i^{GlobalLRU} = U(\alpha a_i) \times BW^{memory} + [U(a_i) - U(\alpha a_i)] \times BW_i^{network} \quad (7)$$

Simultaneously using Equations (3), (4), (6), and (7), the controller solves for a_i, and α, effectively determining the cache allocations a_i^1 and a_i^2, for C1 and C2 respectively. Further constraints can also be added to the problem statement (e.g., imposing a maximum size on either C1, or C2) to limit the solution space.

4. Implementation

We have implemented and deployed a Moirai prototype, comprising all components described in Section 2, on a Windows-based system and made the code publicly available [28]. The controller is implemented in around 6000 LOC of C# and communicates with the caches through RPCs over TCP. The Metrics Engine is implemented as a user-level stage in the hypervisor in around 500 LOC and uses a variant of SHARDS [44] to determine hit ratio curves. Cache modules implement the APIs in Table 1 at user-level in around 2000 LOC in C#.

One implementation challenge is how to classify and direct a tenant's traffic to the configurable caches. We decided to build an extension of the IOFLow framework [38] to implement this functionality. Note that while in its original form IOFlow does keep track of each IO's tenant class, it was designed to provide IO queueing and rate limiting based on IO request headers. By contrast, caching involves inspection and manipulation of the *data* associated with an IO request. We implemented an extension of the IOFlow architecture to add support for data transformations using a version of the Windows messaging API for filter drivers, in around 500 new LOC. IOs are passed to a user-level cache through an upcall, while a kernel-mode thread handling the I/O request blocks pending a return code from the cache. The latter decides whether the request is terminated at the filter driver (hit), or is sent further down the IO stack.

	H.Index	H.Data	H.Msg	H.Log	Exchange
Read %	75%	61%	56%	1%	40%
IO Sizes	64 KB	8 KB	64 KB	64 KB	8 KB
Seq/rand	Mixed	Rand	Rand	Seq	Rand
# IOs	32M	158M	36M	54M	60M

Table 2: Characteristics for 4 Hotmail workloads, part of a 2-day Hotmail IO trace and an Exchange workload, part of a 1-day Exchange IO trace. Seq/rand refer to sequential and random-access respectively. *M*=million.

5. Experimental Evaluation

This section provides an experimental evaluation of some of Moirai's use cases presented in Section 3. Our experimental testbed has 12 servers, each with 16 Intel Xeon 2.4 GHz, 384 GB of RAM and three Seagate Constellation 2 disks or four Intel 520 SSDs in RAID-0. The servers run Windows Server 2012 R2 operating system and can act as either Hyper-V hypervisors or as storage servers. Each server has a 40 Gbps Mellanox ConnectX-3 NIC supporting RDMA and connected to a Mellanox MSX1036B-1SFR switch. We use a combination of real enterprise application traces and benchmarks, as specified in more detail below. As Moirai is transparent to applications and VM's, they can run on our testbed without modifications

We use a mixture of real enterprise application traces and benchmarks in the evaluation. For the former, we use public traces from an enterprise Exchange email server [35] and Hotmail [39]. Key characteristics of these traces are shown in Table 2. The traces are diverse across a number of metrics such as the Read-to-Write ratio, IO sizes, sequentiality of access and number of IOs which allows for a comprehensive evaluation across realistic workload mixes. However, a limitation of these workloads is that they were originally collected *underneath* file caches. As such, they under-represent the amount of application reads.

To account for this limitation, we also use TPC-E [41] and TPC-H [42] to cover a broad class of workloads, ranging from transaction processing OLTP operations with small IO sizes (TPC-E) to large streaming IO from data mining queries (TPC-H). They run over unmodified SQL Server

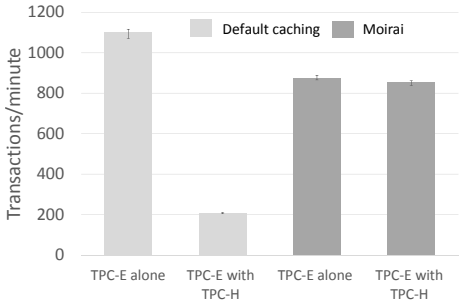


Figure 5: Prioritizing one workload (TPC-E) vs another (TPC-H). With Moirai, the performance of TPC-E is not impacted by TPC-H. In contrast, today, running both workloads together would result in a 5x performance hit for TPC-E.

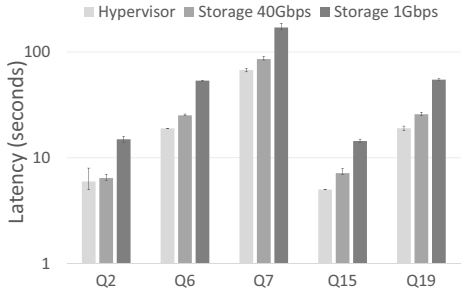


Figure 6: Latency for 5 TPC-H queries. The controller can decide to use file caches in the storage server for fast RDMA-based networks. Y-axis is log scale.

2012 R2 databases. When error bars are shown they represent the average, minimum and maximum from 5 runs.

5.1 Enforcing Priorities

We examine Moirai's ability to *prioritize* a workload using the example of a VM with one SQL Server instance running both TPC-E and TPC-H. The corresponding database files, "TPCE.VHD" and "TPCH.VHD" each have a footprint of 50GB and are stored on *Virtual Hard Drives* (VHDs) on two separate disk-based storage servers.

We run two experiments, one with default caching and one where we use Moirai to prioritize the TPC-E workload, as explained in Section 3.1 and measure the throughput (transactions/min) for the TPC-E workload. The results are shown in Figure 5.

We observe that in the system without Moirai, TPC-E's performance drops by more than 5X when TPC-H runs. On the other hand, we find that with Moirai, TPC-E's throughput running alongside TPC-H is within 2.3 % of its throughput running by itself.

Note that our current implementation of Moirai results in a data plane overhead of 20% (this difference is due to using our user-level cache vs. SQL Server's native cache, which is heavily optimized). The overhead stems in part from extra memory copies between the kernel and the user-level cache. However, we believe that this overhead is acceptable compared to the 5x drop in performance with today's caching infrastructure. Further, note that the controller can detect when no other workloads are running and remove the user-level cache and thus avoid the extra overhead.

5.2 Maximizing Global Hit Rate

We consider the example of maximizing global hit rate using four tenants with 30 VMs each, spread over 10 hypervisors accessing VHDs on an SSD-based storage server. Each tenant's VM uses IOMeter, parameterized with the key characteristics of the Hotmail workloads (Tenants 1-4 are running the Index, Data, Msg and Log workloads respectively).

We compare two approaches of dividing up the cache space. In the first we divide space equally among the four tenants. In the second we use the method described in Section 3.3 to partition the cache and reconfigure the data plane. The results are shown in Figure 2(b).

Interestingly, we observe not only that overall throughput increases by more than 2.5x, but also that this improvement comes at no cost to any of the individual four tenants. The reason is that all tenants benefit from the decreased load at the storage back-end.

We have experimented with other workload combinations as well. In the worst case across all experiments the overall throughput still increased by 35%, but this came at the cost of a small penalty to one tenant, whose throughput dropped by 10%. A cloud provider could feed into the controller a minimum amount of cache space or minimum hit rate it wants to guarantee each workload, and then ask it to divide the remaining cache space to maximize global utility.

5.3 Consolidating Memory Over Fast Networks

In this section we use Moirai on a TPC-H workload running on ten different hypervisors to illustrate the trade-offs for memory consolidation over fast networks. We compare the case where Moirai is used to insert a 50GB cache inside each of the 10 hypervisors, to the case where Moirai inserts one shared 50GB cache at the storage server, which is either accessed at 1Gbps over TCP or at 40Gbps over RDMA. In all cases, all the data resides in memory (100% hit rate). The results are shown in Figure 6.

We observe that the average latency overhead when using a consolidated cache over the fast network is around 26%, compared to using local hypervisor caches. For the slow network the overheads are 153%. Note that in exchange for paying these overheads one gains a 10X reduction in the total amount of cache space allocated for this workload. Also note that with Moirai a provider has the option to seamlessly switch from one cache configuration to another, depending on the state of the system. For example, a provider might switch to a consolidated cache at the cost of some latency penalties when cache space is scarce.

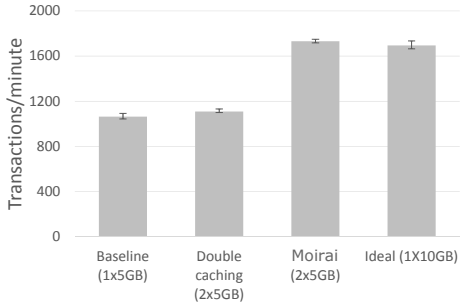


Figure 7: Splitting IOs for TPC-E across two different caches. Today, "double caching" occurs since all IOs flow through all caches. Moirai can prevent this, and match the performance of an aggregate cache.

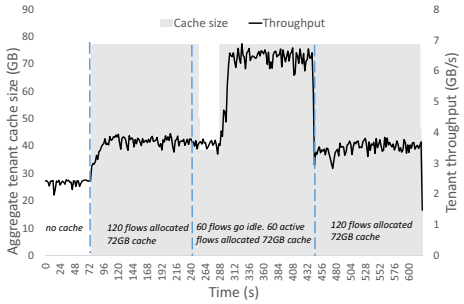


Figure 8: Moirai adapting to workloads dynamically over time. Note there are two y-axis.

5.4 Scaling Out Caches

In this section, we evaluate Moirai's ability to scale out the storage cache as described in Section 3.5. We consider a TPC-E workload on a machine low on memory. The provider wishes to scale out TPC-E's 5GB hypervisor cache to include another 5GB at the storage server.

Figure 7 shows the results when allocating the split cache with and without Moirai. Without Moirai there is little benefit to adding a 5GB cache at the storage server (second bar from the left), compared to having only the 5GB cache at the hypervisor (left-most bar), due to double caching. On the other hand when setting up the two caches with Moirai (third bar from the left), performance is similar to that of an aggregate 10GB cache at the hypervisor (right-most bar).

5.5 Dynamic Workloads

The controller in Moirai continuously monitors the metrics and dynamically reacts to changes after some reaction time *s*, a configurable parameter. For example, Moirai will detect when a cache goes unutilized and reuse the space accordingly. We have worked with values for *s* on the order of 15-30 seconds - we believe that this range presents a good trade-off between responsiveness and unwarranted reconfigurations due to momentary changes in workload demand.

To illustrate Moirai's dynamic capabilities, we evaluate a setup consisting of 10 hypervisors each with 12 VMs, where each VM has a 2GB file stored on an SSD back-end that it accesses through IOMeter. The provider uses Moirai to allocate a total of 72GB cache at the hypervisor level, which is evenly split between VMs (i.e., each receiving 72/120 GB).

Figure 8 shows what happens if half of the VMs on each hypervisor go idle for some time, before they become active again at a later point. Moirai detects when the VMs go idle and re-computes the cache allocation for each VM to distribute spare capacity, hence improving the performance of the active VMs. Once all VMs become active again, Moirai re-computes the initial allocations, and performance goes back to previous levels.

5.6 Control Plane Overheads

We now consider Moirais overheads on the control plane. Moirai implements a version of SHARDS [44] in the Metrics Engine to construct hit ratio curves at runtime. While our current implementation is not as highly-optimized, the original SHARDS paper showed that hit ratio curves with very high fidelity can be constructed online using less than 10MB of memory per workload [44], and marginal (less than 5

We also evaluated the time it takes to compute the optimal cache size allocation as a function of the number of VMs in the system (Algorithm 3.1, described in Section 3.3). We varied the number of VMs from 100 to 5000, and measured the time it took to compute the allocation. For 5000 VMs, it took less than 15s to make that decision, with a water-filling constant ε of 1MB. For ε of 2MB, the time is less than 5s. This highlights the tradeoff between how fine-grained the cache allocation is, and the completion time for the allocation algorithm. However, since cache allocations can feasibly be done at granularities (ε) coarser than 2MB, and they do not need to be re-computed at very short time intervals, we believe this method of cache allocation is very reasonable. We are currently exploring optimizations to reduce the algorithms runtime further.

6. Related work

Application caches. There has been much work recently on caches in data centers. Much of it focused on specialized *application* caches, such as Facebook's photo-serving stack [20], Facebook's social graph store [3], *memcached* [12], or explicit cloud caching services [6, 7]. In contrast, our work is on *system* storage caches for hosting cloud providers that run arbitrary workloads.

System caches. Work on system caches has focused on efficient use of memory for virtual machines through ballooning and sharing techniques [18, 29, 43], which are implemented in state-of-the-art hypervisors like VMware's and Hyper-V. Our work focuses on other caches in the system, beneath the VM abstraction.

Cache replacement policies Some prior work has focused on isolating the cache effects of streams with different access patterns (sequential versus looping) within the

same workload from each other [8, 13, 23, 27]. However, these policies are not workload or tenant aware and cannot prevent a more aggressive workload from occupying more than its fair share of cache. Moreover, each of these policies might actually work better when applied in the context of Moirai, where a cache policy works on per workload segregated cache, as patterns of different workloads don't get interspersed and hence might be easier to detect. Others propose methods to detect changes in workload patterns and dynamically adjust the caching policy used by the system [14]. Moirai provides a perfect vehicle for implementing such an approach and it would be interesting to extend it to support such functionality. Yet another line of work [19, 25] proposes that applications explicitly manage their cache space and its contents, while our goal was to provide a solution that is transparent to the application.

Inefficiencies in cache hierarchies Several other papers have addressed the problem of inefficiencies in cache hierarchies, e.g., some [5, 26] pass hints from the client to better inform caching decisions at the storage server and others [46] extend the SCSI command set by a demote command to avoid double caching. Our goal was a solution that does not require application or VM support, or changes to existing protocols.

Software defined storage. Similar to recent work on software-defined networking (SDNs) [4, 11, 21, 24, 31, 40, 47] and storage (SDS) [38], our architecture is controller-based with a separation between the data and the control plane. Moirai's implementation uses IOFlow's [38] mechanisms for traffic classification, however Moirai's implementation required extensions to IOFlow, e.g., to support arbitrary inspection and manipulation of IO request data, as well as the implementation of the three core components Moirai comprises (as described in Section 2 and Section 4).

7. Summary

Caches are a critical resource in data centers. They improve latency, throughput and reduce the load on networks and storage. But today, caches are implicit, not designed for controlled sharing, leading to severe inefficiencies under multi-tenancy. This paper presents Moirai, a software-defined caching architecture that enables control of caches in a multi-tenant data center. Moirai is transparent to hosted tenants. Their throughput and latency benefit without requiring any tenant input or hints. We show using several different use cases how Moirai can help ease the management of the distributed caching infrastructure and enable the provider to achieve a series of different objectives. We hope that our public release of the code [28] implementing Moirai will help foster future work in this area.

References

[1] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, Aug. 2011.

[2] J.-P. Billaud and A. Gulati. hClock: Hierarchical QoS for packet scheduling in a hypervisor. In *EuroSys*, Apr. 2013.

[3] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of ACM SIGCOMM*, Kyoto, Japan, 2007.

[5] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 145–156, New York, NY, USA, 2005. ACM.

[6] G. Chockler, G. Laden, and Y. Vigfusson. Data caching as a cloud service. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, LADIS '10, pages 18–21, New York, NY, USA, 2010. ACM.

[7] G. Chockler, G. Laden, and Y. Vigfusson. Design and implementation of caching services in the cloud. *IBM Journal of Research and Development*, 55(6):9:1–9:11, Nov 2011.

[8] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An implementation study of a detection-based adaptive block replacement scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 18–18, Berkeley, CA, USA, 1999. USENIX Association.

[9] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 127–141, Stockholm, Sweden, 1985. VLDB Endowment.

[10] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, Seattle, WA, 2014. USENIX Association.

[11] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *Proceedings of ACM SIGCOMM*, Hong Kong, 2013.

[12] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[13] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association.

[14] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, I. Ari, and I. A. . Adaptive caching by refetching. In *In Advances in Neural Information Processing Systems 15*, pages 1465–1472. MIT Press, 2002.

[15] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *FAST*, Feb. 2009.

[16] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, Oct. 2010.

[17] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *ACM CoNEXT*, Nov. 2010.

[18] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[19] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of ACM ASPLOS*, Boston, Massachusetts, USA, 1992.

[20] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of ACM SOSP*, Farminton, Pennsylvania, 2013.

[21] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of ACM SIGCOMM*, Hong Kong, China, 2013.

[22] V. Jeyakumar, M. Alizadeh, D. Mazires, B. Prabhakar, and C. Kim. EyeQ: Practical network performance isolation at the edge. In *NSDI*, Apr. 2013.

[23] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 9–9, Berkeley, CA, USA, 2000. USENIX Association.

[24] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of USENIX OSDI*, Vancouver, BC, Canada, 2010.

[25] C.-H. Lee, M. C. Chen, and R.-C. Chang. Hipec: High performance external virtual memory caching. In *Proceedings of USENIX OSDI*, Monterey, California, 1994.

[26] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-tier cache management using write hints. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[27] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[28] Microsoft. Microsoft research storage toolkit. `http://research.microsoft.com/en-us/downloads/230b8ad4-3340-4a87-8ef0-cf92b376db86/`.

[29] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[30] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *ACM SIGCOMM*, Aug. 2012.

[31] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, S. Vyas, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM*, Hong Kong, 2013.

[32] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 28:1–28:14, New York, NY, USA, 2014. ACM.

[33] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the datacenter network. In *NSDI*, Mar. 2011.

[34] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, Oct. 2012.

[35] SNIA. Exchange server traces. `http://iotta.snia.org/traces/130`.

[36] G. Soundararajan, J. Chen, M. A. Sharaf, and C. Amza. Dynamic partitioning of the cache hierarchy in shared data centers. *Proc. VLDB Endow.*, 1(1):635–646, Aug. 2008.

[37] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, Sept. 1992.

[38] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstrow, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of ACM SOSP*, 2013.

[39] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of Eurosys*, pages 169–182, Salzburg, Austria, 2011.

[40] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of USENIX NSDI*, NSDI'06, San Jose, CA, 2006.

[41] TPC Council. TPC-E. `http://www.tpc.org/tpce/`.

[42] TPC Council. TPC-H. `http://www.tpc.org/tpch/`.

[43] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[44] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, Feb. 2015. USENIX Association.

[45] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI 14), Broomfield, CO, Oct. 2014. USENIX Association.

[46] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of USENIX ATC*, Monterey, California, 2002.

[47] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: a 4D network control plane. In *Proceedings of USENIX NSDI*, Cambridge, MA, 2007.