utmem: Towards Memory Elasticity in Cloud Workloads

Aimilios Tsalapatis, Stefanos Gerangelos, Stratos Psomadakis, Konstantinos Papazafeiropoulos, and Nectarios Koziris

Computing Systems Lab
National Technical University of Athens
Athens, Greece
{etsal,sgerag,psomas,kpapazaf,nkoziris}@cslab.ece.ntua.gr

Abstract. In environments where multiple virtual machines are colocated on the same physical host, the semantic gap between the host and the guests leads to suboptimal memory management. Solutions such as ballooning are unable to modify the amount of memory available to the guest fast enough to avoid performance degradation. Alternatives such as Transcendent Memory allow the guest to use host memory instead of swapping to disk. All these techniques are applied at the memory management subsystem level, resulting in cases where abrupt changes in memory utilization cause unnecessary guest-side swapping. We propose Userspace Transcendent Memory (utmem), a version of Transcendent Memory that can be directly utilized by applications without interference from the guest OS. Our results demonstrate that our approach succeeds in allowing the guests to rapidly adjust the amount of memory they use more efficiently than both ballooning and Transcendent Memory.

Keywords: Virtualization · Virtualized Memory Management · Transcendent Memory

1 Introduction

In cloud computing environments, the lack of cooperation between the guest OSes and the hosts incurs systemwide performance penalties. Because each guest has its own internal mechanisms for managing its resources, unaware of the environment it is running in, all VMs concurrently try to optimize the resources allocated to them as if they were running alone in a physical server. Moreover, hosts are oblivious to any resource management on the guests' side, unless the latter have a paravirtualization mechanism with which they can coordinate with the former [1].

In the case of paravirtualized memory management, the guests and the host use approaches such as memory balloons, which let them pass ownership of physical pages to one another. The balloon, which resides in the guest, tries to use as much guest physical memory as possible. Since the balloon never actually uses the memory that it allocates, the guest's maximum memory usage is reduced.

As a result, it cannot put as much memory pressure on the host. When the guest is in need of memory, it signals to the balloon to free some of the pages it has been given. These can then be once again used by the guest, and the host once again needs to back them with actual memory.

Ballooning allows for dynamically modifying the amount of guest memory, but it is often inefficient for a number of reasons. One of them is that traditional operating systems tend to use as much memory as they can, because they are designed to be the sole tenants of their machine. As a result, the balloon driver constantly competes with the rest of the system for guest physical memory. This causes slow reaction times to spikes in memory usage, degrading overall performance [2]. Apart from that, ballooning also tends to fragment the guest's physical memory map by grabbing free memory areas that rest between used ones, creating holes in the address space [3].

An alternative method for dynamically adjusting the size of guest memory is Transcendent Memory [4] (tmem). Utmem modifies the amount of memory available to the guest without deconstructing and reconstructing the virtual machines' physical address space. The core of the idea is that the guest directly manages only part of the memory allocated to it, and has to indirectly use the rest through requests to the host. As a result, the latter can more easily impose policy, since it does not need to wait for the guests to comply to its requests to redistribute the machine's resources, as in the case of ballooning [5]. This method has up to now found applications as part of the guest's memory and I/O subsystems. An example is the *frontswap* module [6] for Linux guests, which intercepts attempts to swap a page to disk and tries to store it in a tmem pool in the host. Another one is *cleancache*, which functions as a page cache for the guest, but is managed by the host.

While efficient, tmem suffers from the limitation that it cannot be used directly by applications. This in turn means that every tmem request must pass through the I/O subsystem before being serviced. Moreover, the usage patterns of the memory pool are dictated by the swapping subsystem, with the workloads being unable to directly determine which data gets sent to the pool. Current tmem implementations are also confined in the I/O subsystem, and can therefore only be used for storing the kinds of data present in the subsystem, like disk blocks and physical pages.

In this paper we present Userspace Transcendent Memory (utmem), a mechanism which can be leveraged by guest workloads to achieve memory elasticity without the need of balloon drivers. Due to utmem's design, memory does not change ownership between the host and the guests. Performance thus does not degrade as memory usage rises, and in fact stays fixed regardless of the total amount of memory that the guest uses throughout the system.

We also demonstrate that, for guests which consume large amounts of memory, our design performs better under sustained memory pressure than existing tmem mechanisms. The cause of this speedup is that our solution completely avoids traversing the I/O stack, in contrast to ordinary tmem.

In this work we implement the utmem mechanism and prove its efficiency, providing the following contributions:

- We expand the KVM hypervisor to support tmem operations. We do so by adding a new hypercall, as well as support for directly manipulating the system's tmem pools from userspace. We also introduce tmem backend functionality directly to the Linux operating system.
- We construct a device on the guest kernel that exposes the aforementioned pools' functionality in a way directly usable by workloads. We demonstrate this functionality by introducing it to the Redis key-value store.
- We demonstrate that utmem-capable applications demonstrate significant speedups over unmodified applications in cases where guest memory consumption is close to its memory limit.

2 Background

The tmem mechanism can be used to streamline memory management in systems with multiple guests. It specifically addresses the problem of fluctuating guest memory usage, which causes significant performance penalties in systems that rely solely on ballooning. The concept is described in the paper by Magenheimer et al. [7].

Tmem is a key-value store that is accessed by the guests through explicit requests to the host. This store is created by gathering spare memory in the host to create a shared resource called a pool. One of the defining properties of tmem is having its data gated off behind its two main calls: `PUT` and `GET`. By having well-defined entry points, tmem is able to decouple the implementation of its pools from the interface to the users. As a result, the data in the pools can be represented in arbitrary ways, like for example in a compressed form [8]. Even if the data is stored locally and as-is (as in the case of the Xen hypervisor [9]), the guests still cannot access them without a request to the hypervisor. This means that the host remains in control of the system's memory at all times.

Figure 1 demonstrates the location of the tmem pools and the guest-side buffers in the Xen implementation. Each guest has its own tmem frontend (a) , which is the only guest-side component aware of the tmem mechanism. It communicates with the host via hypercalls (b), which are then serviced by the hypervisor in the backend (c). The latter stores the data to (or retrieves it from) its tmem pools.

These pools can be private or shared, depending on whether the data of a guest can be accessed by another one. The tmem specification also describes ephemeral and persistent pools, depending on the permanence of the data stored in them. Because the former may discard the data stored in them at any given time in order to service the rest of the system's memory needs, they can be used to build entities such as victim caches. In contrast, the latter are guaranteed to keep the information stored indefinitely, and are more appropriate for creating high-speed tmem alternatives to more costly storage methods. In this paper we focus on persistent memory pools.

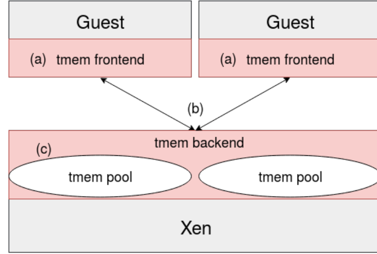

Fig. 1. The location of the pools in the Xen tmem implementation

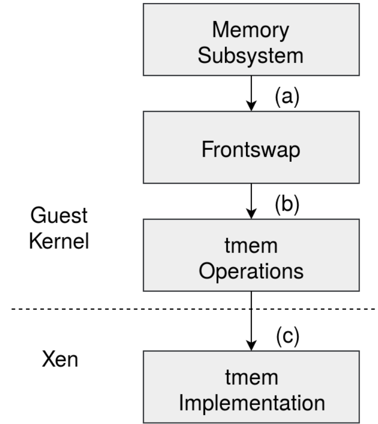

Fig. 2. The layers of a typical frontswap-capable system

Tmem frontends include the *frontswap* and *cleancache* modules for Linux guests running on top of Xen. These consist of hooks on the I/O subsystem that can be used to make a tmem call to the host, instead of a costlier operation like a disk access.

Frontswap's purpose is to lower the rate of swapping and guest disk I/O in general [6]. Figure 2 demonstrates the relationship between the guest's memory subsystem, frontswap, and tmem. Before the guest resorts to swapping to disk, it first sends a request to frontswap (a) to save the page. The module attempts to do so by performing a `PUT` which amounts to looking up the implementation of the operation, typically a hypercall (b). It then makes the appropriate request to the backend (c), which stores the page if it has enough memory, or denies the request, if it does not. The result is passed to frontswap, which then informs the guest kernel whether the operation succeeded. If it did, then there is no need to write the page to swap. As a result, the cost of accessing the guest's disk is replaced by the cost of a single VM exit.

3 Overview

3.1 Design

The limits of tmem are currently defined by its use cases. More specifically, the whole mechanism has been used exclusively in the confines of the memory management subsystem of the guest. As a result, every tmem call presently incurs a performance overhead due to traversing the I/O stack of the guest.

This is not a penalty inherent to the tmem mechanism. In fact, one of its main advantages over approaches like swapping to host-side ramdisks is that successful stores simply consist of a hypercall, without passing through the swap I/O subsystem. One of the main goals of our framework is to completely bypass

the I/O stack of the guest and access host memory through hypercalls, resulting in an overall faster and simpler system.

The main design objective of utmem is to integrate easily with applications which access data by means of explicit storage and retrieval API calls. The mechanism can be interposed between the application and the API, replacing the functions which store or retrieve information. Our goal is to have a design where the host provides a backing store for arbitrary data. The store should be completely opaque to the guests, comprised of discrete tmem pools, and accessible only through tmem calls.

Utmem is split into three parts, which communicate with each other using tmem requests. These parts are:

- The utmem device, which exposes the utmem API to userspace and dictates policy
- The tmem frontend, which implements the communication method with the backend (hypercalls, networking, etc.)
- The tmem backend, which implements the store and services requests

The present implementation is generic and can work with arbitrary data, including physical pages and key-value pairs. It is also superior to both native applications and existing tmem-based solutions in terms of performance. We achieve this speedup without utilizing domain-specific information, like for example the nature of the data being stored. Given a specific client workload, the mechanism can be configured to utilize this kind of information, in order to deliver additional performance gains with little effort.

3.2 Implementation

For the implementation of utmem, we choose to use the KVM hypervisor. Since it does not currently support tmem, we opt to implement that functionality by adding a hypercall. The additions to the hypervisor are about 150 lines of code, most of it due to the hypercall added. The only modification of existing code is an extra 3 lines of code in the main hypercall servicing loop.

Requests propagate across the components of the utmem mechanism, until they reach a tmem backend which can store the data. In our implementation we create a frontend in the guest, which makes a hypercall for each request. This call is received by the hypervisor in the host, which forwards it to a backend that actually possesses tmem pools. After the value is stored, the hypervisor is notified, and in turn notifies the client in the guest that initiated the operation.

The exact series of calls to be made for a utmem request is described in Figure 3. This stack is usable both by guest userspace processes through the utmem mechanism, as well as from kernelspace tmem users like frontswap.

We have chosen Redis as the workload to be used for the evaluation of this mechanism. As an in-memory key-value store, its access semantics are very similar to those of tmem. Our application communicates with the mechanism using

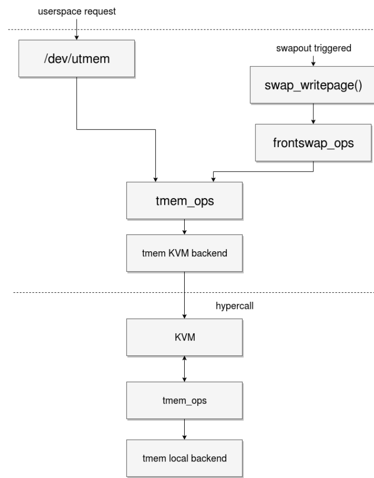

Fig. 3. Architecture of the utmem mechanism

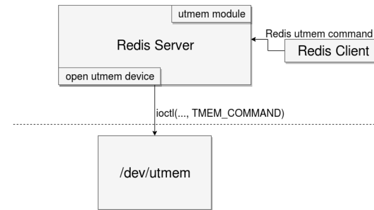

Fig. 4. Architecture of the Redis workload

a device that can be controlled using the `ioctl()` system call. This device is responsible for copying the key-value pairs received from userspace into the buffers used by the guest tmem frontend, and vice versa.

The additions to Redis are shown in Figure 4. They are implemented in a Redis module, which holds the functions which turn the client's requests into `ioctl`s to be sent to the utmem device.

4 Evaluation

To assess our approach, we use the Redis benchmark for the reasons noted above. The utmem mechanism is evaluated on two fronts:

- How well it performs under significant memory pressure. The results confirm our claim that the utmem-capable system is oblivious to guest-side memory pressure, while the unmodified one experiences performance degradation due to swapping.
- How well it performs in situations where memory is abundant. Our results show a slight performance penalty, due to the extra memory copies in our implementation.

To further analyze the behaviour of our approach, we look into a breakdown of the latency for the two basic operations. Our experimental evaluation shows a discrepancy between the latency of `PUT` and `GET`, which stems from their implementation.

We have set up our testbed on a machine with an Intel Xeon X5650 processor with 48GB of RAM. Both the host and the guest are running Linux 4.9.

4.1 Evaluation of utmem under memory pressure

The first set of experiments we performed showcases the advantage of a utmem-capable system against a conventional one, when under pressure. We also demonstrate that utmem exhibits better performance than existing tmem designs.

In the experiment, we fill up the Redis server or the utmem backing store with values of collective size 100% that of the guest's total memory. We keep executing the Redis benchmark until its results converge.

Looking at the results in Figure 5 we confirm that the utmem-capable system outperforms the unmodified one by an order of 2x-3x, depending on the size of the values being stored with each operation. The native case suffers heavy performance degradation due to swapping, even if it is not over its memory limit. On the other hand, utmem's throughput does not lower. Utmem also outperforms systems running unmodified Redis instances that also use tmem by means of the frontswap mechanism. This demonstrates that avoiding disk I/O by using frontswap is not enough to completely avoid performance degradation.

We also benchmarked the guests for working sets of different sizes, with the latter being fractions of total system memory close to unity. Again, by looking at Figure 5 we can see that the performance multiplier of the utmem mechanism against the native case rises together with the size of the working set. Once again, we confirm that frontswap is not suitable for totally mitigating the performance penalties associated with high memory pressure.

4.2 Evaluation of utmem under nonexistent memory pressure

The next experiment we performed concerns the throughput of a system that does not swap. More specifically, we tested the performance of an unmodified and a utmem-capable Redis server. The Redis client is executed outside the guest, in order to avoid influencing the Redis server's performance. There is no memory pressure, because the same key is used for each operation. The benchmark used is the Redis builtin benchmark.

From the graph in Figure 6, we observe that both servers exhibit similar performance. The difference between the two commands is due to the extra copies needed by utmem to finish the storage/retrieval of the value. The performance penalty hovers around 85%, irrespective of the value size. This is expected, considering that the only difference between the operations of different sizes is the number of hypercalls per KB of data moved to the host. The cost of these is negligible compared to the data transfer, so it does not show up in the graph.

In order to look further into the sources of latency for the utmem operation, we create a microbenchmark which is similar to the Redis benchmark. Its function is to continually perform the same operation for 10^5 iterations, using only one key-value pair of size 1024KB. We have normalized the results for the time needed for the microbenchmark to complete for `PUT`, which was 992ms.

Going back to Figure 6, it is obvious from the graph that the costliest part of the operation is copying the data to to and from the backend. The difference between the internal guest call and the guest-to-host communication demonstrates

that the major factor of the operations' latency is the memory copies. Another interesting observation is that there is a slight discrepancy between the two basic operations: `PUT` has a small amount of latency coming from the host backend itself, while the `GET` operation does not. This is because the former needs to allocate memory in order to finish it. The latter does not, because the guest itself provides the buffer into which the data will be written.

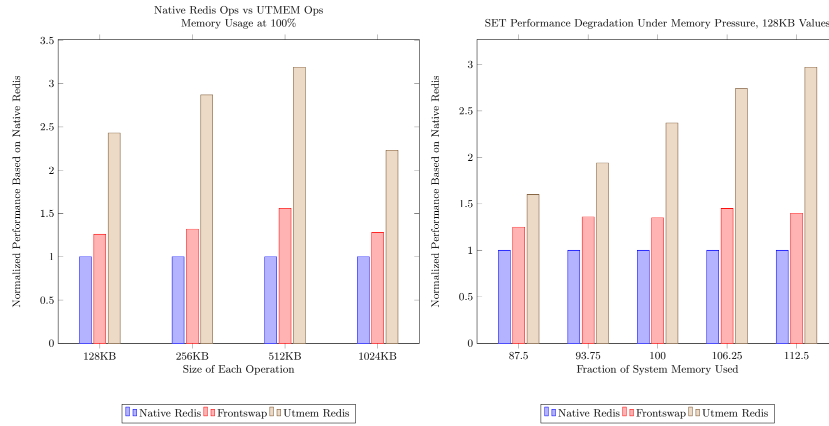

Fig. 5. Performance of native, frontswap-capable, and utmem-capable Redis server for memory usage exactly at and close to unity.

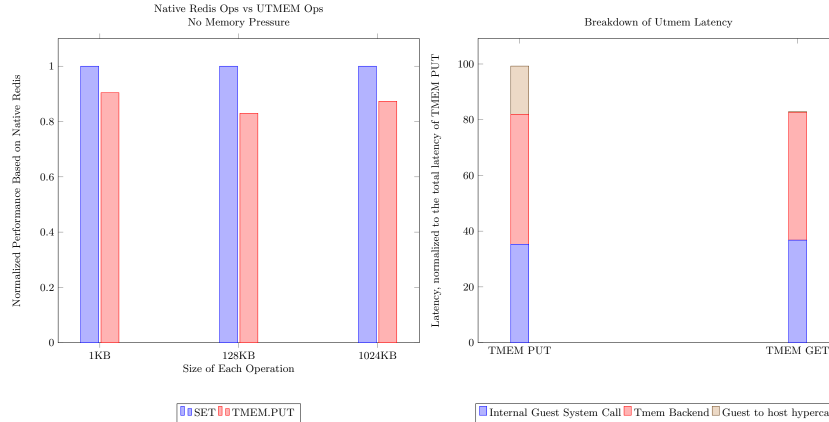

Fig. 6. Left figure: Performance of native and utmem-capable Redis for no memory pressure. Right figure: Breakdown of the latency of the basic utmem operations.

5 Related Work

Transcendent Memory is an idea that has been mainly used to build caches by being able to reserve system memory, which it then uses to serve clients. These can be the guest kernel, or a userspace process. Instances of this design include a two-level cache for systems with nonvolatile memory, as described in [10]. Another example is MemFlex [11], which like frontswap is a cache for swapped-out pages. Unlike frontswap, it is combined with a balloon which deflates when the tmem pool starts getting used, in order to swap the pages in the cache back in, thus freeing host memory.

Another design that draws from the aforementioned work is Mortar [2], which focuses on pools similar to those created by ephemeral tmem. The memory accumulated in the pools is used to build shared caches, which store data that also exists elsewhere. As a result, the caches can shrink instantaneously when memory pressure intensifies. The goal of this approach is to maximize memory utilization by making use of spare resources.

Research efforts have been made to make ballooning more responsive and efficient. These designs aim to avoid needing what amounts to a memory buffer that absorbs spikes in memory demand, in the form of tmem and similar mechanisms. One such approach includes introducing ballooning at the application level [12], so that applications in the guest that manage their own memory, like language runtimes and other middleware, directly control the balloon that exists in the kernel. Another effort to make ballooning more efficient is iBalloon [13], where efficient ballooning is treated as a learning task, with reinforcement techniques being applied to teach the mechanism to reach the desired state in the shortest amount of time.

6 Conclusion and Future Work

6.1 Future Work

While the mechanism presented is already competitive in regards to its performance, there are possible improvements that can be made. An example would be having different backends in the host, each of them with its own tradeoffs, depending on the use case. Another potential expansion is to add a deduplication mechanism for the data stored. Since the values can be objects other than pages, guests running on the same host that store values of the same kind could end up sharing a lot of data, which in turn means that the system's memory limits could be expanded further.

While our approach is appropriate for handling large amounts of data, when used for storing smaller structures the performance penalty from the hypercall may become more apparent. A possible solution to that is to coalesce multiple tmem requests into larger ones, and only flush the data when needed. This would aid in avoiding superfluous VM exits, by keeping inside the guest values that do not take up significant space in memory.

When it comes to combining in-guest storage with utmem, another extension could be an adaptive mechanism that determines whether a value will be stored in the former or the latter. This mechanism would take into account how often a value is accessed, as well as the guest's and the host's memory load.

6.2 Conclusion

In this paper we present utmem, a mechanism for enabling the host to efficiently and flexibly manage system memory. We design utmem by extending the tmem architecture to include a device for communicating directly with userspace. This approach enables the utilization of existing tmem users in KVM, like frontswap.

In order to implement the mechanism, we create a backing store for the tmem service that can reside in Linux itself. We add a new hypercall to the KVM hypervisor, and implement a module in the guest that we can use to send tmem requests from guest userspace to the host.

We evaluate our mechanism and demonstrate that in setups where the guest is under significant memory pressure, the applications which use utmem outperform their native counterparts by a factor up to 3. In the worst case scenario, utmem-aware applications are shown to incur an overhead of 20%, due to the extra data copy needed by our implementation.

References

1. R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.
2. J. Hwang, A. Uppal, T. Wood, and H. Huang, "Mortar: Filling the gaps in data center memory," *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 53–64, 2014.
3. H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C.-z. Xu, "Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines," *IEEE Transactions on parallel and distributed systems*, vol. 26, no. 5, pp. 1350–1363, 2015.
4. "Transcendent Memory in a Nutshell." `https://lwn.net/Articles/386090/://lwn.net/Articles/454795/`. [Online; accessed 26-February-2018].
5. W. Hwang, Y. Roh, Y. Park, K.-W. Park, and K. H. Park, "Hyperdealer: Reference-pattern-aware instant memory balancing for consolidated virtual machines," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 426–434, IEEE, 2010.
6. "Cleancache and Frontswap." `https://lwn.net/Articles/386090/`. [Online; accessed 26-February-2018].
7. D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, pp. 191–200, Citeseer, 2009.
8. S. Jennings, "Transparent memory compression in linux," 2013.
9. D. Magenheimer *et al.*, "Transcendent memory on xen," *Xen Summit*, pp. 1–3, 2009.
10. V. Venkatesan, W. Qingsong, and Y. Tay, "Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pp. 966–973, IEEE, 2014.

11. Q. Zhang, L. Liu, G. Su, and A. Iyengar, "Memflex: A shared memory swapper for high performance vm execution," *IEEE Transactions on Computers*, vol. 66, no. 9, pp. 1645–1652, 2017.
12. T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 337–350, ACM, 2013.
13. J. Rao, X. Bu, K. Wang, and C.-Z. Xu, "iballoon: Self-adaptive virtual machines resource provisioning,"