System-level Implications of Disaggregated Memory

Kevin Lim   Yoshio Turner   Jose Renato Santos   Alvin AuYoung
Jichuan Chang   Parthasarathy Ranganathan   Thomas F. Wenisch*
HP Labs   *University of Michigan, Ann Arbor
{kevin.lim2, yoshio.turner, joserenato.santos, alvin.auyoung, jichuan.chang, partha.ranganathan}@hp.com, twenisch@umich.edu

Abstract

Recent research on memory disaggregation *introduces a new architectural building block—the* memory blade*—as a cost-effective approach for memory capacity expansion and sharing for an ensemble of blade servers. Memory blades augment blade servers' local memory capacity with a second-level (remote) memory that can be dynamically apportioned among blades in response to changing capacity demand, albeit at a higher access latency. In this paper, we build on the prior research to explore the software and systems implications of disaggregated memory. We develop a software-based prototype by extending the Xen hypervisor to emulate a disaggregated memory design wherein remote pages are swapped into local memory on-demand upon access. Our prototyping effort reveals that low-latency remote memory calls for a different regime of replacement policies than conventional disk paging, favoring minimal hypervisor overhead even at the cost of using less sophisticated replacement policies. Second, we demonstrate the synergy between disaggregated memory and content-based page sharing. By allowing content to be shared both within and across blades (in local and remote memory, respectively), we find that their combination provides greater workload consolidation opportunity and performance-per-dollar than either technique alone. Finally, we explore a realistic deployment scenario in which disaggregated memory is used to reduce the scaling cost of a* `memcached` *system. We show that disaggregated memory can provide a 50% improvement in performance-per-dollar relative to conventional scale-out.*

1. Introduction

The continued growth of large-scale scientific computing, data-centric workloads, and virtual machine consolidation all point to a need for greater memory capacity in volume server systems. Whereas Moore's law continues to provide rapid increases in the number of cores per chip, significant cost and technical barriers are impeding commensurate scaling of memory capacity. It is expected that this discrepancy between supply and demand for memory capacity per core will increase by 30% every two years, particularly for commodity systems [11].

Disaggregated memory [11] has been proposed as a cost-effective approach to address the impending memory capacity scaling wall by enabling *independent scaling* of compute and memory capacity. Memory disaggregation introduces a new architectural building block—the memory blade—which augments blade servers' local memory capacity with a second-level (remote) memory that can be dynamically apportioned among blades within an enclosure in response to changing capacity demand. The memory blade comprises a large pool of commodity memory, an ASIC memory controller, and interface logic for communication between memory and compute blades. Software support for memory blades can be provided in a hypervisor to abstract the local and remote memory regions, transparently presenting a large address space to guest operating systems.

While disaggregated memory has been identified to be promising [3, 11], prior work has evaluated disaggregated memory only through simulation, using statically defined estimates of software overheads. Central software implementation issues, such as synchronization in key hypervisor functions, system-level interaction with I/O devices, and the role of software memory optimizations—all of which can substantially impact performance—were not considered.

In this paper, we build on the prior research to explore the software and systems implications of disaggregated memory. We develop a software-based prototype by extending the Xen hypervisor to emulate a disaggregated memory design wherein remote pages are swapped into local memory on-demand upon access. Our prototype faces (and addresses) the same software challenges (e.g., correct locking, I/O, page replacement, etc.) as a complete implementation, and allows us to obtain performance estimates with accurate software overheads for full-scale workloads. We use the prototype to explore two aspects of memory disaggregation critical to its performance and efficacy: the

To appear in 18th International Symposium on High Performance Computer Architecture

page replacement policy and the interaction of disaggregation and content-based page sharing. We then carry out a case study investigating the effectiveness of memory disaggregation at reducing scaling cost for memcached servers, a critical infrastructure component of online data-intensive web-based services.

Our prototyping effort reveals that low-latency remote memory calls for a different regime of replacement policies than conventional disk paging, favoring mechanisms that minimize time spent in the hypervisor, even at the cost of using less sophisticated replacement policies. Whenever a remote address is accessed, the associated page is transferred to local memory. If local memory is full, a local page must be evicted to remote memory. Although both operating systems and hypervisors already have policies for page eviction, these schemes were designed for infrequent paging to a hard disk with millisecond-scale transfer latency. Under memory disaggregation, page swapping is frequent and transfers incur microsecond-scale latencies (4 μs across PCIe 2.0 x2). We find that the overhead of conventional OS page replacement (e.g., the widely-used Clock algorithm) causes it to underperform simpler, faster alternatives.

We next consider the interaction of memory disaggregation with content-based page sharing (CBPS) [22, 9], an orthogonal, software-based method for increasing effective memory capacity that is already available in many hypervisors. CBPS uses the hypervisor's virtual memory mechanisms to store only a single copy of pages with identical content, but incurs extra computation to detect and merge pages and unmerge them upon writes. CBPS and memory disaggregation are synergistic in two ways. First, CBPS in each blade's local memory increases effective local capacity, reducing the frequency of accesses to remote memory. Second, CBPS at the memory blade allows infrequently-accessed pages from the footprints of *different compute blades* to be merged, greatly increasing the scope over which CBPS can be applied. Our study shows that using the two techniques together on a single server provides greater opportunity for workload consolidation and achieves performance within 5% of a system with large local memory at 25% lower cost.

Finally, we explore a realistic deployment scenario in which disaggregated memory is used to reduce the scaling cost of a memcached system. We design a new, data-intensive workload using access trace logs from Wikipedia [20] and mimic their Web-infrastructure to test the memcached caching layer. Memcached is an in-memory key-value store widely used in interactive internet services to cache dynamic web content and reduce load on slower (but consistent/durable) backing stores (e.g., database servers). As both the size and traffic of an internet service like Wikipedia grows, the memcached layer must be scaled up to maintain high cache hit rate and low average service response time. We contrast memory disaggregation with conventional scaling (by adding additional servers) showing a 50% improvement in performance-per-dollar relative to conventional scale-out on a realistic deployment scenario for memcached.

The rest of this paper is organized as follows. Section 2 reviews prior work on disaggregated memory hardware architecture. Section 3 presents the design of our software-based prototype, while Section 4 examines page replacement policies. Section 5 examines the interaction of CBPS and disaggregated memory for a server consolidation workload. Section 6 presents our case study of disaggregation for a memcached system. Finally, we discuss related work in Section 7, and conclude in Section 8.

2. Disaggregated Memory Background

Disaggregated memory expands the memory hierarchy to include a remote level provided by separate memory blades connected over a high speed backplane. Breaking the typical collocation of compute and memory, disaggregated memory allows for independent scaling of compute and memory capacity. Whereas commodity servers are limited in their memory capacity by memory performance requirements and technology scaling trends, memory blades are designed to provide large capacity through the use of buffer-on-board [10] or other fan-out techniques. Each memory blade can connect to multiple servers, dynamically allocating its capacity among those servers. This functionality allows the cost of the memory blade to be amortized across the servers.

A server's local and remote memory together constitute its total physical memory. An application's locality of memory reference allows the server to exploit its fast local memory to maintain high performance, while remote memory provides expanded capacity with increased access latency (but still orders of magnitude faster than accessing persistent storage). We assume industry-standard PCI Express (PCIe) links between the compute and memory blades, hence, interconnect transfer latency dominates remote memory access time. As a consequence, using slower (and potentially more energy efficient) DRAM for remote memory has relatively little performance impact, providing hardware designers the luxury to trade off remote memory bandwidth for increased capacity and power efficiency. In essence, disaggregated memory maintains high performance using fast local memory, while improving capacity and power efficiency using slower remote memory.

Remote memory access can be supported either in hardware at cache-line granularity through the cache coherence protocol, or in software by swapping with local pages through explicit DMA transfers. In this paper, we build

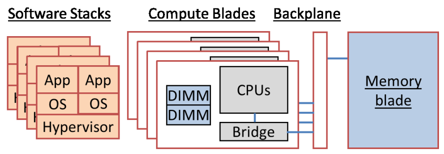


Figure 1. System with memory blades

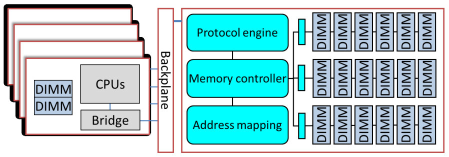


Figure 2. Memory blade architecture

a software prototype of the page-swapping disaggregated memory design, introduced in our prior work [11], because it does not require hardware changes to compute blades and, as we demonstrate, performs well. Access to data stored in remote memory results in a hypervisor trap which transfers the entire remote page to local memory. If local capacity is exhausted, a local page is evicted to remote memory.

Figure 1 shows this disaggregated memory design consisting of a blade enclosure housing multiple compute blades. The compute blades connect over the backplane to one or more memory blades through a PCIe bridge and use the standard I/O interface to access the memory blade. Specifically, memory pages read from and written to the remote memory blade are transferred over the shared high-speed PCIe interconnect in the blade server enclosure. This design maintains cost-efficiency by reusing the pre-existing commodity infrastructure and amortizing the cost of the memory blade across multiple compute servers.

The memory blade, shown in Figure 2, comprises a custom ASIC or lightweight processor, storage for address mapping tables, a PCIe bridge for connecting over the backplane, and multiple DIMMs. The lightweight processor/controller responds to memory accesses as they arrive from the PCIe interconnect. Servers accessing remote memory send a request for a memory location within their own address space, and the memory blade maintains address mapping tables to translate the requested address and requesting server ID to the address of the data on the memory blade. In addition, the memory blade's processor runs management software that coordinates dynamic capacity allocation, as well as discovery and setup phases for initializing the remote memory allocations as servers are booted up.

3. Prototype System Architecture

To reach broad acceptance in the commodity market, disaggregated memory needs to provide benefits to legacy OSs and applications with minimal code modification. We built a software-based prototype by extending a hypervisor to support remote memory to enable arbitrary OSs and applications running in virtual machines to take advantage of the expanded capacity of remote memory without any code changes. Note that in cases where such design constraints are not required, disaggregated memory can be implemented directly in the operating system, avoiding virtualization overheads.

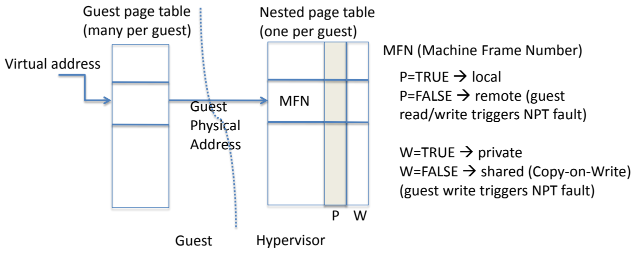


Figure 3. Page tables in our prototype

Our prototype system extends the Xen 3.4.1 hypervisor to support remote memory as a guest-transparent demand-paging store. The prototype sets aside a portion of a server's local memory to act as an emulated remote memory blade with a configurable delay, which mimics the transfer latency of accessing remote memory. A future real memory blade could be substituted for this emulation mode by adding a device driver to the hypervisor.

Our extensions leverage support in modern processors for a new level of page table address translation called Nested Page Tables (NPT) by AMD [2] and Extended Page Tables (EPT) by Intel (for simplicity, we generically use the term NPT in the following). The new level of translation provided by NPTs enables fully virtualized guest VMs to maintain and update traditional OS page tables without hypervisor involvement. Guest page tables map guest virtual addresses to pseudo-physical addresses. These PFNs (pseudo-physical frame numbers) are in turn translated by the hardware to host machine physical addresses, called MFNs (machine frame numbers) using the NPTs. The NPTs are managed exclusively by the hypervisor, which maintains one NPT for each guest VM. The page table structure is shown in Figure 3.

We modified NPT handling to allow dynamic setting of page state to local or remote. A real memory blade would perform DMA data transfers over the PCIe interconnect to migrate pages between local and remote. To avoid spending CPU cycles copying page contents for these transfers, the prototype instead keeps track of the remote or local status of each page and mimics the timing behavior of a real memory blade performing DMA data transfers. Delays for accessing remote memory through a common interconnect are inserted using a small delay loop in the hypervisor with a mutual exclusion lock enforcing serialized transfers.

The prototype keeps track of local and remote pages using the NPT and a bit vector. The bit vector (with one bit per each MFN page in the machine) is marked to indicate which pages are remote. In addition, each PFN that has a corresponding MFN page on the remote memory is marked "not present" in the NPT.

A guest memory access to a remote page causes an NPT page fault in the hypervisor (not in the guest) because the page is marked as "not present". A bit vector lookup determines if the page is remote. If the page is remote, the hypervisor retrieves the referenced page from the memory blade, marks the NPT entry as "present", and then resumes the faulting guest virtual CPU (vCPU). In our prototype, the delay for the retrieval (sending the request to the memory blade and receiving the 4 KB page over PCIe) is modeled through a configurable delay loop, allowing us to model links of different speeds; for the remainder of this work we use a 4 μs delay based on a PCIe 2.0 x2 link. We measured our prototype, on average, to provide link delay accuracy within 1% of the configured 4 μs delay. Unlike with paging to disk, access times to remote memory are fast enough that it is not worthwhile to de-schedule a faulting guest VM; instead the VM blocks on a remote access. If a free local page is not available when a remote page needs to be migrated, a local page is selected for eviction and transferred back to remote memory. The policies and exact functionality of the mechanisms are discussed in Section 4.

With real disaggregated memory hardware, device I/O (e.g., disk, or network interface) initiated by a guest virtual machine needs special handling. DMA operations performed by an I/O device should access the correct memory pages, even in the face of memory page transfers between local and remote memory. Since our prototype just keeps track of the remote/local status of pages rather than copying between memory regions, device DMAs could execute normally without any problems. However, for emulation fidelity, we modified the I/O device virtualization layer (QEMU for Xen fully-virtualized guests) to pin pages used for I/O as local memory. QEMU maintains a software cache of pointers to pages used for I/O. We modified this layer to pin all pages as local when they enter the cache, and unpin them when they leave the cache. This same procedure would be needed with real disaggregated memory hardware.

Direct I/O is a high-performance technique that grants VM guests the privilege to directly access I/O devices without hypervisor mediation of each I/O operation. In this case, the pages used for I/O cannot be determined in the virtualization layer, since without special restrictions the guest is free to use any of its pages for I/O without hypervisor involvement. With disaggregated memory, the hypervisor may change page mappings while the I/O device independently performs DMA, leading to possible inconsistent states. Such inconsistencies can be eliminated with appropriate extensions to the memory blade command set and careful coordination of changes to NPT mappings and I/O Memory Management Unit (IOMMU) page table mappings. Extensions include a command to queue any read/write requests from I/O devices for a specific page, and a command to satisfy all queued requests for a page. The hypervisor, on migrating a page to local or remote, will use these extensions to control the I/O updates to the page contents while page data transfer and mapping table updates are in progress, thereby providing logical transparency of local/remote migration to I/O devices. A detailed description of these mechanisms is omitted due to space constraints.

Our hypervisor extensions required only localized changes to Xen's memory management code, demonstrating implementation feasibility. We also built tools to configure remote memory allocations and delay times, and added statistics counters to the hypervisor to instrument remote memory operations.

In the following sections we use the prototype to explore the system-level implication of disaggregated memory. Specifically, we examine the impact of different replacement policies, the synergy with software-based optimizations, and the use of disaggregated memory on an interactive web-workload of emerging interest.

4. Replacement Policies

Disaggregated memory introduces a new level of memory that must be managed by the system. When data stored in remote memory is accessed, the hypervisor must select a local page to be swapped with the remote page if local memory is fully occupied. Two aspects of the replacement policy for choosing the evicted page impact performance: victim selection quality, and victim selection time. For high quality, victim pages should be carefully selected to avoid evicting "hot" pages that will likely be accessed in the near future. Conversely, the amount of time required to select a victim must not exceed the expected benefits of that selection. One straightforward approach is to reuse policies already implemented for swapping pages to disk; however, those policies were designed with different performance constraints and thus may not provide optimal performance.

Comparison to operating system swapping: Operating systems such as Linux swap pages from main memory to backing storage when an application is requesting more memory but main memory is full. The abstraction provided by virtual memory allows physical memory to be overcommitted, using the storage layer to provide the extra memory capacity (known as swap space). Swap space is typically located on disk, with orders of magnitude slower access latencies than main memory (milliseconds versus nanoseconds). Hence, paging policies are optimized to minimize the number of disk I/O operations, heavily favoring victim selection quality over speed.

While disaggregated memory bears similiarity to disk paging—local memory is overcommitted and pages are swapped to a slower, secondary store—key differences motivate reconsidering replacement policies:

- The latency for remote memory is on the order of microseconds, as compared to milliseconds for disk. The latency difference may change the requirements for the replacement policy—the most accurate decision may no longer be advantageous if the decision takes too long to compute.
- The difference in capacity between local and remote memory is typically smaller than between main memory and storage. Having a relatively large pool of local memory lowers the miss rate impact of non-optimal victim section. Considering the low-latency of remote memory, the replacement policy's performance impact is further reduced.
- When swap space is implemented by the OS, the OS can leverage information to select pages that it knows are not performance critical. In the case of disaggregated memory, the replacement decisions take place at the hypervisor, which cannot take advantage of OS information (e.g., which pages are in a file cache).

Although the previous study [11] evaluated two different replacement policy options (Random and LRU), that study assumed a fixed cost (in cycles) for victim selection and did not implement the actual policy at the hypervisor. Using our prototype, we are able to accurately assess the time spent in replacement policies, which includes actions such as traversing the page table to find replacements, and updating mappings upon swapping local and remote pages.

Policies and schemes: We investigate two well-known page replacement policies: *Round-robin* and *Clock*. We selected these schemes to represent two contrasting classes of policies: fast but inaccurate, and smart but slow.

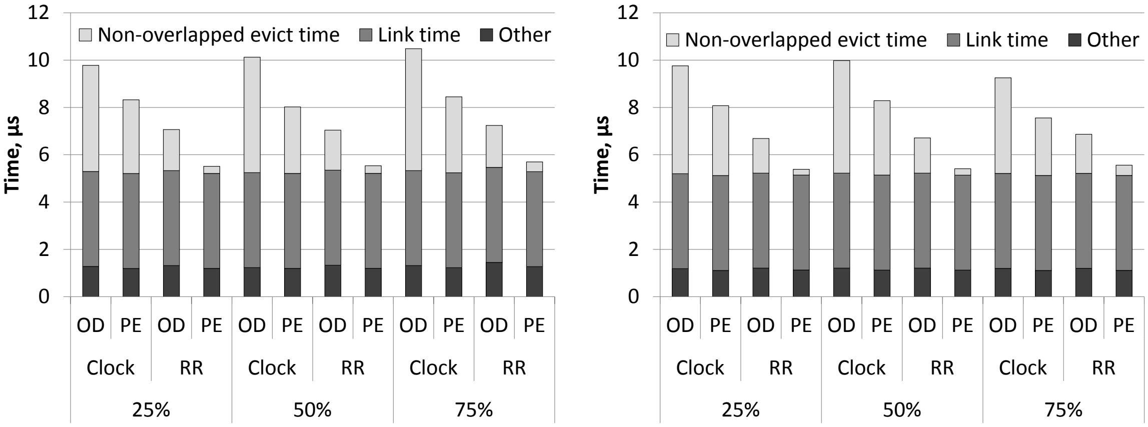
In Round-robin replacement, local pages are selected sequentially by keeping a global pointer to the next victim page. Upon replacement, the pointer is incremented to the next local page, and upon reaching the end of the local address space, the pointer is reset to the start of the address space. Clock approximates LRU by examining the "Accessed" bit of each page in the page table. Like Round-robin, it keeps a global pointer, and uses it to sequentially scan through all local pages. For each page considered, if the accessed bit is not set, Clock will select the page for eviction. Otherwise it will clear the accessed bit and increment the global pointer to the next local page.

Both of these policies are global replacement policies that consider all of local memory for eviction, as opposed to a per-VM replacement policy that limits the choice to pages solely managed by the page-faulting VM. While per-VM replacement would limit cross-VM performance interference, in the case of Clock, a global replacement policy is more likely to uncover "cold" pages that will not be accessed in the near future. In the case of Round-robin, a global policy allows the hypervisor to simply iterate through MFNs with no need for PFN-to-MFN translation. Although outside the scope of our current work, more complex policies can be used with disaggregated memory, including ones that provide quality of service or other measures of locality.

In addition to evaluating replacement policies, which determine *which* page is evicted, we evaluated two eviction schemes, which determine *when* the victim page is selected. The *on-demand* eviction scheme performs victim selection and eviction when remote memory is accessed and the hypervisor is invoked. This scheme is straightforward and can take advantage of the most up-to-date page usage information at the time of remote memory access. Our second scheme, *pre-evict*, keeps a small pool of free pages. Upon remote memory access, one page in the free pool is selected to be the destination of the remote memory page. In parallel with the DMA transfer of the page from remote to local memory, the hypervisor selects and evicts a page to remote memory (adding the MFN back to the free pool). Eviction time is thus partially or completely overlapped with the DMA transfer, minimizing the performance penalty of eviction at the cost of a slight increase in implementation complexity.

Methodology: We evaluate replacement mechanisms (and the studies described in Section 6) using two 8-core AMD Opteron machines, each with 32 GB of RAM. Each machine runs our modified Xen 3.4.1 hypervisor, with Linux 2.6.18-Xen as Dom0, and Debian Linux 2.6.26 guest VMs. Run times were reported by the guest VMs and verified through timing on an external physical machine. Disaggregated memory statistics from the test system's hypervisor were obtained every 30 seconds during the run. We configured a 4 μs link delay, based on the measured time to transfer a 4KB page across a PCIe 2.0 x2 link (done on a separate machine). Accesses to the link were serialized, allowing only one access at a time.

We run the reference inputs of five SPEC CPU2006 workloads with large memory footprints—zeusmp, perlbench, gcc, bwaves, and mcf. When running each workload, we configured a VM with enough memory to fit the workload's entire footprint (determined by examining detailed, simulator-produced memory traces), approximately 500, 650, 850, 900, and 1700 MB respectively. Based on these VM memory sizes, we varied the amount of the total footprint stored on remote memory by changing the amount of local memory. The footprint stored on remote memory ranged from 0% (the baseline all local memory case) to



(a) Perlbench

(b) Zeusmp

Figure 4. Mean total time in hypervisor to handle remote memory access for (a) perlbench and (b) zeusmp. Time is broken down into link delay, non-overlapped eviction, and remaining time.

25%, 50%, and 75%.

For each local-remote memory split, we evaluated our two policies, Round-robin and Clock, and our two schemes, on-demand and pre-evict, yielding 4 different options. In our figures, they are abbreviated by RR, CL, OD, and PE, respectively.

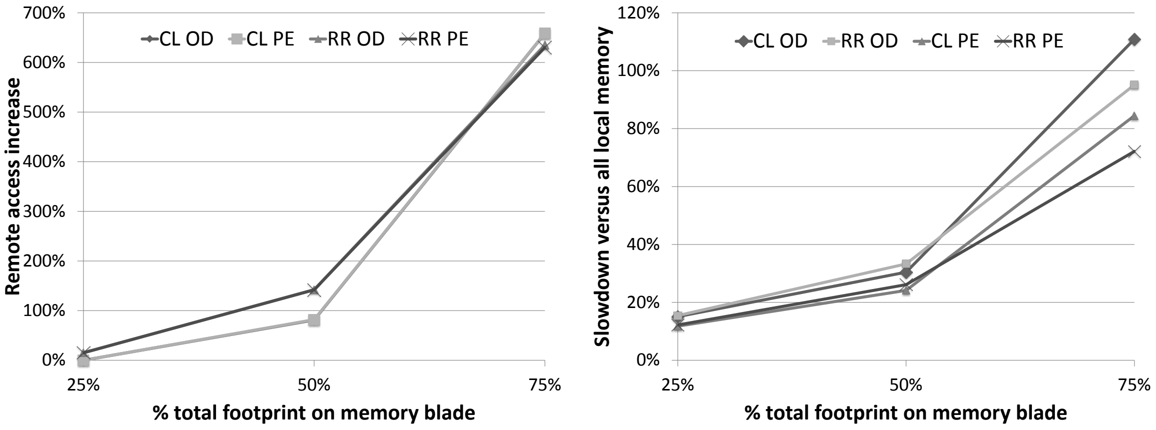
Our workloads fell into two categories: those with a much smaller working set than the footprint (perlbench, gcc), and those with a working set similarly sized to the footprint (zeusmp, bwaves, mcf). These two categories differ in how frequently they access the remote memory. For brevity, we present results for perlbench and zeusmp, which were representative of each category.

Results: The impact of our two replacement policies, Clock and Round-robin, and our two schemes, on-demand and pre-evict, is shown in Figure 4. The figure shows the mean amount of time, in microseconds, spent within the hypervisor to handle a remote memory access, broken down into link time (time to transfer data over PCIe), non-overlapped eviction time (any time in addition to the link time required to select a victim page and handle remapping), and the remaining time (e.g., determining the fault type, updating structures, and bookkeeping). Regarding the replacement policies, Figure 4(b) shows that Round-robin provides 60%-92% faster selection time than Clock, and 26%-35% faster overall handling time for zeusmp. The results are similar in perlbench (Figure 4(a)), indicating that the time to handle remote memory accesses is consistent regardless of the application behavior.

Regarding the eviction schemes, pre-evict provides significant improvement over on-demand, with 20%-24% reduction in access handling time for zeusmp, and a 17%-28% reduction in perlbench. Overlapping eviction time with page transfer is clearly worthwhile, even though it slightly increases code complexity. Ideally, pre-evict would entirely overlap eviction time with link delay. However, this is not always achievable because of variance in eviction time. Thus there remains some non-overlapped eviction time, especially for Clock.

Figure 5 shows the impact of replacement policy and eviction schemes on the number of remote accesses and on application slowdown as local memory size is reduced. We focus on zeusmp as its high level of remote memory activity highlights the major trends well. Other workloads show similar trends, albeit at smaller magnitudes. Figure 5(a) shows that the on-demand and pre-evict schemes yield nearly identical number of remote memory accesses for both Clock and Round-robin, indicating that the schemes do not affect the quality of victim selection. While Clock yields fewer remote memory accesses at 25% and 50% remote memory configurations (16% and 33% lower, respectively), at 75% remote memory Clock is within 3% of Round-robin. However, as shown in Figure 5(b) Round-robin provides application performance within 2% of Clock at 25% and 50% remote memory, and actually improves performance by 12% at 75% remote memory. Moreover, the performance impact of switching from on-demand to pre-evict is greater than switching policies, providing on average a 23% improvement in performance.

These results highlight the greater importance of reducing time spent within the hypervisor over the quality of the victim selection. Thus, using a policy and scheme that minimizes time spent in the hypervisor (Round-robin with pre-eviction) is preferred over more intelligent but time-consuming approaches (Clock with on-demand), reducing slowdown by as much as 39%.



(a) Change in remote accesses

(b) Slowdown

Figure 5. Impact of replacement policies and schemes on (a) amount of remote accesses and (b) application runtime for zeusmp. The baseline for (a) is Clock on-demand with 25% remote memory, and for (b) is a system with all local memory.

5. Synergy with content-based page sharing

Our prototype allows us to implement memory management software optimizations that may augment the benefits of disaggregated memory. In particular, we explore the synergy between disaggregated memory and content-based page sharing (CBPS) [22, 9, 15], which both seek to provide greater effective memory capacity. Applying CBPS to disaggregated memory systems not only improves the usage of local memory on each server, but also has the potential to offer greater benefits by identifying and sharing identical data from multiple systems stored on a single memory blade.

Under CBPS, the hypervisor detects memory pages with identical content, both within a VM and across VMs, and transparently replaces the duplicate pages with only a single copy. CBPS thus increases the total effective memory capacity of the system. In the context of workload consolidation, CBPS helps free up memory to allow more VMs to be co-located on a single machine. The opportunity for sharing is greatest when the VMs are similar, for example running the same OS kernel or having similar data sets.

To implement CBPS, we ported code from Difference Engine [9] to our prototype and modified it to use the NPT framework that we also use to manage local and remote memory. The resulting code detects when guest memory pages have identical content and transparently creates NPT mappings to a single shared MFN with read-only access (writable bit set to false in the NPT entries). The MFNs containing redundant copies are freed. (To isolate CBPS synergies, unlike Difference Engine we only share pages that are fully identical and do not compress page data). A guest write access to a shared MFN causes an NPT page fault in the hypervisor (and not in the guest). The hypervisor creates a private writable copy of the page and changes the guest PFN to MFN mapping for the faulting page. Then the hypervisor resumes the faulting guest vCPU.

Systems with CBPS require memory-overcommit support, where the total physical address space seen by VMs exceeds the host machine's physical memory capacity. When a copy-on-write leads to pages no longer being shared, new host memory pages are allocated and can overflow the host memory capacity. Typical CBPS systems would overflow to disk storage, but we use a memory blade for this purpose. The memory blade has data from multiple systems, offering significant opportunities for cross-host content sharing on the memory blade itself. CBPS can be used at both the local and remote memory levels to increase overall effective capacity at both levels. By reducing the total active working set, sharing at the local level benefits remote memory by decreasing the remote footprint.

The state transition diagram for a single guest PFN is shown in Figure 6, where greyed out transitions correspond to sharing opportunities between local and remote MFNs, an optional feature we do not consider in our current studies. Some state transitions require MFNs to be allocated or freed. For example, when a write access to a shared page occurs, a new MFN needs to be allocated for a local private copy. To obtain this page, the hypervisor may first need to evict some local page by migrating its contents to remote memory. Thus, a PgFault(W) transition from local/remote shared to local private may trigger an eviction action that migrates another PFN/MFN from local private or local shared to remote private/shared.

When a shared page is moved to local or remote memory, the NPT entries for all PFNs that share the page need

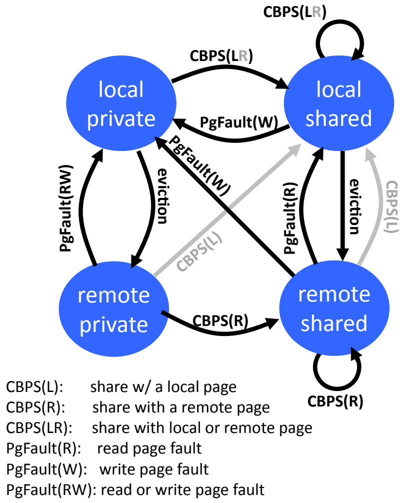

CBPS(LR)
CBPS(LR)
local private
local shared
PgFault(W)
PgFault(W)
PgFault(RW)
eviction
CBPS(L)
PgFault(R)
eviction
CBPS(L)
remote private
CBPS(R)
remote shared
CBPS(R)
CBPS(L): share w/ a local page
CBPS(R): share with a remote page
CBPS(LR): share with local or remote page
PgFault(R): read page fault
PgFault(W): write page fault
PgFault(RW): read or write page fault

Figure 6. PFN State Transitions

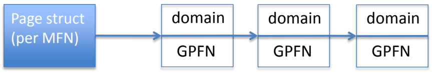


Figure 7. Inverse Mapping

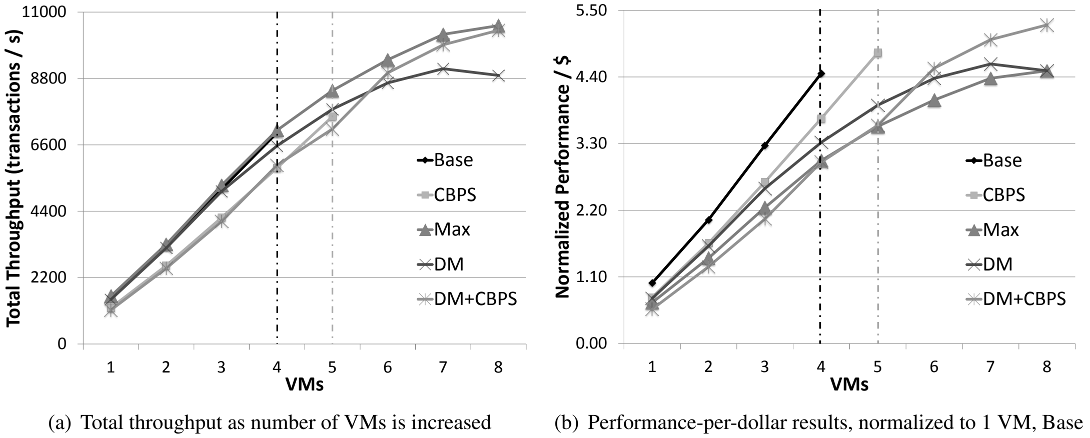
to be updated to ensure proper address translation and triggering of page faults. We add a reverse map data structure (Figure 7) that records the PFNs sharing each MFN. Entries are added when CBPS page scanning detects a sharing opportunity, and removed on guest write accesses causing page faults. These actions can occur concurrently by multiple CPU cores. To avoid lock contention, which we found to be expensive for this structure, we design it to be lock free, allowing concurrency of multiple delete operations and one add operation. Deletes mark entries in the inverse map as invalid rather than de-linking the entries, and entries are later garbage collected by the single CPU core that adds entries when performing periodic CBPS scans. We also had to address other concurrency control challenges; for example, a page fault caused by one guest can trigger eviction that causes an NPT entry of another guest to be modified. Finally, pages used as source or destination of DMA transfers by I/O devices must be kept local and non-shared. To enforce these properties, we added code to Xen's QEMU-based I/O emulation (in a helper "stub domain") to intercept memory pages used by guest VM I/O operations and prevent their eviction or sharing.

Evaluation: We evaluate the synergy between disaggregated memory and CBPS by considering a situation where multiple VMs are consolidated onto a single machine. We vary the number of VMs consolidated on a single large-memory system that has 8 quad-core AMD Opteron processors, with 256 GB of total RAM. Each VM is running VoltDB, an in-memory database running a TPC-C-esque workload that simulates an order-entry environment for a business with multiple warehouses [21]. The database server VMs run on our test system, and a separate server runs the client drivers, which submit transactions to the database. A single workload instance is run per VM; each VM is allocated an 8 GB address space and has unique randomly generated database contents.

To evaluate cost, we estimate the 3-year total cost of ownership (TCO) of both conventional blade and disaggregated memory systems using the methodology outlined in [12], factoring in the primary hardware components (CPU, memory, disk, NIC, memory blade, other), and the 3-year power and cooling cost. Cost and power numbers are collected from public sources, while the cost model for the memory blade is based on prior work [11].

Using this consolidation workload, we measure performance, in terms of total transaction throughput, of several configurations. We vary the number of consolidated VMs from 1 to 8 and consider five server configurations: *Max*, *Base*, *CBPS*, disaggregated memory (*DM*), and *DM+CBPS*. The *Max* configuration has enough local RAM to host all 8 VMs. *Base* is a cost-optimized server that has RAM sufficient for only 4 VMs (32 GB), and *CBPS* is the *Base* configuration with content based page sharing enabled to increase the effective memory capacity. The *DM* configuration uses our disaggregated memory solution with enough RAM to host all 8 VMs, but half of the RAM (32 GB) is local to the server, and half (32 GB) is hosted on the memory blade. *DM+CBPS* uses the combination of disaggregated memory and content based page sharing. Our 256GB memory server is an engineering test machine, and showed instability with certain configurations. Hence we restrict our consolidation study to round-robin, on-demand replacement.

We show performance results in Figure 8(a). *Max* provides near-linear performance scaling with the number of VMs, and *Base* provides the same scaling but is only able to host 4 VMs. The *CBPS* configuration enables memory capacity savings of approximately 40%, allowing one additional VM to be hosted. However, there is a slight performance penalty due to the overhead of page scanning and comparisons, as well as the copy-on-write overhead when a shared page is modified. The *DM* configuration is able to provide performance within 10% of the *Max* configuration, demonstrating the effectiveness of our disaggregated memory design in keeping the working set in the local memory. At 8 VMs, *DM* shows a performance drop off, which is partially from the working set growing too large, and partially from increased queueing delays on the remote memory link. Finally, the *DM+CBPS* provides slightly lower performance at 4 VMs compared to *DM*, due to the same performance penalties as in the *CBPS* case, but provides better scaling at higher VMs, as the 40% memory capacity savings frees up local memory to store a larger



(a) Total throughput as number of VMs is increased

(b) Performance-per-dollar results, normalized to 1 VM, Base

Figure 8. Consolidation results using CBPS. Dashed lines indicate the maximum consolidation achievable for *Base* (black) and *CBPS* (light gray) configurations.

portion of the working set.

In Figure 8(b), we show the performance-per-cost results for each of the configurations. The *Base* and *CBPS* configurations have a server cost of $1680, 3-year TCO of $2434. The *Max* configuration has a $2646 cost ($3618 3-year TCO) due to its larger memory capacity and use of higher density DIMMs. The *DM* and *DM+CBPS* configurations have a $2222 cost ($3052 3-year TCO); higher than *Base* because of the larger total memory capacity. However, the memory blade is amortized over multiple compute blades and enables use of standard density DIMMs. *Base* is able to provide the best perf/$ for 1-4 VMs, because of its lower memory costs, but it is unable to scale to consolidate additional VMs and cannot provide as high peak perf/$ as the other configurations. The *CBPS* option, due to its software overheads, provides lower perf/$ at the same number of VMs as *Base*, but is able to scale to a higher number of VMs. The *DM* and *DM+CBPS* options are able to provide a higher perf/$ compared to the *Max* configuration: *DM+CBPS* provides up to 17% perf/$ improvement versus the *Max* configuration at 8 VMs.

6. Interactive Web workload: memcached

We next consider a case study investigating the effectiveness of memory disaggregation at reducing scaling cost for memcached servers, a critical infrastructure component of interactive data-intensive web-based services. We contrast memory disaggregation with conventional scaling (by adding additional servers) showing that our design can achieve comparable latency at a 50% improvement in performance-per-dollar relative to conventional scale-out. Our case study is designed to approximate the memcached layer used by Wikipedia. We first briefly describe memcached and how we construct our workload using content and access traces from Wikipedia.

Memcached: Memcached is an in-memory key-value store widely used as an application-level cache. Many large and well-known content-serving systems such as Facebook, Twitter, Flickr, Zynga, and Wikipedia use memcached to reduce back-end database load to improve client request latency and throughput. Given its critical role in so many Web 2.0 infrastructures and its emphasis on large-memory, we consider memcached an important application to evaluate on our prototype. A typical deployment comprises a Web server (or application server tier) using a cluster of memcached servers to cache popular objects (e.g., pre-processed text, images, etc), thereby reducing load on the back-end data store, which would otherwise be used (possibly via disk I/O) to store and retrieve these objects.

A memcached cluster provides a lightweight, distributed hash table for storing small objects (up to 1 MB each), exposing a simple set/get interface. Keys are hashed to identify which server within the cluster might store the corresponding value in a way that balances load across the cluster. Individual memcached servers do not communicate (cache contents are maintained by the clients, which have separate connections to each server); hence, memcached clusters scale readily to large sizes (hundreds of servers). Because the servers do not interact, we need only to consider the performance of a single memcached server to generalize entire cluster behavior.

Workload Trace: Although there are several publicly available microbenchmark tools for memcached [1, 17], these benchmarks focus primarily on the raw throughput of get and set operations, and do not generate realistic request interarrival or object locality/size distributions. To capture the interaction of a real workload with memcached, we create an application-level benchmark based on Wikipedia. We use publicly available Wikipedia traffic logs to mimic real Web traffic request locality and the MediaWiki content-serving engine [8] to translate Wikipedia traffic logs into a memcached request trace.

There are two primary sources of available Wikipedia traffic logs. The first (http://dammit.lt/wikistats/) contains aggregated page counters for how many times each Wikipedia page is visited each hour. The second source is made available by the authors of the Wikibench [20] tool, which contains a sampled log (10% sampling) of user requests every hour. To preserve temporal locality, we base our workload trace on the second source.

Given these traces of Wikipedia HTTP traffic, we must translate each HTTP request into an appropriate set of memcached requests. Prior work assumes Zipf-like object popularity distributions when generating memcached request traces [19]. To ensure we capture all aspects of memcached behavior, we emulate the Wikipedia site by running our own local Wikipedia server based on the open-source Mediawiki software. We translate our HTTP request trace into a memcached trace by logging the memcached calls made by Mediawiki. Since Wikipedia itself uses Mediawiki as the content-serving engine, our approach provides an accurate model of how Wikipedia generates memcached traffic (i.e., when objects are allocated, accessed and expire, their size, and their locality/popularity).

We drive our local Wikipedia server using the Wikibench tool [20], and seed it using a 2008 Wikipedia database dump, and traffic traces from January 2008 (the most recent trace data set available from Wikibench). This process produces a memcached request trace consisting of get, set, and add requests. We generate 2.4 million HTTP requests from an 8-hour time window of Wikipedia usage, which translates into 11.6 million memcached requests to 809,272 unique keys, with an average value size of 6.4 KB. The total aggregated unique payload (ignoring time expiration issues) is approximately 5.1 GB.

Evaluation: Before measuring the performance of our prototype, we first establish baseline request latency for our memcached server setup. In a typical deployment, the memory capacity of each memcached server is chosen to achieve a specific latency distribution. In Figure 9, we demonstrate how the object hit fraction (left y-axis) increases when expanding memcached server memory capacity. However, inferring the request latency (right y-axis) requires information about the relative hit and miss penalties of requests. Average memcached request latency can be approximated with the familiar formula: $latency = (hit\_fraction \cdot hit\_latency) + (miss\_fraction \cdot miss\_latency)$. The hit latency is the latency incurred by retrieving an object directly from memcached, whereas the miss latency is the latency incurred by first failing to retrieve the object from memcached, and then retrieving it from the backing store, such as disk or database.

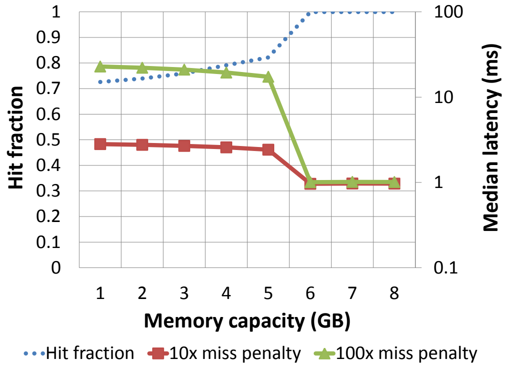


Figure 9. Memcached hit fraction and median latency as a function of available memory capacity

As our trace data does not include the hit and miss latencies of Wikipedia's actual deployment, we instead investigate latency impact by varying the *ratio* of miss to hit latency, and choose this range based on anecdotal data. In practice, the difference between miss and hit latency depends upon both the system architecture and background workload. We approximate the ratio between miss and hit latency as 100x, which is the ratio between a typical back-end disk fetch (10 milliseconds), and a typical memcached cache hit (100 microseconds). On the right y-axis in Figure 9, we see the median request latency with a 100x relative penalty, as well as a more moderate 10x penalty. For the remainder of this section, we consider a 100x penalty.

Applications that rely on memcached are primarily latency sensitive. Therefore, with respect to performance, we determine the additional latency incurred when using disaggregated memory to achieve the target memcached server capacity instead of entirely local memory. From the earlier graphs, we see that for a desired median latency of 1 millisecond, the total memory capacity of the memcached system needs to be at least 6 GB.

In Figure 10, we compare the CDF of request latency in a memcached server using 6 GB local memory and a server using a combination of 1 GB local memory and 5 GB remote memory. We see that using the remote memory blade incurs no discernible latency overhead. Similarly, when using throughput as an additional comparison metric, we find that the request throughput for the remote-memory configuration achieves 95% of the peak throughput of the entirely local memory configuration.

These results imply that when additional memory would improve performance and miss penalties are high, memory disaggregation provides a cost-efficient alternative com-

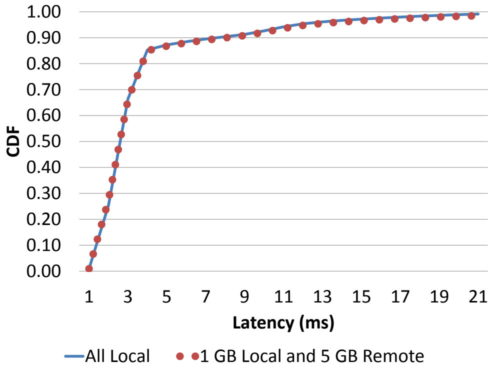


Figure 10. Latency using all local vs local-and-remote memory

pared to the traditional scale-out approach of using additional compute blades.

We contrast the cost efficiency of adding a second `memcached` server to a cluster against expanding its memory capacity through the addition of disaggregated memory, using the cost model from prior work [11]. We assume the same baseline server configuration as a recent `memcached` study [4]: two AMD Opteron 6128HE 1-socket server with 4x 8GB DRAM, 1 SATA drive, 1x 10GbE, and compare it to a single instance of the same AMD server, with the addition of 32GB remote memory. We find that the hardware cost for the pair of compute blades is $3360, and $2410 for the combination of a single compute blade and memory blade. Moreover, when considering operating costs with a 3-year-amortization of the capital costs, the overall total cost-of-ownership (TCO) of the compute blade is $4768, whereas the same for a memory blade is $3148, reflecting a 50% improvement in performance/TCO.

7. Related Work

Our work on a hypervisor-based prototype of disaggregated memory extends prior work [11], exploring system-level implications. It touches upon three primary lines of research, all of which have prior work, but none of which cover the unique context in this paper.

Software approaches toward memory expansion. Similar to our work, Ye et al. [23] extend a hypervisor's memory management functionality to model additional layers in the main memory hierarchy. Our approach differs in that it takes advantage of recent processor hardware support for nested page tables. More significantly, our investigation sheds light on the important tradeoffs for managing disaggregated memory as well as the benefits of using it in latency-sensitive workloads. ScaleMP [18] uses a virtual machine monitor to group a cluster of servers under a single OS image, enabling servers to provide each other with remote memory. ScaleMP relies on OS NUMA support and prefetching to improve remote memory performance. Instead of a hypervisor-based implementation, the MemX project [6] implements a remote memory pager within a VM, providing a service to exploit memory capacity from servers clustered with Ethernet or InfiniBand. Transcendent memory (Tmem [13]) exposes a front-end API for applications to specify their cache pages, allowing the back-end to implement various memory capacity optimizations (e.g., remote paging and page compression) across applications.

Software management of large memory and buffer caches. CBPS has been the topic of multiple publications [22, 9, 15]; this paper demonstrates the complementary benefits of CBPS and memory disaggregation. Memory blades provide a logical platform for integrating Flash as part of the main memory [16], comparable to recent approaches of using Flash as buffer cache [7]. However, the significant latencies introduced by Flash memory require application-aware, non-transparent changes to the OS to mitigate the performance difference versus local DRAM. In contrast, our disaggregated memory design enables transparent memory capacity expansion.

Page replacement policies. Page replacement policies have been a classic memory management topic (e.g., [5, 14]). We leverage existing algorithms such as Round Robin and Clock, and identify the tradeoffs between replacement speed vs. quality for disaggregated memory. Our design uses global replacement and can choose between pages across guest OSes, while MemX and Tmem let applications make local replacement decisions and can potentially exploit application-level knowledge.

While these studies point to the increasing adoption as well as a plethora of interface/implementation options of memory disaggregation, our work is distinguished in the following ways. Transparent to guest VMs and applications, our design preserves the main memory load/store interface to higher-level software, while MemX and Tmem expose remote memory as paging devices and require changes to the application or guest OS. Our design also assumes a new hardware platform that connects to dedicated memory blades via PCIe; the resulting low-latency remote memory leads to different software-level requirements. MemX has recently included data deduplication (CBPS), but only on remote pages, and our work further demonstrates the benefits of CBPS when it frees up local memory.

8. Conclusions

Technology and cost barriers prevent the straightforward use of capacity scaling to meet the growing need in systems for increased memory capacity. Disaggregated memory is a design that can provide a cost-effective way to scale memory capacity. To explore its software implications, we developed a software-based prototype of disaggregated memory by adding hypervisor support for remote memory. We evaluated the impact of two replacement policies, Round-robin and Clock, and two replacement schemes, on-demand and pre-evict, on disaggregated memory. Our findings show that the low latency of remote memory favors approaches that minimize replacement time, with simpler Round-robin outperforming the more accurate Clock which has higher latency for page selection, and with pre-evict greatly outpacing on-demand. We demonstrate that disaggregated memory has synergy with content-based page sharing (CBPS), with the combination outperforming either technique alone. Finally, our case study of an interactive web caching workload (memcached) shows that disaggregated memory provides similar response time performance at a lower cost compared to scaling out on multiple compute blades. Our study demonstrates feasibility of the software infrastructure required for disaggregated memory and shows that it can provide performance-per-dollar benefits for important emerging workloads.

Acknowledgements

The authors would like to thank DCDC project members, Fred Worley, Pari Rajaram, and the anonymous reviewers for their feedback. This work was supported in part by NSF grant CNS-0834403.

References

[1] B. Aker. libmemcached (software download). http://libmemcached.org/libMemcached.html.

[2] AMD. Amd-v nested paging. http://developer.amd.com/assets/NPT-WP-1201-final-TM.pdf.

[3] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proceeding of the 38th annual International Symposium on Computer Architecture*, ISCA '11, New York, NY, USA, 2011. ACM.

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *2nd International Green Computing Conference*, 2011.

[5] R. W. Carr and J. L. Hennessy. WSCLOCK: A Simple and Effective Algorithm for Virtual Memory Management. In *8th Symposium on Operating Systems Principles (SOSP)*, 1981.

[6] U. Deshpande, B. Wang, S. Haque, M. Hines, and K. Gopalan. Memx: Virtualization of cluster-wide memory. In *ICPP'10: Proceedings of the 39th International Conference on Parallel Processing*, pages 663–672, 2010.

[7] Facebook. Flashcache. https://github.com/facebook/flashcache.

[8] W. Foundation. Mediawiki. http://www.mediawiki.org/wiki/MediaWiki.

[9] 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 *OSDI'08: Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[10] Intel. Intel scalable memory buffer. http://www.intel.com/content/dam/doc/datasheet/7500-7510-7512-scalable-memory-buffer-datasheet.pdf.

[11] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA '09: Proceedings of the 36th annual International Symposium on Computer Architecture*, pages 267–278, New York, NY, USA, 2009. ACM.

[12] 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 *ISCA '08*, 2008.

[13] D. Magenheimer. Transcendent memory in a nutshell. https://lwn.net/Articles/454795.

[14] N. Megiddo and D. S. Modha. Outperforming LRU with an adaptive replacement algorithm. *IEEE Computer*, 37(4):58–65, 2004.

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

[16] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *HotOS XII*, 2009.

[17] S. Sanfilippo. An update on the memcached/redis benchmark (blog entry). http://antirez.com/post/update-on-memcached-redis-benchmark.html.

[18] ScaleMP. Versatile SMP (vSMP) architecture. http://www.scalemp.com/architecture.

[19] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: managing server clusters on intermittent power. In *Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 185–198, New York, NY, USA, 2011. ACM.

[20] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[21] VoltDB. Voltdb tpc-c-like benchmark comparison-benchmark description. http://community.voltdb.com/node/134.

[22] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI '02: Proceedings of the 5th symposium on Operating Systems Design and Implementation*, pages 181–194, New York, NY, USA, 2002. ACM.

[23] D. Ye, A. Pavuluri, C. A. Waldspurger, B. Tsang, B. Rychlik, and S. Woo. Prototyping a hybrid main memory using a virtual machine monitor. In *ICCD*, pages 272–279. IEEE, 2008.