Software-Controlled Transparent Management of Heterogeneous Memory Resources in Virtualized Systems

Min Lee Vishal Gupta Karsten Schwan

College of Computing
Georgia Institute of Technology
{minlee,vishal,schwan}@cc.gatech.edu

Abstract

This paper presents a software-controlled technique for managing the heterogeneous memory resources of next generation multicore platforms with fast 3D die-stacked memory and additional slow off-chip memory. Implemented for virtualized server systems, the technique detects the 'hot' pages critical to program performance in order to then maintain them in the scarce fast 3D memory resources. Challenges overcome for the technique's implementation include the need to minimize its runtime overheads, the lack of hypervisor-level direct visibility into the memory access behavior of guest virtual machines, and the need to make page migration transparent to guests. This paper presents hypervisor-level mechanisms that (i) build a page access history of virtual machines, by periodically scanning page-table access bits and (ii) intercept guest page table operations to create mirrored page-tables and enable guest-transparent page migration. The methods are implemented in the Xen hypervisor and evaluated on a larger scale multicore platform. The resulting ability to characterize the memory behavior of representative server workloads demonstrates the feasibility of software-managed heterogeneous memory resources.

Categories and Subject Descriptors D.4.8 [*Performance*]: Measurements

General Terms Performance, Design, Experimentation

Keywords Heteorgeneous memory, Page placement, Virtualized systems

1. Introduction

Die-stacked memories can provide lower access latency and higher bandwidths at lower power levels, in comparison to traditional off-chip memories [6]. However, such die-stacked memories are likely to be constrained in size, i.e., they are projected to have capacities ranging only to a few hundreds of megabytes [7]. This suggests a usage model in which they are combined with off-chip memory to provide higher capacity and low latency capabilities. For enterprise-class or high-performance machines combining a limited amount of fast on-chip memory with additional slower off-chip memory, will therefore, result in the hybrid or heterogeneous memory systems shown in Figure 1.

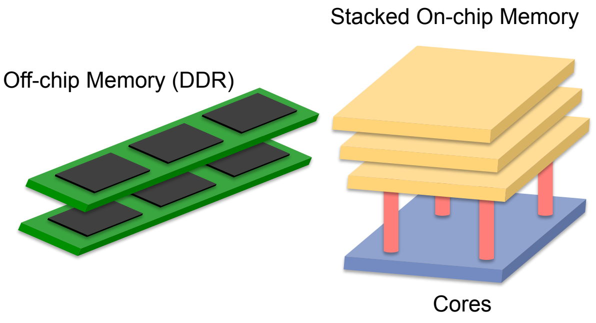

Figure 1. Heterogeneous memory organization consisting of a combination of on-chip and off-chip memories.

Die-stacked DRAM can be utilized as (i) hardware-managed cache or (ii) software-managed memory. The former approach has the advantage of being able to quickly react to changing memory access patterns, and it provides a transparent way to incorporate such memory architecture in ways that support legacy applications. Potential drawbacks and challenges of this approach are that first, it can result in high overhead for managing the tags of such large sized caches, and second, it would require extended coherency support. An additional issue is the consequent lack of software control over memory placement.

Alternative ways to manage stacked DRAM are actively being investigated in the architecture community, but in this paper, we explore how an operating system or hypervisor can use its information about application behavior to manage the heterogeneous memory resources of future multicore platforms. Specifically, we investigate an approach in which stacked DRAM is explicitly exposed as system-visible memory, and we then evaluate the feasibility of software-based memory management for the resulting heterogeneous memory platforms. In particular, given the increasing use of virtualization in server systems, we investigate several challenges for managing heterogeneous memory resources. (1) Hardware provides only limited visibility into the memory access behavior of guest virtual machines (VMs), e.g. x86 provides only one-bit information such as access bit in the page tables. Therefore, efficient methods are required to detect which pages are critical for a guest's performance based on such limited information from hardware. (2) Hypervisor should implement its management scheme transparently to the guest OSes. This may be challenging since the page tables are owned by the guest in paravirtualized environment, thus making it difficult to migrate its pages between memories transparently without guest involvement. Even with hardware virtualization support, such multiple mappings should be handled properly. This also involves TLB management across cores to prevent stale mappings.

This paper presents techniques to address (1) and (2) above. First, we enhance the hypervisor to build an access-bit history for each VM, by periodically scanning the access-bits available in page tables. This 'a-bit history' is then used to detect the guest's 'hot' pages and determine the guest VM's page working-set. Since hot page and working set detection requires periodically scanning page-tables, which can incur high overhead, we maintain additional data structures for quickly accessing page table entries. In addition, scans are done in the virtual time of guest virtual machines, for accurate accounting. Finally, the hypervisor mirrors guest page tables and transparently uses these mirrors, which allows the hypervisor to manipulate guest page mappings by simply changing their mirror page tables, without requiring guest operating systems to be altered in any way, i.e., transparently to guests.

Page access tracking, hot page detection, and mirroring are fully implemented in the Xen open-source hypervisor, thereby enabling experimental evaluation of overheads in realistic server platforms. To emulate such platforms' future memory heterogeneity, we use a multi-socket Intel Westmere platform in which one of its memory controllers is throttled, resulting in the presence of both 'fast' (regular DRAM, emulating future 3D stacked DRAM) and 'slow' (throttled DRAM, emulating future off-chip DRAM) memory in the system. Experimental results obtained on this machine and memory configuration characterize the memory behavior of standard server workloads, in terms of their working set sizes and the performance impact of memory heterogeneity. The page migration mechanism is evaluated with micro-benchmarks, to show the feasibility of software management for future heterogeneous memory systems.

In summary, this paper's technical contributions include:

- A hypervisor-level mechanism to detect guest memory access patterns using access bit information.
- Transparency support for managing heterogeneous memory for virtual machines, implemented by the hypervisor.
- An evaluation of the sensitivity of several server workloads to the performance of heterogeneous memory subsystems.

In the remainder of this paper, we first describe the mechanisms used for tracking guest activity and policies for stacked memory allocation in Section 2. Section 3 describes our evaluation methodology, with experimental results presented in Section 4. Finally, related work and conclusions are described in Section 5 and 6 respectively.

2. Heterogeneous Memory Management

To leverage die-stacked low latency DRAM to reduce an application's overall memory access latencies, it is important to detect and then manage its 'hot' pages. This requires efficient methods for memory access tracking, described next.

2.1 Memory Access Tracking

Current multicore platforms provide limited support for detecting applications' memory access patterns. Specifically, each entry in the page table is associated with an *access bit*. This bit is set by the hardware when the corresponding page is accessed. Software is provided control to reset this bit. This single-bit information is used in our work to determine a VM's memory access pattern, leveraging our earlier work on cache management [4]. Specifically, we periodically scan and collect the access bits in guest page tables, to form a bitmap termed as 'A-bit history' (access-bit history). If a 32-bit word and a 100ms time interval is used, one word amounts to roughly 3.2 seconds of virtual time. Therefore, a dense A-bit history (i.e., many 1's) would indicate the presence of hot pages. Several optimizations are used to minimize overheads, discussed later in this section.

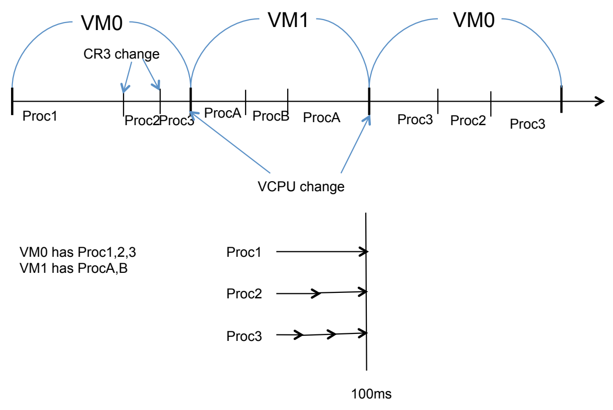

Figure 2. Tracking virtual time: TIMER event (10ms tick) is not shown.

To capture an accurate A-bit history, a process's virtual time rather than wall-clock time is used. This avoids unnecessary page table scans and a more accurate detection of hot pages. The hypervisor is extended to track processes' virtual time across various events, and each time the 100ms boundary is crossed, its page table is scanned for A-bit collection. Figure 2 depicts two VMs — VM0 and VM1 – where each VM has processes Proc1,2,3 and ProcA,B, respectively.

Three events, TIMER, NEW_CR3 and SCHEDULE, occur along its execution timeline. They correspond to the 10ms timer tick, the CR3 switch (process switch), and the VCPU switch, respectively. At these points, the actual time spent in execution is calculated and accumulated for each process. In this fashion, each process' virtual time is tracked, thus enabling the accurate detection of hot pages (this is because virtual time is the actual time spent running on each core).

An implementation in the Xen open-source hypervisor obtains and maintains A-bit histories for arbitrary guest VMs. Since Xen employs frame tables for memory management – large tables in which each entry corresponds to some physical page – we extend this data structure to embed our A-bit history and other information, as shown in Figure 3. The A-bit history is used to hold each frame's access bit history. Next/Prev pointers help form linked lists of pages for efficient access.

In addition, an Rmap structure is used to store reverse mapping information to make it easy to unmap and map some given page. Each physical page (mfn) has one Rmap_list, which is list of Rmap_set. Rmap_set is a fixed size array containing pointers to page table (PT) and page table index (PTI). Therefore by iterating Rmap_list and Rmap_set, all mappings to the given page can be found and changed. Without this Rmap structure it would be too expensive to find mappings to a given physical page, which is needed to change mappings for page migration.

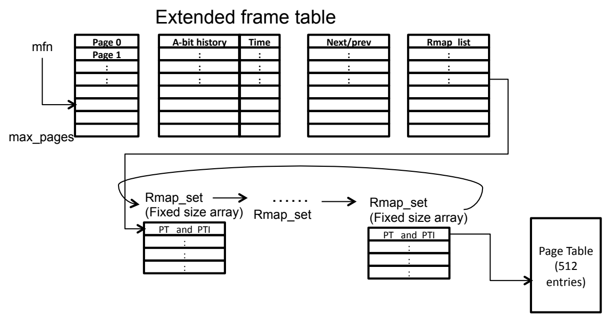

Figure 3. Extended frame table with a-bit history and reverse maps

Further, for guest transparency, the hypervisor mirrors each guest's page table which is installed in the hardware base register (CR3). This is very similar to shadow page table, and this allows us to change virtual-to-physical mappings without changing guest OS. By this, page migrations are transparent to the guest OS.

2.2 Memory Allocation Policy

Stacked DRAM memory management is concerned with both intra-VM and inter-VM memory allocation policies.

2.2.1 Intra-VM Allocation

Our intra-VM page placement policy aims to utilize a limited allocation of fast stacked DRAM for a VM. Pages with the highest hit rate are moved to stacked DRAM. For hot read-only pages, two copies are maintained: a home copy and a satellite copy. The home copy resides in off-chip DRAM, while the satellite copy resides in stacked DRAM. When such read-only pages need to be migrated back to off-chip memory, the satellite copy for these pages is simply discarded, and the home copy is used for accesses thereafter. This saves a page copy for moving data back to off-chip memory. For read-write pages, only a single copy is maintained, and a copy is performed each time when a page is moved back and forth between memories.

2.2.2 Inter-VM Allocation

In a manner similar to allocating constrained physical memory resources across VMs using memory ballooning, the inter-VM allocation policy aims to distribute stacked memory across application based upon there activity. We consider two policies in this work.

Share-based allocation: this policy uses pre-defined shares, e.g., set by the administrator or a cloud allocator, to divide stacked DRAM capacity among VMs. Memory is distributed as a weighted sum of these shares as shown in Equation 1.

$$mem(vm_i) = mem_{total} * \frac{share(vm_i)}{\sum_{i=1}^{n} share(vm_i)} \qquad (1)$$

WSS-based allocation: this policy uses the working set size (WSS) information for each VM to control memory allocation. The allocation are performed by using WSS as the share value in Equation 1.

We are adding these policies to our implementation as part of ongoing work.

3. Experimental Evaluation

3.1 Heterogeneous Memory Emulation

Earlier work on stacked DRAM in the architecture community has relied on architectural simulators. In order to conduct heterogeneous memory research on actual systems with realistic server workloads, we take the alternative approach of emulating heterogeneous memory on a multi-socket platform.

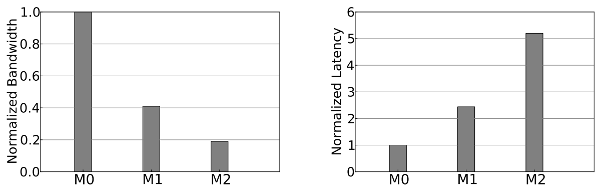

Figure 4. Bandwidth and latency comparison for different memory configurations

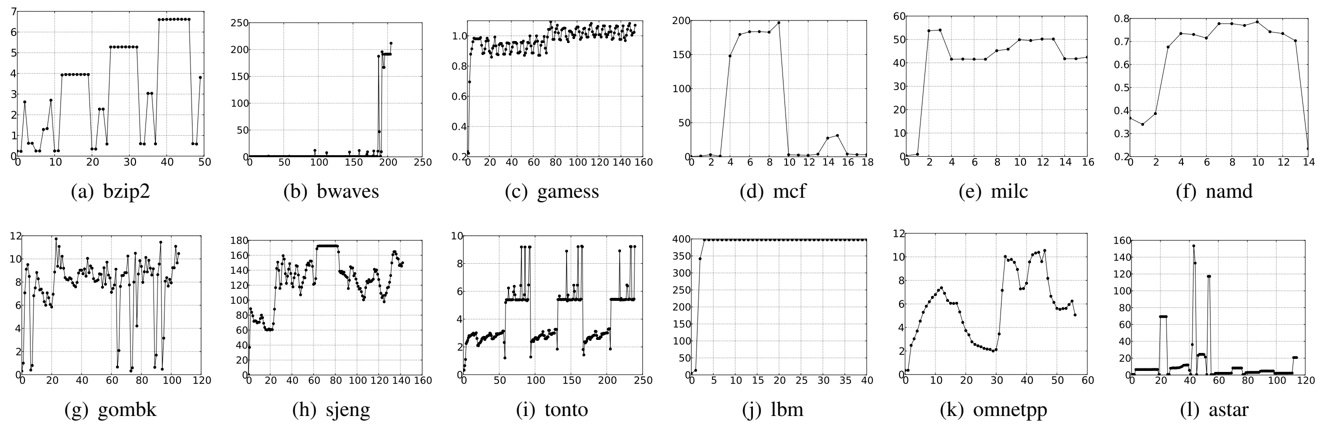

Figure 5. WSS curve for SPEC CPU2006 applications (x-axis = time (s), y-axis = WSS (MB)).

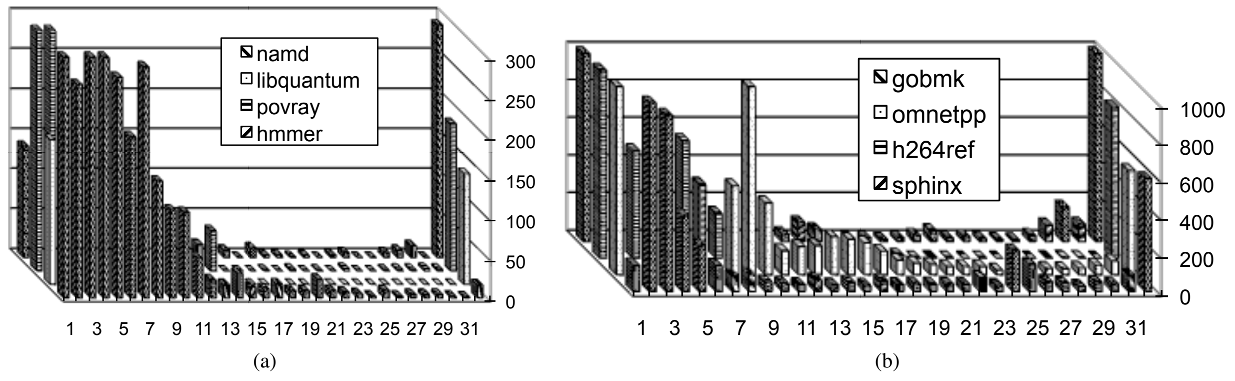

Figure 6. Histograms of pages and corresponding access counts

Our experimental platform consists of a dual-socket 12 core Westmere X5650 server with 12GB DDR3 memory. Cores from the first socket are used for running programs that can access memory from both sockets, i.e., cores from the second socket are kept idle. This NUMA configuration provides an approximate 1.5x difference in memory latency between the two nodes. In order to emulate more heterogeneous configurations, however, we use memory controller throttling on the remote node to slow it down further.

Memory throttling is enabled by writing to the PCI registers (Integrated Memory Controller Channel Thermal Control). By applying different amount of throttling, varied memory configurations can be emulated for emerging memory technologies [10, 14]. Figure 4 shows a comparison of normalized bandwidth and latency for three memory configurations for the memory-intensive stream benchmark [9]. The M0 memory configuration corresponds to no throttling, while M1 implies small throttling, and M2 implies higher throttling. As expected, we see progressively lower bandwidth and higher latency with M1 (2.5x) and M2 (5x) configurations. M0 is used as the base configuration for evaluation.

3.2 Workloads

We evaluate the impact of heterogeneity on server workloads by using a diverse set of server-centric workloads summarized in Table 1. These workloads include CPU-intensive SPEC CPU2006 benchmarks, multi-threaded PARSEC benchmarks, and several MapReduce data processing benchmarks and with data analytics kernels. The MapReduce benchmarks use the shared-memory Phoenix implementation of MapReduce [12], where input datasets are cached in memory.

Workloads	Description
SPECCPU	Single-threaded CPU-intensive benchmarks
PARSEC	Multi-threaded application kernels
Phoenix	Shared-memory MapReduce kernels

Table 1. Workload summary

4. Results

The experimental data shown in Figure 5 depicts working-set size (WSS) graphs as a function of time for several SPEC CPU2006 workloads. As seen in the figure, several CPU-

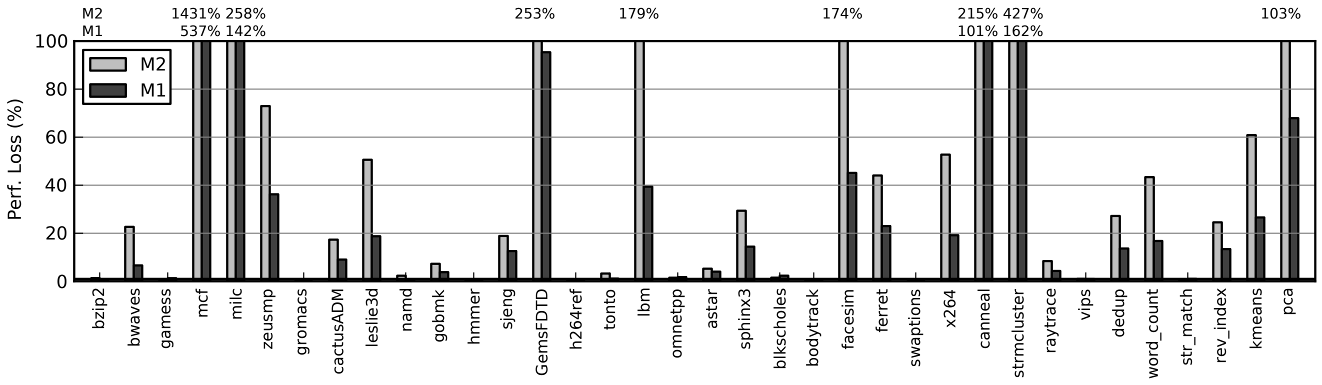

Figure 7. Comparison of performance impact of memory slow down with different memory configurations

bound applications have very small WSS, e.g., $0.8\,$MB for namd. In comparison, memory-intensive workloads like mcf and lbm have much larger working-sets of size $200\,$MB and $400\,$MB respectively. Further, WSS dynamically changes over time for these applications, thereby showing the need for dynamic memory management. Working-set size only represents the amount of memory pages that are accessed by an application concurrently. It, however, does not show how various pages in its allocated memory are used by the application during its execution. Two applications may have the same WSS, but diverse memory footprints. An application may use the same set of pages throughout its execution, while another may keep changing its active memory region.

To illustrate further, Figure 6 presents the access count histogram for various SPEC CPU2006 workloads. Specifically, the X-axis is the number of 1's in the access bit history, while the Y-axis is the number of pages (max Y values are set to 300 and 1000 for better visibility, respectively) Here, we can see some clear distinction between hot pages and cold pages, although some are in the gray area between the two. Basically, the hot page detection mechanism captures such hot pages on the right side ($x > 22$) in Figure 6. For simplicity, we have chosen a threshold value of 22 based on these observations. The behavior shown is robust for different threshold values. These results highlight the fact that only a fraction of the total memory region is actively used by the application which is critical for its performance. These pages should be retained in fast memory, while the remaining pages can be allocated from slow memory.

Our next results evaluate the performance impact of memory slowdown for all of the workloads in Table 1. These applications are executed with different memory configurations, by varying the amount of throttling applied to the memory controller. Figure 7 shows the performance loss (%) for the applications for two memory configurations (M1 and M2) as compared to no throttling (M0) as described in Section 3.1. As we see in the figure, several applications suffer from high performance loss due to memory slowdown, while many others see small impact. Particularly, the mcf, milc, GemsFDTD, and lbm workloads from SPEC CPU2006; the facesim, canneal, and streamcluster benchmarks from PARSEC, and the pca kernel from the Phoenix suite observe severe degradation. As expected, the performance degradation becomes smaller with faster M1 memory configurations. The highest impact is observed for the mcf workloads to be 1431% (15x) and 537% (6x) for the two configurations. Thus, by managing the active memory pages for these applications in the stacked DRAM, substantial performance gains can be achieved. These experiments were also performed with different CPU frequencies, to analyze the correlation between processor speed and memory slowdown on the performance. We observe similar trends for these applications, but with a smaller magnitude due to a slower CPU.

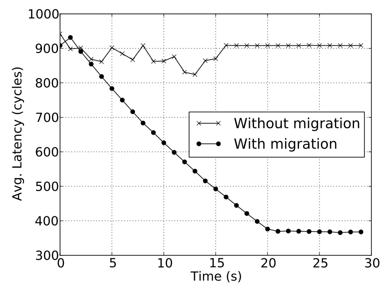

Figure 8. Micro-benchmark results: Memory access latency with and without hot-page migration

We evaluate our page migration mechanisms using a micro-benchmark *memlat*, which allocates a large region of memory and randomly accesses it. The benchmark runs for 30 seconds and reports average access latency for each second. Figure 8 shows the experimental results for two scenarios: when memory is statically allocated from slow memory and when migrations are enabled to dynamically move hot-pages to fast memory. When migration is disabled, the access latency remains high throughout the execution, with an average value of 891 cycles. On the other hand, when hot-page migration is enabled, latency starts to decrease until it reaches a value of 367 cycles and then remains fixed. Pages are initially gradually moved to fast memory, thus the memlat partially accesses pages from fast and slow memory. At t=21s, when all of the memory has been moved to fast memory, the latency reaches a stable value. These results show the feasibility and effectiveness of our software-controlled approach for managing heterogeneous memory resources.

5. Related Work

Concerning memory management in virtualized systems, the VMware ESX server uses a sampling approach to detect working set sizes and manage allocation of system memory among virtual machines using shares [13]. Similarly, Geiger explores mechanisms to monitor the virtual MMU and storage hardware of a VM to provide meaningful information about the usage of buffer cache and virtual memory subsystems [3], while Hypervisor-exclusive cache uses a ghost buffer based approach to predict page miss rates for virtual machines [8]. In comparison, our work uses page-table access bits to detect not only working set size of virtual machines, but also provides 'hotness' information of each page to guide page placement.

Several architectural solutions have also been proposed for tracking memory access patterns and page placement strategies for hybrid memory systems containing traditional DRAM and other memory technologies such as non-volatile memories [1, 11]. Similarly, hardware approaches for managing DRAM caches have also been investigated [2, 10]. Further, efforts have been made to investigate page replacement policies in the context of disaggregated memory platforms, allowing a large pool of memory to be shared by multiple servers [5]. In comparison, our work focuses on system software control on memory management for more efficient utilization of the stacked DRAM. Techniques using sophisticated LRU heuristics for balancing memory across several virtual machines have also been proposed [15]. This work is complementary to our work as similar policies can be used with our approach to guide memory allocation.

6. Conclusions & Future work

This paper presents systems software mechanisms for managing heterogeneous memory resources that consist of a combination of fast 3D die-stacked DRAM and off-chip DRAM. We believe that such stacked DRAM should be managed by software rather than by hardware (hardware managed cache) for flexible management. To this end, we propose and evaluate mechanisms for tracking the memory behavior of virtual machines and managing memory mappings, in a guest-transparent manner. We conduct basic research and evaluation on an emulated heterogeneous memory platform. Preliminary results show the effect of memory heterogeneity on various workloads and our ability to track guest memory access patterns and improve performance by managing how stacked DRAM is used by applications.

As part of future work, we are devising further optimizations of our implementation of the page migration mechanism, and we will next experimentally evaluate different policies for memory allocation.

References

[1] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of SC*. IEEE, 2010.

[2] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of HPCA*, pages 1–12, 2010.

[3] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of ASPLOS*. ACM, 2006.

[4] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of ASPLOS*. ACM, 2012.

[5] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of HPCA*. IEEE, 2012.

[6] G. H. Loh. 3D-stacked memory architectures for multi-core processors. In *Proceedings of ISCA*. IEEE, 2008.

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

[8] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of ATC*. USENIX, 2007.

[9] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. URL `http://www.cs.virginia.edu/stream/`. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[10] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of MICRO*, pages 235–246, Washington, DC, USA, 2012.

[11] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of ICS*. ACM, 2011.

[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*. IEEE, 2007.

[13] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of OSDI*. USENIX, 2002.

[14] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *Proceedings of HPCA*, pages 1–12, 2010.

[15] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *Proceedings of VEE*. ACM, 2009.