Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems

Gabriel H. Loh Nuwan Jayasena Jaewoong Chung
Steven K. Reinhardt J. Michael O'Connor Kevin McGrath
AMD Research
Advanced Micro Devices, Inc.
{gabe.loh,nuwan.jayasena,steve.reinhardt,mike.oconnor}@amd.com

Abstract

Die-stacking technology has the potential to provide significant relief from the Memory Wall by providing a large amount of memory with a high-bandwidth, low-power interconnect. Past studies have mostly assumed that enough DRAM can be stacked to supply the entire system's main memory. Unfortunately, most systems will not be able to economically integrate enough DRAM to provide all of main memory. As such, systems will have a heterogeneous memory space consisting of both in-package and conventional off-chip DRAM. In this paper, we present a simple design study to illustrate some of the potential benefits of such a heterogeneous memory organization. Our results show that using stacked memory under system software control could provide greater benefit than a hardware cache arrangement. Using our example, we also highlight challenges with the approach and use these as a call for more research efforts to help unlock the full potential of die-stacked memories.

1. Introduction

Three-dimensional integration technology enables the combination of multiple silicon die in the same package with high-density, low-power interconnects. The die can be stacked vertically [5] as shown in Figure 1(a), side-by-side on a silicon interposer [2], or both ways as shown in Figure 1(b). Memory vendors are manufacturing memories with multiple layers of DRAM [3, 7, 11, 13].

Die-stacked memories are projected to have capacities from hundreds of megabytes to a few gigabytes. For mainstream computers such as low-to-mid-range PCs, laptops, and netbooks, it may be plausible to 3D-stack the entire system's memory and not provide any conventional off-chip memory. For higher-performance PCs, extreme/gamer configurations, workstations, and especially enterprise, datacenter, and supercomputing servers, additional off-chip memory will still be required for capacity as well as the ability to expand memory. This results in a hybrid or *heterogeneous* memory organization in which some memory is located off-chip and some is located on-chip, as shown in Figure 2. Using stacked DRAM as a very large last-level cache provides some significant performance benefits, but we will later demonstrate that it still falls short of the potential of exposing the stacked DRAM as part of system-visible memory.

This paper explores the trade-offs between deploying a stacked DRAM as a large cache versus exposing the DRAM as part of heterogeneous system-visible memory. We study the performance potential of an idealized DRAM cache implementation compared to a simple approach for hardware-software co-managed system-visible stacked DRAM. The presented techniques are used primarily to highlight the challenges and point out areas industry could benefit from additional academic research and innovation in managing stacked DRAM.

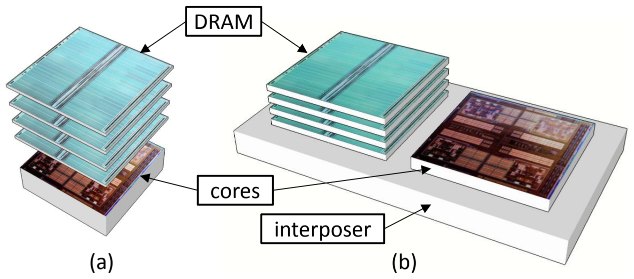

Figure 1: A multi-core chip combined with multiple DRAM chips using (a) vertical die stacking and (b) a combination of vertical and interposer-based stacking.

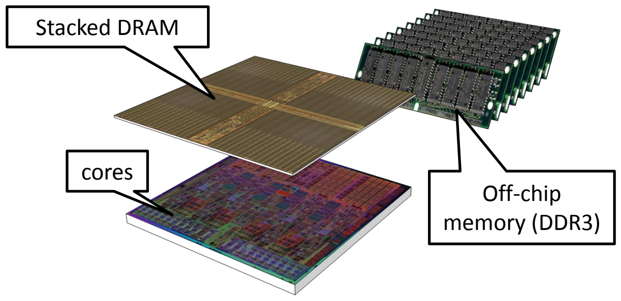

Figure 2: A heterogeneous memory organization with both stacked and off-chip memories.

2. Challenges with Stacked DRAM

Die-stacked memories can provide lower latency and higher bandwidth than is practical with off-chip memories [9, 14]. As a result, in systems with both stacked and off-chip DRAM, an additional level exists in the cache/memory hierarchy before leaving the chip to access off-chip memory. In this paper, we consider using stacked memory as a hardware-managed cache or as an OS-managed portion of a heterogeneous paged memory space. This section discusses the trade-offs between these two options.

2.1 Stacked DRAM as Cache

Managing stacked DRAM as a hardware cache has two key benefits. First, the hardware management mechanism sees all memory accesses in real-time and therefore can quickly react to changing memory access patterns. Second, because no software control is necessary, the benefits of stacked memory are immediately available to legacy applications and legacy OSs.

Managing stacked memory as a cache also has some drawbacks. Naive implementations can result in impractically high overheads for the tag storage. For example, a 1GB cache requires a 96MB tag array, assuming 64-byte cache lines.[1] Techniques such as page caching [6], sub-sectoring [8], and MissMaps [10] can help reduce tag overheads to more manageable levels, but significant on-chip resources (e.g., several megabytes) are still required. Managing stacked memory as an added level of cache may also require that coherency support be extended to another level of caching, with the inherent design and verification challenges.

A key disadvantage, or at least a major missed opportunity, for DRAM caches is that the hardware-only approach fails to exploit other higher-level information that can lead to more effective utilization of the stacked DRAM's capacity and bandwidth.

2.2 Stacked DRAM in a Heterogeneous Memory Organization

Managing stacked DRAM as paged memory under software control mitigates the key drawbacks of managing it as a cache. No tag storage overhead is incurred in this case, the latency of the tag lookup is also removed from the overall access latency, and software-level information can be leveraged to efficiently manage this resource.

A software-managed memory, however, is unable to respond to changes in memory working sets as quickly as a hardware-managed cache. Because any memory management decisions require an interrupt, associated handler overheads, memory copying/migration, and resulting TLB updates, these decisions can be made only infrequently. As a consequence, the OS is largely unable to use the stacked memory to provide any benefit for pages that have high utilization for very short periods of time.

Today, the OS has limited visibility into the usage patterns of memory and may have a difficult time trying to determine a good set of pages to map into stacked memory. Each page-table entry typically contains a "referenced" bit that the OS can clear. On accessing this page, the processor sets the bit. This provides the OS with a low-resolution indication of page usage: it knows whether or not a page has been touched since the bit was cleared, but not how many times. The problem is compounded by the volume of information that needs to be tracked. The OS sees all allocated memory regardless of whether the pages are hot or cold. Among all of these pages, the OS needs to deduce the pages that provide the greatest benefit from being allocated into stacked DRAM.

The limited information available to the OS is an artifact of current hardware and software interfaces. With richer hardware monitoring, feedback from runtime/JIT systems, advanced compiler analysis, and hints or even direct guidance from the programmer, the OS could employ sophisticated algorithms to make much better use of stacked DRAM than any hardware-only cache-based approach ever could.

3. Example Support for Stacked System Memory

This section presents an example approach for supporting system-visible stacked DRAM using a combination of hardware and software support. This appraoch has limitations and weaknesses, but it provides a framework for discussing the open research problems associated with supporting heterogeneous die-stacked memories.

3.1 Hardware Memory Utilization Tracking

The system software[2] is responsible for allocating user virtual pages to machine physical pages. To maximize performance, the OS should attempt to map the most frequently used pages to stacked DRAM memory and the less frequently used pages to conventional off-chip memory. In this context, the memory "uses" are those accesses handled by the memory controller; L1 or L2 hits are not of interest because they are not impacted by the performance of either off-chip nor stacked DRAM.

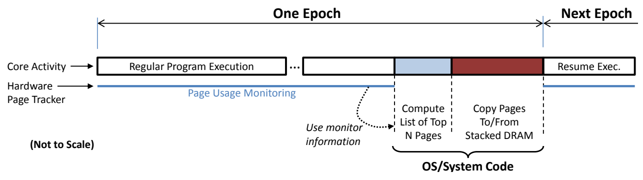

Figure 3: High-level overview of the regular program execution, hardware page tracking, and OS interaction.

The problem effectively boils down to tracking every active page of memory and recording the number of LLC misses per page.[3] For a stacked DRAM with a total capacity of P pages, the OS should choose the P most frequently missed pages and map those into the stacked-DRAM address range. The OS should not generate a new memory mapping too frequently because of the overheads involved in computing the list of the P most frequently missed pages, physically copying pages of memory back and forth between stacked and off-chip memories, and flushing the TLBs. We assume a framework in which the hardware page-tracking mechanisms collect information about page miss counts over the course of an epoch that lasts millions of cycles. At the end of each epoch, the OS uses the collected information to generate a new mapping. A key assumption of this approach is that a page's frequency of misses in one epoch serves as a good indicator or predictor of its miss behavior in the next epoch. Figure 3 illustrates the overall process of both hardware and OS components.

3.1.1 TLB Counter Annex with Page-table Walker Support

Hardware Page-table Walkers: Modern processors employ a logic block called a page-table walker (PTW) that is responsible for handling TLB misses. A given processor typically has a fixed convention for how the OS should lay out its page tables, and uses a few hardware registers to point at the base addresses of the various page-table data structures. As long as the OS organizes the page table following the processor's conventions, the PTW can automatically lookup page table entries on a TLB miss without OS intervention.

An typical page table entry (PTE) contains (among other things) a "dirty" bit and an "accessed" bit. The dirty bit is set in the TLB whenever a write is performed to the corresponding page, and this informs the OS that if the page is ever evicted from memory, then the OS must also write this new/dirty version of the page to the backing store (e.g., disk). The accessed bit is set whenever the page is accessed by either a read or a write. By periodically resetting and then checking the accessed bit, the OS can determine whether or not a particular page has been accessed, which is useful to know for making page replacement decisions.

When the processor suffers a TLB miss, the PTW must evict one of the current TLB entries (assuming all TLB entries are valid and in use). The PTW writes the PTE back to the page table in memory so the updated states of the dirty and accessed bits get propagated to the page table and therefore become visible to the OS. The PTW can then read in the new PTE corresponding to the original TLB miss. The exact sequence of operations depends on whether the

[1]This overhead calculation assumes 48-bit physical addresses, cache replacement meta-data, cache coherence state, and sharer/inclusion state, for a total of six bytes of tag overhead per cacheline.

[2]This could be the traditional OS, but could also include entities such as virtual machine managers/hypervisors.

[3]To be precise, the problem is really to track the pages that will have the most *future* LLC misses, but this is impossible without oracular foresight.

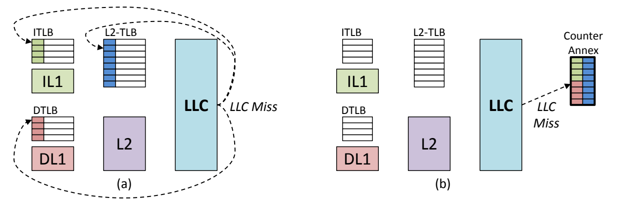

Figure 4: (a) Augmenting each TLB entry with LLC miss counters and the respective update paths (dashed arrows) from the LLC miss-detection logic, and (b) managing the LLC counter update by factoring the counters into a separate Counter Annex structure.

TLB is managed in a write-back or write-through fashion, and these issues are discussed further in Section 5.

The TLB Counter Annex: We modify the PTE to include a field that records the number of LLC misses to the associated page. Conceptually each TLB entry is also augmented with this additional field to store the LLC miss count, and the counter gets incremented by the hardware on each LLC miss, shown in Figure 4(a). Placing the counter directly in each TLB entry is problematic for several reasons. First, TLB traffic increases because each LLC miss now results in an extra read-modify-write of the corresponding TLB entry, leading to port contention that stalls real load and store lookups. Second, a new TLB access path from the LLC must be added, leading to more complexity in the TLB access-arbitration logic and potentially impacting the performance-critical DL1/DTLB timing paths. Third, the LLC is physically distant from the multiple TLBs (ITLB, DTLB, and then multiplied by the number of cores), and so we would also have to add several long global paths that could lead to routing congestion and timing difficulties in the physical design.

We separate the logical per-entry counters into a separate physical table, the *Counter Annex*. The table is located near the LLC's miss-handling logic, shown in Figure 4(b), so it is easily accessible on an LLC miss and avoids all the implementation issues noted previously. There is a slight overhead in that each entry of the Counter Annex must have a tag/physical page number field to facilitate lookups, causing every TLB tag to be replicated (once in the original TLB entry and once again in the Counter Annex).

Loads and stores that hit in the on-chip caches access the normal TLBs to obtain translations, but they do not access the Counter Annex. On an LLC miss, the physical page is looked up in the Counter Annex and the matching entry's counter is incremented. Eventually when the corresponding TLB entry is evicted from the processor core's TLB, the PTW will transfer the TLB's dirty and accessed bits back to the memory copy of the corresponding PTE. In our scheme, the counter field held in the Counter Annex also needs to be written back to the in-memory copy of the PTE. This is accomplished simply by piggy-backing on the dirty/accessed-bit writeback procedure. The PTW logic grabs the counter value from the Annex and writes it back with the other bits in a single operation.

3.2 Operating System Interactions

The hardware support described in Section 3.1 provides the raw usage information for the OS; the OS still must choose an effective set of pages to map to stacked DRAM.

3.2.1 Tracking Only the Referenced Set

While the Counter Annex along with the PTW provides accurate information about which pages incur the most LLC misses, it may still be very expensive for the OS to walk the entire page-table structure each epoch when it decides which pages to remap. To avoid a full page-table walk every epoch, the OS could instead keep a list of only those pages referenced during an epoch. A hardware-based approach to this is to have the PTW also set access bits in the parent nodes of the page table (assuming a multi-level page table organization, which is typical). In this fashion, the OS needs to traverse only the sub-trees that contain pages that were accessed in the most recent epoch, making the overhead proportional to the number of pages referenced per epoch rather than the total number of pages active per application.

An alternative is to have the OS regularly scan segments of the page table and then make note of any pages that have been accessed. This can be incorporated into existing OS routines that are used to collect information to aid page-replacement decisions. This routine adds newly referenced pages to an auxiliary data structure that tracks all referenced pages; this data structure could make use of algorithms such as Bloom filters to add new pages to the set quickly and query for set membership (false positives only cause the consideration of a few pages that were not actually referenced).

3.2.2 Page Selection Algorithm

For a stacked DRAM with a capacity of P pages, we start with the simple approach of selecting the P pages with the most LLC misses observed during the current epoch. The OS walks the list of *referenced* pages and builds a new list of those with the highest counter values. This algorithm assumes that a page that experiences a large number of LLC misses in the current epoch will continue to have a lot of misses in the next epoch and selects pages for mapping to minimize off-chip bandwidth.

To prevent the mapping of pages with insufficient miss traffic, we employ a threshold θ such that any page with fewer than θ LLC misses is not considered for mapping into stacked DRAM. The application of a threshold may result in cases when the list of most frequently missed pages has only $k < P$ items. In this case, we simply keep a random set of $P - k$ of the existing pages from the previous epoch already in stacked DRAM to avoid consuming bandwidth to swap out the page back to the off-chip memory. Any selected pages that are common to the previous epoch and the current one are not moved, so the total data transfer overhead is only proportional to the changes in the sets of mapped pages.

We use epoch lengths on the order of several milliseconds to reduce OS overheads. For multi-programmed workloads, the OS may be able to hide some of this overhead by remapping one application while the remaining programs continue executing. For shared-memory multi-threaded programs, however, all cores must wait until the remapping has completed. The OS can first prepare the list of pages to be remapped by using an idle core (or SMT thread) to walk the list of referenced pages, perform the sort, and determine the list of pages that need migrating. Only at this point does the OS suspend the multi-threaded application and perform the page migrations.

4. Evaluation

4.1 Methodology

Our performance simulations use the gem5 x86 simulator [1]. We model an eight-core server configured as described in Table 1. Each L2 cache is shared between a pair of cores [4], and the L3 is shared across the entire chip.

Stacked DRAMs are projected to have capacities of hundreds of megabytes. To exercise the mixed stacked/off-chip memory organizations discussed in this work, we make use of five multi-threaded server workloads with large memory (mostly 1GB or more) footprints.

Processors		Caches	
Cores	8	**IL1 and DL1**	32KB, 2-way, 2 cycles
Clock	3.2GHz	**L2**	2MB, 8-way, 10 cycles
Issue Width	1 IPC	**L3**	8MB, 16-way, 24 cycles
Off-Chip Memory		**Die-Stacked Memory**	
Bus	64-bit, 1.6GHz DDR	**Bus**	128-bit per R/W, 3.2GHz DDR
Channels	4	**Channels**	8
Banks/Chan	8	**Banks/Chan**	16
Row Buffer	2KB	**Row Buffer**	2048
$t_{CAS}, t_{RCD}, t_{TP}, t_{RAS}, t_{RC}$	9-9-9-36-33	$t_{CAS}, t_{RCD}, t_{TP}, t_{RAS}, t_{RC}$	9-9-9-27-33
$t_{WR}, t_{WTR}, t_{RTP}, t_{RRD}, t_{FAW}$	10-5-5-5-30	$t_{WR}, t_{WTR}, t_{RTP}, t_{RRD}, t_{FAW}$	9-0-5-5-30

Table 1: Simulated server configuration. DRAM timing parameters are specified in multiples of memory bus cycles.

4.2 Performance Results

For performance comparisons, we start with a baseline server without any stacked DRAM. All performance results are normalized to this reference. When using stacked DRAM as a large L4 cache, we assume that the cache is 32-way set associative, and uses 64-byte cachelines, and we ignore the tag overhead that would be required to support such a cache (approximately 12MB of tag per 128MB of DRAM capacity). We generously assume that the tag array has a fixed ten-cycle lookup latency.

We also consider an "oracle" mapping mechanism. We first execute each benchmark and collect a list of each page ever accessed by the program along with the number of accesses (i.e., LLC misses) per page. We then sort the pages by the number of accesses, and then run the simulation again assuming the top P pages are mapped to stacked DRAM. Note that this should not be misconstrued as an upper bound for the general approach of OS-managed stacked memory. A truly optimal solution would also take dynamic program phases into account and continually adjust the mappings based on the remaining future access patterns of the program.

Figure 5 shows the per-benchmark performance for a 128MB stacked DRAM. The data labeled SSM stands for stacked system memory and corresponds to the described OS-controlled heterogeneous memory organization. The first set of SSM data does not include any of the overhead for migrating pages between off-chip and stacked DRAM, whereas the second set of data (SSM w/ Overhead) does. We tuned the algorithm parameters separately for the two cases. When ignoring overheads, we use an epoch length of 10 million cycles and set $\theta = 1$ because overheads are ignored. When overheads are included, we always use an epoch length of 100 million cycles (31.25 milliseconds on a 3.2GHz processor) and set $\theta = 64$.

When overheads are not considered, SSM's simple sorting approach is effective at identifying pages that should be mapped into the stacked DRAM. This is demonstrated by the reasonably close performance between SSM (hashed bars) and the oracle selection approach (black bars). Once the transfer overheads are accounted for, a significant amount of the potential performance is lost. Although on average the SSM approach (with overheads) still performs better than using the 128MB stacked DRAM as a cache, for specific benchmarks, such as SPECjbb05 and SPECweb05, the overheads are too much and the cache approach works better. The trends are similar as the DRAM cache size is increased up to 1GB (not shown due to space constraints).

Effective selection of pages can result in high utilization of stacked DRAM. Figure 6 shows the percentage of memory accesses (LLC misses) that are serviced from stacked DRAM for different stacked DRAM capacities. When using stacked DRAM as a cache, apart from the Web-Index benchmark, increasing the cache size does not significantly increase the amount of traffic served by the stacked DRAM. This is largely due to compulsory misses (we modeled 512-way caches as well, and the miss rates were very similar to

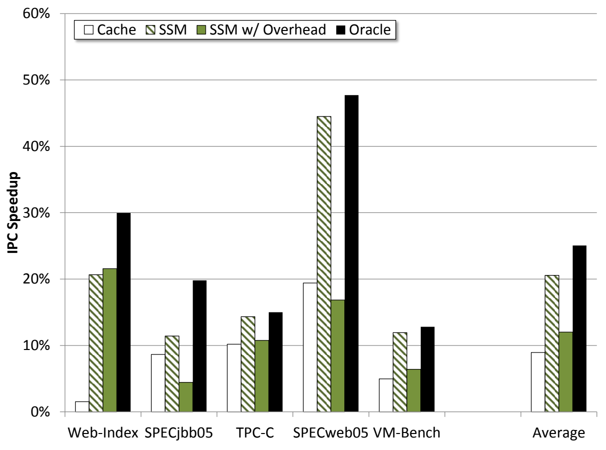

Figure 5: Performance improvement relative to a server with no stacked DRAM (higher is better). Stacked DRAM size is 128MB for these results.

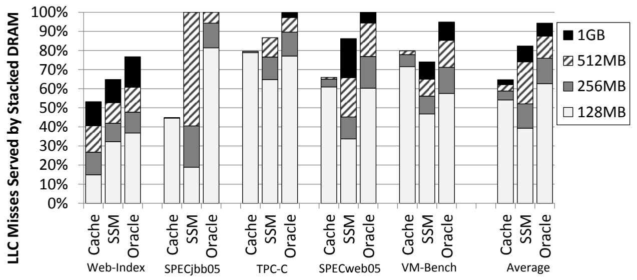

Figure 6: Percentage of LLC misses served by stacked DRAM for different stacked DRAM capacities.

the default 32-way configurations, indicating that the misses are not due to set conflicts). For the SSM approach, observing memory behavior on longer timescales (epochs) effectively filters out short-term variations and allows the sorting algorithm to focus on pages with consistent, long-term re-use. As such, SSM is able to make better use of the stacked DRAM resource as capacity increases.

Note that the range of overall performance increases (approximately 12-48% for oracle) may seem lower than one might expect. This is because, for these benchmarks and the baseline system, four memory channels satisfies a large portion of the applications' bandwidth needs, the applications are largely tuned for today's system configurations, and we have set the stacked DRAM latencies somewhat conservatively. We have experimented with application settings that generate more memory requests as well as reducing the number of off-chip memory channels, and the relative performance benefit does increase accordingly, but the overall trends remain the same.

5. Discussion, Open Issues, Research Needs

The results in the previous section show that managing stacked DRAM as system-visible heterogeneous memory appears to have more performance potential than simply using it as a large cache. The ceiling on performance is going to be higher than our experimental results have shown because the "oracle" policy is not even a true upper bound; it uses a fixed, static set of pages mapped to the stacked DRAM. Despite these encouraging results, there remain many challenges and open questions before developing a full hardware-software cooperative system that can really make great use of stacked DRAM.

5.1 Implementation Concerns

5.1.1 Page Table Format

In some architectures (e.g., x86), the page table does not have very many unused bits, and so placing a multi-bit usage counter in the PTE may not be practical. One alternative is to build an auxiliary data structure in parallel with the page table, although this has some implementation consequences. In particular, the hardware PTW must now know about this structure so the updates can be made correctly. Alternatively, an update mechanism other than the PTW needs to be used.

5.1.2 PTW Write Policy

The description of the PTW approach for updating PTE counters during TLB evictions depends on the TLB using a writeback policy. However, a writeback policy for the TLB can slow the eviction and fill process on a TLB miss. A write-through policy is fairly effective for the TLB because the number of updates is very limited (at most once for the initial access, and once more when the page is modified). Policies and mechanisms that rely on the TLB or PTW need to be carefully coordinated with the update and write policies of these structures.

5.1.3 Operating System Overheads

The effectiveness of an OS page-selection policy depends greatly on the overheads of invoking the algorithm and migrating pages between stacked and off-chip memories. We ran several experiments varying the different parameters of our approach and observed that overall performance is – not surprisingly – quite sensitive to the overheads of page selection and migration. The overall guidance is that any approach for OS-managed heterogeneous DRAM must very carefully control the overheads associated with the page selection algorithm and remapping process. The evaluated scheme uses a somewhat brute-force approach by simply sorting all referenced pages by miss counts; more sophisticated algorithms that make use of other information may be able to predict the best pages to place in stacked DRAM while reducing overheads.

5.1.4 Multi-vendor Cooperation

Hardware-software co-managed stacked memory has the potential to cope with the Memory Wall, but to make this a reality, there needs to be significant cooperation and coordination between hardware and software vendors. Processor companies need to implement the right monitoring hardware to collect useful statistics and information to aid the OS's page-selection process. The OS needs to implement significant support for both the page-selection algorithms as well as the page-migration mechanisms. None of these have significant value in isolation; a commitment to all must be made to unlock the potential of stacked DRAM.

A significant risk in the development of OS-managed stacked-DRAM architectures is the chicken-and-egg problem. The hardware companies may wait until the OS vendors have committed to implementing the software support, and of course the OS vendors do not want to invest development costs in a feature unless the hardware is guaranteed to support it. Such a cyclic dependency could cause delays in developing the necessary technologies. However, this is an opportunity for academic research efforts to make significant contributions. University research groups easily can prototype various hardware approaches in simulation, and modify OS kernels (e.g., using open-source distributions) without needing to support the changes.

5.2 Opportunities

There remain many open questions about how to best implement a hardware-software cooperative architecture to use system-visible stacked memory. Some of these are discussed below.

5.2.1 Richer Monitoring

The evaluated hardware monitoring approach tracked only basic per-page LLC miss counts. It is likely that there exists more useful information that would better aid the OS in making its page-selection decisions. For example, looking at raw LLC misses does not distinguish between the case when one or a few cachelines are responsible for all of the misses versus a case when misses are coming from across the page.

The simple LLC miss counts also do not provide very much temporal information beyond having the OS track how the counts vary from epoch to epoch, which may be too slow to be of much use, especially if epoch lengths are long to keep overheads low. LLC miss counts (or other statistics) using time-decaying moving averages, vectors of counts over some number of time intervals, or other information may provide the OS with a better sense of whether a page is starting to heat up or cool down.

For the purposes of selecting a set of pages to map to stacked DRAM, it is likely that noisy or lossy monitoring of memory behavior is sufficient. For example, swapping the sort order for the most missed and second-most missed pages does not make any difference because both will be selected for mapping to stacked DRAM. This observation could likely be much more deeply exploited to create lighter-weight monitoring schemes that provide just enough information for the OS to perform a good job in selecting pages.

5.2.2 Software Assistance

The preceding discussion focused on hardware-only monitoring schemes. However, the software stack contains a significant amount of information that is likely valuable for making good page-selection decisions. For example, the OS or run-time is aware of memory allocations and deallocations (e.g., malloc/free), and can leverage this high-level information to influence its page-selection process (e.g., avoid selecting pages that were just recently deallocated). The compiler can also assist by managing data placement and data object location in a way that co-locates frequently used items on the same pages. This may be somewhat challenging in general because the compiler needs to know how many LLC misses the different data will generate, which is harder to estimate than the total number of accesses. For some types of data structures, however, the compiler may be programmed to detect common idioms (e.g., large arrays) that are more likely to benefit from being pinned in stacked DRAM.

5.2.3 Other Page Selection Criteria

This sub-section discusses a few other ways that the OS can use the page monitoring and mapping mechanisms. Research in better algorithms, coordinated with research in determining the best complementary hardware monitoring, can potentially provide much better selection decisions with lower OS overheads.

Application-level Guidance: There may be scenarios in which applications contain working sets in which pages frequently cause a large number of LLC misses *and* the sum of all of these pages exceeds the capacity of stacked DRAM. Examples may include databases or web-index search engines that simply deal with very large bodies of data. For such programs, the overhead of computing the set of most frequently missed pages becomes much higher because of the large number of over-threshold referenced pages.

An alternative is to allow the application programmer (or compiler or profiler) to provide input on what pages would likely be beneficial for mapping into stacked DRAM. In the case of databases, it may be desirable to map the root node and first the few layers of the database B-tree (or equivalent data structure) into stacked DRAM. The programmer may also modify the structure of the B-tree so the first few levels have higher branching factors to make better use of stacked DRAM. For web search, structures such as query caches or the first tier of a multi-tiered index [12] could be mapped into stacked DRAM. Similar to the database case, algorithmic changes may be required to get the most benefit out of stacked memory (e.g., splitting the memory-resident tier of an index into two tiers corresponding to the stacked and off-chip memories).

Stack-aware Page Allocation: As described, the evaluated page-selection scheme is reactive and moves pages to stacked DRAM only after demonstrated re-use. The monitoring and subsequent page migration overheads could be removed if the compiler and/or runtime are able to predict which pages will be beneficial to place in stacked memory and then directly allocate these pages accordingly.

Supporting OS Priority Levels and QoS: Different applications have different priorities. For example, a background application performing video transcoding may be relatively memory-intensive, but it may make sense never to map any of its pages to stacked memory at the cost of mapping out a page (even if less frequently accessed) of a high-priority interactive application. One straightforward way to introduce OS priority levels into the page-selection algorithm is to weight all counter values according to priority. The algorithm then simply selects the top P pages based on their weighted LLC miss-count values. Background tasks that should never be mapped into stacked DRAM receive a weight of zero and appear to the algorithm as having no misses at all.

To provide fairness among multiple applications that may have equal importance/priority but exhibit varying numbers of high-LLC-miss pages, the OS can perform its page selection in a fashion that guarantees each application a certain minimum allocation of m pages in stacked DRAM. The algorithm first maps the top m pages from each application, and then allocates the remaining $P - Nm$ pages (for N cores) based purely on the LLC miss counts. Each application i could have a different minimum allocation m_i depending on priority levels. The ability to guarantee certain amounts of stacked memory can also be helpful in reducing the worst-case execution times of real-time applications.

NUMA Systems: OS's for multi-socket multi-processors already have to deal with non-uniform memory architectures in which different regions of the address space exhibit different performance characteristics. Algorithms and heuristics for NUMA systems may be adapted to assist in the stacked-DRAM page-selection problem. The stacked-DRAM problem may be slightly simpler, because the latency to a region of memory in a NUMA system depends on the socket from which a request originates, whereas moving a page from off-chip DRAM to stacked DRAM is faster for everyone. Also note that the non-uniformity introduced by stacking DRAM is orthogonal to conventional NUMA issues; it is a new dimension in that one could have a multi-socket system in which each socket's address range is further divided into stacked and off-chip memories.

6. Conclusions

This paper has explored many of the issues surrounding how to expose a die-stacked, heterogeneous memory system to the OS. The example system demonstrated that this is a generally promising approach: our simple architecture provided better results than an idealized DRAM cache, and the performance potential is likely even greater than what our oracles studies showed. The simplicity of the evaluated scheme also helps demonstrate the plethora of interesting research directions that must be pursued to develop a compelling hardware-software cooperative and co-designed architecture to exploit stacked DRAM. The challenges in coordinating advanced development and research efforts between hardware and software companies makes this area particularly ripe for university efforts to make significant contributions and impact.

7. References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. gem5: A Multiple-ISA Full System Simulator with Detailed Memory Model. *Computer Architecture News*, 39, June 2011.

[2] Y. Deng and W. Maly. Interconnect Characteristics of 2.5-D System Integration Scheme. In *Proc. of the Intl. Symp. on Physical Design*, pages 171–175, Sonoma County, CA, April 2001.

[3] Elpida Corporation. Elpida Completes Development of TSV (Through Silicon Via) Multi-Layer 8-Gigabit DRAM. Press Release, August 27, 2009. http://http://www.elpida.com/en/news/2009/index.html.

[4] T. Fischer, S. Arekapudi, E. Busta, C. Dietz, M. Golden, S. Hilker, A. Horiuchi, K. A. Hurd, D. Johnson, H. McIntyre, S. Naffziger, J. Vinh, J. White, and K. Kilcox. Design Solutions for the Bulldozer 32nm SOI 2-core Processor Module in an 8-core CPU. In *Proc. of the Intl. Solid-State Circuits Conference*, pages 78–80, San Francisco, CA, February 2011.

[5] S. Gupta, M. Hilbert, S. Hong, and R. Patti. Techniques for Producing 3D ICs with High-Density Interconnect. In *Proc. of the 21st Intl. VLSI Multilevel Interconnection Conference*, Waikoloa Beach, HI, September 2004.

[6] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based dram caching for CMP server platforms. In *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, pages 1–12, January 2010.

[7] J.-S. Kim, C. Oh, H. Lee, D. Lee, H.-R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S.-K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, K. Kyung, J.-S. Choi, and Y.-H. Jun. A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os Using TSV-Based Stacking. In *Proc. of the Intl. Solid-State Circuits Conference*, San Francisco, CA, February 2011.

[8] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[9] G. H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proc. of the 35th Intl. Symp. on Computer Architecture*, Beijing, China, June 2008.

[10] G. H. Loh and M. D. Hill. Supporting Very Large Caches with Conventional Block Sizes. In *Proc. of the 44th Intl. Symp. on Microarchitecture*, Porto Alegre, Brazil, December 2011.

[11] J. T. Pawlowski. Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem. In *Proc. of the 23rd Hot Chips*, Stanford, CA, August 2011.

[12] V. J. Reddi, B. Lee, T. Chilimbi, and K. Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proc. of the 37th Intl. Symp. on Computer Architecture*, St. Malo, France, June 2010.

[13] Samsung Electronics Corp. Samsung Electronics Develops New, Highly Efficient Stacking Process for DRAM. April 23 2007.

[14] Tezzaron Semiconductors. Tezzaron Unveils 3D SRAM. Press Release from http://www.tezzaron.com, January 24 2005.