HW assisted migration for hybrid memories OR MemPod (or MemoPod): Memory Clustering for Efficient and Scalable Migration in Flat Address Space

ABSTRACT

Die-stacked DRAM is an emerging technology that has been annoruced to be included with off-package memories resulting in a hybrid memory system. A large body of recent research has investigated the use of die-stacked DRAM as a hardware-manged last-level cache. This approach comes does not expose die-stacked memory for application use which could be beneifical to memory capacity constrained workloads. An alternative approach is to manage both memories as flat address space as part of main memory. Performance of flat addresss space requires efficient page placement and migrations such that most memory access are from the higher performance die-stacked memory. One apporach is for the operating system (OS) to monitor memory access and periodically migrate pages, however OS is limited to coarser granularities migration periods due to overheads of page table updates, and TLB shootdowns. In this paper we describe a clustered hardware migration architecture that transparently migrates pages that can scale to arbutrary number of channels. We also design scalable solutions to storing page access tracking and page remapping tables that can scale to future systems with terabytes of memory. Our results for evaluation of multi-programmed workloads of parsec CPU and rodinia GPU applications show that our solution has **XX%** better performance over recent flat-address space management schemes.

Keywords: Memory architecture, Die-stacked memory

1. INTRODUCTION

In recent years improvement of system performance has been impeded by the memory wall problem [19]. To alleviate the memory wall problem there has been a lot of recent research in the use of 3-D die-stacked memory to provide high performance. Placing 3D memory stacks in the same package as the processor can provide significant improvements in bandwidth and lower power consumption [1]. There has has been significant advancement in the industry including the development of die-stacked memory standards and consortia [6, 9, 15], and various announcements from several processor companies [5, 14, 1].

Current stacking technology may provide on the order of eight 3D DRAM stacks, each with 2GB capacity, for a total of 16GB of fast DRAM [5]. However, many server systems already support *hundreds* of GB of memory and so a few tens will not suffice for the problem sizes and workloads of interest. The resulting system will therefore consist of two types of memory: a first class of fast, in-package, die-stacked memory, and a second class of off-package commodity memory (e.g., double data rate type 3 (DDR3)).

Given such a *Two-Level Memory* (TLM) hybrid memory organization, the challenge then comes from determining how to best organize and manage this system. The goal of any management is to give the performance of the fast in-package memory while still providing the capacity of the larger off-package memory. A large body of recent body of recent research has focused on utilizing the stacked DRAM as a large, high-bandwidth last-level cache (e.g., an "L4" cache), coping with the challenges of managing the large tag storage required and the relatively slower latencies of DRAM (compared to on-chip SRAM) [10, 16, 7, 8, 20, 18, 13, 3, 4]. Such a hardware caching approach has some immediate advantages, especially that of software-transparency and backwards compatibility. As the stacked DRAM is simply another cache that is transparent to the software layers, any existing applications can be run on a system with such a DRAM cache and potentially obtain performance and/or energy benefits [11]. One drawback of the caching approach is that stacked memory is not available for application use. While the capacities of die-stacked memory is likely to be insufficient to serve as the entirety of a system's main memory it is still non-trivial in size and would be beneficial to memory capacity constrained workloads [2].

An alternative configuration is to expose the die-stacked memory as part of the main memory capacity. In this configuration the management and efficient use of the TLM hybrid memory lies on the application and the underlying management option. One recent research explored management by the operating system (OS) [12]. The OS monitors memory usage and periodically performs page migrations to move frequently accessed pages to stacked memory. One of the many challenges that OS or any runtime implementation faces is that it is constrained by the overheads of management. The OS needs to take an interrupt to do anything, additionally the OS then has to traverse the page tables to read the access frequency information, migrate pages and update the page table entries to reflect the change of virtual to physical address

mapping and flush the translation lookaside buffers (TLB). As we show in our evaluations, these costs are non-trivial and constraint the OS to set the minimum management intervals or epochs to 0.1 seconds. Such coarse grained intervals leaves a lot of opportunities on the table.

Most recently [17] proposed a transparent hardware schem to manage TLM as a flat address space. The main contributions of that work was the use of storage efficient structures to track page access frequency and remapping tables. They allow migrations to happen between any aribitrary channels and the use of single centralized migrator can become a bottleneck when systems scale to terabytes of memory and 10s of channels. To this end we in this paper we introduce the migration cluster architecture that is specifically designed to scale to systems of future. Our migration clusters group channels of different memory levels into smaller cluster sets and migration is limited to within cluster only. In this clustered architecture we also use page access counting hardware that can store information for millions of pages as well as we use remap tables that can remap millions of pages in the cluster efficiently. We also describe and evaluate novel migration algorithms that are able to optimize for latency and bandwidth sensistive workloads in a CPU-GPU system.

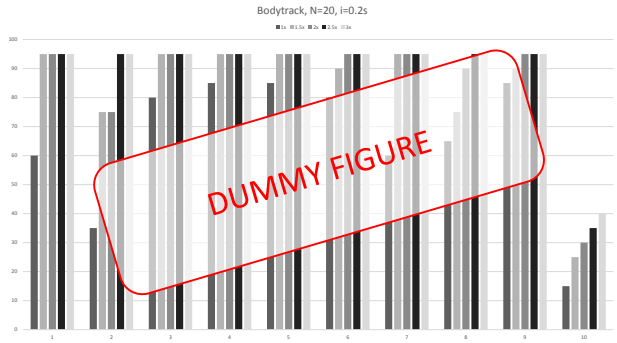

Figure 1: Dummy figure caption 2

Show graph with motivational result.
List the paper's main contributions.
Overview of paper's chapters.

2. BACKGROUND

Description of memory organization
Differences between DDR, HBM, PCM, on-chip/off-chip, benefits and drawbacks of each memory type.
Present the experiment where we identify the optimal NLM organization

3. RELATED WORK

Present the papers in a structured way. Organize them in categories and discuss a category at a time. E.g. HW-assisted, SW-assisted, Hybrid. Separate category for irrelevant papers that worked as motivation.
List all the papers we think are relevant. Dicuss each one briefly.
List the elements that make our work better than the rest.

4. ARCHITECTURE

Our clustered migration mechanism was carefully designed to address key challenges associated with the migration problem. In this section, we present a complete description of our micro-architectural design, followed by a breakdown of all important design decisions made, along with the corresponding challenge addressed by each one.

4.1 Clustered Migration Architecture

Figure 2 presents an overview of MemPod. MemPod's design was kept modular to facilitate system integration. A number of memory pods are injected between the LLC and the system's MCs. Each pod clusters a number of MCs and restricts migration within the pod. To the rest of the system, pods are seen as MCs. With MemPod's transparent design, each pod will now be receiving all the requests originally addressed to any of the pod's MCs.

A pod's operation when a memory request arrives would be to monitor the request, update any necessary migration-related counters and forward the request to the intended recipient MC. The migration logic within a pod does not need to be invoked during a response from any MC and could potentially be bypassed, saving some cycles. An obvious drawback of clustering MCs into pods is the serialization of potentially parallel requests to different MCs and as such, any counting scheme used by the pod – as well as the actual forwarding of the request – have to be as efficient as possible.

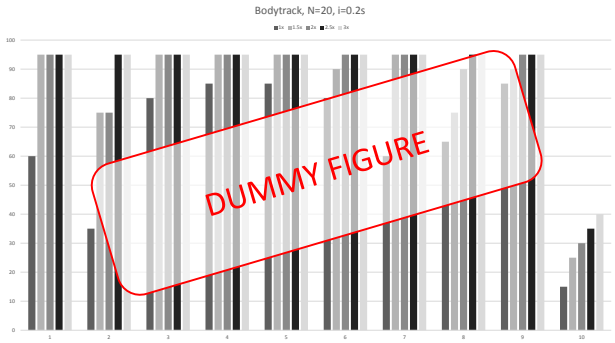

Figure 2: MemPod high-level architecture

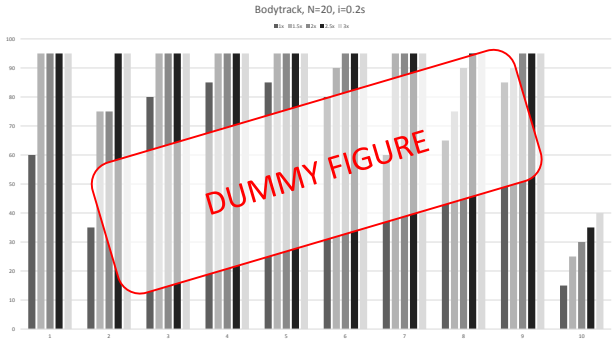

Figure 3: Major architectural Pod elements

Memory Pod

The major architectural elements of a Pod are presented in Figure 3. A Pod's remap structure includes the remap table as well as necessary decoding and hit detection logic. Counting logic is equally important for migration as the existence of a remap table and it's responsible for identifying the hot pages which will later be candidates for migration. Without reliable hot page detection, migration would be random and often counter-productive. Finally, migration logic is responsible for orchestrating a page swap between two MCs and updating the Pod's state.

During the design of a new system, the number of pods can vary arbitrarily given different constrains. A design with just one cluster would be equivalent to a centralized migration controller allowing any-to-any[1] migration, while a design with a pod number equal to the number of MCs would imply that migration is disabled. The latter option would be completely redundant and is only used as an example. A reasonable number of Pods would be equal to the number of slow-memory MCs. In Figure 2 we present a system with eight MCs for the fast, on-die stacked memory and four MCs for the slow off-chip memory. The use of four pods imposes few restrictions on migration possibilities, while forbiding migration between two slow-memory MCs. For the remainder of this paper, we set the number of pods to four.

The dollowing subsections provide detailed descriptions of all major elements of a migration management mechanism. We present MemPod's, THM's and HMA's approach to each element's design, as well as alternative designs. It's important to note that each of the following elements is independent from all the rest and future mechanisms could choose almost any element design combination in a plug-and-play fashion, with some exceptions. For example, the use of Majority Element Algorithm (MEA) activity tracking cannot work with threshold-based triggers.

4.2 Page Relocation and Remap Table Size

Migration of memory pages can provide maximum benefits when no restrictions are imposed on the available migration locations. In other words, the optimal scenario for a migration policy would be the option of potentially filling the entire fast memory with migrated hot pages. On the other hand, more options require more bookkeeping and incur a higher cost. Flat address space memories do not have the luxury of a backing memory, like a 3D-stacked DRAM cache. Consequently, migrating a page implies swaping two pages to ensure the existence of exactly one copy for each of the participating pages. As such, migrating and swaping will be used interchangeably for the remainder of this paper.

A traditional remap table is a hash structure, indexed by a page's address and pointing to the migrated address if one exists. On a page migration, the remap table is updated to reflect the new address of a migrated pge. However, such a remap table is not enough when re-migration of pages is allowed. Figure 4 presents a scenario where a "naive" remap table fails. Figure 4(i) shows the starting state of our memory before any migration, as well as the starting state of the remap table. For simplicity, we present the three memory locations needed by our example. Page 10 is assumed to be a fast memory page, while pages 100 and 200 are slow memory pages. The numbers inside the memory locations represent the content page's id. Figure 4(ii) shows the state of memory and remap table after swaping pages 10 and 100. The content of page 10 is now page 100, and the content of page 100 is now 10. The remap table correctly states that requests to page 10 should be relayed to page 100 and vice versa. Everything works as it should during this first migration, however Figure 4(iii) shows the state after the second migration. Page 10 is now swaped with page 200. Such a migration would imply that page 100 (now held at 10) became cold and page 200 became hot. The contents are swaped and now page 10 holds page 200 and page 200 holds page 100. However, the state of the remap table is inconsistent. A request to page 10 would get forwarded to page 200, returning the wrong page.

The reason this remap table design fails is simply because pages are allowed to re-migrate – like page 10 in our previous example – while the remap table "assumes" the content held at a page address matches the page's ID. There is only one solution to this problem: The migration logic needs to be aware of exactly where each page's contents are located at any given time. Such a requirement can be implemented in various ways:

Safe and slow: Always restore a forwarded page's contents before it participates in a new migration. For such an implementation, hot page count will have to be kept based on the content page instead of the real page Id. In the earlier example in Figure 4, the second migration of page 10 implies that page 100 is cold, but in order to offer page restoration support, the second swapping of page 10 will have to imply that page 10 is cold. The cold page (10) will be restored back to address 10 from 100 and then moved to its new location. It's a minor modification that does not affect any other aspect of a migration mchanism.

THM approach: Migration is restricted in segments. In a memory configuration with a 1:8 fast:slow memory ratio, exactly 9 pages compete for a position at the one fast memory page available. This solution is elegant enough to allow re-migration without extremely high storage overhead, limiting however the migration potential of memory pages. If two or more hot pages coexist in a segment, only one can reside in fast memory at any given time. At the same time, if none of the segment's pages are hot, the fast memory page slot cannot be utilized by some other segment's page.

HMA approach: Any page can migrate to any other page address without the need of dedicated a remap table structure. The OS-based migration scheme imposes no limitations, since the OS takes care of updating the page tables and flush the TLB, however the cost of HMA's intervention and the penalties incurred from a cold TLB could sum up to very high values.

MemPod approach: Migration is restricted within a Pod, but no intra-pod location restrictions are applied. A Pod's remap table is extended with a second field that

[1] Any-to-any Migration: Page migration without limits on source and destination. All MCs can migrate a page to any MC.

holds the content page's id. During the second migration in our previous example, the issue was caused because we updated the remap entry of the holding page (10) instead of the entry of the content page (100). An attempt to recursively follow the remap table's entries until we figure out which one we should update risks looping infinitely because of cycles (For Example if page 10 is re-migrated back to address 10). Even if a smart algorithm is utilized to delete entries of non-migrated pages, the complexity of the recursive algorithm will only be bounded by the size of the remap table since in the worst case the entire table will be traversed. Figure 5 shows the memory and remap table states after the same page mirations as in Figure 4. At the end of the second migration, the states of memory and remap table are consistent. It's important to note that a remap table entry is now points to a pair of values: (1) *relay address* (i.e. where is the requested page located) and (2) *content address* (Which page is currently held).

Alternative approach: Recent works propose the use of arbitrarily small remap tables. When the remap table inevitably gets full two possible solutions exist: (a) Migration is disabled and (b) the OS is invoked to update page tables and flush the TLB. After OS's intervention, the remap table is cleared and migration remains active.

Discuss the cost of MemPod and THM approach in terms of storage overhead. They should be similar.

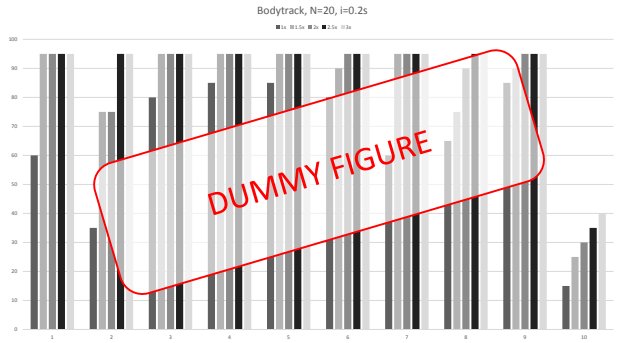

Figure 4: Naive remap table operation

4.3 Activity Tracking Mechanism

Activity tracking could be considered the most important element of any migration mechanism. In most migration studies, activity tracking becomes a synonym of identifying hot regions by counting the number of accesses. In a more generalilzed approach, it could potentially be extended to track patterns, parallelism, bit flips or any other information useful to the underlying mechanism. Along with the remap structure, activity tracking is the limiting factor in any migration policy. The overhead of maintaining a set of counters per migration segment, is often a bottleneck.

MemPod utilizes an important observation to maintain a low activity tracking cost. Moving the hottest pages into fast

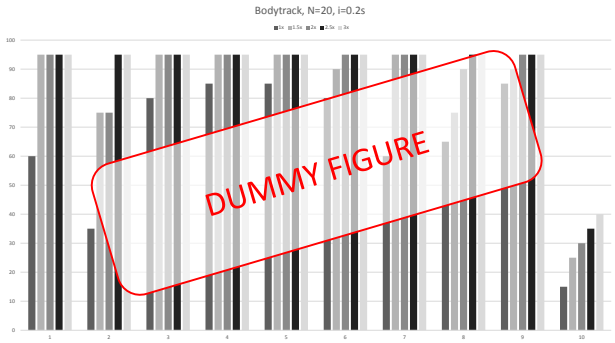

Figure 5: Remap design that allows re-migration

memory is a commonly accepted approach, but not necessarily optimal. Several scenarios expose the failure of such an approach. For example, a page might become cold soon after it's migrated, wasting a space in fast memory. Another example arises when interval based migration policies are used. A cold page of the previous tracking cycle could become hot during the next interval. Strong indications exist that a combination of temporal as well as spatial locality has the potential of exposing better results.

Frequently encountered solutions in the literature consist of increasing the activity tracking granularity (i.e. track a group of pages together), limiting the bits for each tracking counter, or simply paying the overhead for a complete tracking mechanism at the finer granularity. MemPod's activity tracking is designed with a novel approach, using a Majority Element Algorithm (MEA) to track the hottest pages at a low cost. To the best of our knowledge, such an algorithm was never used in this context. THM also presents an interesting tracking approach, by utilizing competing counters for each segment.

Using counters for every memory segment supported by the migration mechanism obviously imposes extremely high area overhead but benefits in accuracy. Identifying the hottest pages however, also requires the often-overlooked sorting complexity. With the introduction of new memory technologies and the continuous capacity increase in memory capacities, it won't be long before even the most efficient sorting algorithm will require more time than we are willing to spend.

THM approach: One 8-bit competing counter tracks each memory segment. As described in Section 4.2, THM restricts migration within segments. The competing counter is incremented by one when a page in slow memory is accessed and decremented by one when the segment's fast page is accessed. The counter's value is then monitored and can trigger migration. Competing counters represent a tradeoff between area overhead and accuracy. THM requires 8 bits per fast memory page making THM the most area-efficient as far as tracking is concerned. However, competing counters are susceptible to some error, since a cold page could potentially trigger migration and be placed in the fast memory.

HMA approach: Full activity tracking per OS page (4KB)

for all memory regions. THM uses the least efficient tracking mechanism in exchange for perfect knowledge at a fine granularity. Full activity tracking also introduces the complexity of sorting all the counters to identify hot pages. THM and MemPod do not require sorting.

MemPod approach: MemPod requires one counter per hot memory page ranking close to THM's area efficiency. However, the set of MemPod's counters is capable of tracking pages *from all memory regions.* using the Majority Element Algorithm presented in **cite**. For the remainder of this paper we will use the term "MEA counters" as a shorthand for MemPod's activity tracking. MEA is a simple, streaming algorithm which returns the (K) most frequently occuring elements in an array in linear time. As described in Section 3, MEA is guaranteed to return the set of K hottest pages under certain assumptions that are not commonly held in a stream of memory requests. A sensitivity analysis of MemPod's tracking mchanism's accuracy is presented in the experimenal evaluation section. MEA strikes a balance between most frequently occuring and most recently used page addresses, a fortunate and welcomed consequence in locality exposure. As already discussed, asking for the K hottest pages does not necessarily mean that every other page is completely cold. For example, the K+1 page will usually be a good choice for migration as well. A new limitation arises when MEA counters are used: The system will be presented with the set of K hottest pages, however the counters' values cannot provide an order. The absolute hottest page could have a lower counter value than any other page.

Even with the most efficient tracking mechanism, future designs will soon be required to cache some of their counters while the rest are stored in memory to aleviate the area overhead. THM's segment-based counters are automatically cached along with their corresponding SRT entry and restored whenever necessary, at the cost of a memory access. It is also important to note that any counter update should be moved off the critcal path. Extreme accuracy is not necessary for correct operation. Even if the migration mechanism is not as accurate as it could be, all memory requests will be able to retrieve correct information as long as the state of memory and remap structure are consistent. Updating a remap table entry however, needs to be performed accurately without any ambiguity to aleviate the risk of inconsistent state.

IDF: With the recent growth in PIM (Processing-In-Memory, sometimes called Near-Memory Computation) mechanisms **[cite]**, future migration designs could rely on the PIM module to update the necessary counters, off the critical path and stored entirely in memory until it's time to use the actual counter values. The PIM approach *eliminates* the need to store counters in SRAM circuits. Hypotheticaly, PIM could also be invoked for sorting the tracking counters.

4.4 Migration triggers

Deciding when to perform migration is as important as knowing which pages to migrate. Migration adds a significant delay to a system and as such it must be used wisely. Any penalties incurred whould be amortised by the performance improvement when placing a page in the fast memory. Requests that arrive while migration is performed have to be delayed to ensure functionaly correct behavior. Two very common triggers are used throughout the literature whenever state must be updated based on tracking information (such as MC scheduling, NUMA, DVFS etc.). Interval-based (or epoch-based) triggers occur with a steady frequency, while threshold-based solutions trigger without a predetermined frequency, whenever a threshold value is passed.

Both interval-based and threshold-based approaches face the same challenge of identifying the optimal interval or threshold value. Identifying the appropriate value is not usually a trivial task. Factors like a system's architecture, application's behavior, as well as semi-random factors (for example DRAM will refresh more frequently under higher temperatures) make the optimal value differ from system to system. Designers usually opt for the value that provides the best results on average. The optimal value should not be too small since it will trigger some potentially expensive procedure frequently, but it cannot be too large since that usually leads to potential performance loses.

As far as memory migration in flat address spaces is concerned, the state-of-the-art mechanisms trigger their migration procedures based on:

THM approach: THM uses a threshold-based mechanism. When the competing counter described in section 4.3 exceeds a threshold value, migration is triggered. THM will swap the page that triggered the event with the page currently residing in the segment's fast memory page. As a result, a small chance exists that a cold page was accessed at the right time to trigger migration and now it resides in fast memory. Such a mistake should be quickly get resolved however, since the cold page in fast memory should get remigrated soon enough. Each segment can trigger migration independently and asynchronously since no interval is used. THM risks very frequent migrations that will stall the stream of incoming requests until each swap is finished. **Check if they studied any of the previous two effects.** THM's authors identified the optimal threshold value as **XX**.

HMA approach: HMA uses an interval based mechanism. Upon each interval, HMA attempts to migrate as many pages in order to fill the fast memory. However with the high cost associated with the OS's intervention, management and the penalties of cold TLB force the interval value to be much larger. HMA authors identified the optimal timing interval to be as high as 1ms. **VERIFY**.

MemPod approach: MemPod uses timing intervals. Like HMA, MemPod tried to fill the fast memory with hot pages on each interval. However, MemPod is transparent to the system and as such the need for costly OS intervention is waived. With the cost of a migration cycle kept to lower values, MemPod offers the possibility of a smaller interval time, which could potentially result in better performance. The optimal interval value is evaluated in the results section.

4.5 Decentralization of migration logic

The use of multiple MCs and multiple channels in today's memory organizations serves the purpose of exposing channel-level parallelism. Each channel can issue requests independently without any knowledge of other channels' states. Some migration mechanisms in the literature inherently "assume" a centralized migration controller in charge of monitoring traffic, while others attempt to implement a completely distributed mechanism. A centralized approach can be severely limiting. Current HBM memory technology allows up to 8 channels, while many processors are already designed with two or even more off-chip memory channels. Our evaluated system in this paper features a total of twelve memory channels. Channel number is predicted to increase in the near future **[cite]**. The channel parallelism capabilities will be entirely lost if we enforce request serialization due to migration-related activity tracking or remap table lookups. On the other hand, a fully distributed solution will eliminate all serialization, at the cost of all-to-all communication between each channel. An alternative to the communication cost would require OS intervention. MemPod's novel clustered architecture attempts to balance this tradeoff.

As with most of the essential elements for migration presented in this Section, the system's designer can choose any level of centralization desired according to the specific design's constraints. As such, the body of work on migration covers the entire range:

THM approach: Even though not clearly stated in the THM proposal, it appears the authors opted for a centralized unit and consequently all channel parallelism potential is lost, placing THM last in our list in terms of parallelism potential. Decentralizing THM's migration controller appears to be possible due to the possibility of caching SRT entries, but in that case cache coherency becomes a concern. Race conditions could occur if the same SRT entry is simultaneously cached in different locations and its counter is modified. For this paper's evaluation section, we assume THM is fully centralized, as presented through figures in its proposal publication.

HMA approach: HMA ranks at the top of our list, featuring a fully decentralized mechanism. It's important to note that HMA does not require a remap table and consequently one possible source of request serialization is automatically removed. Activity tracking is performed at each MC individually, where a controller will monitor the activity of its own pages. When HMA's migration is triggered, the OS collect all activity monitors from all the MCs before it proceeds with migration. Of course, collecting all this information from each MC by the OS consumes a considerable amount of cycles.

MemPod approach: Clustering memory channels into Pods comes with significant benefits. First, assigning exactly one off-chip (slow) memory channel to each independent Pod ensures those channels can still issue requests in parallel. Being the slowest part of our memory hierarchy, keeping slow channels independent does not add delay to a potential bottleneck. Furthermore, each group of two on-chip (fast) channels can still operate in parallel. Some serialization is introduced between sibling fast channels, however with a Pod's light design and the high-bandwidth potential of those channels, the delay is amortized. Beyond channel parallelism, each Pod can hold all the migration-related structures, eliminating the need of retrieving information from each of its MCs at the beginning of a migration interval. **I'm trying to point out that a Pod keeps the structure size manageable because it handles less controllers. Other policies could hold all the information at a central point, but structures for 12 channels will be a log bigger thus slower. I think I'm not presenting my point correctly here.**

4.6 Migration Cost

Regardless of the choice for each migration element described so far, once migration is triggered, any migration manager has to follow the same steps: First, migration candidates need to be identified. Traditionally, one page (or a segment depending on the migration granularity) from the slow memory and one from fast memory. Both candidates need to be swapped. First they will be read and stored in temporary buffers and then written at their remapped locations.

We assume that channel parallelism is utilized when reading and writing the candidate pages. In other words, the two read commands will be sent in parallel, as well as the write commands that follow. Consequently, The hardware penalty for one page swap is the time required to read from and then write to the slow memory. We also assume that writing the two candidates back occurs a row buffer hit since the page was just opened in the previous step. When consequtive swaps happen, as in the case of HMA and MemPod, all swap reads are assumed to result in row buffer misses. In this study, we evaluate all mechanisms under the same memory organization and as such, the hardware swap penalty is the same regardless of the mechanism. However, each mechanism introduces some unique penalties:

THM does not introduce a cost for identifying the candidate pages since it follows a deterministic algorithm. The page that caused the competing counter to exceed the set threshold will be the slow memory candidate, while there is exactly one fast page candidate per segment. Furthermore, only one swap is executed at every triggered migration, setting te cost per migration equal to the hardware cost.

HMA attempts to fill the entire fast memory with migrated pages. The number of swaps that will be performed per interval can be as high as the number of pages in fast memory. Inevitably some hot pages will already reside in the fast memory. We modeled HMA to not attempt migration of those hot pages. Such an approach could compicate finding a fast-memory candidate page, although with HMA's full activity tracking counters, and since sorting is a necessary operation at every interval, this problem can be reduced to the simple task of

following the sorted activity list backwards (i.e. It's easy to find the coldest fast-memory page). Unfortunately, HMA introduces costs that are hard to estimate. Traversing and updating page tables and flushing TLBs is part of the cost introduced by the OS. On top of that, the effect of a cold TLB can penalize severely all running applications.

In MemPod, similar to HMA, each Pod tries to fill the entire fast memory assigned to it. However, MEA counters make it *impossible* to identify the coldest pages. In addition, the algorithm cannot be modified to identify the *least frequently occuring* addresses. In order to overcome this ssue, MemPod follows a naive algorithm. Each Pod will receive the list of N hottest pages, where N is the number of fast pages in a single Pod. Then it will iteratively switch all pages in order, even the pages ehich are already in fast memory. To introduce some amount of parallelism, the fast memory page candidates are selected using a round robin algorithm between the two fast channels. In other words, the hottest page will be swaped with the first page of the first fast channel. The second hottest page will go to the first page of the second channel. The third page will be swaped with the second page of the first channel and so on. Under this model, the assumption that reads and writes can be performed in parallel is no longer valid, as well as the assumption of row-buffer hits for writes. In case the two page candidates are in the same channel, serialized reads and writes will be performed at the speed of the fast channel, all of them resulting in row buffer misses. A possible solution would be to pay the full price and use HMA's activity tracking scheme. However, since the migration process is parallelized inherently by the use of Pods and given the several benefits of MEA counters, we decided to keep the naive migration algorithm. **I just realized we have a small bug here. I need to update the source code in the case of MEA counters.**

How about we only update the remap table when such a conflict occurs????? Think about it.

Give numbers for each case of migration and update source code

4.7 Scalability to Future Memory Capacities

4.8 Discussion and comparison

Talk about limits in the size of remap tables/counters just fr the sake of argument. How would each mechanism handle it?

5. RESULTS

5.1 Experimental Methodology

5.2 Result1

5.3 Result2

5.4 Result3

6. CONCLUSIONS AND FUTURE WORK

7. REFERENCES

[1] Bryan Black, "Die Stacking is Happening," in *Proc. of the Intl. Symp. on Microarchitecture*, Davis, CA, December 2013.

[2] Aamer Jaleel Chiachen Chou and Moinuddin Qureshi, "CAMEO:A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *Proc. of the 47th Intl. Symp. on Microarchitecture*, Cambridge, UK, December 2014.

[3] Michel El-Nacouzi, Islam Atta, Myrto Papadopoulou, Jason Zebchuk, Natalie Enright Jerger, and Andreas Moshovos, "A Dual Grain Hit-miss Detector for Large Die-stacked DRAM Caches," in *Proc. of the Conf. on Design, Automation and Test in Europe*, 2013, pp. 89–92.

[4] Fazal Hameed, Lars Bauer, and Jörg Henkel, "Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies," in *Proc. of the*, 2013.

[5] Intel, "KnightsLanding," http://www.realworldtech.com/knights-landing-details/.

[6] JEDEC, "Wide I/O Single Data Rate (Wide I/O SDR)," http://www.jedec.org/standards-documents/docs/jesd229.

[7] Djordje Jevdjic, Stavros Volos, and Babak Falsafi, "Die-stacked dram caches for servers," in *Proc. of the Intl. Symp. on Computer Architecture*, 2013.

[8] Xiaowei Jiang, Niti Madan, Li 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*, January 2010, pp. 1–12.

[9] Joint Electron Devices Engineering Council, "JEDEC: 3D-ICs," http://www.jedec.org/category/technology-focus-area/3d-ics-0.

[10] Gabriel H. Loh and Mark 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] Gabriel H. Loh, Nuwan Jayasena, Kevin McGrath, Mike O'Connor, Steven Reinhardt, and Jaewoong Chung, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*, New Orleans, LA, February 2012.

[12] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories." in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2015.

[13] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, July 2012.

[14] NVIDIA, "NVIDIA Pascal," http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/.

[15] J. Thomas 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.

[16] Moin Qureshi and Gabriel H. Loh, "Fundamental Latency Trade-offs in Architecturing DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, Canada, December 2012.

[17] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim, "Transparent hardware management of stacked dram as part of memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014.

[18] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, Canada, December 2012.

[19] William A. Wulf and Sally A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, March 1995.

[20] Li Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *Proc. of the 25th Intl. Conf. on Computer Design*, October 2007, pp. 55–62.