MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories

ABSTRACT

In the near future die-stacked DRAM will be present alongside off-chip memories in hybrid memory systems. A large body of recent research has explored possible uses and performance-improving mechanisms regarding such a memory configuration. The first approach is to use this new memory as a last level cache and it has been shown to be beneficial for system performance. Another approach is to use the on-chip memory as an extension of the off-chip one in a "flat address space" configuration, exposing more memory capacity to applications leading to performance improvement under capacity constrained workloads.

In this paper we propose MemPod, a scalable and efficient memory management mechanism for flat address space hybrid memories. MemPod monitors main memory activity and upon predefined intervals migrates the most frequently accessed memory pages to the faster on-chip memory. We propose a novel architectural organization and the use of "Majority Element Algorithm" for activity tracking.

Our results with multi-programmed workloads show MemPod to improve Average Main Memory Time (AMMT) by up to 29% and 11% on average compared to the state-of-the-art, while our analysis shows that MemPod is the most viable option compared to the state of the art, as memories become faster in the future. MemPod's novel activity tracking approach leads to significant cost reduction (∼6000x lower space requirements) and incredible improvement on future prediction accuracy (58% average improvement) over traditionally used full counters.

Keywords: Memory architecture, Die-stacked memory

1. INTRODUCTION

The memory wall problem has been known to impede system performance [?]. 3D stacked memory is considered as a viable solution and has been studied extensively. Placing memory stacks in the processor package was shown to provide significant improvements in terms of bandwidth and power consumption [?]. Processor manufacturers already began including 3D-stacked memories in their announced products [?, ?, ?] and die-stacked memory standards have been developed [?, ?, ?]. The use of die-stacked memory will undeniably be part of many future systems. Products such as **Cite Fury-X, any other existing product (AMD?)** are already manufactured with stacks of memory in the same package as the processing cores.

~~The current technology for incorporating stacked memory, as well as the current protocols allow up to 16GB of fast DRAM [?].~~ **Find the current limits of HBM, mention them and cite.** Commodity server systems often need *hundreds* of GB of memory. Consequently, at its current state, in-package memory cannot solely support today's memory requirements, leading to the emergence of *hybrid memories*, usually with fast 3D-stacked memory and the traditionally used off-chip memory such as DDR4. Often, hybrid configurations with two memory technologies are called *"Two-Level Memories"*, ~~which implies a cache-like organization, even though that's not always the case~~ or *"N-Level Memories" (NLM)* for configurations with N memory technologies. It is not clear yet how this newly added memory is best utilized. Recent research proposed mechanisms to manage the 3D-stacked memory as a high-bandwidth last level cache, while other proposals attempt to manage this memory as an extension of the main memory. Each approach comes with its own challenges, benefits and drawbacks.

Recent proposals in the literature [?, ?] demonstrate that when stacked memory is used as a cache, we need to re-evaluate traditional SRAM cache optimizations. Tag placement and granularity must be carefully studied and double access (tag check and data retrieval) in order to serve one request must be avoided due to the high latency of accessing a DRAM structure. Despite the associated challenges, state-of-the-art proposals manage to achieve high performance improvement through elegant solutions. Such a management scheme offers transparent operation and does not require added hardware structures to support it. On the downside, the extra memory capacity is not available to the software and some space is wasted to store tags instead of useful information. ~~As memory capacities increase, the size of tag information could result in wasting even larger parts of our fast memory.~~ In a DRAM cache configuration, tags consume **??%** of the on-chip memory's capacity.

Instead of using stacked-memory as cache, some recent research [?, ?] proposed both hardware and hardware-

supported OS dynamic memory management mechanisms to manage it as a *"flat address space memory"*. Memory accesses are being monitored in hardware and the goal is to identify "hot" memory pages and migrate them into the fast memory, in order to improve performance. Such a configuration extends the exposed main memory capacity, serving capacity-constrained workloads better than a DRAM cache organization, while at the same time eliminating the issue of tag placement and retrieval. Additionally, a flat address space organization could be incorporated in a system without the presence of a dynamic memory management mechanism and it would still operate correctly and boost performance simply because of the presence of a fast memory region and static memory allocation from the OS. Like a DRAM cache, this memory organization is also transparent to the application programmer. However, applications will occasionally need to be interrupted for the required migrations to take place.

Flat address space dynamic memory managers need to overcome significant challenges. The main drawback comes from the required book-keeping. Monitoring memory regions and keeping track of migrated pages in order to relay incoming requests transparently, often comes at very high storage and power consumption overheads. Unlike a DRAM cache, there is no backing store memory. As such, each migration is a swap of two pages in order to ensure that exactly one copy of each participant page exists in main memory. In the presence of a flat memory with two regions, one fast and one slow, it could be expected that the fast one is fully utilized before we start using the slow one. However, modern OS's assign memory addresses randomly for security reasons **Find and cite** and as such no guarantees can be made for a "smarter" memory allocation policy.

In this paper, we propose MemPod, a dynamic memory manager for flat address space memory configurations that is efficient in terms of area requirements and performance, as well as scalable to future technology advancements in memory capacity and speed. MemPod's novel microarchitectural design clusters existing memory controllers into memory "Pods" allowing for better scalability and integration to future systems with larger and faster memories. For MemPod's activity tracking requirements we incorporated a *"Majority Element Algorithm" (MEA)* heuristic, originally proposed for database management and big data analytics. Our evaluation shows MEA to be capable of high prediction accuracy with very low hardware overhead. To the best of our knowledge, such an algorithm has not been utilized before for activity tracking.

Our evaluation results under homogeneous and mixed 8-core multi-programmed workloads show MemPod to outperform the current state-of-the-art by 11% on average and up to 29% in terms of Average Main Memory Time (i.e. the average time a request spends in main memory). Under an overclocked memory configuration, our results show MemPod to be the most scalable mechanism as memory technology improves. The use of MEA activity tracking requires $\sim 0.01\%$ of the storage space required by ~~traditionally used~~ Full Counters (FC) used in previous research studies, while at the same time achieving 58% higher ~~hot page identification (future prediction)~~ prediction accuracy.

A goal of this paper is to identify the basic building blocks of *any* flat address space dynamic memory management mechanism. System designers can choose any solution for each of these elements and create a new management scheme in a plug-and-play fashion. Each building block is associated with its corresponding trade-off. Later in this paper we provide a description of each block, along with a side-by-side comparison of MemPod's approach and what the state-of-the-art mechanisms propose.

The contributions of this paper are:

- Novel microarchitectural clustered design.
- Novel activity tracking algorithm.
- Breakdown of the basic building blocks of *any* flat address space dynamic memory management mechanism.

In Section **??** of this paper we present background and related work regarding memory management schemes. Section **??** presents our novel activity tracking in detail along with an in-depth evaluation of its capabilities compared to traditionally used mechanisms. Section **??** gives a detailed overview of MemPod's architecture, presented side-by-side with the state-of-the-art mechanisms we compare against. Our architecture section is organized based on the fundamental basic blocks of management mechanisms in order to present the management problem in its entirety. Section **??** presents our experimental methodology and evaluation results and finally, Section **??** concludes the paper.

2. BACKGROUND & RELATED WORK

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. MAJORITY ELEMENT ALGORITHM

Memory management mechanisms need the ability to monitor and profile accesses to memory in order to identify hot regions and migrate them. Traditionally, a full set of counters is used, with one counter per physical page – or region depending on the mechanism's granularity – in order to accurately keep count of all accesses to all regions. Periodically these counters must be sorted in order to identify the regions with the highest counts. The identified regions will serve as a "prediction" for the next interval (i.e. the hottest page of the current interval will be amongst the hottest pages of the following intervals).

MemPod uses the "Majority Element Algorithm" (MEA) for its activity tracking needs. MEA attempts to identify the *majority* elements in a set. For example in an

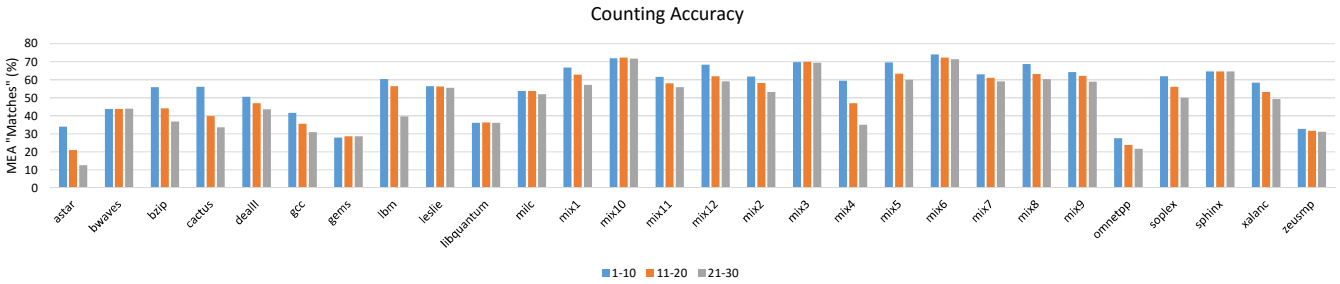
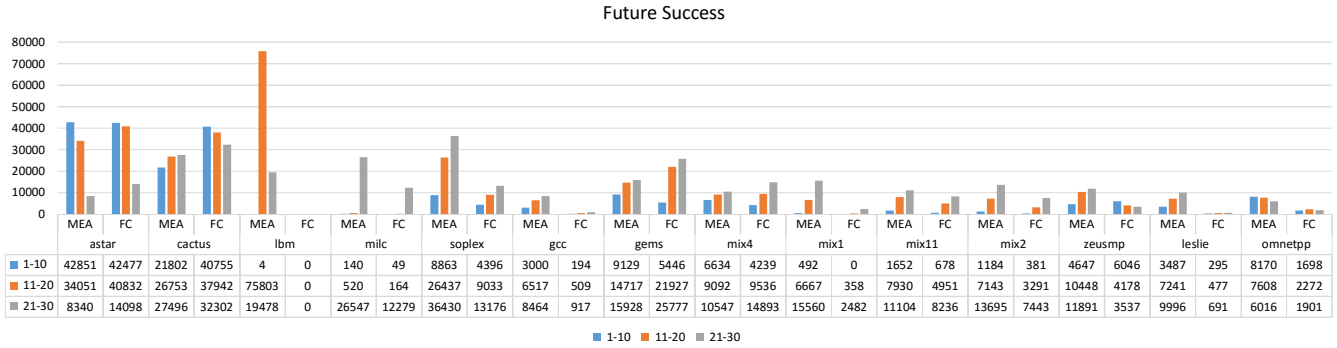


Figure 1: MEA counting accuracy compared to Full Counters on the top three tiers (ranks 1-10, 11-20, 21-30)



	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC
	astar		cactus		lbm		milc		soplex		gcc		gems		mix4		mix1		mix11		mix2		zeusmp		leslie		omnetpp	
1-10	42851	42477	21802	40755	4	0	140	49	8863	4396	3000	194	9129	5446	6634	4239	492	0	1652	678	1184	381	4647	6046	3487	295	8170	1698
11-20	34051	40832	26753	37942	75803	0	520	164	26437	9033	6517	509	14717	21927	9092	9536	6667	358	7930	4951	7143	3291	10448	4178	7241	477	7608	2272
21-30	8340	14098	27496	32302	19478	0	26547	12279	36430	13176	8464	917	15928	25777	10547	14893	15560	2482	11104	8236	13695	7443	11891	3537	9996	691	6016	1901

Figure 2: MEA prediction accuracy (part 1) compared to Full Counters on the top three tiers (ranks 1-10, 11-20, 21-30)

array with integers, MEA can be used to identify the K *majority numbers*. Majority elements are the most frequently occurring elements, as long as they exist more than $\frac{N}{K+1}$ times in the information stream (i.e. they have majority), where N is the number of elements in the array.

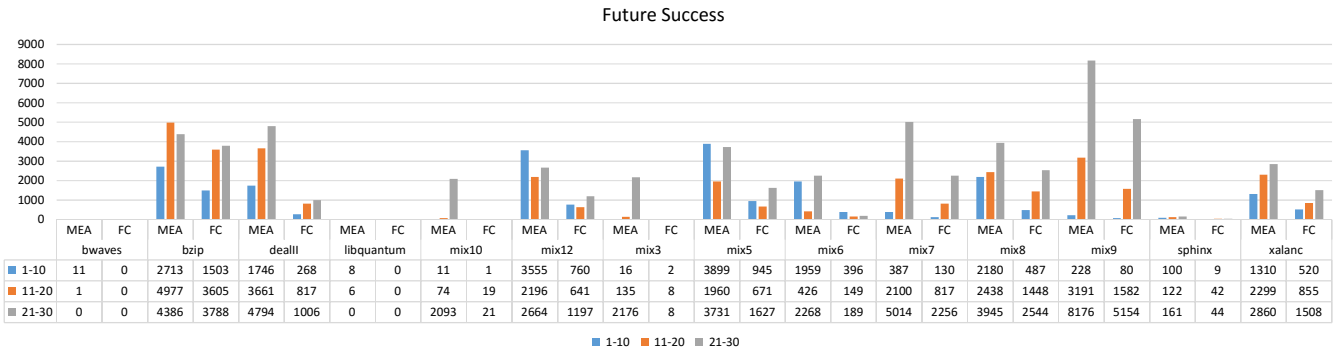
The MEA algorithm was originally proposed in [**?**] and studied in-depth in [**?**] as a heuristic capable of efficiently identifying majority elements in a stream. This algorithm is formally proven to be 100% accurate, as long as the K most frequently occurring elements it must identify have majority. With complexity $O(N)$ it could be an ideal candidate for real-time streams of information, such as a stream of memory requests.

The MEA algorithm is presented in Algorithm **??**: A map structure of K entries holds the element's ID (in our integer array example that would be the integer's value) and maps it to a counter. Looping through the array, if the next integer exists in the map it increases its counter by 1. Otherwise, if there's enough room in the map it adds the new entry with a count of 1. If the number does not exist in the map and all K counters are occupied, the algorithm subtracts 1 from every counter, removes the entries with a counter value of 0 and proceeds to the next integer. Once the entire array is processed, the map entries hold the majority elements.

Even though this heuristic is 100% accurate, in the absence of its main assumption no guarantees can be made. The outcome of this algorithm relies on several uncontrolled variables, such as the order our re-

Algorithm 1: Majority Element Algorithm

;
Input: X: Set of N elements
Input: K: Number of elements to output
Data: T: Map structure with K entries
Result: Set of K majority elements
;
Initialization: $T \leftarrow \emptyset$;
;
foreach $i \in X$ **do**
  if $i \in T$ **then**
    $T[i] \leftarrow T[i] + 1$;
  else if $|T| < K - 1$ **then**
    $T[i] = 1$;
  else
    forall $j \in T$ **do**
      $T[j] \leftarrow T[j] - 1$;
      if $T[j] == 0$ **then** $T \leftarrow T \setminus j$
    end
  end
end



	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC	MEA	FC
	bwaves		bzip		dealII		libquantum		mix10		mix12		mix3		mix5		mix6		mix7		mix8		mix9		sphinx		xalanc	
1-10	11	0	2713	1503	1746	268	8	0	11	1	3555	760	16	2	3899	945	1959	396	387	130	2180	487	228	80	100	9	1310	520
11-20	1	0	4977	3605	3661	817	6	0	74	19	2196	641	135	8	1960	671	426	149	2100	817	2438	1448	3191	1582	122	42	2299	855
21-30	0	0	4386	3788	4794	1006	0	0	2093	21	2664	1197	2176	8	3731	1627	2268	189	5014	2256	3945	2544	8176	5154	161	44	2860	1508

Figure 3: MEA prediction accuracy (part 2) compared to Full Counters on the top three tiers (ranks 1-10, 11-20, 21-30)

quests appeared in. However, the nature of the algorithm presents a very welcomed side effect: Elements accessed repeatedly can be evicted from the map by elements that were accessed less times but more recently. This observation reveals MEA's favoritism towards temporal locality. Furthermore, the area overhead of this algorithm implemented in hardware remains constant, regardless of how many elements need to be profiled (i.e. regardless of how many pages exist in main memory).

In a memory management scheme that uses Full Counters and a scenario where we want to identify the 100 "hottest" pages, we would need one full counter per memory page and on top of that we would have to periodically sort all those counters to pick the top 100. With the use of MEA counters we only need a map with 100 entries regardless of the actual number of pages in main memory. In an 8GB memory with 2kB pages and looking for the top 100 pages, MEA needs ~5K times fewer bits than the full counters' storage requirements (4MB Vs 850B). Considering all the potential benefits MEA can offer in theory, we compared its counting and prediction accuracy against the Full Counters (FC) scheme.

To evaluate MEA as a possible candidate for activity tracking, we used traces captured from multi-programmed 8-core workloads (the same traces used and described in Section **??**) and simulated MEA and FC side-by-side with an in-house off-line simulator that provides oracle knowledge of future intervals. The interval size for both MEA and FC was set at 5500 requests which is the average number of requests serviced within a 50us window in our timing experiments. We compared the two mechanisms based on their counting and prediction accuracies regarding the top 3 tiers of memory pages. Pages ranked in the 1-10 hottest pages were grouped in tier 1, tier 2 holds pages ranked 11-20 and finally tier 3 holds pages 21-30.

First we compared MEA's counting accuracy against FC's guaranteed correct counting. In other words we evaluate MEA's capability of identifying the hottest pages of the current interval accurately. Figure **??** shows the counting accuracy of MEA in the top 3 tiers. In the best cases, MEA scored about 75% accuracy in the top three tiers and about 50% on average. Our results show that in our use cases MEA is not a viable option in terms of accurate counting. However, correct counting accuracy does not necessarily translate to accurate predictions during future interval(s), even though intuitively, these two aspects might seem correlated.

Figures **??** and **??** present a comparison of MEA and FC in terms of prediction accuracy and demonstrates MEA's true potential. Following the same structure as Figure **??**, we compare each mechanism's "predictions" against the top three tiered pages of the following interval (oracle knowledge). First we can observe that in almost all workloads MEA strikes more hits in the top tiered pages. It's also worth noting that FC scored zero in several occasions. In most of those occasions, MEA scored relatively low, but not zero. The *lbm* workload presents a very interesting case. FC was completely unsuccessful in predicting the future, while MEA achieved one of its highest scores throughout all workloads.

Proposed in 2003, to the best of our knowledge, this algorithm has not been used in any micro-architectural proposal before. Our results clearly show that it can well outperform the very expensive option of full counters at greatly reduced hardware cost.

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, along with the corresponding challenge addressed by each one.

4.1 Clustered Migration Architecture

Figure **??** presents an overview of MemPod. MemPod's design was kept modular to facilitate system integration and scalability. A number of memory "Pods" are injected between the LLC and the system's Memory Controllers (MCs). Each Pod clusters a number of MCs and enforces migrations to only occur between its member MCs. Pods do not communicate with each other in any way, restricting inter-Pod migrations. To the rest of the system, Pod are exposed as MCs. With MemPod's transparent design, each Pod will now be receiving all

Challenge	Tradeoff	THM	HMA	MemPod
Page Relocation	*Flexibility / Time*	Only 1 Candidate *Min / Very Low*	No Restrictions *Max / High*	Intra-Pod Migration *High / Medium*
Remap Table Size	*Flexibility / Area*	1 entry per fast page (~2.4MB) *Min / Medium*	No remap table *Max / Min*	1 entry per page (~54MB) *High / Max*
Activity Tracking	*Accuracy / Area*	8 bits per fast page (64kB) *Medium / Low*	16 bits per page (9MB) *Max / Max*	128 MEA entries (640B) *High / Very Low*
Migration Trigger	*N/A*	Threshold	Interval	Interval
Tracking Organization	*Simplicity / Parallelization*	Fully centralized (Serialized requests) *High / Min*	Fully distributed *High / Max*	Semi-distributed (Pods operate independently) *High / High*
Migration Driver	*Latency*	CPU *High*	CPU (OS) *Max*	Pod *Low*
Migration Cost	*Time*	HW cost + CPU time *Medium*	HW + SW + cold TLBs + CPU *Max*	HW *Min*

Table 1: Breakdown of state-of-the-art designs

the requests originally addressed to any of the Pod's member MCs.

A Pod's operation when a memory request arrives would be to monitor the request, update any necessary migration-related activity tracking 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 activity tracking scheme used by the Pod – as well as the actual forwarding of the request – have to be as efficient as possible.

MemPod's clustered architecture also aids in reducing global traffic during migrations compared to non-partitioned mechanisms. Higher global traffic could require the global switch (or crossbar) to have higher bandwidth resulting in increased energy costs and could also introduce performance penalties. Furthermore, with Pods servicing migrations transparently, the system's CPUs will be able to keep executing non-memory instructions during migrations.

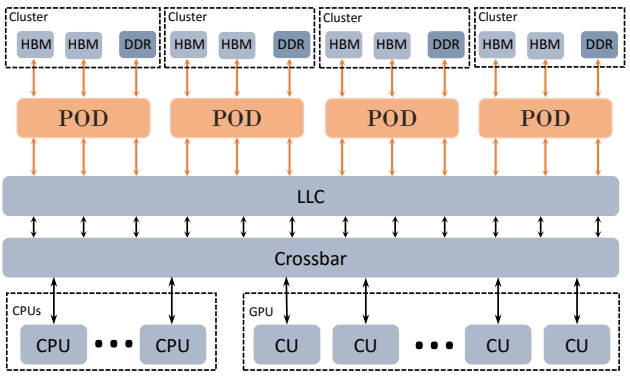


Figure 4: MemPod high-level architecture

Memory Pod

The major architectural elements of a Pod are presented in Figure **??**. A Pod includes an activity tracking unit

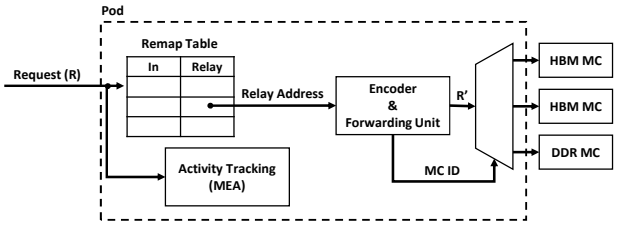


Figure 5: Major architectural Pod elements

(MEA), a remap table for keeping track of migrated pages and a forwarding unit that can re-encode a request with the relay address and then based on that address send the request to the appropriate Member MC.

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. Such a configuration inherently restricts migration between slow off-chip channels, while at the same time maintains full channel-level parallelism on the system's bottleneck – the slow MCs. In a configuration where the number of fast-memory MCs are not a multiple of slow-memory MCs, Pods can be configured asymmetrically or some MCs could be members of multiple Pods, with their capacity partitioned to avoid crosstalk. In Figure **??** we present a system with eight MCs for the fast, on-die stacked memory and four MCs for the slow off-chip memory. Throughout this paper we use HBM2 die-stacked memory **Cite** and DDR4-1600 as our off-chip memory. The use of four pods imposes few restrictions on migration possibilities, while forbiding migration between two slow-memory channels without any additional logic. For the remainder of this paper, we set the number of pods to four.

The design of a complete memory manager can be broken down into the following 5 "building blocks":

- Migration flexibility: Defines which areas in memory can be used as migration candidates. Higher flexibility offers more potential for performance improvement and increases book-keeping cost.
- Remap table: A structure that keeps track of migrated pages and is able to provide a relay address given a requested address.
- Activity tracking: Logic and structures needed to profile memory requests and predict future "hot" pages.

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

- Migration trigger: Defines when migration occurs. Usually the trigger can be interval-based or threshold-based.

- Migration driver/datapath: Defines the path followed and the hardware modules involved in performing migrations.

Each basic block includes some trade-off. For example, allowing more flexibility in migration locations can lead to higher performance benefits at the cost of larger book-keeping structures. The following subsections provide detailed description of each building block. For each block, we present MemPod's, THM's and HMA's approach, as well as alternative designs found in the literature. It's important to note that each of the following building blocks is independent and future mechanisms could choose almost any element design combination in a plug-and-play fashion, with some exceptions such as the use of MEA activity tracking cannot be combined 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.

A traditional remap table is a hash structure, indexed by a page's address and pointing to the migrated (or relay) address if one exists. On a page migration, the remap table is updated to reflect the new address of a migrated page. However, a "naive" remap table design can fail when re-migration of pages is allowed. A content-aware remap table is necessary in order to support re-migration. Figure **??** presents a side-by-side preview of the operation of these two tables and demonstrates how a naive table can fail.

The first row shows the starting state of our memory before any migration, as well as the starting state of the two remap tables. For simplicity, we only 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. The second row shows the state of memory and remap table after swapping[2] 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 the third row 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 right-hand side of Figure **??** demonstrates the operation of a content-aware remap table able to support re-migration.

This remap table design fails 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 **??**, 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. Minor modifications are required to track activity based on the content page's id, that does not affect any other aspect of a migration mechanism.

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 with low 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. THM requires ∼5.6MB for its *Segment Remap Table*.

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 flushing 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 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. It's

[2]In the absence of a backing store used by DRAM caches, a migration implies a swap of two pages

important to note that a content-aware remap table entry 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). MemPod requires 8MB per Pod and 32MB total for its sontent-aware remap table structure.

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 or (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.

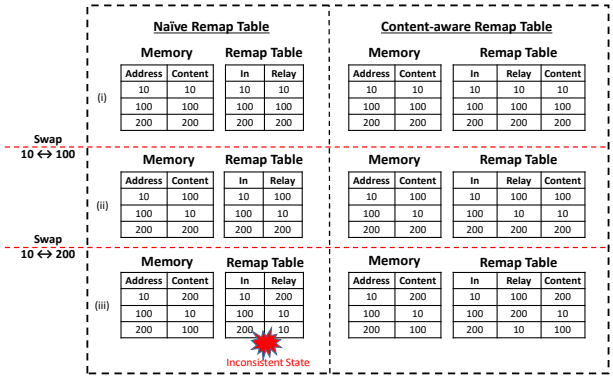


Figure 6: Naive Vs content-aware remap table operation

4.3 Activity Tracking Mechanism

Activity tracking could be considered the most important element of any memory management mechanism. In most studies on the subject, activity tracking becomes a synonym of identifying hot regions by counting the number of accesses. In a more generalized 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 for most memory management mechanisms. The overhead of maintaining a set of counters per memory page (or any other granularity), is often a bottleneck.

MemPod utilizes an important observation to maintain a low activity tracking cost. Moving the hottest pages of an on-going interval into fast memory is a commonly accepted prediction technique, 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. Our results in Section **??** demonstrate how Full Counters often conclude on a non-accurate prediction.

Frequently encountered solutions in the literature consist of increasing the activity tracking granularity in order to reduce the number of counters needed (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 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 *counting* 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 **??**, 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 when it exceeds a dynamically-set threshold. Competing counters represent a trade-off between area overhead and accuracy. THM requires 8 bits per fast memory page making THM the extremely 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. HMA uses the least efficient tracking mechanism in exchange for perfect counting 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 an MEA map structure of K entries, where K is the number of hot pages we wish to identify at each interval. Our evaluation determined that 128 is the optimal number of MEA counters. Each entry maps a page's address to a counter. Through our evaluation, we identified the optimal counter size to be 4 bits and 62 bits are needed to address each page within a Pod, leading to a total of 1.03KB of space requirements. Using the MEA counters, MemPod's activity tracking profiles *the entire memory* at extremely low cost.

As described in Section **??**, 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. As demonstrated, MEA strikes a balance between most frequently occurring and most recently used page addresses, a fortunate and welcomed consequence in locality exposure. 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 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 main memory to alleviate the area overhead. THM's segment-based counters are automatically cached along with their corresponding SRT entries and restored whenever necessary, at the cost of a memory access. By the description the MEA algorithm presented earlier in Section **??**, we can be certain that this algorithm cannot work well with the presence of a cache since it frequently needs to loop through *all* entries of the MEA structure. Fortunately, the required space needed by the algorithm is small enough to fit on chip and since its size does not scale with any increase in memory capacities it can be used with future systems.

It is also important to note that any *activity tracking* updates should be moved off the critical path. Extreme tracking accuracy is not necessary for correct operation. Even if the tracking 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 alleviate the risk of inconsistent state.

4.4 Migration Triggers

Deciding when to perform migrations is not always a trivial task. Migrations add significant delays to a system and as such it must be used wisely. Any penalties incurred should 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. Throughout the literature, two triggers are most commonly used whenever state must be updated based on tracking information (such as MC scheduling, migrations, Dynamic Voltage and Frequency Scaling (DVFS) etc.). Interval-based (or epoch-based) triggers occur with a set 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. Factors like a system's architecture, application's behavior, as well as semi-random factors (for example higher temperatures can lead to more frequent DRAM refreshes) 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 loss.

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 **??** 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, since the cold page in fast memory should get remigrated soon. 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. To overcome this issue, the authors attempt to dynamically predict the future benefit of using a threshold value from a set of some pre-defined values. A "sampling region" is used where migrations are virtually executed allowing THM to extrapolate and associate a potential benefit with each one of the pre-define threshold values. When THM expects that migration costs will be amortized it updates the threshold value of a segment.

HMA approach: HMA uses an interval based mechanism. Upon each interval, HMA attempts to migrate as many pages in order to fill the entire 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.

MemPod approach: MemPod uses timing intervals. At each interval each Pod will migrate up to K pages into the fast memory, where K is the number of MEA counters used. MemPod is transparent to the system, rendering costly OS intervention unnecessary. Since each one of the 4 Pods will attempt to migrate up to K pages, up to 4×K migrations can happen within each interval. However, since each Pod is independent and there is no cluster overlap, each Pod can issue migrations in parallel. The time required by MemPod to execute 4×K migrations is equal to the time to execute K migrations.

4.5 Decentralization of Migration Logic

The use of multiple MCs and multiple channels in modern 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 four 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 se-

rialization 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 paper, 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. To ensure a fair comparison in our evaluation of THM, we assumed it can utilize the full channel-level parallelism potential of a system.

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 (requiring hardware support), where each controller will monitor the activity of its own pages. When HMA's migration is triggered, the OS will 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 number 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.

4.6 Migration Datapath

Regardless of the choice for each migration building block 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. The two identified candidates need to be swapped. First they will be read and stored in temporary buffers and then written at their remapped locations.

Describing the actual migration datapath is often overlooked in migration publications. Without dedicated page migration driver hardware, migration will have to be orchestrated by some CPUs. Consequences include communication delay, potentially some delay introduced at the processor's cache levels and the performance degradation caused by stalling those CPUs until migration is over. MemPod implements the migration driver within each Pod. Since the Pod has direct communication with the MCs, added delays are kept to a minimum. In HMA, the OS orchestrates everything. Some CPUs have to be stalled and used to service the OS interrupt, causing the migrated pages to traverse through communication mediums and caches on each way. THM does not describe its datapath in detail. ~~We assume the CPUs are used in this case too.~~ For fair comparison, we do not model the penalty introduced by using CPUs for migration in our HMA and THM simulations presented in Section **??**.

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 an entire page from and then write it to the slow memory. We also assume that writing the two candidates back incurs row buffer hits since the page was just opened in the previous step. ~~When consecutive 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~~ the time required for one migration is the same across all mechanisms. 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 the 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 complicate 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. Sorting all the activity tracking counters, 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, with the use of MEA counters, identifying the fast-memory page candidate is as simple as checking that it's not part of the K hot pages. The identification algorithm starts at the very first fast memory location and iterates sequentially until it detects a page address that is not in the set of hottest pages. For the next migration, the identification algorithm simply continues from where it was left. If a hot page already resides in the fast memory it's ignored.

4.7 Scalability to Future Memories

Assuming memory capacity in the order of tens of Terabytes, SRAM requirements for activity tracking and remap tables could become unfeasible. Caching part of the migration logic and using part of main memory as backing storage seems necessary. An analysis on the impact of such caching, as well as the optimal cache size is presented in the experimental evaluation section. MemPod's semi-distributed architecture allows caching without any action required to protect against race conditions because of the utilization of independent Pods that never share information.

At such capacity levels, centralized migration controllers will no longer be sustainable due to the severity of the introduced serialization penalty. Assuming the driving force behind memory capacity's tremendous scaling are future applications, it's safe to assume that memory traffic would also scale up, as well as sustainable bandwidth expectations.

Alternative approaches described in this Section limit the size of the remap table at the cost of disabling migration when full, or invoking the OS before migration resumes. This approach could also be incorporated in future migration mechanisms. We should note that with a limited remap table size, or even an available number of counters less than the number of fast pages, MemPod would still offer the potential of migrating *any* slow memory page to the limited set of fast pages (as long as it's within the same Pod) available for migration, in contrast to THM, where only those segments that happened to fit in the new limited size will be able to migrate, forbidding migration of the rest of the slow pages. HMA would remain intact from this limitation since no remap tables are necessary.

5. RESULTS

5.1 Evaluation Framework

The goal of our evaluation framework is to quantitatively and qualitatively assess MemPod's capabilities and compare it against state-of-the-art proposed mechanisms. Throughout our evaluation section, we study MemPod's performance running as part of an eight-core CPU. We extended Ramulator [**?**] to support flat address space hybrid memories and with MemPod for our memory simulations. HMA and THM were also implemented in our simulation framework for comparison purposes. Ramulator allows for cycle-level memory simulation and includes a simple CPU front-end capable of approximating resource-induced stalls. We chose to evaluate MemPod under a realistic memory configuration consisting of 1GB 3D-stacked HBM2.0 **[Cite]** and 8GB of off-chip DDR4-1600. Table **??** Provides a more detailed description of the simulated system's configuration.

5.2 Experimental Methodology

We used benchmarks from the SPEC2006 suite [**?**] as our workloads. Using Sniper [**?**], we extracted memory request traces while simultaneously executing 8 benchmarks on a simulated 8-core CPU. We then feed these multi-programmed memory traces in Ramulator, executing all workloads to completion. Our complete set of workloads consists of 16 "homogeneous" workloads, where the same benchmark runs 8 times in parallel (we simply call this workload with the benchmark's name in later results), as well as 12 workloads featuring a random mix of 8 benchmarks each (marked as mix1-12). A breakdown of the mixed workloads is shown in Table **??**.

We also extended Ramulator with caches needed for activity tracking and/or remap tables depending on the simulated mechanism. Cache misses inject a memory request into the stream of requests fed by our trace files to retrieve the missing information. No priority is given to cache miss requests over the rest of the requests. When caches are disabled, the simulator assumes that any information needed by any mechanism exists on chip and is accessible without any delay. The migration process was implemented in detail as well. In order to read an entire page from memory, 32 read requests need to be sent for each of the two migration candidates and then another set of 32 requests for each of the two write-backs.

Since we used Ramulator with recorder traces, we chose to report Average Main Memory Time in our results instead of IPC. Even though Ramulator has the ability to approximate IPC, AMMT is a more accurate metric since it models the memory in detail. AMMT is the average time spent in main memory by each request (lower is better). Due to space limitation we are not able to show results from all our workloads in most of the graphs in this paper. In those graphs, we only present the results from mixed workloads, the average of all mixed workloads, average of all homogeneous workloads and overall average.

5.3 Simulation Results

5.3.1 Optimal Parameter Values

MemPod exposes 3 variables that allow fine-tuning it based on a particular system or expected workloads: (1) The number of MEA counters, (2) interval length and (3) the size of each MEA counter. Identifying the opti-

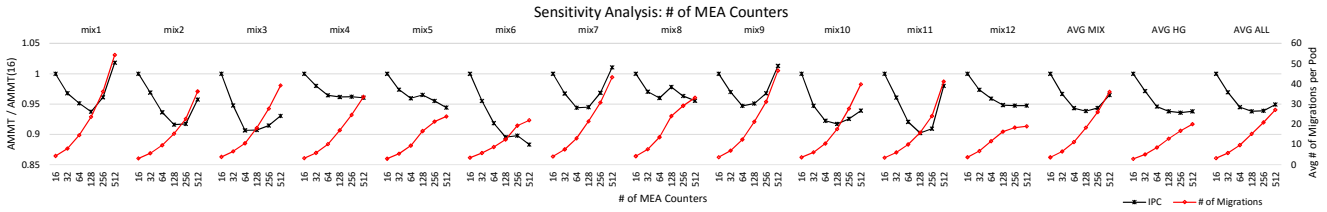


Figure 7: # of MEA Counters Vs Normalized AMMT (primary axis) and average # of Migrations per Pod per interval (secondary axis)

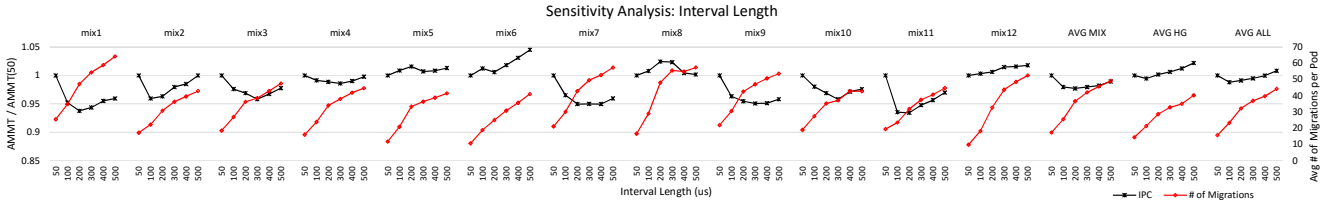


Figure 8: Interval Length Vs Normalized AMMT (primary axis) and average # of Migrations per Pod per interval (secondary axis)

mal values for each parameter can maximize MemPod's capabilities. The number of MEA counters dictates the highest possible number of migrations that can be performed at each interval, while the epoch length will determine MemPod's ability to better adapt to phase changes in a workload. The size of each MEA counter can affect performance due to the loss of information when smaller counters are used but can also save space on the chip.

We first identified the optimal number of MEA counters, ~~by setting the epoch length to 500us~~ by keeping the epoch length constant and exponentially increasing the number of counters from 16 to 512. In order to minimize the impact of other factors, we executed this experiment with 16 bits per counter and caches disabled. In other words, each counter was given more than enough space and all required information such as the remap table resides entirely on the chip and is accessible at no cost.

Figure **??** shows normalize Average Main Memory Time (AMMT)[3], along with the average number of migrations per Pod per epoch (secondary axis). The results indicate that each Pod utilizes the higher number of counters and consecutively performs more migrations per interval, however performance begins leveling off when more than 128 counters were used. More migrations can directly be translated into higher power consumption and communication cost. Using 128 counter, MemPod improves AMMT by 7% over the baseline with 16 MEA counters. Based on our observations we conclude that the optimal value for this parameter is 128 and will be used for the remainder of this section.

After identifying the optimal counter number, Figure **??** displays the same measurements as Figure **??** with a varying epoch length. Caches were again disabled, each counter was given 16 bits and the number of MEA counters was set to the optimal value of 128. In most of our benchmarks a smaller epoch length leads to higher performance. In accordance to the previous experiment, higher epoch lengths also lead to higher number of migrations. On average, an epoch length of 100us outperforms the baseline (50us) by 2%. For comparison purposes, HMA ?? identified the optimal epoch length to be 1ms (10x larger) in order to support all lengthy processes that take place during a migration event.

As previously explained in Section **??**, the MEA algorithm cannot be cached efficiently and as a result, the entire activity tracking structure needs to be on chip. The size (in bits) of each counter defines the area requirements of our MEA tracking mechanism. We modified the MEA algorithm ~~to overflow to the value of 1 instead of 0~~ to remove map entries with counter values equal to or less than zero (instead of strictly equal to zero) in order to support counter saturation. We opted not to immediately remove an overflowed counter simply because its value is now zero since the existence of the correct counter set is crucial to the algorithm's accuracy. For this experiment we used the optimal parameter values identified in the previous experiments and set the number of MEA counters to 128 and interval length to 100us. All caching was disabled in order to study the direct impact of this variable.

Figure **??** presents the impact of counter size on AMMT and average number of migrations. We first observe that 8 bits are sufficient for the majority of our workloads, since larger sizes report identical results. Our second observation is that two bit counters report a negligible performance degradation (0.3% on average) and a reduced average number of migrations.

The most interesting observation comes from assigning 4 bits to each counter. On average, performance is boosted slightly (up to 3% and 1% on average) even

[3]AMMT: The average time a request spends in main memory

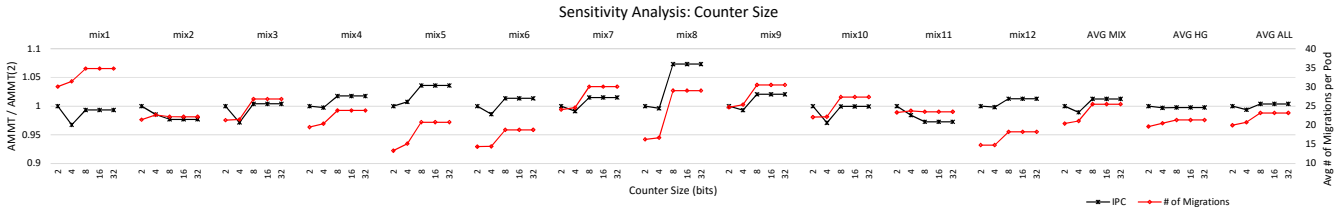


Figure 9: Counter size (in bits) Vs Normalized AMMT (primary axis) and average # of Migrations per Pod per interval (secondary axis)

Processor	
Cores	8 @ 3.2GHz
Width	4 wide out-of-order
Caches	
L1 I-Cache(private)	64KB, 2 way, 4 cycles
L1 D-Cache (private)	16KB, 4 way, 4 cycles
L2 Cache (shared)	8MB, 16 way, 11 cycles
HBM2	
Bus Frequency	1 GHz (DDR 2 GHz)
Bus Width	128 bits
Channels	8
Ranks	1 per Channel
Banks	16 per Rank
Row Buffer Size	8KB (open-page policy)
tCAS-tRCD-tRP-tRAS	7-7-7-17 (mem. cycles)
DDR4	
Bus Frequency	800 MHz (DDR 1.6 GHz)
Bus Width	64 bits
Channels	4
Ranks	1 per Channel
Banks	16 per Rank
Row Buffer Size	8KB (open-page policy)
tCAS-tRCD-tRP-tRAS	11-11-11-28 (mem. cycles)

Table 2: Experimental framework configuration

compared to using larger counters, while migration count is smaller. Since a difference is observed, we can conclude that these smaller counters "lose" information that in turn benefits overall execution. The reported result is a welcomed artifact of the MEA algorithm combined with application behavior.

Based on our results, we identify 4 bits per counter to be the optimal value. Each one of the 128 MEA entries needs 21 bits for addressing the 1125K pages per Pod and 4 bits for its counter, leading to an area cost of only 400B per Pod and ∼1.5KB total. Compared to the state of the art, MemPod's activity tracking requirement is ∼341x smaller than THM's (512KB) and ∼6100x smaller than HMA's (9MB).

5.3.2 *Performance Comparison*

Figure **??** presents a performance comparison of MemPod, HMA, THM and a configuration with 9GBs of on-chip HBM memory, normalized to the performance of a hybrid memory configuration without migration capabilities. We evaluated all mechanisms with caching disabled.

	mix1	mix2	mix3	mix4	mix5	mix6	mix7	mix8	mix9	mix10	mix11	mix12
astar		✓					✓	✓	✓✓		✓✓	
bwaves					✓		✓	✓✓	✓			✓
bzip				✓	✓	✓	✓	✓				✓✓
cactus				✓	✓✓	✓		✓				✓✓
dealII				✓✓	✓		✓	✓	✓			✓✓
gcc	✓	✓	✓	✓		✓✓				✓✓		
gems	✓	✓					✓		✓		✓	
lbm	✓		✓			✓				✓		
leslie	✓	✓	✓				✓		✓		✓✓	
libquantum			✓			✓				✓✓		
mcf	✓	✓	✓	✓✓	✓	✓				✓		
milc	✓		✓	✓						✓		
omnetpp	✓	✓						✓			✓	
soplex			✓		✓	✓				✓		
sphinx		✓	✓						✓		✓	
xalanc					✓		✓	✓				✓
zeusmp	✓	✓					✓		✓		✓	

Table 3: Mixed workloads description

Based on the results we can derive some interesting observations:

- In some workloads migration is harmful to performance, as observed with the bwaves workload, where a no-migration scheme reports higher performance (lower AMMT). We observe that in those cases, MemPod leads to deteriorated performance compared to THM. However, in the case of zeusmp, MemPod increases performance, while THM and HMA report higher AMMT than the no-migration scheme.

- MemPod outperforms the state-of-the-art competitors in the majority of our workloads, and in several cases scoring very close to an HBM2-only configuration.

- On average MemPod reports 25% higher AMMT than HBM2-only, while THM and HMA report 39% and 41% respectively.

- All mechanisms outperform HBM2-only when executing the libquantum experiment. We attribute this result to a combination of correct timing, application behavior and workload size. In the case of libquantum, to working set size fits entirely in

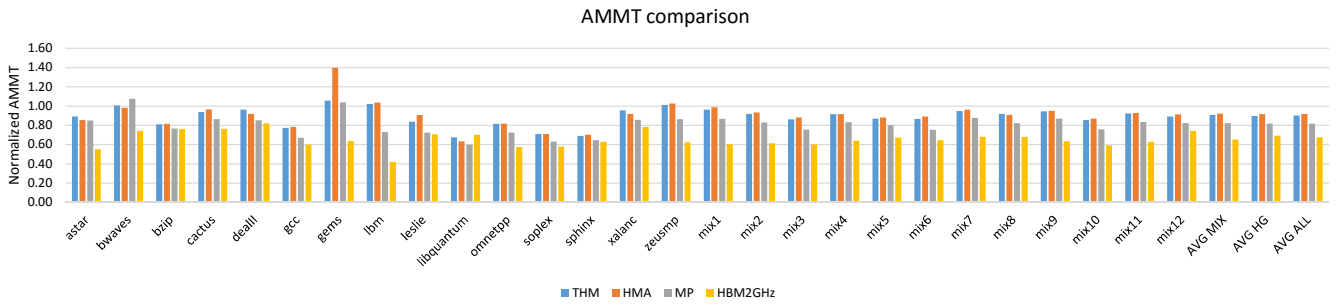


Figure 10: Performance Comparison: AMMT is normalized to a hybrid memory without any migration mechanism.

our fast memory. As a results after some migrations, the entire working set will be present in our HBM. However, correct timing is the driving factor behind this impressive performance increase. Our results show the row-buffer hit ratio to be ??x times larger than having HBM2-only (and random page assignment). Apparently, page migrations resulted in an in-memory page order that exploits almost every bit of memory parallelism from the application. This result could be further explored and intentionally recreated in some future work.

5.3.3 *Caching Effect*

I need to run one small experiment to complete this subsection. We also need to finalize how we want to model HMA's interrupt, PT updates and cold TLBs. Based on previous discussions, we can assume that sorting HMA's counters is completely overlapped by servicing requests (in the best case) and as such HMA doesn't stall for sorting.

Migration mechanisms will be forced to include a cache since activity tracking and remap table structures are commonly too large to hold on-chip. The use of a cache will unavoidably hinder performance. In this experiment we evaluate the impact of a cache on each mechanism's performance. As described in Section **??**, each mechanism has different cache requirements. THM caches its counters and remap table together with its "Segmented Remap Table" **Verify the name** structure. HMA has no need for a remap table however it has high storage requirements for its counting mechanism. MemPod only needs to cache its large remap table since MEA counters will be on chip. For the purposes of this experiment, we simulate all mechanisms with a 64kB direct-mapped cache. For MemPod, cache is divided equally over four Pods (16kB per Pod).

HMA's design further complicates this study, since sorting all activity counters at each epoch is performed by the OS, utilizing the cpu's cache instead of the dedicated migration cache. Our HMA results do not include any penalties related to sorting, OS interrupts and cold TLBs and Page Tables. As such, the reported HMA values can be considered overly optimistic.

Part of our study was to determine if the slow off-chip memory could be an acceptable solution to serve as the backing store location of each mechanism's structures. If the reported performance is acceptable, using the DDR4 memory would be ideal since we won't be reducing the effective capacity of the small HBM.

Figure ?? shows each mechanism's relative slowdown, normalized to the performance of the corresponding mechanism when no cache is simulated. **DISCUSS**

6. CONCLUSIONS AND FUTURE WORK

MemPod is a scalable, modular and efficient dynamic memory management mechanism. Our analysis demonstrated significant and encouraging results compared to state-of-the-art proposals. The modular design achieved with the use of Pods allows for a more scalable migration mechanism while at the same time enforcing small limitations on migration opportunities.

MEA counters have not been previously used in the micro-architectural context. Our analysis demonstrates it can be beneficial in more than just a migration mechanism. Any proposed mechanism that needs some sort of activity tracking or even identifying frequently-occuring phenomena could possibly gain significant performance benefit at lower cost through the use of MEA.

Not sure what to keep/remove here. I'll think a little more about it. In addition, MemPod leaves a lot to be investigated in future work. We intend to focus on extending our design to support even more complex memory configurations, with the addition of non-volatile memory which adds a reliability aspect. Ordering of migrations could result to incredible increase in performance, as demonstrated in experiment ?? and libquantum, allowing for future research. MemPod could also be used to accommodate memory scheduling mechanisms by utilizing its internal structures, or on the other hand, we could focus on "Pod-aware" memory scheduling.