Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories

Mitesh R. Meswani Sergey Blagodurov David Roberts
John Slice Mike Ignatowski Gabriel H. Loh

AMD Research
Advanced Micro Devices, Inc.

{mitesh.meswani, sergey.blagodurov, david.roberts, john.slice, mike.ignatowski, gabriel.loh}@amd.com

Abstract

Die-stacked DRAM is a technology that will soon be integrated in high-performance systems. Recent studies have focused on hardware caching techniques to make use of the stacked memory, but these approaches require complex changes to the processor and also cannot leverage the stacked memory to increase the system's overall memory capacity. In this work, we explore the challenges of exposing the stacked DRAM as part of the system's physical address space. This non-uniform access memory (NUMA) styled approach greatly simplifies the hardware and increases the physical memory capacity of the system, but pushes the burden of managing the heterogeneous memory architecture *(HMA) to the software layers. We first explore simple (and somewhat impractical) schemes to manage the HMA, and then refine the mechanisms to address a variety of hardware and software implementation challenges. In the end, we present an HMA approach with low hardware and software impact that can dynamically tune itself to different application scenarios, achieving performance even better than the (impractical-to-implement) baseline approaches.*

Keywords: Memory architecture, Die-stacked memory

1. Introduction

Die-stacked memory is an emerging technology that has the potential to significantly attack the memory wall problem [34]. In particular, placing one or more 3D stacks of memory inside the same package as the processing units can provide orders of magnitude more bandwidth at significantly lower power costs [2]. In recent years, there has been significant advancement in the industry including the development of die-stacked memory standards and consortia [13, 16, 24], and various announcements from several processor companies [12, 23, 2].

One key challenge of using in-package memory is that the current integration levels are still insufficient to satisfy a high-end system's memory capacity requirements. For example, 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 [12]. 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)).

A large 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) [18, 25, 14, 15, 36, 28, 20, 8, 10]. 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 [19].

Hardware caches, however, are not without their challenges. In particular, the implementation complexity is quite significant for several reasons:

- The most effective DRAM cache proposals involve tag organizations quite different from conventional SRAM-based on-chip caches. Some place tags directly in the DRAM for scalability [18, 25], while others use page-sized cachelines with sectoring and prefetching [15, 36, 14], all of which require from-scratch engineering efforts to implement the new cache control logic.
- The cache controller must also perform traditional memory controller tasks such as issuing row activation and precharge commands, respecting DRAM device timing constraints, scheduling command and address buses, and managing DRAM refresh. Designing a single unit that simultaneously handles the cache controller functionality while juggling the low-level 3D DRAM controller issues is likely much more difficult than designing either one in isolation.
- A hardware cache implementation is also "baked in" to a particular processor design, making the cache organization, policies, etc., inflexible once built.
- Last, and possibly the most costly obstacle, is that the verification effort required to ensure the correct implementation of hardware-based DRAM caches is daunting and has largely not been discussed in the DRAM cache literature. There are so many combinations of functional operations (e.g., cache hit, cache insertion, writebacks) with DRAM timing scenarios (e.g., row buffer hit, row conflict but t_{RAS} not elapsed, bus write-to-read turnaround not sat-

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

isfied) at different operating points (e.g., temperature dependent refresh rates, entering/exiting DRAM low-power modes) where the DRAM cache controller must be verified to operate correctly.

Furthermore, the capacities of die-stacked DRAMs, while insufficient to serve as the entirety of a system's main memory, still provides a non-trivial amount of capacity. Ideally, the stacked DRAM would be available to augment or increase the size of the system's total physical memory, whereas caching does not provide this benefit. The addition of even modest amount of capacity was shown to substantially improve performance of capacity-limited workloads [6].

In this work, we explore a different direction for die-stacked DRAM, which is to keep the hardware simple, but to push the management of the memory system up into the software layers. We consider a heterogeneous memory architecture (HMA) where both the fast, stacked DRAM and the conventional, off-package DRAM are all mapped to the same physical address space, somewhat similar to a non-uniform memory architecture (NUMA) approach [4]. We consider minimal hardware support to *aid* the software layers in managing the HMA, but the physical interface to the stacked DRAM remains simple. At the same time, we maintain *application-level transparency* in that all of the necessary software changes to use the HMA are constrained to the operating system (or runtime system) so that existing applications can run unmodified. In this manner, we retain the programmer productivity advantages of conventional hardware caching techniques, while keeping the hardware implementation cost down.

2. Heterogeneous Memory Architecture (HMA)

In this section, we first go through the hardware implications of a non-cache die-stacked memory organization, and then we discuss the software challenges that arise when hardware management has been removed from the picture.

2.1. HMA Hardware

2.1.1. Package-Level Organization In our baseline hardware organization, we assume a high-performance accelerated processing unit (APU) consisting of multiple CPU cores and a GPU. The APU is 2.5D-integrated on a silicon interposer [7], along with multiple 3D stacks of DRAM. Figure 1 shows the packaging of these components. As discussed earlier, the 3D DRAM does not provide sufficient capacity for a server-class system, and so conventional off-package memory (e.g., DDR) is also provided, as shown in the figure.

Recently announced systems integrate 16GB of in-package 3D DRAM; over the next few years with continued DRAM density scaling and increases in 3D DRAM stack heights, achieving 32-64GB of in-package memory is not unreasonable. With main memory capacities in the range of 256GB-1TB (or more), the size of the stacked DRAM would be in the approximate range of 1/16 (64GB vs. 1TB) to 1/8 (32GB vs. 256GB) of the off-package memory capacity. For the

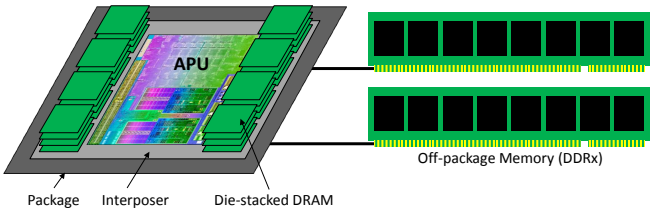

Figure 1: Package-level organization of the target system consisting of a high-performance APU 2.5D-integrated with multiple DRAM stacks, along with conventional off-package DDR memory.

majority of the studies in this paper, we assume a 1:8 ratio of stacked vs. off-package memory.

Typical bandwidths for off-package DDR memory is in the tens of GB/s. For example, a single DDR3 channel clocked at 1600 MHz can provide a peak bandwidth of 12.8GB/s. Most systems are equipped with only a few channels (e.g., 2-4). With die-stacked DRAM, the bandwidths are significantly higher. As an example, a single stack of JEDEC "high-bandwidth memory" (HBM) provides eight channels of 128 bits each, with a data transfer speed of 1Gbps [16], for a total of 128GB/s. When considering that a system could include eight stacks of 3D DRAM, that would push the aggregate peak bandwidth to 1TB/s.

The result is a heterogeneous memory architecture (HMA) consisting of a small portion of very high bandwidth in-package memory, with a much larger amount of lower-bandwidth commodity off-package memory.

2.1.2. Address Mapping Without loss of generality, we assume that the stacked DRAM is mapped into a first contiguous range of the physical address space, and that the off-package DRAM is mapped to a second contiguous range in the same address space. An example physical address mapping is shown in Figure 2(a). During the system boot process, the standard memory discovery and mapping mechanisms are used to detect the stacked DRAM (as well as how much there is) and assigns it to a particular region of the physical address space. This is effectively no different than what is already done today when the system boots up and must take inventory of how much memory has been populated in each of the memory slots for each channel on the system motherboard, but with the process extended to also consider the die-stacked DRAM.

2.1.3. Memory Controllers Modern processors already contain multiple memory controllers (MC). The addition of die-stacked DRAM would require additional memory controllers that were properly designed to handle the timing parameters specific to the stacked memory. In the best case, this could simply be an existing memory controller reconfigured with new timing parameters. Figure 2(b) shows a portion of the processor with the additional die-stacked DRAM controllers (shaded) along with their physical-level interfaces (PHYs).[1] The processor also already contains logic to route memory

[1] The stacked DRAM PHYs are shown smaller (but not to scale), as they only need to drive a point-to-point link across the interposer, as contrasted to a conventional DDR PHY that must drive a higher impedance load through the C4 bumps, the package substrate, out the package pins, across the motherboard, and to the memory DIMMs.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

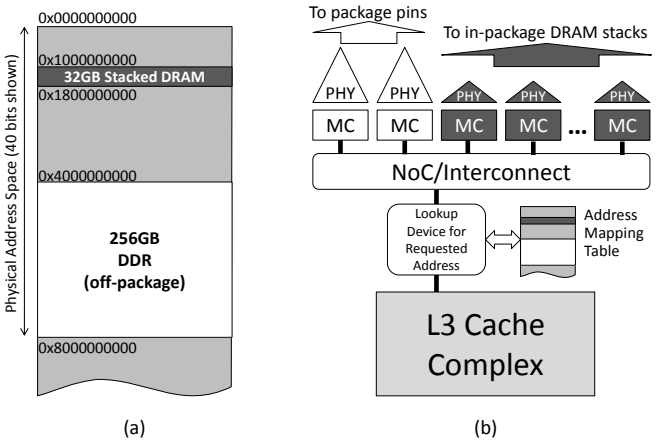

Figure 2: (a) Example mapping from a physical address space to the conventional off-package and die-stacked memories, drawn approximately to scale. (b) Hardware modifications (shaded) to support an HMA organization, including additional memory controllers and memory physical-level interfaces (PHYs) and additional entries in the address-to-device mapping table.

requests to the correct memory controllers. In a conventional processor without die-stacked memory, the processor uses the mapping that was set up during the boot process (Section 2.1.2) to perform an address range check that determines which memory controller "owns" the address, and then the processor forwards the request to the respective memory controller. In a system supporting an HMA, the mapping table simply contains a few additional entries corresponding to the memory ranges that map to the die-stacked DRAM. When a request targets one of these ranges, the processor uses the same mapping table to then forward the memory request to the corresponding die-stacked DRAM memory controller. Overall, the hardware changes necessary to support an HMA organization are relatively minor and heavily leverage existing mechanisms.

2.2. HMA Software

2.2.1. Roles and Responsibilities A hardware cache controller can store copies of individual cache lines, whether small (64B) [18] or large (2KB) [14], but at the operating system (OS) level all memory is managed at page granularity (typically 4KB for x86 architectures). The operating system (or other runtime software) must somehow decide which pages should be placed in the fast die-stacked memory, and which ones should be kept off-package.

The hardware cache controller has full visibility into each and every memory request, whereas the OS has only a few bits of coarse-grained information through its page table entries (PTEs). A typical PTE includes a "referenced" bit and a "dirty" bit. The OS can clear a PTE's referenced bit, and later when the processor accesses the page, the hardware page-table walker will set the referenced bit. By observing this bit, the OS can determine that the page was used. However, the OS cannot determine how *recently* the page was accessed beyond the broad interval of "sometime since the referenced bit was last reset", which could be an instant ago or much further in the past. Similarly, the single bit does not provide any differentiation between a page that was merely accessed a single time versus one that was repeatedly reused over and over again. Both the recency and frequency of page usage would likely be critical inputs for the OS to use in deciding which pages should be placed in the fast memory, but neither are readily available. Additionally, the referenced bit is set for any access to this page, but not all accesses result in traffic to main memory (for example, a page that is frequently accessed but that also almost always hits in the on-chip caches would not benefit much from being migrated to fast memory).

Apart from deciding what pages should go where, the OS must then also actually move the pages between fast and slow memories, which introduces a host of performance challenges discussed below.

2.2.2. Overheads and Performance Challenges Even assuming that the OS could make a good selection of pages to place in the fast memory, the OS is still at a significant disadvantage compared to a hardware cache. The DRAM cache controller simply inserts a copy of the cacheline into the cache after the requested data have been retrieved from the off-package main memory. The OS must first take a processor interrupt to do *anything*. Pages must be copied/moved between the fast and slow memories, the corresponding PTEs must be updated (even a single physical page could have multiple virtual addresses pointing to it), and then a translation lookaside buffer (TLB) shootdown must be issued to each core that may have cached copies of stale PTEs. The interrupt latency alone can take several microseconds (e.g., average of $\sim 2\mu$s on Real-Time Linux [29]), and the TLB shootdown can take many microseconds as well.[2]

At least without some significantly more complicated mechanisms [3], this entire migration and remapping process must occur while all of the target application threads are suspended. Correctness issues may arise if an application is allowed to continue running while its pages are being moved around. For example, a write operation could be lost if a thread writes to a page that was already migrated but whose PTE had not yet been updated (or if the processor held a stale PTE). This means that every microsecond of OS overhead related to HMA page migration directly adds to the application's execution time. This overhead can potentially be offset by performing migrations infrequently, but then this runs the risk of a migration/page-selection policy that is so slow that by the time a page is migrated into fast memory, it may no longer be hot. The migration frequency must be carefully balanced to ensure a sufficiently reactive page selection policy while keeping interrupt and TLB shootdown overheads under control. We discuss specific policy implementation in Section 3.

A pure OS-driven page caching approach is likely difficult to implement for similar performance reasons. The idea would

[2] We measured the TLB shootdown latency on an AMD 32-core platform running the Linux OS, and found that the latency grows with the number of cores involved in the shootdown. For 4, 8, 16, and 32 cores, the shootdown latency takes approximately 4, 5, 8, and 13μs, respectively.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

be to dynamically swap pages between the stacked DRAM (which acts like a conventional "main memory") and the off-package memory (which acts more like conventional storage or swap). Only pages currently in the stacked DRAM have valid PTEs, and so any miss would result in a page fault; the OS would then step in, swap the requested page in, update the page table, and perform a TLB shootdown if necessary (i.e., if a victim page was removed from the fast memory, thereby rendering its PTE invalid). This amounts to taking a page fault *on every single stacked DRAM miss*, likely crippling performance. This type of approach has been attempted [9], but in the context of paging between DRAM and a fast memory-mapped solid-state drive (SSD) device (where the alternative would be the even slower path of going through the file system).

2.3. Isn't This Just NUMA?

The described HMA has many similarities to a NUMA memory organization, in that different regions of the physical address space have different performance characteristics. However, the nature of the non-uniform performance due to a hybrid of memory technologies is different from a traditional NUMA scenario. In a multi-socket NUMA system, it is desirable to allocate memory on the same sockets as the threads that will be accessing that memory the most. This can be challenging when threads from multiple sockets access a particular object with similar levels of intensity. Placement of the object at one sockets helps some threads and hurts the other, and placement at the other socket hurts the former and helps the latter. The HMA problem differs in that all threads run "close" to the fast-DRAM, and so the desire is to allocate *all* memory in the die-stacked DRAM. The problem is not about localizing data near its compute, but rather that there is more data than will fit in the local "NUMA domain".

3. Managing the HMA

The operating system or runtime must somehow detect the pages that will be used most often, where "used" is with respect to the memory traffic, not in terms of memory instructions that could hit in the on-chip caches. We now detail some possible HMA management policies.

3.1. Baseline Algorithms

3.1.1. Oracular Page Selection The first policy that we describe attempts to fill up pages in stacked memory based on perfect future knowledge. It provides an upper bound for the best possible performance of an HMA management policy for a unified address space. We divide a program's execution into epochs, where each epoch is a fixed-length interval (e.g., 0.1 seconds). During the course of each epoch, the hardware tracks the memory traffic (i.e., from last-level cache misses) associated with each page.

Oracle is an idealistic reference policy, at the start of each epoch it selects the N pages of memory that will serve the most traffic (reads or writes) during the upcoming epoch (assuming a first-level memory capacity of N total pages). These N pages are then loaded into the first-level memory, and all remaining pages are placed in the second-level memory. The page tables are updated to reflect the new virtual-to-physical memory assignments (and all translations set to be valid), and so no further page faults are required. The only overhead is a single OS intervention (equivalent to a migratory fault) to update the mappings once per epoch.

3.1.2. History-based Page Selection Rather than relying on unavailable information about the future (as in the case of the Oracle policy), the history policy that we describe next selects pages based on past traffic patterns. At the end of an epoch, the OS sorts all pages, and for a stacked DRAM with a capacity of N pages, the OS selects the top-N pages responsible for the greatest amount of main memory traffic to be placed into the stacked DRAM. Any pages that were in the fast memory but did not make the top-N cut for the next epoch must be migrated back out to off-package memory, and the page table must be updated to reflect all new page placements. A similar history-based approach was previously explored [19]. The key implementation challenge is in devising a scalable mechanism that allows the hardware to maintain per-page traffic counts; this will be revisited below in Section 3.2.

This history-based page selection relies on the assumption that a page that was heavily used in the recent past will continue to be heavily used (and similarly, cold pages remain cold). The selection of a shorter epoch length allows the history policy to adapt more quickly to changes in a program's working set, but then it also increases the inter-epoch OS overhead.

3.1.3. First-touch Page Selection The second baseline policy is derived from a common NUMA memory allocation strategy. The "first touch" NUMA allocation policy places a page of memory on one of the memory channels belonging to the same socket of the thread that first requested the page (if there is room). For private, per-thread data, this ensures that such pages are co-located with the only thread that will ever use them. For shared pages, the heuristic also works well in many scenarios.

For an HMA system, at the start of each epoch, all pages are initially marked invalid in the page tables (and all TLBs must be shot down). Then, as pages are accessed, there are two scenarios. In the first case, the page is currently in the off-package memory, and so the page is migrated to the fast memory, the page table is updated to reflect the new mapping, and the PTE valid bit is set. In the second case, the page is already in the fast memory (i.e., it was mapped there during the previous epoch), and so only the PTE's valid bit needs to be set. Any subsequent accesses to this page will proceed normally (i.e., without a page fault) as the corresponding PTE is now valid. As new pages are accessed for the first time during the epoch, more pages are migrated into (or re-enabled) in the fast memory until the stacked DRAM is completely filled up. Note that as pages are migrated in, pages from the fast memory may also need to be migrated out. Note

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

that any such pages still have invalid PTEs, and so the OS is free to change their mappings without worrying about TLB consistency issues (i.e., no need for shootdowns). Eventually after the stacked DRAM has been filled, the OS re-enables the valid bits on all remaining pages that will now be stuck in the slower off-package memory for the rest of the epoch.

This first-touch HMA policy does not require any additional hardware support to track page access counts, but it may select a sub-optimal set of pages to place in the fast memory as the first pages touched in an epoch are not necessarily the ones that are responsible for the most memory traffic. Another challenge for first-touch is that each "first touch" event during an epoch incurs a page fault that includes an OS interrupt. Contrast this to the history-based approach that takes a long time to sort and migrate a large number of pages at the start of an epoch, but then does not interfere until the next epoch.

3.2. Hot Page Policy

The history-based policy described above is not immediately practical because it requires sorting *every* page in memory based on access counts and then selecting the top N pages (where N is the size of the fast, die-stacked DRAM).[3] Our next policy simplifies the problem by simply dividing pages into "hot" versus "cold" pages using a simple threshold. Any page that has an access count that exceeds a fixed threshold θ is classified as hot (for the current epoch). By choosing θ correctly, the number of pages classified as hot will hopefully be close to N.

In the ideal case, if the hot-page count is *exactly* N, the sorting operation can be completely skipped because the choosing of the top-N out of a set of N simply amounts to choosing the entire set. Sorting is also unnecessary when the size of the hot set is less than N. Under normal operation, the hot set will usually be different than N. In the case where there are more hot pages than fit in the fast memory, the OS can choose the top-N out of this smaller set (i.e., not *all* of memory). In the case where the set of hot pages fails to use up all of the fast memory, the remaining capacity is filled using a first-touch approach.

A key advantage to this approach is that, similar to the history-based policy, pre-loading pages and making the corresponding PTEs valid at the start of the epoch cuts down on faults related to migrating pages via the first-touch mechanism.

This *first-touch hot-page* (FTHP) policy is effectively a generalization of both the history-based and first-touch algorithms. When θ is set to zero, then *all* pages are considered hot, and so sorting the hot pages is the same as sorting all of memory, which then is equivalent to the history-based policy. When θ is set to ∞, then no pages are ever in the hot set, so the entirety of the fast memory is populated via the first-touch mechanism. Setting θ to finite, non-zero values strikes a balance between the two approaches: some number of hot pages are pre-loaded into the first-level memory to reduce faults, and the remaining capacity can be populated dynamically based on whichever pages the program accesses.

Tracking Page Access Counts The FTHP policy still requires tracking the per-page memory traffic. To support this, we propose an extension to the existing TLBs and hardware page-table walker (PTW) logic, along with some potential changes at the software level. We logically extend each PTE with a count field that records the number of memory accesses for that page. We likewise extend each TLB entry with a field to track the number of accesses to the page corresponding to the cached PTE. On each last level cache (LLC) miss, the counter is incremented. On a TLB eviction, the counter value is written back and added to in-memory count. The OS can then use these access count values in choosing its hot pages.

For some architectures, the PTEs do not have enough free bits to store an access count. In these cases, the counts may need to be placed in an auxiliary data structure that parallels the page table [17]. In other architectures with enough free bits in the PTEs (or in a system where large pages are used, which frees up several PTE bits), the counters may be directly placed in the PTEs. In any case, the hardware PTW must be modified to perform this read-modify-write on the counters, which is not currently supported. However, the changes are minimal as the PTW already has all of the logic it needs to read and write PTEs, and so the primary additional hardware is simply a single adder and the relevant control logic/finite-state-machine updates. Overall, the PTW changes are relatively minor, but the software changes could be much more challenging in practice (especially when the hardware companies are not the ones writing the OS software!). We revisit this later in this section.

3.3. Setting the Hotness Threshold

The FTHP policy relies on setting the hotness threshold θ. Setting the value too low can cause the HMA to have too large of a hot set, thereby increasing the overhead of choosing the top-N; setting the value too low results in a small hot set, which in turn increases the faulting overhead due to the first-touch mechanism. The best threshold can vary from application to application, within phases of an application, and also depending on the actual size of the die-stacked DRAM in a given system. It is undesirable to have the OS vendor hand tune this parameter to support a wide range of applications and platforms.

We propose a dynamic feedback-directed HMA policy that can dynamically adjust the hotness threshold θ to achieve a best-of-both-worlds approach between history-based and first-touch policies. At the start of each epoch, the size of the hot set is compared to the size of the die-stacked DRAM (N). If the hot set is too small to fill the fast memory, then θ is lowered which causes more pages to be classified as hot. Likewise, if the hot set is too large, θ is increased which causes fewer pages to be put in the hot set. If the feedback mechanism works well, then the size of the hot set should converge to N.

[3] A full sort is not strictly necessary, as the problem is actually to choose the top-N, which is algorithmically faster than sorting all objects in a set.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

Having the hot set size equal (or come very close to) N is very desirable. In the case that both are exactly equal, then the number of first-touch faults is reduced to zero (because the entire fast memory has been populated and so there is no room to bring in any more pages via first-touch), and there is no need to sort the pages to find the top-N as discussed earlier.

There are a variety of ways to update θ each epoch. We simply use a proportional linear feedback control system. This is but one possible algorithm for dynamically adjusting θ; we do not claim it is optimal, but it provides an effective proof of concept of the approach. The key result is that the parameter θ need not be hand-tuned, and this allows the mechanism to be more easily applied to a broader set of applications and platforms.

3.4. Low-cost Hot Page Detection

The hardware and software support required to track the hottest pages may be too invasive for main-stream systems (e.g., those based on the x86 ISA). Here, we propose an approximation that greatly simplifies the hardware support. We also explain how to adjust the FTHP algorithm to work with the simplified hot-page tracking.

The first part is on the OS side. Instead of using the PTE's accessed bit to mean that a page has been referenced by the processor, we instead re-interpret it to mean that the page has been classified as "hot", where hot is the same as before, meaning that the page has generated more than θ memory requests to it. This new-interpretation provides backwards compatibility; if the OS sets θ to zero, the hardware PTW will update the accessed bit on the first reference to the page.

We maintain per-TLB-entry access counts to track the memory traffic associated with each page. However, rather than accumulate the total traffic in an in-memory counter (which requires new OS data structures or many unused PTE bits), the processor monitors the count in the TLB entries. When the TLB's access count for a page exceeds the threshold θ, it invokes the hardware PTW and sets the accessed/hot bit in the PTE.

At the end of the epoch, all pages that had more than θ memory accesses will have their respective accessed/hot bits set in the PTE. These now form our hot set of pages. Similar to FTHP, if the hot set is smaller than N, then we use first-touch to populate the remainder of the die-stacked memory. If the hot set is larger, then we simply take the first N pages (as we have no other way to differentiate them as they all only have a single hot bit that is set). The dynamic-feedback approach can be applied here as well to adjust the value of θ to try to make the size of the hot set match N as closely as possible.

This "hot bit" HMA policy requires minimal hardware changes. Setting the hot bit in the in-memory PTE uses the exact same mechanism that is used today to set a PTE's accessed bit. A new model-specific register (MSR) is needed for the OS to specify the current threshold θ. This is necessary, because when the processor increments the in-TLB access counts, it must compare the count against θ to know if the page has crossed over the hotness threshold and therefore have its accessed/hot bit set.

This hot-bit policy is also just an approximation of the FTHP policy. It could happen that a TLB entry's access count reaches θ-1, and then the TLB entry gets evicted. When the page is re-accessed, there is no saved history for previous access activity, and so the count starts again at zero. In a pathological scenario, the page is accessed frequently enough that it should be classified as hot, but its hot bit never gets set. As mentioned earlier, having only a single bit of discrimination leaves the policy unable to "sort" the hot pages when the hot set is greater than N, which is another possible source of behavioral deviations from FTHP.

Detecting hot pages is an area of ongoing research, for example the IBM POWER8 architecture [11] stores memory reference counts and reference history for a page in the page table. An OS-level page manager was proposed in [35] to manage a DRAM+PRAM architecture. The OS page manager uses multiple least recently used (LRU) queues [37] to find the set of pages that are most frequently written and migrates them to the DRAM. We note that the differences and the goal of our hot-page detection was to take advantage of the existing page table walk hardware as much as possible, and to explore approximate solutions for a practical implementation. Overall our research focus is on various methods to manage HMA (hardware cache, page cache, and unified address space) for a multi-tiered memory hierarchy in which stacked memory is only a portion of the total memory footprint of the system.

3.5. Other policies

We have also evaluated a number of hardware cache and OS page cache policies. We model a hardware managed DRAM cache based on the Alloy Cache model [25], with a direct-mapped organization with tags and data (64-byte cachelines) read out together to minimize latency. By caching only blocks that are missed on by the higher-level caches, a hardware cache by itself would not be able to exploit the spatial locality in the access streams. To address this, our cache implementation employs a prefetcher that pulls 1 kilobyte worth of data along with the missed bock.

For the OS page cache policy, the crucial aspect is deciding when a page needs to be evicted. We implement a multiple first in first out (Multi-FIFO) page replacement algorithm borrowed from PerMA and DI-MMAP projects [32] and [31]. In this replacement policy the evicted page is first put on one of two queues (hot-page or eviction) and pages are evicted to make space only if the level 1 memory is full. Evictions only occur from the eviction list.

4. Experimental Methodology

4.1. HMA simulation

We implemented a memory trace-driven HMA simulator that models the two different types of memory, along with the

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

different management policies. To drive the simulations we collected traces for memory accesses that miss the last level cache for workloads executing on a system with an AMD A10-5800K APU clocked at 3.8GHz and 16GB memory. The traces have information about reads/writes, timestamps and physical address being accessed. The timestamps were used to generate interrupts for our epoch driven policies.

From the memory traces along with additional information gathered from hardware performance counters, we develop simulator that functionally models the behavior of the individual HMA policies that can determine whether the individual requests hit or miss in the fast memory, migration activity, whether interrupts or TLB shootdowns are needed, etc., and these events are coupled with an analytical model to project performance and energy. The model uses the Leading Loads method [26, 30] to split an application's execution time into CPU time and memory time using performance counters. The leading load model calculates the memory time as the time spent servicing the leading (first in case of many outstanding loads) non-speculative load that misses the last level cache. This model has shown to be fairly accurate for predicting execution time for dynamic voltage frequency scaling (DVFS) scaling on AMD processors [30]. For each load instruction, the appropriate latency is added to the total execution time based which memory the load went to. This estimate represents a worst-case execution time assuming that the additional latency cannot be hidden by concurrent computation or pre-fetching. The DRAM page hit ratio[4] (from performance counters) is taken into account during this calculation, as page hits typically have a lower latency than page misses. We assume the same row buffer hit ratio for both types of memories, although differences in channel counts and other memory parameters could cause this to vary in practice. Writes are not considered to impact execution time, assuming that there is sufficient write-buffering in the processor chip, writes can be drained to memory during periods where the memory bus is idle. A fixed 5μs time penalty is charged for each page fault [27] to cover the basic interrupt costs, and then another 3μs penalty is applied whenever a TLB shootdown [33] is required. Page migration time is also added to the execution time, as the migrations do not overlap with computation because this is performed by the OS page fault handler. The time for migration is assumed to be bounded by the off-package memory bandwidth.

Once total (worst-case) execution time is derived, we estimate active and background power consumption for the memory system (not the APU compute units). The current and voltage parameters for 3D-stacked DRAM are based on measurements from a high-bandwidth memory (HBM) device. DDR4 power is estimated based on the Micron DDR3 DRAM power calculator [21], using DDR4 parameters from [22].

[4]Except for Windows applications, the data were not available and we used a hit ratio of 30%, which was the average for Rodinia and PARSEC.

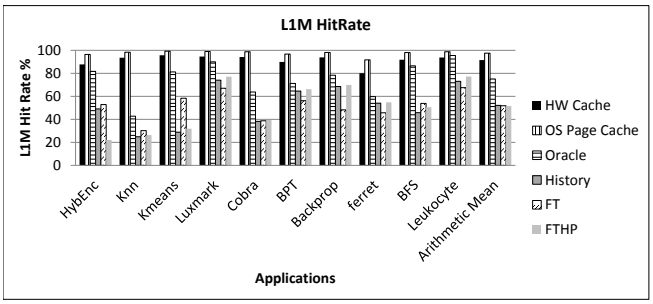

Figure 3: Hit rates in die-stacked, level-one memory.

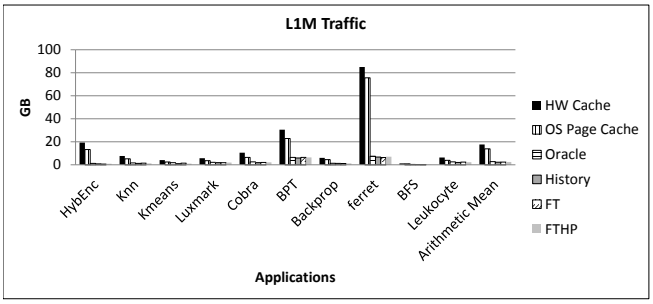

Figure 4: Total traffic at the die-stacked, level-one memory.

4.2. Benchmarks

We evaluated a total of 23 different workloads from PARSEC [1], Rodinia [5] and some Windows desktop applications. We only focused on the applications that exhibited high levels of memory traffic. PARSEC is a suite of multi-threaded CPU workloads composed of diverse applications from emerging areas ranging from computer vision to financial analytics. We ran the simlarge input dataset for PARSEC. Rodinia is a suite of benchmarks that are designed to test accelerators such as GPUs. We evaluated a subset of six Rodinia benchmarks that represented areas such as data mining, machine learning, to graph traversal. Rodinia was run with its standard default input dataset. In addition to benchmarks we also evaluated real-world Windows applications for video transcode and Windows SDK. Together these workloads provide a mix of operating systems, devices (CPU, GPU), and a wide array of application domains that provide a rich suite to test our management policies.

5. Experimental Results

We plot and discuss results of the top 10 of a total of 25 applications that generated the most traffic. Most of the remaining applications generate significantly lower memory traffic and as such the memory policy does not have a significant impact on speedup. The results in this section are presented for a baseline L1M:L2M ratio of 1:8. Additionally, for the FTHP policy we show results for epoch of 0.1s and threshold of 32, which were the best static value on average for performance. We discuss sensitivity sweeps for FTHP policy, for L1M:L2M ratio, epoch length and threshold in Section 6.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

5.1. Analysis of Hit Rates and traffic

We first analyze the three performance metrics shown in Figure 3, Figure 4 and Figure 5. Each of these figures plots for each application on the X-axis the performance metric on the Y-axis. Also shown in each of these plots is the arithmetic mean across all applications for a given management policy. The hit rates in the L1M is the highest for both the caching policies, with the OS Page Cache being on average the best performer. Among the HMA policies, oracle policy comes quite close to the hit rate of the caching policies. The oracle policy shows that there is some epoch level locality that can be leveraged, however, oracle is unattainable and as such gives the upper bound on performance for the HMA policies. The caching policies do come at the price of page migrations and in particular the OS page cache policy also incurs page faults and TLB shootdown overheads. The plots for traffic to each memory level reflects the sum of all traffic going to a given memory level. For the hardware cache, traffic is generated for reads/writes as well to satisfy misses. Similarly page cache has traffic for reads/writes but also incurs significant traffic to migrate pages on page faults in L1M. On average, the traffic generated by both the caching policies is between 3x to 5x as compared to the remaining HMA policies. This can be attributed to the significant migration traffic for misses in the caching policies. In contrast, the HMA policies generate migration traffic only at epoch boundaries. From these three performance metrics it is clear that while the caching policies have very high hit rates they come at a price of significant increase in migration traffic. Hence, discussed next in Section 5.2 is our evaluation using performance and energy models.

5.2. Performance and Energy results

The performance and energy model, discussed in Section 4.1, takes the performance metrics from simulation as well as additional information from performance counters and estimates performance and energy. Show in Figure 6 and Figure 7 are the speedup and energy consumption (memory only) for HMA architecture with HBM as L1M and DDR4 for L2M memory. Figure 6[5] shows for each application on the X-axis the speedup over a memory composed of only DDR4 memory. Also shown in this figure is the arithmetic mean speedup across all applications for a given policy. For comparison, the figure also includes the speedup of an HBM-only memory over DDR4, which serves as a ceiling for speedup. For these workloads an HBM only memory has a speedup of about 20% on average over all DDR4 memory. The oracle policy is the most effective with an average speed up of about 15% and comes very close to the HBM speedup. The hardware cache did not show any significant improvements and the worst performing is the OS page cache policy with an average factor of *3x slowdown*. The advantage of high L1M hit rate for this policy is overshadowed by the overheads due to page faults and associated costs. Thus, for an HMA system the OS page cache policy illustrates that hit rate is not the only optimization target and other factors such as migrations and faults play a big role as well. After oracle, the history and FTHP policy perform nearly the same, with history being marginally better. The history policy has its own set of challenges associated with devising a scalable mechanism to track per-page access count and sorting at epoch boundaries. The FTHP policy reduces some of these complexities by just tracking pages that have crossed some θ accesses. Thus, if number of pages that are accessed more than θ are less than size of L1M then no sorting is required. Hence by tuning θ we can reign in the cost of sorting and find the right balance with the desire to pre-load hot pages and reduce overheads. The simple FT policy shows an average of 18% slowdown which shows that some sort of hot page selection is required to take advantage of the faster HBM memory.

Figure 7[6] shows for each application on the X-axis the consumed energy for memory. The figure also shows the arithmetic mean for a given policy across all applications. In general the trends follow those observed for speedup in Figure 6. The OS page cache consumes 4x more energy than the HMA policies. Hardware cache is better than OS page cache but it has significant energy expenditure and has 2x more energy burn than HMA policies. All HMA policies have lower energy usage than page cache and in particular FTHP has similar usage as oracle and history policies. Hence by using FTHP policies we can get performance comparable to history, but with lower complexity. Next, we discuss some approximations of detecting hot pages which may show the path for an even more practical solution.

5.3. Discussion

While maintaining a hot page list using our θ threshold reduces complexity over history policy it still requires some sorting overhead if we have more hot pages than the size of L1M. Instead here we discuss a simpler variant of detecting hot pages by just marking pages as hot if they are accessed more than θ times. The hot bit splits the pages into hot and cold sets. If the hot set is larger than the L1M size we choose the hot pages on a first-come first-serve basis (i.e., the pages that were marked hot first are preloaded). The main difference between this hotbit version of FTHP and the one presented in previous sections is that some pages that are accessed later in the epoch may get left out even if they have high access counts. The adjustment to θ can alleviate and balance those issues to some extent. Choosing a static θ value may not be practical as it may be sub optimal for some applications and may require prior profiling. That is why we have also devised a variant that adjusts θ dynamically.

To that extent, we have deployed a simple proportional

[5]The Y-axis is truncated at -100 so that trends for other points are visible and we provide data label for points below -100.

[6]The Y-axis is truncated at 4.5e09 so that trends for other points are visible and we provide data label for points above 4.5e09.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

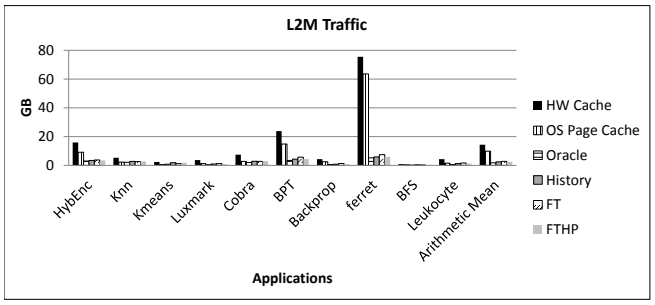

Figure 5: Total traffic at the off-package, level-two memory.

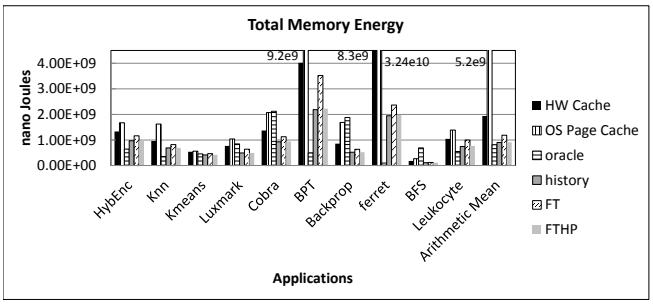

Figure 7: Energy consumed for memory.

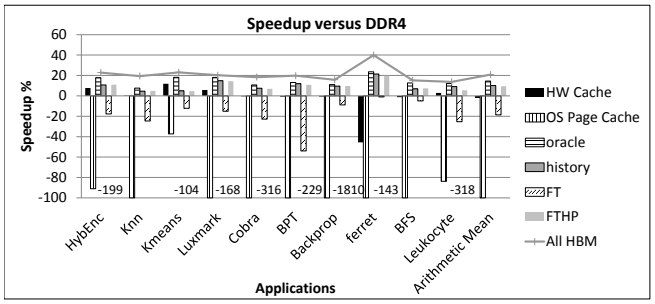

Figure 6: Speedup over DDR4 memory.

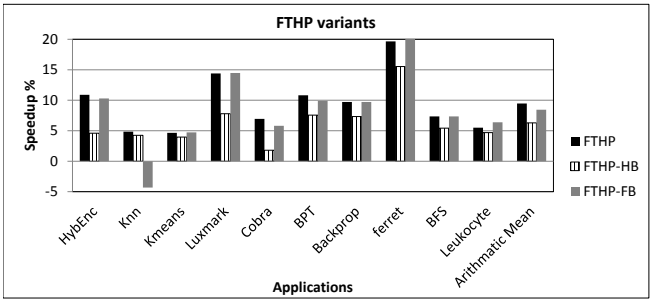

Figure 8: Speedup for FTHP, FTHP hotbit, and FTHP with dynamic threshold adjustment.

linear feedback controller that aims to bring the hot set close to the size of the first level memory in pages. That way, the need for sorting is eliminated, and the hotbit approximation acts as the history policy. At the end of each epoch, the controller monitors the hot set identified with the presently chosen threshold θ. The relative difference between this set and the size of the fast memory region is then calculated. The resulting number is the rate by which the threshold θ is adjusted for the next epoch (to the smaller value if the hot set fits within level one memory with some space left for FT, and to the bigger value, if the hot set is too big for L1M). Such reactive proof-of-concept implementation, though useful, can be replaced with a more sophisticated threshold adjustment method, e.g. interpolation based on prior θ values.

We compare FTHP, FTHP-hotbit and FTHP with dynamic θ adjustment in Figure 8. FTHP with hot bit gets within 2/3 of the speedup of FTHP. The dynamic feedback algorithm allows to further improve the results and approach FTHP most of the time. By its nature, the feedback however can be detrimental to performance at times, in case the application is highly dynamic (the feedback control is unable to keep up with the hot set) or if there is a significant hot set overreaction to the parameter adjustment.

6. Sensitivity Analysis

In Section 5, we had focused mostly on the best static parameter values for some of the knobs that we have to set for running FTHP. In this section we analyze the sensitivity of those knobs for speedup. We present a one-factor analysis by sweeping one knob at time, and fixing the remaining knobs to their best static value. First, presented in Figure 9 is the sensitivity of performance to the chosen epoch interval. In this figure for each application shown on the X-Axis we plot the speedup over DDR4 for four different epochs. The epoch determines the ability to capture application phases. The smaller the epoch, the more fine grained the phases we can detect, but this comes at the cost of overhead of handling epoch related management activity. The lowest epoch of 0.001 seconds is set based on the value of the Linux scheduling timer. This timer is used by the operating systems process scheduling algorithms among other things. Reducing granularity below this value will cause overheads in the system software which we would like to avoid. Conversely our largest epoch is topped off at 1 second since some of our applications run only for a second or so. The epoch sweep shows that very fine granularities cause more overheads to handle epoch related activities. On an average the lowest 0.001 epoch results in 5x slowdown, and going to 0.01 improves it but it still has 100% slowdown. Epoch lengths of 0.1 and 1 second yield similar speedups, with 0.1 epoch being the best. These results show that epoch is an important parameter for a system that manages Heterogeneous Memory Architecture in software. It should not be too fine grained, as the systems software overhead will outweigh any potential benefits from a more robust memory scheduling.

Next we sweep different values of FTHP thresholds to study its effect. Figure 10 shows for each application on the X-axis its corresponding speedup at three different thresholds, 4, 32, 4096. The threshold θ sets the criteria for classifying a page as hot. Setting this value too low can result in too many hot

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

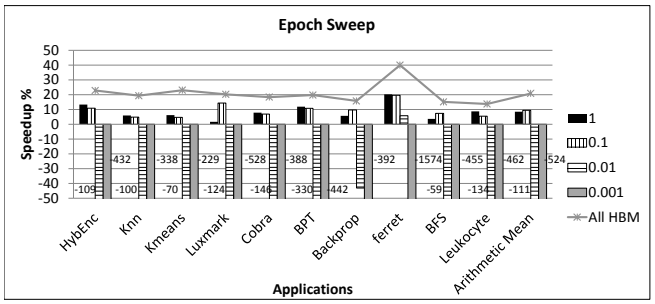

Figure 9: Figure shows speedup over DDR4 for FTHP with different epochs.

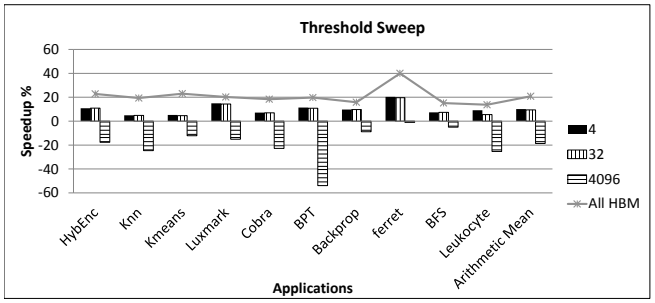

Figure 10: Figure shows speedup over DDR4 for FTHP with different hot page thresholds.

pages and as a result we may not choose the correct set of hot pages to preload at epoch boundaries. On the other hand choosing a threshold too high will preclude a lot of hot pages. The threshold sweep shows that on average threshold of 4 gives the best performance and threshold of 32 gives nearly the same performance. When going to a higher threshold we start observing a slowdown as demonstrated by a threshold of 4096. It is important to note that these results hold for a certain L1M size and a particular workload, and are provided here for illustrative purposes only. For a different stacked memory and different applications, other values for the threshold are likely more appropriate. This highlights the importance of the dynamic parameter adjustment for the software-orchestrated memory management.

Lastly we sweep different values of L1M sizes. We expect that as L1M size increases we should expect better performance. For this sweep we dynamically also adjust the threshold so as to take advantage of larger L1M memory. Figure 11 shows the speedup for each application for 6 different L1M sizes which are 1/32, 1/16, 1/8, 1/4, 1/2 and all HBM. As anticipated as we increase the size of the L1 memory we see improvements in performance. The improvements in performance is almost linear on average when going from 1/4 to all HBM; however for the lower ranges it does not increase linearly, for example a speedup of 1/4 is 7.5% and 1/8 is 6.3%. We conjecture that to take advantage of bigger memory we not only have to set the correct threshold but we may also have to tweak epoch lengths.

We have presented one-factor sensitivity analysis that shows

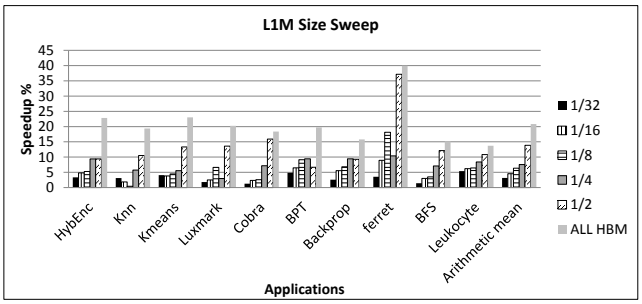

Figure 11: Figure shows speedup over DDR4 for FTHP with different L1M sizes.

that three important parameters for FTHP have impact on speedup. A larger study, which is out of the scope of this paper, is to perform a k-factor analysis to find the optimal settings of these factors and potentially adapt them dynamically per application. We leave k-factor analysis and sensitivity studies for energy as part of future explorations.

7. Conclusions and Future work

Due to several pressing constraints, the memory subsystem of the future will likely contain a heterogeneous mix of memory technologies, such as emergent die-stacked DRAM alongside with more conventional DDR4. This heterogeneous memory architecture prompts the design of management policies that place workload data across levels of memory judiciously. Previous solutions have focused on hardware caching techniques to make use of the stacked memory. These approaches, however, require complex changes to the processor and also cannot leverage the stacked memory to increase the system's overall memory capacity. In this work, we have considered exposing the stacked DRAM as part of the system's physical address space to simplify the hardware implementation and increase the physical memory capacity of the system. We have shown that the overhead incurred by the software layers in this case must be carefully weighed in order to prevent severe performance degradation due to low die stacking hit rates, excessive data migration and page fault servicing delay. Furthermore, we have presented an efficient hybrid memory management system in software that dynamically adapts itself to different applications with minimal hardware support. We also described how such a policy can be shaped by gradually easing the assumptions of several more informed (and less practical) methods.

In the future we would like to explore non-volatile random access memory (NVRAM) as part of main memory and extend our analysis to more than two memory levels. We would like to further investigate methods to detect hot pages and continue our explorations with dynamic switching between different management policies (subject to active workload requirements) and API support for preferential binding of the frequently accessed memory objects to the faster memory regions in the heterogeneous memory subsystem.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.

Acknowledgment

We would like to sincerely thank P. Conway and J. Jillella for their help with the tracing tools and for providing us with some of the memory traces used for HMA simulation. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

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

[3] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Concurrent Page Migration for Mobile Systems with OS-Managed Hybrid Memory," in *Proc. of the Computing Frontiers*, May 2014.

[4] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for numa memory management," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 5, pp. 19–31, Nov. 1989.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[6] A. J. Chiachen Chou and M. 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.

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

[8] M. El-Nacouzi, I. Atta, M. Papadopoulou, J. Zebchuk, N. E. Jerger, and A. 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.

[9] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale, "DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications," in *Proc. of the ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, November 2012, pp. 731–735.

[10] F. Hameed, L. Bauer, and J. Henkel, "Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies," in *Proc. of the*, 2013.

[11] *POWER8 Processor User's Manual for the Single-Chip Module*, IBM.

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

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

[14] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers," in *Proc. of the Intl. Symp. on Computer Architecture*, 2013.

[15] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, January 2010, pp. 1–12.

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

[17] M. Lee, V. Gupta, and K. Schwann, "Software-Controlled Transparent Management of Heterogeneous Memory Resources in Virtualized Systems," in *Proc. of the the Workshop on Memory Systems, Performance, and Correctness*, 2014.

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

[19] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*, New Orleans, LA, February 2012.

[20] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. 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.

[21] Micron Technology Inc., "Micron DDR3 Power Calculator," http://www.micron.com/products/support/power-calc.

[22] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and mitigating refresh overheads in high-density ddr4 dram systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485927

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

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

[25] M. Qureshi and G. 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.

[26] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski, "Practical performance prediction under dynamic voltage frequency scaling," in *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE, 2011, pp. 1–8.

[27] M. Saxena and M. M. Swift, "Flashvm: Revisiting the virtual memory hierarchy," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, ser. HotOS'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855568.1855581

[28] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. 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.

[29] T. Straumann, "Open Source Real Time Operating System Overview," in *Proc. of the Int'l. Conference on Accelerator and Large Experimental Physics Control Systems*, San Jose, CA, 2001.

[30] B. Su, J. L. Greathouse, J. Gu, M. Boyer, L. Shen, and Z. Wang, "Implementing a leading loads performance predictor on commodity processors," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USENIX Association, 2014, pp. 205–210.

[31] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "Dimmap—a scalable memory-map runtime for out-of-core data-intensive applications," *Cluster Computing*, pp. 1–14, 2013.

[32] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of nvram in data-intensive architectures: an evaluation," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 703–714.

[33] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 340–349. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.65

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

[35] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, Raleigh, NC, September 2009, pp. 101–112.

[36] L. 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.

[37] Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *In Proceedings of the 2001 USENIX Annual Technical Conference*, 2001, pp. 91–104.

Authorized licensed use limited to: University of Illinois. Downloaded on October 05,2023 at 08:41:31 UTC from IEEE Xplore. Restrictions apply.