Understanding the Performance of Parallel Applications in a Tiered-memory System

Kaiwen Xue
kaiwenx@andrew.cmu.edu

1 Summary

Data center applications often require large amounts of memory up to terabytes. This scaling of memory has two implications: 1) memory cost will account for the major server cost of data centers instead of compute cost [18], and 2) virtual memory performance must scale with the larger memory. Therefore, modern data centers employ various techniques to use memory more wisely. This includes: 1) leveraging a tiered memory system, where the traditional main memory is divided into heterogeneous devices with different latency, bandwidth, and cost, to balance the cost and performance, and 2) using huge pages to reduce the TLB miss rate to improve address translation overhead.

The architecture and systems community has proposed various ideas in terms of both approaches. However, current research often overlooks scenarios where huge pages and tiered memory systems are simultaneously employed. Because both approaches are widely adopted in today's data center [13, 18], we conjecture that lacking attention to such an environment could potentially hurt application performance. Studies focusing on huge pages typically assume a homogeneous main memory, whereas those centered on tiered memory mainly address the migration of standard pages across different tiers. A system with homogeneous main memory that is extremely optimized for minimizing TLB miss rate using huge pages might fail to offload proper cold pages to lower tiers because they are promoted to a huge page and cannot be directly offloaded. On the other hand, if the hotness of smaller regions is uneven within a huge page, candidate pages selected for offloading to lower tiers using existing algorithms developed by tiered-memory researchers could be sub-optimal.

In addition, we found that no approach can avoid answering an important question: how to detect which memory regions are hot? The "hotness" of a memory page is useful in both approaches because the system needs to choose which regular pages should be promoted to huge pages, and which cold pages should be offloaded to the memory of the lower tiers. Nevertheless, the proposals in existing works are often limited. Their hotness detection approaches either incur huge overhead or are limited to small memory size.

One important category of applications running on modern data centers is big data applications. In addition to their intensive memory usage, they are often executed in parallel. For example, graph applications often use the shared memory paradigm of parallelism using OpenMP when executing on a single node. It is thus possible that a poor memory placement or promotion policy causes the application performance to be even worse in parallelism because it might cause more workload imbalance if the memory is not managed fairly per core.

This project serves as a motivation for research that aims at solving these problems. We first discuss the memory management background and review the issues with existing solutions in detail (section 2). To overcome these problems, we propose *hotness walk*, a design that provides architectural support for detecting the hotness of pages in any range (section 3). Then, we extend a full-system simulator to prepare us to implement offloading and page size promotion algorithms (section 4). Using the simulator, we profile the performance of the parallel applications to profile the challenge and discuss the feasibility of the design (section 5). Finally, we use the evaluation results to direct the future work to be done to complete the details of this design.

2 Background and Related Work

In this section, we cover the background and current literature on huge page management and tiered-memory systems. We also describe why existing approaches fail to detect the hotness of pages optimally in the scenario where both multiple page sizes and heterogeneous memory exist.

2.1 Virtual Memory and Management of Huge Pages

Modern computer systems use virtual memory to provide the software with the illusion of a unified address space, such that the applications can operate on memory in isolation. Under this scheme, whenever a virtual address is referenced, the memory management unit (MMU) in the processor will walk through multiple levels of page tables, tables translating virtual to physical addresses. During this page table walk, the process utilizes specific bits of the virtual address as offsets to identify the next level's physical page table address. The number of bits after the last-level page table determines the page size. The PTE table is normally the last-level table with 12 offset bits, resulting in 4KB pages. The page table walk process is inefficient [19] since each memory access requires multiple additional references. Therefore, modern processors often cache the translation results in the translation lookahead buffer (TLB). The TLB lookup can happen in parallel with the L1 cache, reducing memory access overhead significantly.

Since TLB is essential in improving application performance, a high TLB miss rate is unacceptable. However, it is very easy for applications with large memory footprints to fill up the TLB and cause a high miss rate. The solution is to use huge pages by walking one or two fewer levels. The information about whether a page is huge is stored in each page table entry by the operating system and will be respected by the MMU when performing the page walk. With 2MB or 1GB pages, the entries of translations to be cached in TLB can be reduced vastly. The state-of-the-art techniques aiming at allocating huge pages smartly to use TLB more efficiently while limiting memory bloating and preserving process fairness include Ingens [8], HawkEye [15], and PCC [12]. While these works are effective in reducing TLB misses and improves application performance in a homogeneous memory system, they do not optimize performance for a system with multiple tiers of memory. For example, they might merge normal pages into huge pages with some of the merged pages being better off offloaded to lower tiers.

2.2 Tiered-memory System and CXL

To increase the memory capacity and reduce stranding, modern data centers use disaggregated memory to serve as either shared memory pool [10] or lower-tier backup memory [13]. While multiple design strategies exist for tiered-memory systems, our project will emphasize the Compute Express Link (CXL) [1] approach. CXL is a cache-coherent interconnect protocol. Connecting the processor to memory chips with CXL, the processor can access more than what the main memory provides with the cost of longer latency, effectively a trade-off of cost and capacity with latency. The development of CXL is heavily supported by industry and is considered to be the basic building block of the server processors of the next generation [1, 10, 13].

One of the main questions in designing a tiered memory system is placing hot memory in the higher tiers. Since CXL memory can be exposed to the software as zero core NUMA (zNUMA) memory, the OS can manage CXL inside its NUMA subsystem. TPP [13] proposes to augment the NUMA subsystem by intercepting the memory swapping operation and first redirecting the memory to be swapped to the zNUMA node, effectively the CXL. While TPP uses the access bit in the page table to determine whether a page is hot, the detection is in the OS-level page size granularity and thus cannot deal with the case where normal page hotness is not balanced within a larger page. That is, if the OS maps a 2MB huge page, TPP cannot detect any hotness information in a 4KB region within this huge page.

2.3 Detection of Hot Pages

To determine both the best page sizes to be used in a system as well as the best placement of the pages in a tiered memory system, it is important to detect the hot pages. Useful information on which pages are hot can help with grouping the hot regular pages to be a huge page, or offloading cold pages to a slower tier. Through a survey of existing literature, we list four requirements for a successful hotness detection mechanism:

1. The detection should be accurate and of fine granularity. As discussed before, if the hotness information detected is only of huge page granularity, the system will miss the opportunities to offload only the cold pages in the area to a lower tier.

2. The detection method itself should be of low overhead and should scale with memory size. For example, methods such as PEBS [3] that pin a core are unlikely to scale with the increasingly large amount of memory [14].

3. After having the hotness information, the system software, such as the OS, should be able to determine the pages that need action efficiently. That is, even if the memory becomes huge, the system should not scan all the memory for their hotness information.

4. A full system often consists of more than one process. To preserve fairness across the processes, page offloading and promotion decisions should be made where other processes' performance is not negatively affected. This means the hotness information should be maintained per process, so that both the hottest and coldest pages can be found in each address space.

We now identify how the existing works cannot satisfy all these conditions simultaneously. Ingens [8] and HawkEye [15] both require scanning of the whole address space to determine targets to be promoted. The access-bit approach employed by TPP [13] is efficient, and TPP uses the LRU list to decide which pages are to be offloaded, which avoids scanning the whole address space. However, as discussed in the last section, TPP failed to detect hotness at a fine granularity. PCC [12] employs architectural support to write the page numbers of several normal 4KB pages most suitable for promoting to 2MB. However, PCC provides hotness information suitable for promotion and not any information about the **subpages**, making it very hard to make informed decisions on demoting (splitting) the huge pages. MEMTIS [9] has attempted to manage the memory of multiple page sizes in a tiered memory system. However, while MEMTIS claims it controls the overhead of tracking memory accesses, it still uses a PEBS-based mechanism. When the memory scales to terabytes, the limited overhead is bound to decreased accuracy. Telescope [14] proposes to avoid address space at scanning at the scale of terabytes by utilizing the access-bit information in upper levels of page tables. Nevertheless, this Telescope still uses PEBS and employs the assumption [2] that large memory regions tend to have homogeneous hotness, which is proved not effective in [9].

Note that many of the above approaches failed to consider process colocation. For example, TPP uses an LRU list that selects offloading victims on a physical level. PCC's hotness structure contains information about only one process. Not accounting for multiple processes running in a system will result in the unfairness of processes. Works such as MaxMem [17] and HeMem [16] take the data center QoS of the colocated workload into consideration so that the system can use an appropriate amount of memory of the faster tier according to the QoS requirement set by the system administrator.

In summary, we believe none of the current work as of our writing successfully meet all the requirement of detecting useful hotness information accurately and fairly with no overhead. In this project, we propose a design called *hotness walk* that satisfies these requirements. Limited by the scope and time of a course project, we extend an existing simulator that can be used for future development of the design.

3 Hotness Walk Design

In this section, we propose a design called hotness walk, a novel mechanism to detect the per-process page hotness of any range with low overhead. Hotness walk requires modification to the microarchitecture of the memory management unit (MMU). In certain cases of translation, the MMU generates additional memory requests to the page table and increments a hotness counter right in the memory that can be addressed by the page table.

3.1 Tracking Page Hotness at Fine Granularity

Figure 1 shows where the hotness information is held when page walks are performed in a 4-level radix page table. In particular, we store the hotness information of a 4KB page right under (physically contiguous) the PMD table that addresses it. Note that one PMD table addresses 1GB of memory. When a huge page is mapped, a PMD entry is a physical page number (PPN). In Intel x86-64, this can be easily checked by the hardware as the PS-bit of a PMD entry is set for a huge page.

When an MMU (page table walker or TLB) request happens, it first goes through the normal process to find the appropriate PMD entry. When the page is a 2MB huge page, the MMU additionally calculates a physical address (the blue temperature box in Figure 1) based on the end of the PMD table and offsets it with the 12-20 bits of the virtual address, the usual PTE offset should the page be 4KB. It then increments the counter addressed by this address by 1. The counter thus represents the hotness of a 4KB range, and since it is stored in the page table of a particular process, the information it provides is also per-process.

The MMU cannot increment the counter every time it receives a memory request, because that will incur the prohibitive overhead of memory accesses. Therefore, different policies can be configured to sample the memory accesses. We explore these policies in **??**. The hotness walk only keeps additional data for pages mapped to 2MB. This can save the memory overhead used to store the hotness information. All the access-bit-based approaches can still be easily incorporated for 4KB pages.

3.2 Identifying Promotion and Demotion Candidates

The hotness walk hardware directly stores hotness information in the memory that can be accessed through page table walk. Therefore, once the OS would like to periodically select a certain number of hot pages to promote, it performs a page table walk in the software, going to the lower levels of the radix tree only when the access bits of the higher levels are set. Then it can select any hottest 4KB ranges to promote to huge pages or migrate them to a faster tier. Once the operation is done, the OS halves all the counters in the hotness structure so that it only obtains the hotness information of the closest past accesses.

Note that there is no fixed design in the OS or other user space system software at this moment. However, with the new architectural support of additional hotness information, the OS has a rich design space for implementing a variety of policies as it sees fit.

We argue that the hotness walk mechanism satisfies all four requirements in section 2. First, it can track memory access at a 4KB granularity, even if the range is mapped as 2MB. Second, both the tracking and identification process of the page hotness are efficient - no PEBS or address space scanning is used. Finally, since the hotness information is placed in the page table, it is naturally private to processes. The software has the flexibility to choose any policies it wants using this information, such as satisfying a dynamic QoS requirement as in [17].

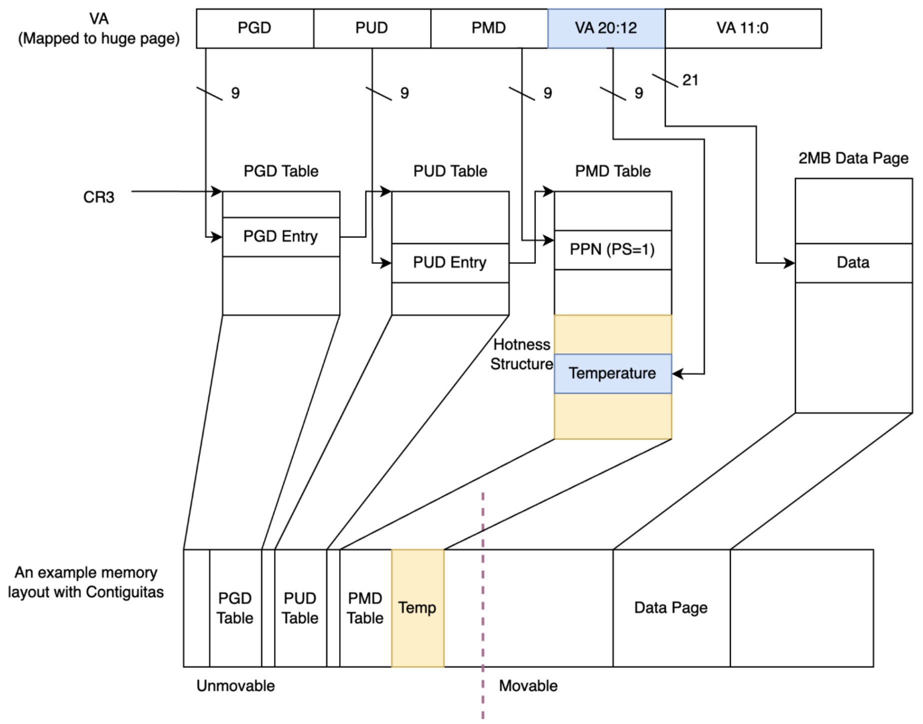

Figure 1: Hotness walk structure memory layout

3.3 Hotness Walk for Parallel Applications

One potential weakness of the hotness walk, if it were to be used with parallel applications, is that it directly updates memory. To perform memory updates in parallel, additional cycles must be spent on locking the interconnect and other synchronization measures, to avoid race conditions when multiple processors try to perform the hotness walk to the same region. We model the additional cycles needed to perform this operation and discuss its implication in section 5.

4 Simulator Design

In this section, we describe how we extend a full-system simulator's functionality to prepare us for the design of implementation in section 3.

We use QEMU [4] as the front-end to functionally emulate the instructions running on top of the Linux kernel of 5.12. For the back-end, we use the Structural Simulation Toolkit (SST) [6] to perform cycle-accurate simulation and use DRAMSim3 [11] as the main memory back-end. The front end, upon receiving instructions from the OS, sends the instructions to SST via a piece of file-backed shared memory. The SST picks each instruction and simulates them in a cycle-accurate behavior. This communication was done as a project in an earlier iteration of this course (section 7).

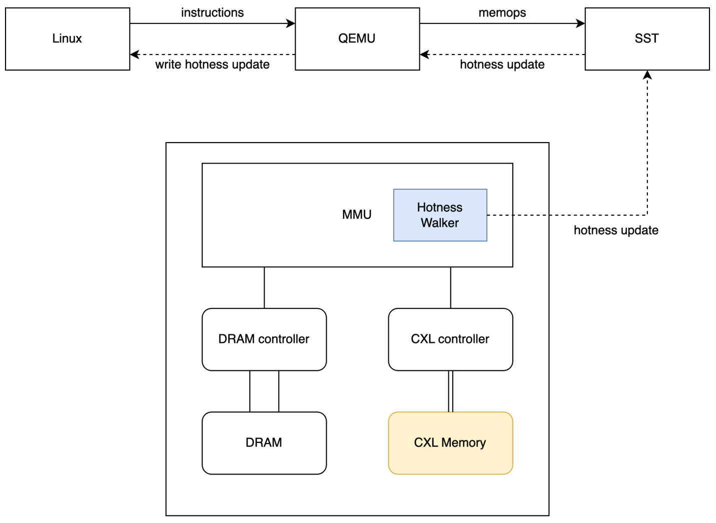

Figure 2: Simulation Infrastructure

As shown in Figure 2, we augmented the infrastructure discussed above. First, we added the backward channel that enables SST to communicate data with QEMU. This is useful because we "write" the hotness counter in SST, but would like to make that value visible in the kernel. Thus, we added the hotness walker to the MMU. Once the hotness walker has incremented the hotness count for a number of cycles, it writes the addresses of the hotness back to the QEMU. Since

QEMU has access to the address space of the Linux kernel it emulates, it can write the update back to Linux. Again, all communications are done per-core in shared memory.

The hotness walker is called when the TLB is processing the memory instructions it received from the simulated core. The instruction contains everything needed for a page table walk, including the physical and virtual addresses, the page table entries of all levels, the page size, and whether the page is a kernel page or not. Using this information, the hotness walker calculates the memory location to send the hotness request to, and sends the memory request, incrementing the hotness count for that memory location. The SST keeps a list of the counters saved in all the physical addresses belonging to the hotness structure. Upon a certain number of cycles, it sends the updates of these saved data back to QEMU, as described above.

In addition, we configured CXL memory using the same DRAMSim3 back-end. Although flat-mode CXL memory should be connected to the core through PCIe, we do not need that amount of detail, as we only need to model the additional latency CXL brings. One potential limit of our current setup, however, is that we use DDR4 for CXL memory, instead of exploiting CXL's potential to save memory cost by using older generations of memory. In SST, the MMU is connected to the memory hierarchy, which ends up reaching the memory controller. We assign one memory controller to each type of memory. Figure 2 omits the cache hierarchy, which should lie in the middle of the MMU and the memory controller, but emphasizes we tune the latency of accessing different tiers of memory by tuning their link latency. We add 89ns of latency to the link between CXL memory and its memory controller, drawn from [10].

The extension of the SST prepares us for implementing and tuning our design. However, the actual implementation of the design, especially the policy made by the OS, is out of the scope of this project. We instead perform different motivation experiments to demonstrate the necessity and potential of our design and show how it would impact the execution of parallel applications.

4.1 Multicore Simulation

We extend the simulator so that it can simulate the behavior of parallel applications. The initial implementation of SST communicates with QEMU in a piece of shared memory. When there are no instructions to run, the core simulated on SST sleeps until more instructions are available. The instructions are available when the QEMU fetches more instructions on that core. When these instructions are fetched, QEMU wakes up SST through a semaphore in the shared memory. However, when there are multiple cores, SST might be waiting for instructions on one core that does not have instructions available, while the QEMU tries to produce more instructions for a core whose instruction buffer is full, thereby causing deadlocks. We thus wake up the SST cores that cannot fetch instructions and insert no-ops to those cores, such that SST can proceed to the cores whose instruction buffers are full and enable QEMU to make progress. In our experiments, we discard the no-ops and only count the actual instructions as the instruction count.

4.1.1 Limitation of Multicore Simulation

While the above approach can estimate the relative performance for each core, it does not accurately simulate the multiprocessor behavior. This is because inserting no-ops when one core has no instructions to run prevents these cores from fetching additional instructions. Failure to do so might cause inaccuracy when multiple processors are using the same resource, e.g., contending for locks. All of our multicore experimental results are produced by the flawed approach, as we only found out the problem right before finishing the report. This leaves the accuracy of the results in question, though they are still somehow representative of the relative performance. We leave re-running necessary experiments related to the hotness walk in future work.

5 Profiling Parallel Applications with Tiered-memory Systems

In this section, we use the simulator mentioned in the last section to profile three parallel applications. Specifically, for a parallel application, we would like to explore the following questions:

1. What is the performance of parallel applications running on a tiered-memory system with multiple page sizes?
2. Will workload be more imbalanced due to the longer latency of accessing the slower memory tier? Is the tiered-memory environment something that requires the parallel application programmer to react upon?
3. What are the TLB miss rates under different configurations of page sizes and memory tiers?
4. What is the cost to synchronize the hotness update of multiple cores?
5. Why is it necessary to track hotness in a fine granularity? How imbalanced is page hotness among the address space?

We use the Entropy server of PDL to run our simulator. We use three shared-memory-based parallel applications that use OpenMP, 605.mcf_s (MCF) from the SPEC Benchmark Suite [5] and Page Rank (PR) and Breadth First Search (BFS) from the GAP Benchmark Suite [7]. We use the standard input provided by SPEC in MCF (memory footprint 4GB) and the Twitter dataset for the two graph applications (footprint

12GB). We use 4 cores whenever we claim multicore is used. We use 16GB for total system-wide memory.

We measure 1 for all of them but study 2-5 for only PR in detail due to the page limit. **Originally, we would like to also study the performance scaling relative to the number of cores as well, but due to the limitation to multicore simulation we just found out, we determined the results are too inaccurate, we have moved that part from the project, including the report and the posters.**

5.1 Performance of Tiered-memory Systems

To see the potential performance gain of a tiered memory system with multiple page sizes, we first measure the application performance with the simulation using relative CPI (cycle per instruction) as a metric. The instruction is measured as the total number of instructions aggregated from all four cores.

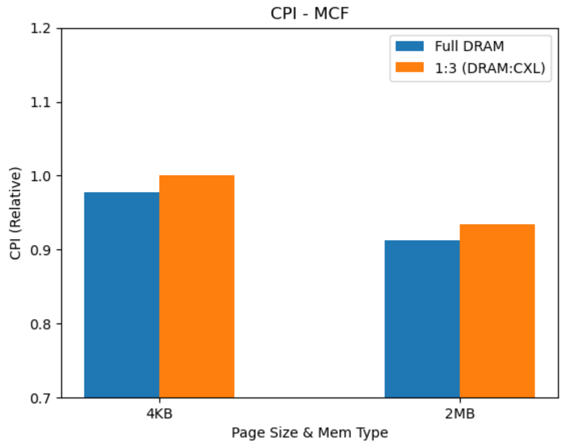

Figure 3: Performance - MCF

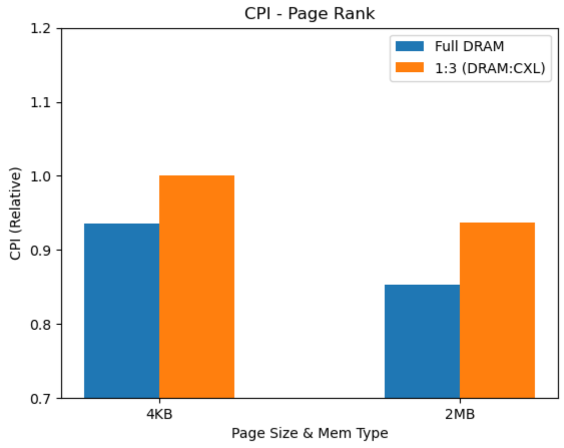

Figure 4: Performance - PR

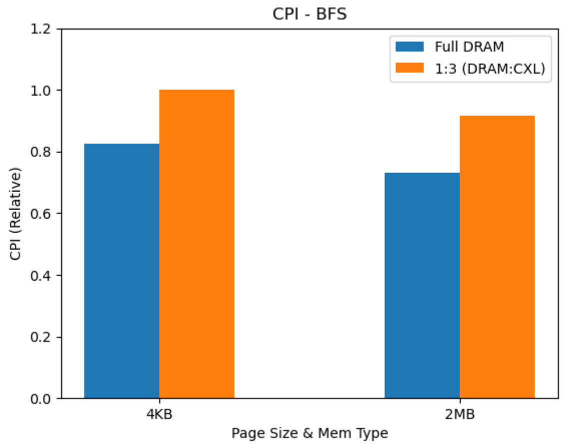

Figure 5: Performance - BFS

Figure 3, Figure 4, and Figure 5 shows the performance difference for all three applications. To get an estimation of where no proper placement policy is used, and no page sizes are tuned by the OS, we turned off NUMA balancing and transparent huge page. We also remap all kernel pages to 4KB. We find that using 4KB pages with tiered memory can reduce performance to 10-25% for graph applications (PR and BFS) and 8% for MCF. We interpret this difference as MCF is not as memory intensive as the graph applications, so less cycles are wasted on memory stalling and TLB miss latency.

5.2 Work Imbalance Across Cores in Tiered-memory Systems

We study the work imbalance across cores with PR. We measure the number of instructions processed by each core for the same number of cycles.

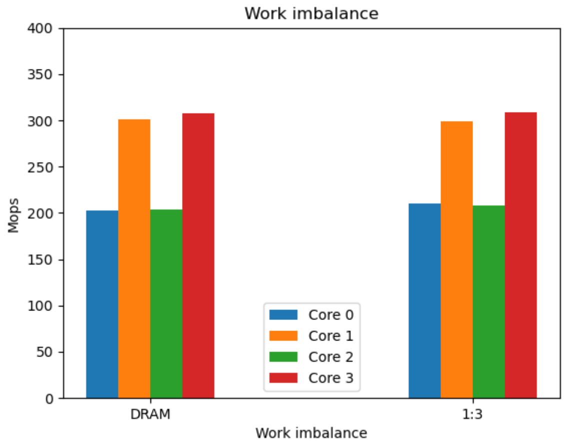

Figure 6: Work imbalance - PR

Figure 6 shows the work imbalance of PR of four cores with two configurations, DRAM and 1:3. The work imbalance situation does not change significantly with the different configurations. Therefore, we think it is the algorithms themselves that are important (e.g., Task 3 in Assignment 4), and the system software should deal with the placement of pages, not the application programmers.

5.3 TLB Miss Rate

In this section, we look at the TLB miss rate for PR in one core. Other cores have similar relative TLB miss rates of each configuration.

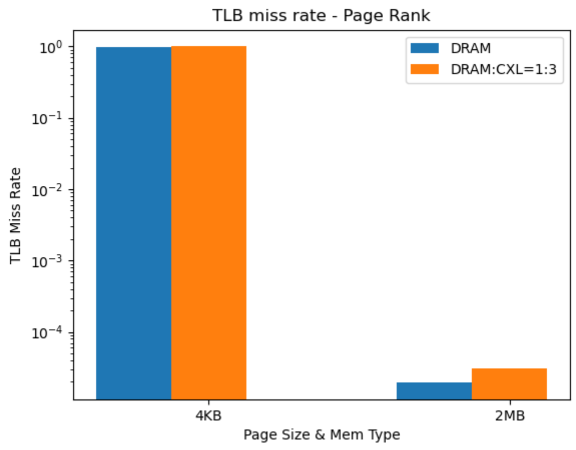

Figure 7: TLB miss rate - PR

Figure 7 shows the TLB miss rate of PR. We measure the TLB miss rate in a homogeneous DRAM setup and a 1:3 setup. In both cases, the TLB miss rate gets lower for several orders of magnitude. This is consistent with our expectations because a 2MB page can save the number of TLB entries by 512×. The same idea goes for both the homogeneous and a mixed setup. Since simulations are ended manually, there are some subtle differences in the exact memory access patterns. We believe the difference in the different configurations of tiers in Figure 7 is caused by variation instead of the tiers themselves.

5.4 Synchronization Cost of Hotness Walk

Accessing the hotness page is a read-modify-write operation, so it is not atomic. Synchronization should be in place to avoid incorrect hotness updates. The synchronization may include any cache invalidation traffic as well as locking the interconnection among the cores. We measure the overhead introduced by synchronization as follows using a simplified model: assuming the hotness update happens per L1 TLB miss, we add 100 cycles of extra latency to each miss to model all the contention caused by synchronization.

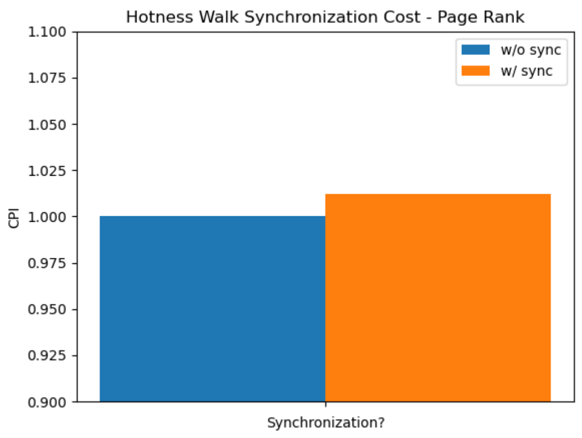

Figure 8: Synchronization Cost - PR

Figure 8 shows the CPI difference after synchronization cost is modeled, for a 1:3 configuration of 4KB pages of PR. The overhead is 1.2% when synchronization is used. We expect the synchronization cost for multiple page sizes to be lower because synchronization happens with TLB misses. This suggests that as long as we can keep the TLB miss rate low and design a policy that does not update the hotness structure too often, we can get a fairly low overhead caused by synchronization.

5.5 Necessity for Hotness in Fine Granularity

We assess the need for tracking in fine granularity by looking at how the hotness changes from time to time for each 4KB range. In general, tracking in fine granularity helps with page demotion (either from huge pages or from faster tiers). In this experiment, we simulate 0.3–0.4 billion cycles, around 60-90ms, on MCF and PR. The hotness of 4KB ranges is halved every 100,000 cycles.

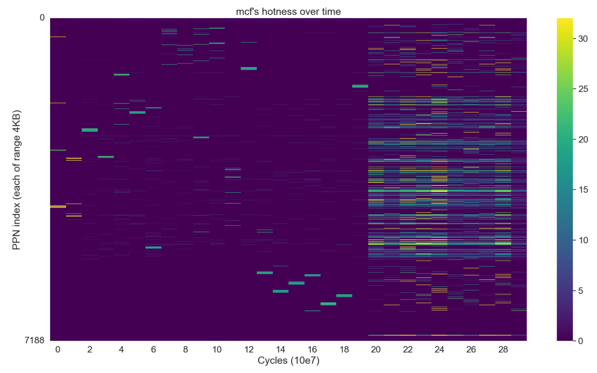

Figure 9: Hotness over time - MCF

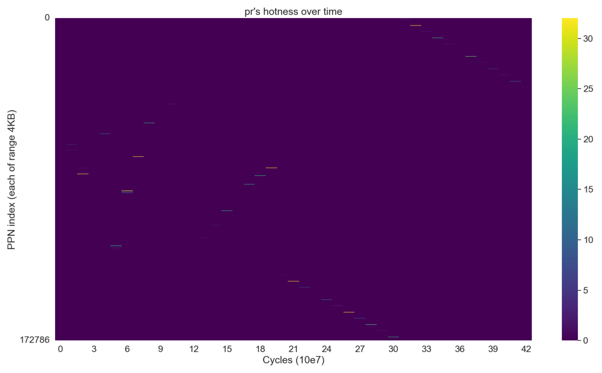

Figure 10: Hotness over time - PR

Figure 9 and Figure 10 demonstrates the 4KB hotness overtime. MCF and PR have distinct memory access patterns, as the former solves a combinatorics optimization problem, while the latter calculates the degree of each graph vertex and scatter to neighbor graphs. In the plots, each block on the y-axis is a PPN of a 4KB range. The lighter the color is, the hotter the page is. As we can see the pages that are hot during a period can quickly become cold. Therefore, it is essential to detect this change as often as possible, by knowing the detailed hotness inside each 4KB range. The demoted page can either stay 4KB or be offloaded. However, even if the tracking and scanning processes are both efficient, the page migration overhead might still be large. We leave the evaluation of page migration overhead in the software for future work.

6 Conclusion

In this project, we surveyed various approaches to placing memory in a tiered memory system with multiple page sizes. We found that these approaches are limited either by efficiency, accuracy, or fairness when it comes to tracking the hotness of pages. To this end, we propose a preliminary design to solve this problem. We extended a simulator and used the extended simulator to conduct experiments to show the necessity of our design, focusing on the implication of parallel applications, an important application category in data centers. We also pointed out the possible inaccuracy or even mistakes in our evaluation, due to a flaw in our multi-core simulation design. In the future, in addition to fixing the issue in simulation, we aim to finalize the details design and implement it in both the architecture and operating systems.

7 Potpourri: Acknowledgment, Source Code, Reuse

Source code. This workload and evaluation code is hosted on `http://kevinrsx.github.io/memory-gapbs`. The GitHub repository for this project is available at `https://github.com/KevinRSX/memory-gapbs`. The source code of simulation used in this project is hosted internally at the CAOS Group at CMU. We plan not to release the code publicly until the project is complete, though the instructors are welcome to request them.

Acknowledgment. This project is part of ongoing research by Prof. Dimitrios Skarlatos. Kaiyang Zhao and Ziqi Wang implemented the initial version of the simulation infrastructure. Sum Yin Kwok implemented part of the backward communication. Prof. Skarlatos pointed out the flaws in multicore simulation.

Distribution of Credit. This is an individual project. Outside collaborators are acknowledged as above.

Project Reuse. A part of this project report is reused in my 15-740 final project report this semester. We believe the reuse is reasonable because of the amount of work done. The two projects have significant parts that do not overlap. Instructors are welcome to request a draft of another report.

References

[1] Compute express link. `https://www.computeexpresslink.org/`.

[2] Damon: Data access monitor. `https://www.kernel.org/doc/html/v5.17/vm/damon/index.html`.

[3] Intel® 64 and ia-32 architectures software developer manuals. `https://www.intel.com/content/`

www/us/en/developer/articles/technical/intel-sdm.html.

[4] Qemu, a generic and open source machine emulator and virtualizer. https://www.qemu.org/.

[5] Spec benchmarks and tools. https://www.spec.org/benchmarks.html.

[6] Structural simulation toolkit. http://sst-simulator.org/.

[7] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.

[8] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association.

[9] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[10] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[12] Aninda Manocha, Zi Yan, Esin Tureci, Juan Luis Aragón, David Nellans, and Margaret Martonosi. Architectural support for optimizing huge page selection within the os. In *Proceedings of the 56th International Symposium on Microarchitecture (MICRO)*. IEEE, 2023.

[13] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Alan Nair, Sandeep Kumar, Aravinda Prasad, Andy Rudoff, and Sreenivas Subramoney. Telescope: Telemetry at terabyte scale, 2023.

[15] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Amanda Raybuck, Wei Zhang, Kayvan Mansoorshahi, Aditya K. Kamath, Mattan Erez, and Simon Peter. Maxmem: Colocation and performance for big data applications on tiered main memory servers, 2023.

[18] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.