SAND2022-9655C

Automatic HBM Management: Models and Algorithms

Daniel DeLayo*
Stony Brook University
ddelayo@cs.stonybrook.edu

Kenny Zhang*
Stony Brook University
kzzhang@cs.stonybrook.edu

Kunal Agrawal
Washington University in St. Louis
kunal@wustl.edu

Michael A. Bender
Stony Brook University
bender@cs.stonybrook.edu

Jonathan W. Berry
Sandia National Laboratories
jberry@sandia.gov

Rathish Das
University of Waterloo
rathish.das@uwaterloo.ca

Benjamin Moseley
Carnegie Mellon University
moseleyb@andrew.cmu.edu

Cynthia A. Phillips
Sandia National Laboratories
caphill@sandia.gov

ABSTRACT

Some past and future supercomputer nodes incorporate High-Bandwidth Memory (HBM). Compared to standard DRAM, HBM has similar latency, higher bandwidth and lower capacity.

In this paper, we evaluate algorithms for managing High-Bandwidth Memory automatically. Previous work suggests that, in the worst case, performance is extremely sensitive to the policy for managing the channel to DRAM. Prior theory shows that a priority-based scheme (where there is a static strict priority-order among p threads for channel access) is $O(1)$-competitive, but FIFO is not, and in the worst case is $\Omega(p)$ competitive.

Following this theoretical guidance would be a disruptive change for vendors, who currently use FIFO variants in their DRAM-controller hardware. Our goal is to determine theoretically and empirically whether we can justify recommending investment in priority-based DRAM controller hardware.

In order to experiment with DRAM channel protocols, we chose a theoretical model, validated it against real hardware, and implemented a basic simulator. We corroborated the previous theoretical results for the model, conducted a parameter sweep while running our simulator on address traces from memory bandwidth-bound codes (GNU sort and TACO sparse matrix-vector product), and designed better channel-access algorithms.

In our simulations, we found two consistent results: (1) at low thread counts, when there is less competition for HBM, FIFO outperforms Priority by up to 37%. (2) at high thread counts, Priority outperforms FIFO by up to 3.3×.

We also generated artificial traces not based on bandwidth-bound code where FIFO's makespan was 40× larger than Priority, but thanks to Priority's provably good bounds, could not manufacture similarly bad ratios for Priority.

To mitigate (1), we designed new versions of Priority, called Dynamic Priority, that periodically shuffle the priorities of the threads. Choosing an appropriate reshuffling frequency removes an inherent "unfairness" in the original Priority approach: we can reduce the standard deviation of the response time for a DRAM request by an order of magnitude without increasing the makespan. This makes Dynamic Priority unambiguously better than both FIFO and Priority in all our simulations.

CCS CONCEPTS

• **Hardware** → **Emerging architectures**; • **General and reference** → **Experimentation**; *Empirical studies*; *Validation*; **General conference proceedings**; **Design**; **Metrics**; • **Computer systems organization** → **Multicore architectures**;

KEYWORDS

HBM; High-Bandwidth Memory; Far-Channel Arbitration; Queue management; Priority Queue; Scheduling; Memory Management; Memory Hierarchy; FCFS; First-Come-First-Serve; FIFO; First-In-First-Out; KNL; Knight's Landing; Xeon Phi; Sapphire Rapids; LRU; makespan; Dynamic Priority; Timeliness; Fairness

ACM Reference Format:
Daniel DeLayo, Kenny Zhang, Kunal Agrawal, Michael A. Bender, Jonathan W. Berry, Rathish Das, Benjamin Moseley, and Cynthia A. Phillips. 2022. Automatic HBM Management: Models and Algorithms. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490148.3538570

1 INTRODUCTION

To improve memory performance, a new hardware technology has been introduced known as ***high-bandwidth memory*** or ***HBM*** [34, 57].[1] HBM has higher bandwidth than DDR4 (today's DRAM technology) but similar latency. HBM's bandwidth is so high because it is placed directly onto the processor package (unlike DRAM). HBM thus augments the existing memory hierarchy by providing a memory that can be accessed with up to 5x higher bandwidth than DDR4 when feeding a CPU [1], and up to 20x higher bandwidth when feeding a GPU [59].

*co-first author

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07…$15.00
https://doi.org/10.1145/3490148.3538570

[1] From [24], "hardware vendors use various brand names such as High-Bandwidth Memory (HBM), Hybrid Memory Cube (HMC), and MCDRAM for this technology."

HBM is not a replacement for DRAM since it is generally about five times smaller than DRAM.[2] Moreover, HBM does not fit into the standard cache hierarchy because its latency is no better than DRAM's. (With standard "pyramid-shaped" hierarchies, the bandwidth and latency get better as the sizes get smaller.)

Central problem of HBM management. Any system that has HBM must decide what data gets evicted from HBM and in what order HBM fetches requests via a limited number of channels to DRAM. Least-recently-used (LRU) [54] and other traditional replacement policies [17, 18, 29, 54] have been used in conjunction with First-In-First-Out (FIFO) DRAM-access policies, such as First-Come-First-Served (FCFS) for multicores with traditional cache. This combination works well for regular caches but much worse for HBM [24]. We empirically observe LRU with FCFS performing poorly for HBM in this paper.

Machines that use HBM, such as Intel's Knights Landing [56], boot in multiple modes specifying how much of the HBM is system managed and how much is user managed. In ***cache mode*** the system controls HBM as a last level of cache, in ***flat mode*** the programmer explicitly copies data in and out of HBM, and in ***hybrid mode*** the HBM is split into a "flat" piece and a "cache" piece. Intel Knight's Landing has long been depcrecated [58]; However Intel's upcoming (2022 planned release) Sapphire Rapids [50] Xeon will also have HBM. Sapphire Rapids further adds ***HBM-only mode*** for systems without DRAM. Another variation (to which our model does not apply) exists and is being used in Deep Learning: HBM+NVram without DRAM [37].

The vision of this paper is that a system could automatically manage HBM as efficiently as a standard cache. This would free the programmer from explicitly managing HBM. In high-performance computing environments such as DOE supercomputers running scientific codes, system cache mode sometimes yields little if any advantage, but software is too complex and mature to be rewritten to manage HBM explicitly. Since HBM was first commercialized it hasn't been clear whether this is indeed even possible. The algorithmic plus empirical results in this paper provide strong optimistic evidence in support of provably good automatic HBM management.

Recent theoretical results offer hope for practical automatic HBM management. Das et al. [24] give a theoretical performance model of HBM, which captures the high (on-package) bandwidth between the cores and HBM and the lower (off-package) bandwidth to DRAM. It is unknown how well the theoretical abstraction introduced by Das et al. [24] predicts empirical performance, especially when constants are a first-order concern in real systems.

HBM+DRAM model. In an HBM, the channel(s) to DRAM (***far-channels***) are a sequential bottleneck [24]. The p cores can simultaneously send a memory request in each time step. Up to p requests can be fulfilled by the HBM in parallel, but only a small number q of requests can use the channel to DRAM at a time. Thus, if multiple cores simultaneously request memory from DRAM, the channel accesses need to be scheduled and serialized. We call this problem ***far-channel arbitration***. Waiting for access to DRAM dominates the running time over other considerations. For current systems, the number of DRAM channels per processor is typically not more than 8.

A natural performance objective is ***makespan*** which, given a batch of running processes, is the time when the last process completes. Minimizing cache misses is not the same as minimizing makespan, and can be far from it [24, 43]. In fact, a workload could have few cache misses, but because it does not take advantage of the parallelism between cache and HBM, it has a poor makespan.

1.1 Two Components of HBM Management

A system-controlled HBM has two algorithmic policies to set:

- **HBM replacement policy.** When the system brings a new block from DRAM to HBM, it first needs to decide which block to evict from HBM.
- **Queuing policy for far-channel arbitration.** Multiple requests (up to p disjoint requests: one request per core) for blocks on DRAM can occur simultaneously, but at most q requests ($1 \leq q << p$) can be fulfilled per time step. The system must decide in each time step which of the outstanding block requests to fulfill (and which ones to keep in the queue).

HBM replacement is not the problem. The traditional way we think about replacement policies (Least Recently Used [54], or LRU) turns out also to work with HBM. That is, LRU can also be used in the context of HBM management to help obtain theoretically good performance guarantees [24]. In actual implementations of HBM as a last level of cache, LRU is not the replacement policy, since when HBM is used in cache mode, it has limited associativity or is even direct mapped. However, we show that the same good theoretical guarantees from [24] can be retained, given certain assumptions on the mapping from DRAM addresses to locations in HBM (see §2). The bottom line is that LRU and variants work asymptotically well, not only in regular caches, but also in HBM.

FIFO queue for far-channel arbitration. In contrast, a natural and intuitive policy for serializing the outstanding requests to DRAM—simply to queue them up in First-In-First-Out (FIFO) order—is provably bad. Moreover, in this paper, we show that workloads that are bad for FIFO queue management are easy to generate and, based on how we generate them, we expect to see similar workloads commonly in practice. We find it unsurprising that traditional replacement policies (e.g., LRU [54]) continue to work with HBM, but we find it surprising that FIFO can perform poorly given its prevalence. Intel has previously used a FIFO variant called "adaptive-open-page-policy," and much of the literature [32, 38] focuses on optimizations to the basic FCFS policy. We believe that our results justify cycle-accurate simulations that might influence hardware vendors to consider the disruptive change of modifying far-channel arbitration policies.

Far-channel arbitration is the problem. How to determine which pages to transfer in each time step (and which to delay) is a new algorithmic challenge not faced in traditional caching, but critical to HBM management.

This algorithmic challenge can be restated as follows: how to partition the pages of the HBM among all processes and then change this allocation dynamically in each time step. This is because if in

[2] From [24], this is due to constraints such as heat dissipation, as well as economic factors.

each timestep we have control over which process's page is brought into HBM, and which process's page is ejected, then we are exactly determining the partitioning of HBM among the processes.

The problem with the FIFO policy for serializing DRAM access is that it tends to have the effect of spreading out HBM evenly and thinly among all the processes. The HBM becomes too "stretched, like butter scraped over too much bread."

At any time step, a good partitioning of HBM may allocate HBM space to processes unevenly. Some of the processes may be assigned a zero fraction of space in HBM, momentarily starving them in order to give enough capacity of the HBM to other processes so that they do not thrash. In principle, good parallelism means having as many processes running and as few starved as possible. But if the HBM partition gives too tiny a sliver of the HBM to each process, then no process significantly benefits from HBM, and the bottleneck becomes the channel to DRAM. But determining how many processes to run and exactly how to divide the HBM, is extremely sensitive to the processes's request streams and a good solution changes in each time step.

Priority queue for far-channel arbitration. An alternative policy called ***Priority*** was recently proposed by Das et al. [24]. In this scheme, each thread is assigned a fixed priority. These priorities effectively determine at each step, which thread gets to use the DRAM channel. Specifically a page request from a high–priority thread always takes precedence over a page request from a lower-priority thread, regardless of which page requests was made first.

What is interesting about this priority-based scheme is how apparently unfair it is. Low-priority threads can get delayed by higher-priority threads—but the reverse does not happen. Despite the pecking order among the threads and this seeming unfairness, Priority's makespan is within $O(1)$ times the optimal makespan. This is a major improvement over FIFO's $\Omega(p)$ worst case.

The Priority scheme has a good makespan because the priority scheme naturally does a good job of partitioning the HBM among the threads. If there is not enough space in the HBM to satisfy all of the threads, then the lower-priority threads starve until the higher-priority threads do not need as much as space. As the priorities are allowed to change over time, randomly permuting the priorities can mitigate excess starvation.

In this paper, we show that for large processor counts, Priority is favorable over FIFO queue management. For small processor counts, Priority does as well as FIFO or better when periodically randomly permuting the priorities. This also has the benefit of mitigating some of the unfairness inherent in a Priority scheme. We show this on workloads that are based on common memory-bandwidth bound computation kernels.

1.2 Results

In this paper, we evaluate priority-based methods for managing High-Bandwidth Memory automatically.

Explanation of Knight's Landings performance using the HBM+DRAM Model. We validate the HBM+DRAM model Knight's Landing (KNL) [56], an example system with HBM, has a performance profile explained by HBM+DRAM model. We develop and run a series of microbenchmarks on KNL in its different modes. In our measurements HBM has a similar but slower access latency than DRAM by 24ns (roughly 10 percent outside of shared L2), HBM has a higher bandwidth by about 4.8×, and accessing HBM in cache mode can incur an extra latency cost.

We run the benchmarks in Flat mode HBM, Flat mode DDR, and Cache Mode. We use the srGUPS microbenchmark (see §5) to measure bandwidth and we chase pointers to measure latency.

In order to access DRAM in Cache Mode, we must first cross the mesh and miss shared L2, then cross the mesh again and miss in HBM. Thus, this third mesh crossing adds a 50% overall latency penalty, but a 100% latency penalty when just considering the time to access HBM.

The latency penalty on a cache miss is roughly the time it takes to traverse the mesh on KNL systems. This causes a 1.5× change in overall latency, but a 2× change in latency when discounting the initial mesh search across shared L2.

Simulation and Extension to Multiple Channels. We built a simulator in C++ using the HBM+DRAM model. We instrumented two memory-bandwidth-bound applications, TACO Sparse Matrix-Matrix Multiplication [23, 40] and GNU sort [53], to obtain page-access sequences for use as simulator workloads. In our instrumentation we used several techniques, such as overloading C++ operators, to log memory accesses.

In our simulations, we varied the size of HBM, the source of the access traces (GNU sort, quicksort, Sparse and Dense Matrix Multiplication), the number of cores, the distribution of work across the cores, the method by which we permute priorities (none, cycle, cycle-reverse, interleave, Dynamic Priority), how often we remapped priorities (some parameter times the HBM size), the number of channels to DRAM (1-10), and whether the DRAM queue is FIFO or Priority. In this paper, we present an interesting subset of them in depth and briefly present our results for several others.

Evaluation of FIFO versus Priority. We compare FIFO and Priority's makespan when running the TACO Sparse Matrix-Matrix Multiplication and GNU sort workloads.

In our instrumented traces, at low thread counts where HBM is plentiful, Priority gives a worse makespan than FIFO by up to 37%. When the number of threads increases and HBM becomes more scarce, FIFO gives up to a 3.3× worse makespan than Priority. We also design a request sequence to be bad for FIFO, hold the amount of memory per core constant, and get a linearly worse makespan. We observe up to 40× worse makespan. Because Priority is provably good, we cannot create a bad request sequence for it.

Demonstration that periodically changing priorities fully eliminates FIFO's advantage. We propose Dynamic Priority, which retains the theoretical guarantees of Priority. Unlike Priority, Dynamic Priority is either as good as FIFO or outperforms FIFO in all of our simulations. We also find that Priority suffers from having highly variable response times, where the ***response time*** of a page request is the duration between sending the page to the DRAM queue and the page being served. Dynamic Priority reduces this variance by occasionally permuting the priorities. Changing priority more often decreases the standard deviation of response times but may increase the makespan. We characterize this tradeoff and, for our experiments, identify a broad range of

parameters where variance is small and the makespan is as good as or better than both Priority and FIFO.

1.3 Related work

HBM-tuning and cache mode. Our HBM model is consistent with the behavior of the Multi-Channel DRAM (MCDRAM) in Intel's Knights Landing (KNL) processor [34]. In KNL, arbitration of HBM misses is handled by the DRAM controller. Although the actual protocol is proprietary, it is likely a solution based on [49]. Such arbitration is commonly called "first-ready first-come-first-served (FR-FCFS)." As the name implies, this is a variant of FCFS.

The first algorithmic work for MCDRAM, done before KNL existed, used a simulator to predict speedup for a flat-mode sorting algorithm. [13, 14]. This was validated on KNL by Butcher, et al. [20]. Several recent papers have documented runtime improvements of 3-4x using KNL when problem instances fit entirely in the MCDRAM. For example, Li et al. studied kernels from scientific computing [42] such as sparse matrix-vector multiplication. Byun, et al. observe KNL speedup for dense matrix-matrix multiplication [21]. Laghari, et al. designed flat-mode algorithms for computational kernels such as STREAM on KNL [41]. Slota and Rajamanickam [55] obtained 2-5x speedups for graph algorithm instances larger than HBM.

The above work gives HBM-aware algorithms for structured kernels. However, many scientific workflows are too complex to be completely rewritten using such kernels [20]. Das, et al., offer theory predicting computational speedups on HBM systems by changing the DRAM controller design. [24].

Hierarchical memory models. There are several hierarchical memory models for both sequential and parallel settings [2–4, 8–11, 15, 16, 22, 25, 35] including models with private caches. In our HBM model, we do not include private caches. HBM does not fit well with standard hierarchical memory models [13, 31] because DRAM and HBM have about the same latency.

Feuerstein and Strejilevich de Loma [28] work on a multi-threaded paging problem. Their model has only one core, and they interleave request sequences of multiple threads into a single thread to minimize the number of cache misses. Loma [26] and Seiden [51] give randomized algorithms for the same setting as described by Feuerstein and Strejilevich de Loma.

Paging in multicore systems is investigated throughly [5, 33, 39, 43]. Hassidim [33] introduces a paging model where p cores share a cache. His objective is to minimize the maximum running time of all the processors. López-Ortiz and Salinger [43] use Hassidim's model, but they minimize the total number of cache misses incurred by all the cores instead of the running time. They present several dynamic programming algorithms; however, their running times are exponential in model parameters.

Katti and Ramachandran [39] work on a constrained model where the interleaving of the processors is given as part of the input. They present competitive online algorithms in their restricted model. Very recently, Agrawal et al. [5–7] give $O(\log p)$-competitive online algorithm in the general parallel paging model.

Although related, these models are incomparable to the Das et al. HBM model since latencies differ by level.

HBM hierarchies by application: deep learning and scientific computing. Recent work in large-scale deep learning leverages multi-level memory hierarchies involving HBM/NVRAM. [12]. Stochastic gradient descent computations in HBM withing GPU's can be adequately fed from NVRAM. of such hierarchies have vastly different latencies, so the model of Das, et al. does not apply. However, scientific computing requires more interaction between memory levels. Therefore, HBM/DRAM hierarchies persist, as represented by Intel Sapphire Rapids. The latter has huge amounts of HBM bandwidth. Under certain expected configurations, Sapphire Rapids could have 3.68 TB/s of peak memory bandwidth with 128GB of HBM [52].

2 HBM MODEL AND MANAGEMENT

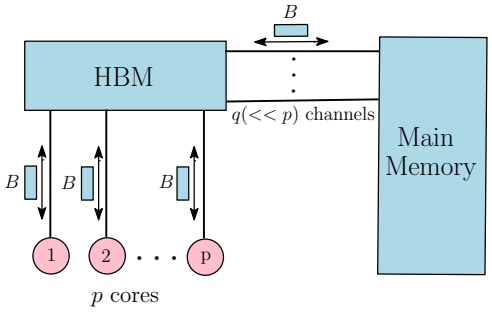


Figure 1: The HBM model with p cores.

In this section, we describe the theoretical model of HBM on which our simulator is based. This model slightly generalizes the model from [24]—the primary difference is that we consider multiple channels from HBM to memory while the prior model only considered a single channel.

HBM Model. The model consists of p cores connected to the HBM of size k blocks by p parallel channels. The HBM is connected by $1 \leq q << p$ channels to DRAM of unbounded size. Data is transferred in blocks of size B both from DRAM to HBM and from HBM to the cores. Thus, the size of HBM is $k \cdot B$. We model the increased bandwidth of HBM as p parallel channels between HBM and the p cores. However, due to the bandwidth bottleneck of DRAM, at most q blocks can be transferred in parallel along the channel to DRAM. The similar block-transfer time from HBM to the cores and from DRAM to HBM is captured by setting all block-transfer times to 1.

In reality, instead of a 'DRAM-HBM' far-channel as depicted in Figure 1, there is a mesh connecting the cores to both DRAM and HBM. When the HBM is used as a cache, a core's memory access is first directed to the HBM. On an HBM miss, the memory access then goes to DRAM. Modeling this behavior as a separate channel between HBM and DRAM is both algorithmically clean and predictive (see §5).

Each core p_i requests a sequence of blocks $R^i = r^i_0, r^i_1, r^i_2, \ldots$ on its dedicated channel to HBM. Each core's requests are disjoint; that is, satisfying a request on sequence p_i does not progress any other sequence. Core p_i requests blocks in the order of sequence R^i and does not request the next block until the previous block in the sequence is served to the core. If core p_i requests block r^i_j at time step t and if r^i_j is in the HBM (this is an ***HBM hit***), then at the

next time step $t + 1$, the requested block is transferred to core p_i. If the requested block is not in HBM (this is an ***HBM miss***), then the block must be transferred from DRAM to HBM and then from HBM to core p_i. This takes at least two time steps but may take arbitrarily longer if the request from HBM to DRAM must wait to get access to (one of the q) DRAM channels.

HBM Management Policies. The HBM-management algorithm must consider two resources: (1) q ***far channels*** between HBM and DRAM; and (2) the k blocks within the HBM (see §1.1).

A far-channel arbitration policy decides which (of the potentially many) waiting requests are served using the q channels between HBM and DRAM. When there are more than q outstanding requests for blocks that are not in HBM, all of them cannot be fetched from DRAM to HBM in parallel due to limited bandwidth between DRAM and HBM. The block requests are kept in a queue called the ***request queue***. Blocks are fetched from the DRAM in the queue order determined by the far-channel arbitration policy and up to q blocks can be served in parallel.

A block-replacement policy decides which blocks stay in HBM and which blocks are evicted when HBM is full and new blocks are brought from DRAM to HBM. This is analogous to cache replacement. Several block-replacement strategies such as LRU, FIFO, CLOCK [36] have been proposed in the caching literature. For cache management, LRU guarantees constant-competitive performance with the optimal given constant-factor resource augmentation [54].

Makespan as our performance metric. On a single core with normal cache, the metrics of ***makespan*** (maximum completion time) and ***number of cache misses*** are closely aligned. Approximately minimizing the number of cache misses would approximately minimize the makespan. However, this correlation does not extend to HBM. López-Ortiz and Salinger [43] and Das et al. [24] show that the number of HBM misses is the wrong objective, since different policies that have the same number of misses can have wildly different makespans. Instead, they argue that we should directly optimize makespan. Formally, given an HBM of size k blocks and p disjoint request sequences to the p cores, the objective is to find a far-channel arbitration policy (for the HBM-DRAM channel) and a block-replacement policy for the HBM to minimize makespan.

Theoretical Results for Automatic HBM management. Das et al. [24] analyzed HBM-management policies under this model for the special case where $q = 1$, that is, only one channel from HBM to DRAM. Their results indicate that HBM management is fundamentally different from Ideal-Cache management [30, 31] which is only concerned with the block-replacement policy. They show that designing a far-channel arbitration policy is fundamental for designing good automatic HBM-replacement algorithms: Combining the natural far-channel arbitration policy of First Come First Serve with the natural block replacement policy of LRU is terrible theoretically. On the other hand, a priority-based policy (also combined with LRU) performs well. This policy arbitrarily assigns a pecking order on the cores and always satisfies requests from high-priority cores before lower-priority cores. Therefore, for the same block-replacement policy (LRU), the far-channel arbitration policy makes all the difference in theoretical performance.

The performance of the two channel-arbitration policies Priority and FCFS with the same block-replacement policy LRU are:

Theorem 1 ([24] Performance of Priority for $q = 1$). *Priority is $O(1)$-competitive for the makespan-minimization problem (even without any memory-augmentation).*

Theorem 2 ([24] Performance of FCFS for $q = 1$). *There exists p block request sequences such that even with d memory augmentation and s bandwidth augmentation the makespan of FCFS+LRU is $\Theta(\frac{p}{ds})$-factor away from that of the optimal policy.*

Therefore, in the worst case, FCFS+LRU can be factor of $\Omega(p)$ worse than Priority+LRU.

Extension to multiple channels between HBM and DRAM. We now present a relatively straightforward extension from one channel to q channels between HBM and DRAM and present a $O(q)$-competitive online algorithm for the generalized HBM model.

Theorem 3. *If there are q channels between DRAM and HBM, then Priority achieves an $O(q)$-competitive ratio for the makespan-minimization problem (even without any memory-augmentation).*

Generalizing fully-associative HBM results to direct-mapped implementations. Prior HBM results applied to fully-associative caches [14, 24]. However, practical implementations of HBM are usually direct mapped [50, 56]. We now explain how to take a program designed for a fully-associative HBM with LRU (or optimal) replacement and automatically transform it into another program that runs asymptotically as fast on a direct-mapped cache. This direct-mapped cache only need be a constant-factor larger. From this transformation plus resource augmentation, we conclude that the scheduling asymptotics for HBM are the same on a fully-associative cache and a direct-mapped cache.

Frigo et al. [30, 31] and Prokop [48] show that a fully-associative sequential cache of size M with LRU replacement can be simulated with a direct-mapped cache of size $O(M)$. We give a similar result for HBMs—the main difference is that in a sequential cache, at most one page is accessed at a time while multiple pages may be accessed concurrently in an HBM with p channels to the cores.

Lemma 1. *There exists an automatic transformation from a program running on a size-k fully-associative HBM with LRU or FIFO replacement to another program that simulates these policies on direct-mapped cache of size $\Theta(k)$ using a constant factor more bandwidth from HBM to DRAM. Specifically, assuming that the direct mapped cache has $O(1)$ more HBM and bandwidth, (1) each HBM hit in the original program causes $O(1)$ hits and no misses in the transformed program (in expectation) and (2) each miss in the original program causes $O(1)$ misses in the transformed program (in expectation).*

Proof. We will use two data structures, similar to those used by Frigo [30, 31, 48]. The first data structure is a hash table (with chaining to resolve collisions) which allows us to simulate full-associativity. The other is a doubly-linked list which allows us to simulate LRU or FIFO. The transformation is essentially identical to the one described by Frigo; we describe it here for completeness.

The HBM is divided into two $\Theta(k)$ regions — one for maintaining the meta-data (hash-table and linked list) and another for keeping the actual pages from the program (program data). All manipulation

is done by accessing DRAM addresses and changing the data stored in these addresses. We first reverse the mapping from direct-mapped HBM and designate a single location in DRAM whose data will be stored in each location in the HBM — that is, we have a bijection between the direct-mapped HBM and DRAM.

The hash table is a size k hash table which maps each block that the program might access in DRAM to some location in the program data region of the HBM — we call it the Cache DRAM address (even though it might not always be cached). That is, each HashTable key is a user-supplied DRAM address. Each HashTable value contains a DRAM address which is part of the above mentioned bijection. By using a 2-universal family of hash functions [45], one can ensure that if we have k blocks in HBM at a time, then the chain length is $O(1)$ in expectation.

We will use this hash-table to simulate full-associativity as follows: When the program accesses a user-supplied DRAM address, we search the hash-table in $O(1)$ time to see if the page is in HBM. If not, we copy over data from the user-supplied DRAM address to the corresponding Cache DRAM address and then bring it into HBM. When evicting a page, we copy data from the Cache DRAM address to the user-supplied DRAM address. If the page is found in the hash table, we can then access this page by accessing the corresponding Cache DRAM address which is cached within HBM.

To pair the hash-table and linked list, each hash-table node points to a linked-list node; each linked-list node also has a corresponding back pointer. The linked list is ordered based on eviction policy. In FIFO, the front of the linked list is the node corresponding to the first-in page and the back is the node corresponding to the last-in page. In LRU, the front of of the linked list is the LRU page and the back of the linked list is the MRU page.

When we encounter an HBM miss, the page at the front of the linked list is evicted, or removed from the hash table and from the linked list. The data is then copied back from the Cache DRAM address to the original DRAM address. At this point, the user-supplied DRAM address is copied to the corresponding Cache DRAM address of this page, brought into HBM, and inserted into the hash table and into the back of the linked list. All this causes at most a constant number of pages to be brought into HBM. On an HBM hit, no new pages are brought into HBM. □

Theorem 4. *The makespan of the transformed program running on the direct-mapped cache is at most $O(\log q)$ factor larger than the original program running on a fully associative cache with FIFO and at most $O(\log p)$ larger than the original program running on a fully associative cache with LRU, where q is the number of channels from DRAM to HBM and p is the number of processors. If q is a constant, as in the original HBM model, then the direct-mapped cache is asymptotically equivalent to FIFO.*

Proof. The above transformation works for both FIFO and LRU. For simulating FIFO, the linked list is only modified on an HBM miss—since at most q blocks can be transferred from DRAM to HBM on each time step, we have to add up to q blocks to the front of the linked list on any one step. However, LRU order changes on HBM hits as well. Therefore, if p processors access p pages which are all within HBM, then p corresponding blocks must move to the head of the linked list in one time step.

We must move x items concurrently to the front of the linked list. $x \leq q$ for FIFO and $x \leq p$ for LRU. There are two components: remove x items concurrently from the linked list and insert the x items to the front of the linked list. Removal is easier—we mark the items removed without physically removing them and periodically run garbage collection to physically remove the items. As we never traverse this linked list to find an item, we allow it to get large as long as it fits in HBM.[3]

We keep a q (correspondingly p) element auxiliary array. At a high-level, we wish to ensure all x processors can write their item into a *different* location of this array. If we can achieve this, then we can create a "mini" linked list of these x items in $O(1)$ time (each item can link itself to the item before and after itself concurrently). This mini linked list can be linked to the front of our master linked list in $O(1)$ time.

The remaining problem is to ensure that each item can be written at a unique location in this auxiliary array concurrently. This can be achieved in $O(\log q)$ (correspondingly $O(\log p)$) time using prefix-sums. In brief, each processor must get a unique number between 1 and x (which is equivalent to updating a shared counter) in parallel and then write their element in the location they get by updating this shared counter—prefix-sums is exactly designed to perform this operation.[4] Therefore, each core will write its item in a unique location in the auxiliary array in $O(1)$ time, link itself to its neighbors in $O(1)$ time, and then the mini list is linked to the original list in $O(1)$ time. □

The following corollary follows since FIFO can be used as a replacement policy instead of LRU in the original HBM proof from [24] without changing the competitive ratio asymptotically.

Corollary 1. *One can achieve $O(1)$-competitive makespan with a direct-mapped HBM versus a fully-associative HBM, when $q = O(1)$.*

3 SIMULATING HBM AS CACHE

We built a simulator of the HBM Model from §2 to understand how Priority and FIFO behave in typical cases and how their constants compare. Following the model:

Property 1. *The sets of pages accessed by each core are mutually exclusive.*

Property 2. *We track page references and ignore computation. Thus, any advantage in minimizing makespan has meaning only when the code being simulated is memory-bandwidth bound.*

Property 3. *HBM is fully associative.*

These properties are notably different from KNL hardware. We intend not to understand KNL's implementation but the constants and performance predicted by the model.

We argue that these properties are reasonable. Property 1 means that we do not simulate true parallel programs—we argue that p cores sharing a common HBM and processing reference streams

[3] In principle, it can get even larger since the unaccessed items will logically move out of HBM and never be accessed. We need only keep the items representing the pages currently in HBM—that is, the logically un-removed items—and their neighbours. That is, at most $O(k)$ items of the linked list have to be in cache even if the linked list is longer than $O(k)$.

[4] Some theoretical models assume an $O(1)$-time fetch-and-add (FAA) hardware operation. With such an operation, this update can be done in constant time.

from the same serial code is a reasonable surrogate for multi-threaded execution. Our two test codes, sorting and sparse matrix multiplication (SpGEMM), are both memory-bandwidth bound, satisfying Property 2. Corollary 1 shows that Property 3 is reasonable.

3.1 Simulating HBM

We simulate FCFS and priority-based[24] management of HBM under LRU with a simple C++ program that ingests address traces from serial runs of annotated code. In a preprocessing step, each array dereference in the annotated code is mapped to its page reference. The resulting sequence of page references forms the input to our simulator.

Simulation. All cores share a single HBM of size k slots, each of which can hold a single page. The simulation operates on ticks. We define r_*^i as the currently requested page in R^i. When processor i is served the page it requests, r_*^i, then on the next tick processor i will request the next page in its queue, changing r_*^i. Let t be the current tick, on which the following occurs:

(1) If t is a multiple of the remap period T, remap the priorities. Increase t by 1.
(2) For each r_*^i, if r_*^i is not resident in HBM, add r_*^i to the DRAM request queue.
(3) If there are more requests in queue then empty slots in HBM, evict up to q pages by LRU.
(4) For each r_*^i, if r_*^i is resident in HBM, serve r_*^i to processor i.
(5) Retrieve up to the next q pages in the DRAM request queue from DRAM into HBM. Remove these pages from the queue.

3.2 Generating Data

We run 1 independent run of a program per processor to generate p independent ***access traces***. These traces form a ***workload***; in our workloads, we assume that all processors are working on sufficiently similar tasks and each trace is generated from the same program with different randomness. Datasets 1 and 2 are both from memory-bandwidth bound applications, and they're therefore amenable to HBM. Dataset 3 is designed to stress FIFO.

Dataset 1: Sorting. We generate GNU sort [53] memory access traces by running GNU sort on randomly generated sequences of 500,000 integers. Since GNU sort takes iterators as input, we created a logging iterator class that logs every dereference to a file, and passed these logging iterators to GNU sort.

Since sorting is perhaps the most ubiquitous computing kernel of all, any advantage is of interest.

Dataset 2: Sparse Matrix Matrix Multiplication. Our SpGEMM code is based on TACO Sparse Matrix-Matrix Multiplication [23, 40]. We replaced the arrays used in this code with our own array-like objects that log all accesses to a file. We generate the access traces by running this modified version on two sparse matrices of size 600 by 600 where approximately 10% of the elements exist. These elements are randomly generated.

SpGEMM is the cornerstone of many computations in scientific computing and data science, and has been shown to benefit from many-core parallelism of 200 cores and beyond [19, 27].

Dataset 3: Traces designed to be bad for FIFO. FIFO performs asymptotically poorly when run on a long sequence of unique pages, repeated many times. We generate the sequence $1, 2, 3 \ldots 256$ and repeat it 100 times. FIFO performs poorly on this sequence when there is insufficient memory to keep everything paged in. See §4.

While this sequence is specifically designed to make FIFO look bad, it is still a simple sequence that generalizes nicely to everyday usage. For example, this trace could be generated by a program that needs more memory than the working set size to perform well. If it has less memory than the working set size, it starts to thrash.

4 MODEL SIMULATION RESULTS

We simulate FIFO, Priority, and Dynamic Priority and establish:

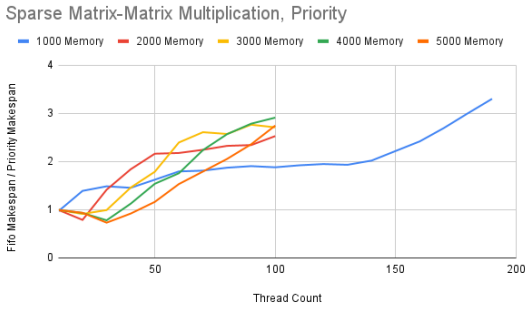
- Priority is generally better than FIFO in terms of Makespan, often much better (3.3×), except at small thread counts, where Priority can be slightly worse(1.37×).
- Dynamic Priority gives the same or better Makespan than both FIFO and Priority for all tested workloads.
- Priority may end up starving threads for long periods of time (metric formalized later). Dynamic Priority starves threads for much shorter periods of time.
- Dynamic Priority, by changing thread priorities more or less often, can increase or decrease thread starvation at the cost of additional overhead in Makespan, but a large range of values lead to essentially the same Makespan.

Simulation and explanation of FIFO's poor performance. We now show the results of our simulation of FIFO and Priority on Dataset 3 (see §3.2), where FIFO performs asymptotically worse than Priority. We plot the results in Figure 3; FIFO yields a 40× worse makespan that linearly scales with thread count. To make FIFO fail so catastrophically, the HBM size k is set to have enough memory to fit only $\frac{1}{4}$ of all the unique pages across all the threads.

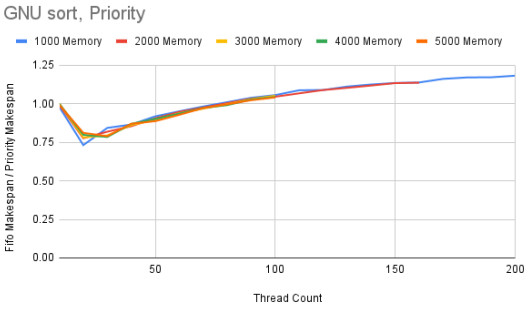
When running on FIFO, we never have a cache hit—by the time a thread repeats a page in its sequence, the page has been long evicted. In contrast, Priority will have a much higher cache hit rate. The first two threads (as one thread cannot saturate the channel in the HBM+DRAM model) load all of their pages into HBM. They complete their work quickly while the next two threads load pages into HBM. This repeats until all of HBM is filled up, at which point some least recently used page is evicted. When a higher priority thread runs into a cache miss due to this eviction, the lowest priority thread stops making progress instead of continuing to sabotage other threads. Generalizing this examples makes it clear why FIFO performs much worse than Priority when HBM is especially sparse.

We show similar but less extreme cases where FIFO performs badly on Datasets 1 and 2 (see §3.2). Figure 2 shows FIFO vs Priority on one instance of sparse matrix-matrix multiplication and one instance of GNU sort. In Figure 2a, we see that FIFO can give a makespan of up to 3.3× as large as Priority for large thread counts between 50 and 200. Similarly, for GNU sort in Figure 2b, FIFO gives a 1.2× larger makespan at high thread counts.

When and why FIFO outperforms Priority for Makespan. There are cases where Priority yields a worse makespan than FIFO. In Figure 2a, Priority yields a slightly larger makespan up to 1.33× as large as FIFO. For GNU sort in Figure 2b, Priority gives a 1.37× larger makespan at low thread counts.



(a) SpGEMM, 600 × 600, 90% sparsity. The SpGEMM results are particularly promising since SpGEMM has been shown in the literature to scale beyond 100 cores.



(b) GNU sort of 500,000 integers.

Figure 2: Simulation results for priority vs. FIFO, with HBM sizes ranging from 1000 to 5000 slots. The y-axis shows the ratio of FIFO's makespan to priority's makespan. Values greater than 1.0 show an advantage for priority. In both cases, FIFO can dominate at low processor counts but priority always dominates at high processor counts.

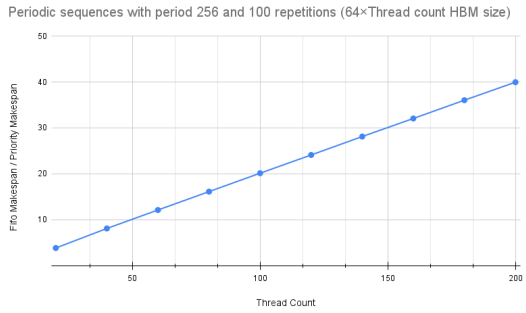


Figure 3: FIFO vs Priority for 100 repetitions of the sequence $1, 2, 3 \ldots 256$, but only $\frac{1}{4}$ of the memory required to fit every page in HBM. FIFO misses every page and Priority starves threads. FIFO yields a higher makespan by as much as 40×.

For these low thread count results, HBM is plentiful—all the threads can run simultaneously without thrashing. Since all the workloads are running the same problem of approximately the same length and characteristics, it is natural for all the threads to be worked on at approximately the same speed to end at the same time and minimize Makespan. When running Priority, we instead work on some threads slowly and some threads quickly, causing some threads to be left behind. The issue isn't utilization, but an artifact of this inconsistency when running balanced workloads. Dynamic Priority aims to reduce this inconsistency, which we quantify below. We also discuss Priority's performance on a different metric (that does not have such an artifact) at the end of this section.

Quantifying thread starvation. To capture some notion of thread starvation, we define 'response time' and 'inconsistency' as follows.

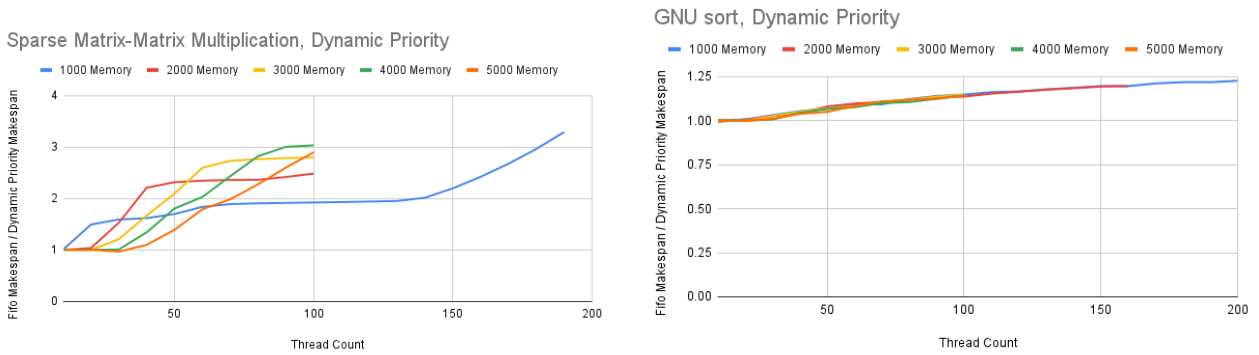
Let $r_0^i, r_1^i, r_2^i, \ldots$ be the page reference sequence running on thread p_i. Let the ***response time*** w_j^i of any page reference r_j^i be the number of simulation ticks between when the page is requested and when the page is serviced. For an HBM cache hit, $w_j^i = 1$, since it takes one tick to transfer the page from HBM to the thread. For an HBM cache miss, $w_j^i \geq 2$, depending on when it is served by the DRAM request queue. It takes one tick to transfer the page from DRAM to HBM, and one more tick to transfer the page from HBM to the thread (see §2). We define ***inconsistency*** to be the standard deviation of w_j^i over all i, j.

Thread starvation occurs when some threads have disproportionately higher response times than other threads. In FIFO, no threads are starved, and both response time and inconsistency are $O(p)$. In Priority, low-priority threads are starved. Their initial page requests are blocked by higher-priority threads, causing some requests to have large (possibly unbounded) response time and inconsistency. Thus, the inconsistency for Priority is high.

Changing priorities gives a better makespan and response time than both FIFO and Priority. By changing priorities, we gain two main advantages: threads are starved less often and the mixed results from Figure 2 become unambiguously positive. We present Dynamic Priority, a scheme which randomly permutes the priorities of the threads every fixed interval T. We also consider Cycle Priority, a deterministic scheme which cycles priorities on these same intervals. These schemes are based on the observation in [24] that priorities can be periodically re-assigned without violating the theoretical bounds if the interval is longer than the size of HBM ($T \geq k$). Thus, we talk about T as a multiple of k. We define P as the set of processing thread ids and k as the HBM size.

We consider Cycle Priority as it has practical advantages over Dynamic Priority: ease of implementation and ability to trivially bound the response time. Coordinating and generating the shared randomness required to implement Dynamic Priority in hardware may not be desired, especially if a simpler and easier to implement scheme suffices. Cycle Priority does not need shared randomness; it only requires processors to agree when to change priority.

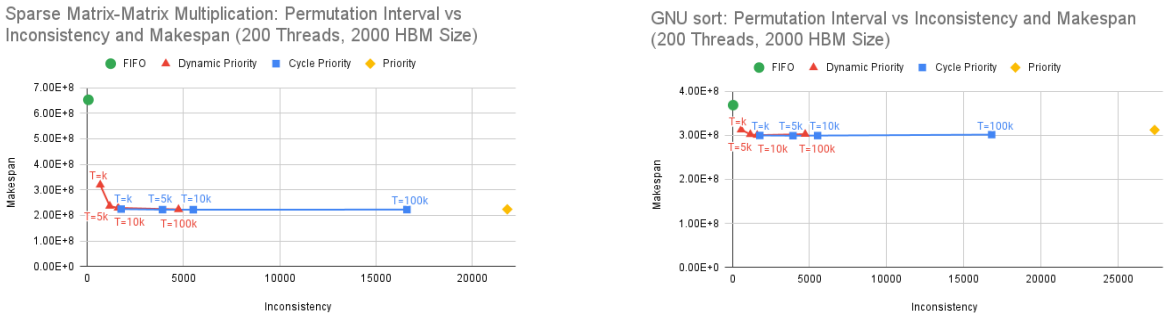
There is also a trivial upper bound on the response time of a page reference. A thread is guaranteed to become the highest priority thread within p priority permutations. We therefore bound the longest a page reference can wait in the request queue by $p \cdot T$. The bound on inconsistency follows from this bound on response time.



(a) SpGEMM, 600 × 600, 90% sparsity.

(b) GNU sort of 500,000 integers.

Figure 4: Simulation results for Dynamic Priority versus FIFO, with HBM sizes ranging from 1000 to 5000 slots. Randomized remapping has mitigated any advantages that FIFO held in Figure 2. The results for deterministic remapping are similar for balanced workloads.



(a)

(b)

Figure 5: Effect of scheme and T on inconsistency. Starting at the bottom right of each subfigure, find the Priority point (yellow). This is our point of maximum performance and maximum inconsistency. Moving to the left, we show the priority permutation strategies for decreasing permutation interval. Most of the inconsistency can be removed with minimal loss in performance.

Definition 1.
π: $P \to P$. *A permutation mapping thread ids to priorities.*
Priority: *π is always the identity permutation.* $\pi(i) = i$.
Dynamic Priority: *replace π with random permutation π'.*
Cycle Priority: *replace π with π'.* $\pi'(i) = (\pi(i) + 1) \mod |P|$.

Figure 4 shows the affect of randomizing priorities every $10 \cdot k$ ticks. At low thread counts, where Priority previously lost to FIFO, Dynamic Priority either performs as well as FIFO or outperforms FIFO on Makespan. At high thread counts, Dynamic Priority performs as well as or better than Priority and FIFO. For balanced workloads (where every thread has a comparable task), Cycle Priority also performs similarly to Dynamic Priority and may be simpler to implement in hardware. When the work is asymmetric, Cycle Priority continuously places the same thread behind the most demanding thread, causing small amounts of starvation. This can likely be mitigated on sufficiently long sequences by instead cycling through all permutations or shuffling, but would be more complex to implement. Even though Cycle Priority does not violate the theoretical guarantees of [24], we believe Dynamic Priority to be a more robust scheme.

Dynamic Priority reduces starvation and maintains Makespan. We empirically show that periodically permuting priorities (such as in Dynamic Priority or Cycle Priority) gives us significantly lower inconsistency than FIFO and as good or better makespan than Priority. We plot the inconsistency and makespan for various ***permutation intervals*** T of FIFO, Priority, Dynamic Priority, and Cycle Priority in Figure 5. As $T \to \infty$, Dynamic Priority approaches Priority. As $T \to 1$, Dynamic Priority approaches purely random selection, which has the same expected waiting time in the DRAM queue for each thread as FIFO.

For both Figure 5a (sparse matrix-matrix multiplication) and Figure 5b (GNU sort), FIFO has the highest makespan and Priority has the highest inconsistency. For T in the parameter range $10k$ to $100k$ for Dynamic Priority and $5k$ to $100k$ for Cycle Priority, the makespan is similar to Priority but the inconsistency is much lower. However, when Dynamic Priority gets too small (less than $10k$), the makespan increase as T decreases. Therefore, there is a large

Table 1: FIFO has lowest inconsistency and highest average response time. Priority has highest inconsistency and lowest average response time. More frequent permutation decreases Priority's inconsistency and increases its average response time. This differs from the results using makespan, where Priority's makespan is about the same as or worse than Dynamic Priority.

(a) Inconsistency and average response time for sparse matrix multiplication using permutation intervals k, $5k$, $10k$, and $100k$.

Queuing Policy	Inconsistency	Response Time
FIFO	69.303	30.273
Dynamic Priority $T = k$	683.817	14.744
Dynamic Priority $T = 5k$	1178.274	10.824
Dynamic Priority $T = 10k$	1612.298	10.430
Dynamic Priority $T = 100k$	4744.975	9.745
Cycle Priority $T = k$	1768.970	10.430
Cycle Priority $T = 5k$	3916.114	10.323
Cycle Priority $T = 10k$	5512.070	10.275
Cycle Priority $T = 100k$	16597.218	9.837
Priority	21804.684	5.464

(b) Inconsistency and average response time for GNU sort using permutation intervals k, $5k$, $10k$, and $100k$.

Queuing Policy	Inconsistency	Response Time
FIFO	45.021	12.712
Dynamic Priority $T = k$	569.941	10.902
Dynamic Priority $T = 5k$	1163.263	10.513
Dynamic Priority $T = 10k$	1606.777	10.454
Dynamic Priority $T = 100k$	4722.316	10.303
Cycle Priority $T = k$	1776.058	10.451
Cycle Priority $T = 5k$	3929.010	10.436
Cycle Priority $T = 10k$	5528.542	10.428
Cycle Priority $T = 100k$	16823.080	10.242
Priority	27396.798	5.822

parameter range for T where Dynamic Priority's makespan is as good as Priority but the inconsistency is much lower.

Our results indicate that T should be greater than $10k$ to allow closer behavior to Priority and more page reuse between permutations. We observe large ranges of T such that Dynamic Priority can reduce Priority's inconsistency while retaining its makespan.

Priority gives a better average response time, even when makespan is the same. Another metric, average response time, measures the performance of a scheduler. We chart Figure 5 but with average response time instead of Makespan in Table 1. We see that, for the same Makespan, Priority has the lowest average response time for both GNU sort and Sparse Matrix-Matrix Multiplication. For both datasets, FIFO has the highest (worst) average response time and Dynamic Priority and Cycle Priority both give about the same average response time for reasonable values of T. This provides further evidence that this low-thread count anomaly in Figure 2 that is eliminated in Figure 4 by Dynamic Priority is due to an artifact of using Makespan and Priority.

5 MODEL VALIDATION EXPERIMENTS

We validate our algorithmic model from §2 on Xeon Phi Knight's Landing processors, which have HBM accessible both directly (in flat mode) and as a cache for DRAM (in cache mode).

One abstraction the model makes is, instead of a mesh connecting all of HBM, DRAM, and the cores [56], HBM sits between DRAM and the cores. This leads to several algorithmic properties:

Property 1. *HBM and DRAM have a similar latency when accessed directly.*

Property 2. *HBM has higher bandwidth than DRAM (which can be detected by normal programs).*

Property 3. *The latency to access DRAM through a cache miss is approximately double accessing HBM.*

Property 4. *When too many HBM misses occur in cache mode, the channel to DRAM becomes the bottleneck.*

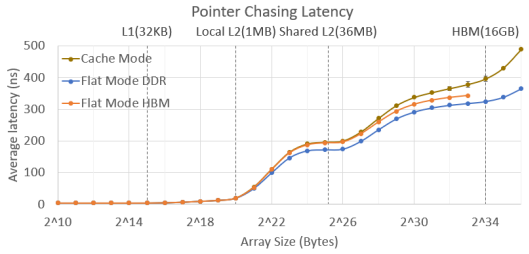
To show that Knight's Landing has Properties 1-4 and is consistent with the HBM Model, we measure and compare the latency and bandwidth of HBM, DRAM, and HBM as a cache for DRAM. Validating some of these properties, especially the benefits of HBM (Property 2 and Property 4), has been done before [46, 47]. We present our findings for due diligence. We perform two microbenchmarks (one latency-bound and one bandwidth-bound, described below) on a range of allocation sizes, some of which fit within HBM and some of which exceed HBM.

We perform these experiments on Xeon Phi Knight's Landing CPUs with 272 threads (4 hyperthreads per core) and 16GiB of HBM. Each CPU has 6 DDR Channels and 8 HBM connections, all connected to the cores via a mesh. We ensure we are accessing HBM (or DRAM) when the machine is in flat mode by using `numactl --membind`.

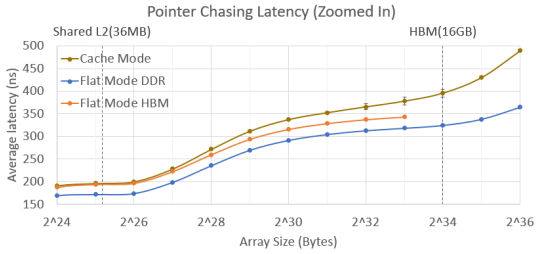
5.1 Microbenchmarks

Measuring Latency: Pointer Chasing. To measure the latency to a level of the KNL's memory hierarchy, we record the average time to chase a pointer on an array of a fixed size. We call the operation $x := a[x]$ ***pointer chasing*** on the array a. In order to map the latency across the memory hierarchy (L1, L2, shared L2, HBM, or DRAM), we run our pointer chasing experiment for arrays whose sizes are the powers of two from 1KiB to 64GiB. We stop the experiment early for HBM, which can only allocate an array of size 8GiB. Each element in the array is initialized to the index of a random element. To avoid loops without causing significant CPU usage from generating random numbers, we add a bit of randomness every 32 pointer chasing operations. In total, we measure the time to perform 2^{27} pointer chasing operations, then divide by 2^{27} to get the average.

Measuring Bandwidth: GLUPS. To measure the bandwidth of various levels of the KNL memory hierarchy, we record the average MiB/s that can be read, xor'd, and written in randomly chosen blocks of length 1024 bytes. To measure bandwidth, we introduce ***GLUPS***, or Giga-Large Updates per Second. GLUPS are closely related to the standard GUPS benchmark (formally referred to as RandomAccess) [44], but operates on sequential blocks of 1024 bytes (128 doubles) to ensure we fully saturate the HBM channels.



(a) Pointer chasing latencies for arrays sizes up to 64GiB.



(b) Figure 6a zoomed in for array sizes larger than shared L2. See Table 2a for data.

Figure 6: Pointer chasing on HBM, DRAM, and HBM as a cache for DRAM. Cache tier and size is marked by vertical dotted lines with annotations above. There is a drastic change in latency after exceeding each level of cache.

Table 2: GLUPS and pointer chasing performance for array sizes within and exceeding HBM.

(a) Pointer chasing latency test for DRAM, HBM, and HBM as Cache. Units are nanoseconds per update. See plotted version in Figure 6b.

Array Size	DRAM (ns)	HBM (ns)	Cache (ns)
16MiB	168.9	187.6	190.6
32MiB	171.9	194.1	196.1
64MiB	174.0	196.5	199.8
128MiB	198.8	222.3	228.1
256MiB	235.6	259.8	271.6
512MiB	269.7	293.8	311.9
1GiB	291.4	315.5	337.5
2GiB	304.4	328.6	352.8
4GiB	312.7	337.2	365.7
8GiB	318.3	343.1	378.3
16GiB	324.4	-	396.1
32GiB	338.0	-	430.5
64GiB	364.7	-	489.6

(b) 272 threads GLUPS bandwidth test for DRAM, HBM and HBM as Cache. Units are MiB/s. HBM and Cache have a much higher bandwidth than DRAM, but HBM as Cache drops off sharply once the working set exceeds HBM.

Array Size	DRAM (MiB/s)	HBM (MiB/s)	Cache (MiB/s)
512MiB	70,627	299,593	308,103
1GiB	67,874	262,208	302,974
2GiB	66,459	315,227	313,730
4GiB	67,025	323,989	319,459
8GiB	67,118	323,318	309,988
16GiB	67,534	-	272,787
32GiB	67,931	-	148,989
64GiB	67,720	-	146,600

We use GLUPS instead of GUPS due to the machine quirks of KNL. GLUPS ensures we're using all the channels to HBM and therefore all the bandwidth by loading in sufficiently large chunks. It's a machine specific reason. We expect GLUPS to be bandwidth-bound when run multi-threaded (272 threads on KNL).

To measure GLUPS, we randomly pick a spot on the array, sequentially read, xor with a fixed but arbitrary number, and write each of the next 128 doubles. For KNL machines, 128 doubles is 16 cache lines each of size 64 bytes. We perform this operation until the entire array's worth of data has been updated—that is, for a 2GiB experiment, we update a total of 2GiB of data. We implement this benchmark in C++ using OpenMP for parallelization.

5.2 Results

Similar access latency to HBM and DRAM. Our model sets the access latency to DRAM and HBM chips as the same (Property 1). In the pointer chasing results in Figure 6a, the latencies for Flat Mode DRAM and Flat Mode HBM differ by approximately 24ns for array sizes between 16MiB and 8GiB. While not exactly the same, the latency difference is still sufficiently small to invalidate standard caching assumptions. For the purposes of modeling, KNL hardware is consistent with Property 1.

Bandwidth advantage of HBM over DRAM. One key reason to use HBM over DRAM is the Higher Bandwidth offered (Property 2). This property is well studied [46, 47]—we present our findings for due diligence. The model in §2 has p channels between HBM and the cores. We validate this in Table 2b by showing a $4.3 - 4.8\times$ bandwidth improvement over DRAM for array sizes between 512 MiB and 8GiB. While not the full $p\times$ bandwidth improvement the model predicts, KNL has a sufficiently large bandwidth such that the bottleneck is not transferring data from HBM to the processor (or cache to processor) but transferring data from DRAM to HBM. Thus, we find that KNL hardware is consistent with Property 2.

Latency penalty for HBM misses in cache mode. When in cache mode, a memory access that misses HBM and goes to DRAM will incur double the latency of an HBM hit (Property 3). In real hardware (specifically KNL), there are several different caches to miss (L1, L2, shared L2) before HBM is accessed. In order to better understand the penalty of missing various caches, we perform pointer chasing on array sizes from 2^{10} to 2^{32} bytes in Figure 6a. We zoom in on the same data in Figure 6b and tabulate it in Table 2a to better show the latency to access HBM when the problem size is larger than the previous levels of cache.

When accessing a random element of an array that is twice the size of HBM in cache mode, there is a 50% chance of getting a cache miss. Therefore, we expect to see the additional latency to DRAM only half the time. However, as the array sizes grow far beyond each boundary in the memory hierarchy, the latencies plateau. Using the differences in heights of these plateaus, we can get an estimate of the latencies to each level of the memory hierarchy.

Specifically, in the cache mode latencies, a miss out the CPU's local L2, which requires traversing the mesh to another tile's L2, incurs about 200ns. This is a baseline latency that we subtract off for measuring latency to HBM and DRAM. Memory accesses that miss shared L2 cache and go into HBM take about 160ns. While we can't see the final plateau for missing HBM and going into DRAM, the final data point suggests that misses to DRAM take 300ns or more. This shows the double latency penalty we expected—KNL hardware is consistent with Property 3.

Bandwidth reduction for HBM misses in cache mode. HBM bandwidth suffers when too many HBM misses occur in cache mode because of the bandwidth bottleneck between DRAM and HBM when using HBM as cache (Property 4). We perform the GLUPS experiment and plot the results in Table 2b. We see that bandwidth halves when the array is 32GiB (2x larger than HBM), but still has a higher bandwidth than DRAM. Thus, KNL hardware exhibits enough of this bottleneck to be consistent with Property 4.

6 CONCLUSION

In this paper, we make a case for the benefits of cycling priorities as a better method of managing HBM. We analyze how cycling affects both fairness and makespan and determine that cycling priority schemes are likely preferable to both FIFO-like and static-priority management schemes. We investigate how to cycle by focusing on two schemes, Dynamic Priority and Cycle Priority. These schemes both have constant-competitive makespan when compared to optimal; our experiments further show that they have sufficiently low constants to outperform FIFO and Priority on common workloads. We find compelling evidence that Dynamic Priority and Cycle Priority may out-perform current FIFO-like HBM management and should be studied further. As Cycle Priority performs well and is likely easier to implement in hardware than Dynamic Priority, we find that Cycle Priority is especially promising.

6.1 Future Work

Our theoretical model is intentionally simple and does not admit all of the complexity of real architectures. One important simplifying assumption is that access sequences are disjoint. Theory on non-disjoint access sequences is a promising avenue for future work.

We test our schemes on similar workloads across all cores. Future work may test different workloads; it will be especially interesting to see how Cycle Priority behaves on different distributions of work.

While we focused on KNL as a motivating example, we did not attempt to simulate KNL hardware. Cycle-accurate simulations will be essential to making informed decisions for specific architectures.

ACKNOWLEDGMENTS

This work was supported in part by the Laboratory-Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Kunal Agrawal was supported by the Department of Computer Science and Engineering at Washington University in St. Louis as well as the NSF grants CCF-2106699, CCF-1733873, and SPX-1725647.

Michael Bender was supported by the National Science Founation grants CCF-2118832, CCF-2106827, CSR-1763680, CCF-1716252, CNS-1938709, and CCF-1725543.

Rathish Das was supported by the Canada Research Chairs Programme and NSERC Discovery Grants.

Benjamin Moseley was supported in part by NSF grants CCF-1824303, CCF-1845146, CCF-2121744, CCF-1733873 and CMMI-1938909. Benjamin Moseley was additionally supported in part by a Google Research Award, an Infor Research Award, and a Carnegie Bosch Junior Faculty Chair.

We would like to thank Mike Ferdman (Stony Brook University), Gwen Voskuilen (Sandia National Laboratories), and especially Si Hammond (Sandia National Laboratories) for helpful conversations and pointers to relevant resources.

REFERENCES

[1] 2015. High-performance on-package memory. http://www.micron.com/products/hybrid-memory-cube/high-performance-on-package-memory Archived at https://web.archive.org/web/20150921170652/http://www.micron.com/products/hybrid-memory-cube/high-performance-on-package-memory.

[2] Alok Aggarwal, Bown Alpern, Ashok K. Chandra, and Marc Snir. 1987. A Model for Hierarchical Memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 305–314.

[3] A. Aggarwal, A.K. Chandra, and M. Snir. 1990. Communication Complexity of PRAMs. *Theoretical Computer Science* (March 1990), 3–28.

[4] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127.

[5] Kunal Agrawal, Michael Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2020. Brief Announcement: Green Paging and Parallel Paging. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*.

[6] Kunal Agrawal, Michael Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2022. Online Parallel Paging with Optimal Makespan. In *Proc. 34th ACM on Symposium on Parallelism in Algorithms and Architectures*.

[7] Kunal Agrawal, Michael A Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2021. Tight bounds for parallel paging and green paging. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 3022–3041.

[8] Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Mohammad Mahdi Javanmard. 2021. Low-Span Parallel Algorithms for the Binary-Forking Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 22–34.

[9] Matthew Andrews, Michael A. Bender, and Lisa Zhang. 1996. New Algorithms for the Disk Scheduling Problem. In *Proc. 37th Annual Symposium on Foundations of Computer Science (FOCS)*. 580–589.

[10] Matthew Andrews, Michael A. Bender, and Lisa Zhang. 2002. New Algorithms for the Disk Scheduling Problem. *Algorithmica* 32, 2 (February 2002), 277–301.

[11] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. 2008. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 197–206.

[12] Mahsa Bayati, Miriam Leeser, and Ningfang Mi. 2020. Exploiting GPU Direct Access to Non-Volatile Memory to Accelerate Big Data Processing. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[13] Michael A. Bender, Jonathan Berry, Simon D. Hammond, K. Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A. Phillips, David Resnick, and Arun Rodrigues. 2015. Two-Level Main Memory Co-Design: Multi-Threaded Algorithmic Primitives, Analysis, and Simulation. In *Proc. 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad,

INDIA.

[14] Michael A. Bender, Jonathan W. Berry, Simon D. Hammond, K. Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A. Phillips, David S. Resnick, and Arun Rodrigues. 2017. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. *J. Parallel and Distrib. Comput.* 102 (2017), 213–228. https://doi.org/10.1016/j.jpdc.2016.12.009

[15] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2019. Small Refinements to the DAM Can Have Big Consequences for Data-Structure Design. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Phoenix, AZ, 265–274.

[16] Guy E Blelloch, Rezaul A Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 501–510.

[17] Allan Borodin and Ran El-Yaniv. 1998. *Online Computation and Competitive Analysis*. Cambridge University Press.

[18] Allan Borodin, Prabhakar Raghavan, Sandy Irani, and Baruch Schieber. 1991. Competitive paging with locality of reference. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. Citeseer, 249–259.

[19] Aydin Buluç and John R Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.

[20] Neil Butcher, Stephen L Olivier, Jonathan Berry, Simon D Hammond, and Peter M Kogge. 2018. Optimizing for KNL Usage Modes When Data Doesn't Fit in MCDRAM. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 37.

[21] Chansup Byun, Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, et al. 2017. Benchmarking data analysis and machine learning applications on the Intel KNL many-core processor. *arXiv preprint arXiv:1707.03515* (2017).

[22] Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 105–115.

[23] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[24] Rathish Das, Kunal Agrawal, Michael A Bender, Jonathan Berry, Benjamin Moseley, and Cynthia A Phillips. 2020. How to manage high-bandwidth memory automatically. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 187–199.

[25] Rathish Das, Shih-Yu Tsai, Sharmila Duppala, Jayson Lynch, Esther M Arkin, Rezaul Chowdhury, Joseph SB Mitchell, and Steven Skiena. 2019. Data races and the discrete resource-time tradeoff problem with resource reuse over paths. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*. 359–368.

[26] Alejandro Strejilevich de Loma. 1998. New results on fair multi-threaded paging. *Electronic Journal of SADIO* 1, 1 (1998), 21–36.

[27] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2018. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Comput.* 78 (2018), 33–46.

[28] Esteban Feuerstein and Alejandro Strejilevich de Loma. 2002. On-line multithreaded paging. *Algorithmica* 32, 1 (2002), 36–60.

[29] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. 1991. Competitive paging algorithms. *Journal of Algorithms* 12, 4 (1991), 685–699.

[30] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *Proc. 40th Annual ACM Symposium on Foundations of Computer Science (FOCS)*. 285–297.

[31] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1 (Jan. 2012), 4. https://doi.org/0.1145/2071379.2071383

[32] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján. 2016. Happy: Hybrid address-based page policy in drams. In *Proceedings of the Second International Symposium on Memory Systems*. 311–321.

[33] Avinatan Hassidim. 2010. Cache Replacement Policies for Multicore Processors. In *Proc. Innovations in Computer Science (ICS)*, Andrew Chi-Chih Yao (Ed.). 501–509.

[34] Nicole Hemsoth. 2014. Micron, Intel reveal memory slice of Knight's Landing. http://www.hpcwire.com/2014/06/24/micron-intel-reveal-memory-slice-knights-landing/.

[35] Mohammad Mahdi Javanmard, Pramod Ganapathi, Rathish Das, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury. 2019. Toward Efficient Architecture-Independent Algorithms for Dynamic Programs. In *International Conference on High Performance Computing*. Springer, 143–164.

[36] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement.. In *USENIX Annual Technical Conference, General Track*. 323–336.

[37] Wenbin Jiang, Pai Liu, Hai Jin, and Jing Peng. 2020. An Efficient Data Prefetch Strategy for Deep Learning Based on Non-volatile Memory. In *International Conference on Green, Pervasive, and Cloud Computing*. Springer, 101–114.

[38] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 24–35.

[39] Anil Kumar Katti and Vijaya Ramachandran. 2012. Competitive cache replacement strategies for shared cache environments. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 215–226.

[40] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[41] Mohammad Laghari and Didem Unat. 2017. Object placement for high bandwidth memory augmented with high capacity memory. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 129–136.

[42] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 26.

[43] Alejandro López-Ortiz and Alejandro Salinger. 2012. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*. ACM, 113–127.

[44] Piotr Luszczek, Jack Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. 2004. Introduction to the HPC Challenge Benchmark Suite. (12 2004).

[45] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press, Cambridge, England.

[46] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. 1–10.

[47] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. Rthms: A tool for data placement on hybrid memory system. *ACM SIGPLAN Notices* 52, 9 (2017), 82–91.

[48] Harald Prokop. 1999. *Cache-Oblivious Algorithms*. Master's thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[49] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, Vol. 28. 128–138.

[50] Ruchira Sasanka. 2022. Enabling High-Bandwidth Memory in Future Intel Processors. https://hpcevents.intel.com/devhub/Enabling-High-Bandwidth-Memory-in-Future-Intel-Processors.

[51] Steven S Seiden. 1999. Randomized online multi-threaded paging. *Nordic Journal of Computing* 6, 2 (1999), 148–161.

[52] Anton Shilov. 2021. Intel Shows Off Multi-Chiplet Sapphire Rapids CPU with HBM. https://www.tomshardware.com/news/sapphire-rapids-with-hbm-pictured.

[53] Johannes Singler and Benjamin Konsik. 2008. The GNU libstdc++ parallel mode: software engineering considerations. In *Proceedings of the 1st international workshop on Multicore software engineering*. 15–22.

[54] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (Feb. 1985), 202–208. https://doi.org/10.1145/2786.2793

[55] George M Slota and Siva Rajamanickam. 2018. Experimental Design of Work Chunking for Graph Algorithms on High Bandwidth Memory Architectures. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 875–884.

[56] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *Ieee micro* 36, 2 (2016), 34–46.

[57] TACC. 2022. Stampede2 User Guide. https://portal.tacc.utexas.edu/user-guides/stampede2

[58] Tiffany Trader. 2018. Requiem for a Phi: Knights Landing Discontinued. https://www.hpcwire.com/2018/07/25/end-of-the-road-for-knights-landing-phi/

[59] Jack Wells, Buddy Bland, Jeff Nichols, Jim Hack, Fernanda Foertter, Gaute Hagen, Thomas Maier, Moetasim Ashfaq, Bronson Messer, and Suzanne Parete-Koon. 2016. *Announcing supercomputer summit*. Technical Report. ORNL (Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States)).