How to Manage High-Bandwidth Memory Automatically

Rathish Das
Stony Brook University
radas@cs.stonybrook.edu

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

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

Jonathan Berry
Sandia National Laboratories
jberry@sandia.gov

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

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

ABSTRACT

This paper develops an algorithmic foundation for automated management of the multilevel-memory systems common to new supercomputers. In particular, the High-Bandwidth Memory (HBM) of these systems has a similar latency to that of DRAM and a smaller capacity, but it has much larger bandwidth. Systems equipped with HBM do not fit in classic memory-hierarchy models due to HBM's atypical characteristics.

Unlike caches, which are generally managed automatically by the hardware, programmers of some current HBM-equipped supercomputers can choose to explicitly manage HBM themselves. This process is problem specific and resource intensive. Vendors offer this option because there is no consensus on how to automatically manage HBM to guarantee good performance, or whether this is even possible.

In this paper, we give theoretical support for automatic HBM management by developing simple algorithms that can automatically control HBM and deliver good performance on multicore systems. HBM management is starkly different from traditional caching both in terms of optimization objectives and algorithm development. Since DRAM and HBM have similar latencies, minimizing HBM misses (provably) turns out not to be the right memory-management objective. Instead, we directly focus on minimizing makespan. In addition, while cache-management algorithms must focus on what pages to keep in cache; HBM management requires answering two questions: (1) which pages to keep in HBM and (2) how to use the limited bandwidth from HBM to DRAM. It turns out that the natural approach of using LRU for the first question and FCFS (First-Come-First-Serve) for the second question is provably bad. Instead, we provide a priority based approach that is simple, efficiently implementable and $O(1)$-competitive for makespan when all multicore threads are independent.

CCS CONCEPTS

• **Theory of computation** → **Scheduling algorithms**; **Packing and covering problems**; **Online algorithms**; **Caching and paging algorithms**; **Parallel algorithms**; **Shared memory algorithms**; • **Computer systems organization** → **Parallel architectures**; **Multicore architectures**.

KEYWORDS

Paging, High-bandwidth memory, Scheduling, Multicore paging, Online algorithms, Approximation algorithms.

ACM Reference Format:
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 (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3350755.3400233

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SPAA '20, July 15–17, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6935-0/20/07…$15.00
https://doi.org/10.1145/3350755.3400233

1 INTRODUCTION

Enabled by the recent innovations in 3D die-stacking technology, vendors have begun implementing a new approach for improving memory performance by increasing the bandwidth between on-chip cache and off-package DRAM [34, 39]. The approach is to bond memory directly to the processor package where there can be more parallel connections between the memory and caches, enabling a higher bandwidth than can be achieved using older technologies. Throughout the rest of this paper, we refer to the on-package 3D memory technologies as ***high-bandwidth memory*** or ***HBM***.[1]

The HBM cannot replace DRAM ("main memory") since it is generally about 5 times smaller than DRAM due to constraints such as heat dissipation, as well as economic factors. For example, current HBM sizes range from 16 gigabytes per compute node (the Department of Energy's "Trinity" [26]) to 96 gigabytes per compute node (the Department of Energy's "Summit" [47]), several times smaller than the per-node sizes of DRAM on those systems (96 GB and 512 GB, respectively). HBM therefore augments the existing memory hierarchy by providing memory that can be accessed with up to 5x higher bandwidth than DDR4, today's DRAM technology, when feeding a CPU [1], and up to 20x higher bandwidth when feeding a GPU [47], but with latency similar to DDR4.

As the number of cores on a chip has grown in the past two decades, the *relative memory capacity*, defined as memory capacity divided by available gigaflops, has decreased by more than 10x [35]. Thus, processors are becoming more starved for data. HBM, with its improved ability to feed processors, provides an opportunity to overcome this bottleneck, if application software can use it.

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

HBM has a critical limitation besides its constrained size: since it uses the same technology as DRAM, it offers little or no advantage in latency. Therefore, the HBM is not designed to accelerate memory-latency-bound applications, only memory-bandwidth-bound ones.[2]

How HBM is managed. Intel's Knights Landing processors [46] on Trinity can boot into multiple modes. In "cache mode," the HBM is system-controlled and is integrated into the memory hierarchy as the "last level of cache." In "flat mode," the programmer controls HBM by explicitly copying data in and out of HBM, trying to squeeze as much performance from the system as possible. Hybrid mode splits the HBM into one "flat" piece and one "cache" piece.

This proliferation of HBM modes exists because there is no consensus on how to automatically manage HBM efficiently, or whether this is even possible.

On the other hand, on most systems, on-chip cache is automatically managed by the hardware and works well enough that application programmers can treat the cache hierarchy a a black box. They can also assume sufficient support from libraries of cache-aware and cache-oblivious algorithms. Ideally, we would like a system controlled HBM to also work well-enough to free the programmer from the need to manage it.

However, managing the HBM is not identical to managing caches and introduces complications that do not exist in more traditional memory hierarchies. In particular, cores compete not only for HBM capacity, but also for the more limited channel capacity between HBM and DRAM. HBM does not fit into a standard memory hierarchy model [12, 30], because in traditional hierarchies, both the latency and bandwidth improve as the levels get smaller. This is not true of HBM.

The question looms: Are there provably good algorithms for automatically controlling HBM?

1.1 Results

Multicore HBM model. We propose a multicore model for HBM that captures the high bandwidth from the cores to HBM and the much lower bandwidth to DRAM. There are p parallel channels connecting p cores to the HBM but only a single channel connecting HBM to DRAM; see Figure 1. This configuration captures the high (on-package) bandwidth between the p cores and HBM and the much lower (off-package) bandwidth between HBM and DRAM. Data is transferred in blocks — there are up to p parallel block transfers from the HBM to the cores, but only one block transfer at a time between DRAM and HBM. The roughly comparable latencies are captured by setting all block-transfer costs (times) to 1. See Section 8 for a more nuanced discussion of this.

HBM management. We focus on instances where the multicore's threads access disjoint sets of blocks. This emphasizes the cores' competition for HBM and the limited bandwidth between HBM and DRAM.

What should the performance objective be? The high-level objective is to improve the performance by finishing all threads as quickly as possible — in other words, we want to minimize the makespan. In sequential caching, we generally use minimizing cache misses as a performance objective since it is a good proxy for makespan. We might be tempted to use HBM misses as the performance objective here. Surprisingly, it turns out that minimizing HBM misses correlates poorly with makespan for HBM for two reasons (formally proved in Section 7). First, unlike caches, HBM has the same access latency as DRAM. Second, multiple processors are accessing the HBM and potentially contending for the channel between HBM and DRAM and the amount of contention can have an impact on the makespan, not just the sheer number of HBM misses. Therefore, we focus directly on minimizing makespan. We establish the following:

- *Minimizing page faults does not correlate well with minimizing makespan.* An algorithm that optimizes the total number of HBM misses may have bad makespan (the time the last thread completes)— up to a $\Theta(p)$-factor worse than optimal. This is not true for the standard cache-replacement problem where the total number of cache misses is the surrogate objective for makespan [19, 28, 44].
- *Minimizing makespan is strongly NP-hard.*
- *Sharing the HBM-to-DRAM channel fairly does not work.* We consider how to design block-replacement policies for the HBM coupled with the First-Come-First-Serve (FCFS) algorithm for determining the order of accesses from HBM to DRAM. We show that even though LRU is a very good block-replacement policy, if we use FCFS in the HBM-to-DRAM channel, LRU performs poorly. In particular, with any constant amount of resource augmentation the makespan of using FCFS with LRU is $\Omega(p)$ away from the optimal policy in the worst case. This negative result establishes that more sophisticated management of the channel between HBM and DRAM is central to the designing a good algorithm for the problem. The seemingly fair FCFS policy is bad.

Our main positive results are simple online and offline algorithms for automatically managing HBM. What is interesting about HBM management is how starkly it differs from traditional caching policies, which are well understood in a serial setting but are challenging in multicore settings [31, 38].

- *Priority-based mechanism for managing the HBM-to-DRAM channel.* We give a priority-based policy for managing the channel between HBM and DRAM. We impose a pecking order on the cores, so that a high-priority core never has a request to DRAM blocked by a request from a lower-priority core. Our algorithms for HBM management are built around this priority-based mechanism.
- *$O(1)$-competitive algorithm for HBM management.* As a first step towards our online algorithm, we give an offline approximation algorithm for the makespan objective. We build upon the offline algorithm to obtain a simple online algorithm (with a more complicated analysis) that is $O(1)$-competitive, even without resource augmentation. The online algorithm is non-intuitive —it often preferentially allocates the HBM-to-DRAM channel to a single core while depriving other cores — but guarantees nearly optimal makespan. (The algorithm can treat cores fairly over time by periodically changing the pecking order among the cores.)

[2] HBM does not increase *off-package* DRAM bandwidth. It does not accelerate a scan of a large chunk of data in DRAM that does not fit into the HBM, because the operation is limited by the DRAM bandwidth. Therefore, HBM improves some memory-bandwidth-bound computations but does not automatically improve all.

1.2 Related Work

HBM-tuning and cache mode. Intel's Knights Landing (KNL) processor [34] features an implementation of HBM. Several recent papers have documented runtime improvements of 3-4x using this HBM in "cache mode" compared to using DRAM alone, when problem instances fit entirely in the HBM. For example, Li et al. studied eight kernels from scientific computing [36] on KNL and found 3-4x speedups on some instances of sparse matrix-vector multiplication, Cholesky decomposition, and dense matrix-matrix multiplication. They observed more modest 1-2x speedups for sparse matrix transpose and sparse triangular solve. Byun, et al. corroborate the KNL speedup for dense matrix-matrix multiplication [21]. Slota and Rajamanickam [45] observed 2-5x speedups in graph algorithms on instances far larger than HBM. Butcher et al. [20] also studied problems that are too large to fit into HBM. They optimized sorting on KNL and concluded that GNU parallel sort run in cache mode is not nearly as fast as a custom sorting algorithm (based upon concurrent calls to GNU serial sort), also run in cache mode [20].

This result is in keeping with the predictions made by Bender et al. [11, 13, 14]. Before KNL existed, they gave HBM-optimized sorting algorithms and obtained simulation results that predicted good speedups for these algorithms. However, this does not settle the question of automatic management of HBM. KNL's arbitration of HBM misses is handled by the DRAM controller. Although the actual protocol is proprietary, it is likely a solution based on [41]. Such arbitration is commonly called "first-ready first-come-first-served (FR-FCFS)." As the name implies, this is a variant of FCFS. We show in Section 5 that FCFS is not a good arbitration policy for HBM misses, and we conjecture that a future cache mode informed by this paper may perform significantly better.

Multi-thread/multi-core paging. There exists a rich literature on multi-threaded and multi-core paging models. Feuerstein and Strejilevich de Loma [27] consider a paging model where there are multiple threads but only a single core. They optimize the number of cache misses. Loma [25] and Seiden [42] give randomized algorithms for the same setting.

Hassidim [31] considers a paging model where there are multiple threads and multiple cores. He minimizes the makespan. López-Ortiz and Salinger [38] consider a similar model but minimize cache misses. They give lower bounds and an offline algorithm with a runtime exponential in the size of cache. Katti and Ramachandran [33] give a competitive algorithm for multi-core paging assuming that the interleaving of the request sequences of the cores are fixed. These classic paging results differ from our HBM model because in these prior results access times of near and far levels are different but there are still p channels between the cores and shared cache and between cache and DRAM. Therefore, these prior parallel caching results do not carry over to our setting, and vice versa.

When multiple threads access the same shared cache, the fraction of cache dedicated to any given thread can vary [5, 9, 10, 17]. Peserico [40] and Bender et al. [16, 17] formulated models for page replacement in a fluctuating cache, with the latter model serving as an algorithmic foundation for cache-adaptive analysis [16, 17, 37].

There are several analysis frameworks based upon assuming an underlying optimal paging algorithm. These include the seminal ideal cache-model of Frigo et al. [29, 30] and Prokop [30], which was based on Sleator and Tarjan's [44]'s classic paging results; cache-adaptive analysis [16, 17, 37]; and parallel caching models based on work stealing [23]. We can view this paper as proposing an alternative setting for HBM, where the programmer can assume optimal paging for HBM, and then let the system make all decisions.

Finally, we note that there are many sequential and parallel models of the memory hierarchy [2–4, 6, 7, 15, 18, 22, 24, 32]. These include models where there are private caches associated with each core [8]. In contrast, our HBM model explicitly does not need to consider a private cache. Whatever happens in the private caches of each individual core is independent of the optimization problem in this paper. As mentioned in [12], the performance characteristics of HBM set it apart from most other memory-hierarchy models, so that in some ways, HBM and DRAM are like siblings on the same level of the hierarchy, and in other ways they are stacked. This helps explain why the present optimization problem is so surprisingly different from prior work, and also why prior work provides little insight into how to deal with HBM.

2 HBM MODEL

Our model of HBM comprises a multi-core machine with p parallel channels connecting p cores to the HBM. The HBM has size M, and the DRAM (main memory) has no space limitation.

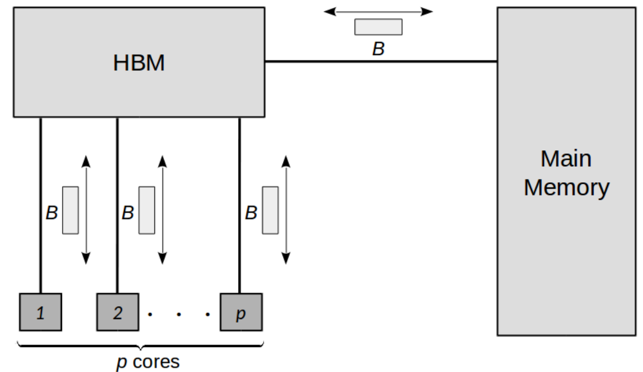

Figure 1: The HBM model with p cores and two levels of memory.

The increased bandwidth of HBM comes from multiple channels between it and the cores. There is only one channel between HBM and DRAM. Data is transferred along any of these channels in blocks of size B; see Figure 1. HBM can hold $k = M/B$ blocks. There can be up to p parallel block transfers from the HBM to the cores, but only one block at a time is transferred between DRAM and HBM. The roughly comparable access costs are captured by setting all block-transfer cost to 1. That is, it takes one time step to transfer a block from HBM to a core or from DRAM to HBM.

Unlike the Ideal Cache model [29, 30], our HBM model has two resources to manage: the HBM itself and the ***far channel*** between HBM and DRAM. The HBM is managed by a block-replacement policy, and the DRAM channel is managed by a ***far-channel arbitration policy.*** For example, we might consider LRU block replacement and FCFS far-channel arbitration.

Parallel program execution in HBM model. Each core runs its own stream of instructions, which for the purposes of the model, is a sequence of block requests. Thus, we denote $R^i = r_0^i, r_1^i, r_2^i, \ldots$ as the sequence of the blocks requested by core p_i on its dedicated channel to the HBM. We omit the core number when it is understood. We analyze the *disjoint* case in which the cores access disjoint sets of blocks, that is, for $i \neq j$ and $\forall q, s$ $r_q^i \neq r_s^j$, as in prior work on parallel caching [31, 38]. This case emphasizes the computational issues that arise when the p programs compete for their own share of the HBM. Furthermore, the most common case when executing multithreaded programs is that the threads are disjoint or nearly disjoint [12, 20].

We say that a request r_j^i is ***served at time step t***, if the previous request to be served was r_{j-1}^i and the requested block r_j^i is transferred to core p_i at time step t. Request r_j^i is served as follows. If r_j^i is in HBM, then core p_i receives that block exactly one tick after the request. Otherwise, the block must be retrieved from DRAM via the far channel which takes at least one additional tick and may take many more depending on the request streams of other cores and the far-channel arbitration policy. Note that HBM hits have nonzero cost in our HBM model (since HBM latency is no better then DRAM latency) and are thus infinitely more expensive than the zero-cost cache hits of traditional caching models.

Objective: minimizing makespan. Given an HBM of size M and p disjoint request sequences of p cores, the objective is to find a contention-resolution policy for the HBM-DRAM channel and a block-replacement policy for the HBM so that the makespan is minimized.

3 TECHNICAL OVERVIEW

In the rest of the paper, we propose methods to automatically manage HBM and we analyze them with our new HBM model. Our contributions, per section, are as follows:

- In this section, we explain how to navigate the algorithmic issues that distinguish HBM management from traditional cache management and we informally justify the makespan metric for HBM. We add a formal justification in Section 7.
- We give online algorithms for managing HBM asymptotically optimally with respect to the makespan objective (Section 4).
- We analyze the natural strategies for managing the two resources of the HBM model (variants of which are canonical results of computer architecture cited more than 1000 times [41]) and prove that they are not asymptotically good at managing HBM (Section 5).
- We prove the strong NP-hardness of the makespan minimization problem for HBM (Section 6)

Metrics for HBM management. HBM management must comprise policies for (1) dividing HBM's storage capacity among cores or DRAM regions, (2) evicting blocks from HBM, and (3) deciding which DRAM block requests to satisfy first, which we call "far-channel arbitration." In this paper, we do not constrain (1). However, we note that KNL's cache mode employs direct-mapped caching (each block in DRAM has a fixed and unique destination in HBM), constraining both (1) and (2). Crucially, we will show that effective far-channel arbitration is the key to reducing the running time of a program in the HBM model.

We know that in single-core and multi-core paging models, minimizing the number of cache misses leads to a good approximation of the running time of a program [19, 28, 44]. The interesting difference in our HBM model is that the number of HBM misses no longer gives a good approximation to the running time. In particular, in Section 7, we show that there exist request sequences from p cores such that any policy that minimizes the number of HBM misses has a running time that is a factor of $\Theta(p)$ larger than the optimal running time. Moreover, unlike in traditional caching, resource augmentation in the form of larger HBM sizes is not necessary.

Hence, we turn directly to the makespan metric. We show that the problem of minimizing the makespan in HBM model is strongly NP-hard. The limited bandwidth between HBM and DRAM plays a pivotal role in the hardness proof.

So how should we deal with contending requests for the HBM-DRAM channel?

Natural far-channel arbitration policies do not work in the HBM model. One intuitive (but doomed) far-channel arbitration policy is to queue DRAM requests in First-Come-First-Serve (FCFS) order. Fairness would seem to dictate some sort of FCFS queue, and a canonical variant of this from [41] has been influential with vendors of DRAM controllers. However, we prove that FCFS is *not* a good far-channel arbitration policy for HBM. Even if we have a good eviction policy, such as LRU, a far-channel arbitration policy of FCFS queueing is provably non-optimal. In particular we show that even with d memory augmentation and s far-channel bandwidth augmentation, there exist p request sequences in which the makespan of FCFS with LRU is a $\Theta(\frac{p}{ds})$-factor away from that of the optimal policy.

A better strategy is to assign priorities to the cores. That is, there is a pecking order among the cores so that a high-priority core always beats a lower-priority core when one of its requests needs access to the HBM-DRAM channel. In Section 4 we analyze an online HBM-management strategy where the eviction policy is LRU but the far-channel arbitration policy is based on the pecking order. We prove that this simple scheme, which we call PRIORITY, is constant competitive with the optimal policy for minimizing makespan (without needing resource augmentation of either the HBM size or the far-channel bandwidth).

The theoretical wedge that this paper drives between FCFS and priority-based HBM-DRAM channel arbitration is an important contribution of this paper. Given that it is natural to have FCFS buffers queue up requests for the HBM-DRAM channel (and that FCFS variants are widely implemented in today's hardware), we believe that this negative queuing result could be quite useful to hardware designers. A priority-based scheme is straightforward to implement in real hardware and leads to provably good algorithms for HBM management under our assumption of disjoint reference streams. Furthermore, we note that priority-based schemes are not inherently unfair. Our analysis still works if we change the priorities periodically over time.

Analysis of online competitive algorithm for the makespan. While the priority-based mechanism seems algorithmically simple, its analysis is more complicated. Thus, we explain the online analysis through an intermediary, an offline algorithm with a more complicated mechanism but a somewhat simpler analysis. We show

that this algorithm, which we call k-PACKING, is an $O(1)$ approximation algorithm to makespan.

The offline algorithm k-PACKING divides the execution into *phases* of $\Theta(k)$ steps, where k is the size of the HBM. In each phase, each core makes an all-or-nothing decision about whether to execute its thread: either the thread makes $\Theta(k)$ progress or it does not run. If a thread runs, it grabs all of the resources from HBM that it needs by allocating space in the HBM and a channel bandwidth equal to the number of blocks that it accesses in that phase. Thus, in a phase, some threads make essentially full progress and others make none. k-PACKING performs a maximal packing of the threads into the phase. k-PACKING is a "very" offline policy because it requires $\Theta(k)$ look-ahead for each core, and k is very large.

The online algorithm PRIORITY does not have any lookahead, in contrast to k-PACKING. Hence, some cores may progress $\Theta(k)$ steps in a $\Theta(k)$-length phase, while others do not. If a core does not progress a full $\Theta(k)$ steps, we say that it *wastes* HBM capacity and bandwidth. We prove that the priority scheme guarantees at most a constant factor of these resources are wasted. This delivers the same performance guarantees as k-PACKING.

4 $O(1)$-COMPETITIVE ONLINE ALGORITHM FOR HBM BLOCK MANAGEMENT

In this section we present an $O(1)$-competitive online algorithm for the makespan-minimization problem. We first give an offline $O(1)$-approximation algorithm. Then we show how to transform the offline strategy into an online strategy while retaining constant competitiveness.

One of the exciting aspects of our makespan-minimization problem is that proving constant competitiveness does not require resource augmentation. This result stands in stark contrast to most online caching problems, where resource augmentation is necessary to achieve good competitive ratios. Nonetheless, the optimization problem is delicate. In Section 5 we show that some seemingly natural HBM caching policies achieve a competitive ratio as large as $\Theta(p)$.

In the rest of this section we prove the following theorem:

THEOREM 1. *There exists an $O(1)$-competitive online algorithm for the makespan-minimization problem (without resource augmentation).*

4.1 Constant-approximation offline algorithm

This section gives an offline $O(1)$-approximation algorithm for the makespan-minimization problem, which we call the k-PACKING algorithm. We show the following:

Lemma 1. *There exists an offline constant-approximation algorithm for the makespan-minimization problem (without resource augmentation).*

We divide the request sequence $R^i = r_1, r_2, r_3, \ldots$ for each thread i into ***chunks***, where each chunk C_{ij} (except possibly the last) contains exactly $k/4$ requests[3]. Specifically,

$$R^i = \overbrace{r_1, r_2, \cdots, r_{k/4}}^{C_{i1}}, \overbrace{r_{1+k/4}, r_{2+k/4}, \cdots, r_{2k/4}}^{C_{i2}} \cdots$$

Executing a chunk C_{ij} means servicing each request in C_{ij}. A chunk C_{ij} is ***ready to run*** as soon as $C_{i,j-1}$ is executed. We associate a request with the block of memory needed to service the request.

Algorithm k-PACKING. The k-PACKING algorithm proceeds in ***phases***. In each Phase ϕ, k-PACKING executes at most one chunk from each thread. No chunks are partially executed. Let C be a set of chunks. Define the ***working set*** of C, denoted $\mathcal{B}(C)$, to be the set of blocks requested in set C. It is the union of the set of blocks over all the chunks in C.

In Phase ϕ, k-PACKING executes a set C_ϕ of ready-to-run chunks such that

1. $|\mathcal{B}(C_\phi)| \leq k$,
2. Each thread executes either zero or one chunk in Phase ϕ, and
3. C_ϕ is maximal. That is, no additional chunk can be added while satisfying Constraints 1 and 2.

Although chunks can be chosen greedily within a phase, k-PACKING itself is not greedy. That is, it may be possible for a thread to make forward progress in a phase without hindering any other thread—but k-PACKING does not execute any thread unless it can complete an entire chunk for that thread in the phase.

Because a phase ϕ is defined by the chunks C_ϕ that it runs, we will overload notation, letting $\mathcal{B}(\phi)$ denote the set of blocks served in Phase ϕ. For generic input $\mathcal{I}$, let k-PACKING($\mathcal{I}$) denote running k-PACKING on instance $\mathcal{I}$.

Definition 1. *Each Phase ϕ in k-PACKING($\mathcal{I}$) has one of the following types.*

Contested: $3k/4 \leq |\mathcal{B}(\phi)| \leq k$
Uncontested: $|\mathcal{B}(\phi)| < 3k/4$.

Lemma 2. *If $|\mathcal{B}(\phi)| < 3k/4$, then all unfinished threads execute a chunk in Phase ϕ.*

PROOF. We prove this by contradiction. Let ϕ be a phase such that $|\mathcal{B}(\phi)| < 3k/4$. Assume that there is an unfinished thread p_i that does not execute a chunk in ϕ. However, a chunk of p_i has at most $k/4$ blocks. So we can add the ready-to-run chunk of p_i to Phase ϕ without violating Constraints 1 or 2 for phases. Therefore phase ϕ is not maximal, a violation of the third constraint. □

Lemma 3. *Suppose that k-PACKING($\mathcal{I}$) has X_1 contested phases and X_2 uncontested phases. Then the makespan of k-PACKING($\mathcal{I}$) is at most $\frac{5k}{4}X_1 + kX_2$.*

PROOF. In every contested phase, at most k blocks are transferred from DRAM to HBM, requiring at most k time steps. Once the blocks are in HBM, they are served to the cores in at most $k/4$ time steps. So a contested phase finishes in $5k/4$ time units. Similarly, in every uncontested phase, at most $3k/4$ blocks are fetched from DRAM to

[3] Our proofs assume that k is a multiple of 4, fairly common for memory size, but we can adjust the proofs for general k.

HBM using at most $3k/4$ time steps. Once all the blocks are in the HBM, they are served to the cores in at most $k/4$ more time steps. So a uncontested phase finishes in k time units. □

For a set of phases Φ for an algorithm A we define $\eta_A(\Phi) \equiv \sum_{\phi \in \Phi} |\mathcal{B}(\phi)|$. When the set of distinct blocks in any phase fits in HBM (i.e. total at most k), $\eta_A(\Phi)$ gives an upper bound on the total time to bring blocks in from DRAM over all phases in Φ. This is the cost of running the algorithm normally, but artificially emptying HBM at phase boundaries. The system may need to bring in a block once for each phase it participates in. It can be served to its core from HBM multiple times within a single phase. For any instance k-PACKING($\mathcal{I}$), let Φ_1 be the set of contested phases.

Observation 1. *Suppose k-PACKING($\mathcal{I}$) has X_1 contested phases. Then $\eta_{k\text{-PACKING}}(\Phi_1) \geq (3k/4)X_1$.*

Analysis of OPT. Let OPT denote the optimal algorithm for the makespan-minimization problem. As with k-PACKING, we divide the execution OPT($\mathcal{I}$) on instance $\mathcal{I}$ into ***phases***. Each phase has a fixed length of exactly $k/4$ time steps (except possibly the last phase). Thus, Phase 1 contains the requests serviced in the first $k/4$ time steps, Phase 2 contains the requests serviced in the next $k/4$ time steps, and so on.

Observation 2. *Suppose OPT($\mathcal{I}$) runs in Y phases. Then its makespan is at most $(k/4)Y$ and at least $(k/4)(Y-1)+1$.*

We now compare the number of phases in OPT($\mathcal{I}$) versus k-PACKING($\mathcal{I}$).

Lemma 4. *Suppose k-PACKING($\mathcal{I}$) has X_2 uncontested phases and OPT($\mathcal{I}$) has Y phases. Then $Y \geq X_2$.*

PROOF. From Lemma 2, in a uncontested phase, all the unfinished threads execute a chunk, which contains $k/4$ block requests. Let p_i be a thread that executes during the last uncontested phase. This implies that in all X_2 phases, p_i executes a chunk. As OPT can execute at most $k/4$ requests of p_i in each phase (length of each phase in OPT is $k/4$), OPT needs at least X_2 phases to serve thread p_i. □

Let Φ_{OPT} be the set of phases for the optimal algorithm for some instance $\mathcal{I}$. OPT has only one kind of phase. Since the algorithm is clear from context, use the shorthand $\eta(\Phi_{OPT})$ instead of $\eta_{OPT}(\Phi_{OPT})$.

Lemma 5. $\eta_{k\text{-PACKING}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$.

PROOF. Consider an arbitrary phase ϕ of OPT. Let B be a block that is requested by thread p at least once during phase ϕ. Let the first and last requests of block B in phase ϕ be the fth and jth reference to B respectively in thread p's request sequence. All $j-f+1$ references to Block B in phase ϕ together contribute exactly 1 to $\eta(\Phi_{OPT})$. Let r'_f and r'_j be the references in thread p's request stream corresponding to the fth and jth reference to block B respectively. Because each phase of OPT has $k/4$ time steps, there are at most $k/4$ requests between r'_f and r'_j in thread p. Let ϕ_f be the phase in k-PACKING that contains reference r'_f and let ϕ_g be the next phase of k-PACKING that contains a reference to Block B. Both phase ϕ_f and phase ϕ_g execute $k/4$ requests for thread p (unless phase ϕ_g is the last phase for thread p.). Thus reference r'_j, is either in phase ϕ_f or phase ϕ_g. Thus all references to block B in phase ϕ of OPT are in at most two phases (of any type) in k-PACKING. They contribute at most 2 to $\eta_{k\text{-PACKING}}(\Phi_1)$. Summing over all phases in Φ_{OPT} proves the lemma. □

Lemma 6. *Suppose that OPT has Y phases. Then $\eta(\Phi_{OPT}) \leq (5k/4)Y$.*

PROOF. Because a Phase ϕ of OPT has length $k/4$ (except a truncated last phase), at most $k/4$ blocks can be transferred from DRAM to HBM. In addition, there are at most k distinct blocks already present in HBM at the start of Phase ϕ. These can be accessed in parallel by the threads. Thus, altogether, in the phase, at most $5k/4$ distinct blocks can be accessed. Summing over all Y phases proves the lemma. □

Lemma 7. *Suppose that k-PACKING has X_1 contested phases and OPT has Y phases. Then $X_1 \leq (10/3)Y$.*

PROOF. From Lemma 5, we know $\eta_{k\text{-PACKING}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$. From Observation 1, $(3k/4)X_1 \leq \eta_{k\text{-PACKING}}(\Phi_1)$ and from Lemma 6, $\eta(\Phi_{OPT}) \leq (5k/4)Y$. Combining these, we get $X_1 \leq (10/3)Y$. □

PROOF OF LEMMA 1: Suppose that k-PACKING has X_1 contested and X_2 uncontested phases. Let $T(\mathcal{A})$ denote the makespan of an algorithm $\mathcal{A}$. Then from Lemma 3,

$$T(k\text{-PACKING}) \leq \frac{5k}{4}X_1 + kX_2.$$

Suppose OPT has Y phases. Then from Lemma 7, $X_1 \leq (10/3)Y$ and from Lemma 4, $X_2 \leq Y$. Combining these, we get the following.

$$T(k\text{-PACKING}) \leq \frac{5k}{4}\frac{10}{3}Y + kY.$$

However, from Observation 2, $T(\text{OPT}) \geq (k/4)(Y-1)$. Thus,

$$\begin{aligned} T(k\text{-PACKING}) &\leq \frac{5k}{4}\frac{10}{3}(Y-1) + \frac{50k}{12} + 4\frac{k}{4}(X_2-1) + k \\ &\leq \frac{50}{3}T(\text{OPT}) + 4T(\text{OPT}) + \frac{62k}{12}. \end{aligned}$$

Hence, the lemma follows as $T(k\text{-PACKING}) = O(T(\text{OPT}))$.

□

4.2 Online algorithm

In this section we introduce an online algorithm PRIORITY, which automatically guarantees that its execution on an instance $\mathcal{I}$, PRIORITY($\mathcal{I}$), has some of the structural properties that k-PACKING($\mathcal{I}$) does. Unlike k-PACKING, PRIORITY does not need to know the HBM size k or any future requests from the request sequences.

Specifically, the k-PACKING algorithm guarantees that in a phase (1) there are $\Theta(k)$ steps, (2) either the working set size is $\Theta(k)$ or every unfinished thread makes $\Theta(k)$ progress, (3) any thread that makes progress (without finishing) completes $\Theta(k)$ requests.

What makes the HBM model algorithmically interesting is the bandwidth bottleneck between HBM and DRAM. Given this bottleneck, the algorithmic concern/challenge is how to break ties when multiple block requests compete for the limited bandwidth.

In the offline setting, we managed the tie-breaking issue by using size-$\Theta(k)$ chunks, but PRIORITY does not know k.

FCFS (First-Come-First-Serve) is a naturally fair policy for managing DRAM accesses. As we show in the next section, FCFS works poorly, at least when paired with an LRU page-replacement policy —see Section 5.

A better idea, at least when using LRU page replacement, is to assign priorities to threads, so that a high priority thread can never be blocked by a low-priority thread. This leads to constant competitiveness. Specifically, we prioritize block requests based on *which thread* made the request, with the highest-priority thread granted access to the DRAM. The specific priority order does not matter. With this far-channel arbitration policy, PRIORITY naturally does what we explicitly designed k-PACKING to do. Moreover, unlike most caching problems, resource augmentation is not necessary for constant competitiveness.

Let $R^i = r_1^i, r_2^i, r_3^i, \ldots$ be thread p_i's request sequence. Suppose that at time step t of some algorithm execution, thread p_i has served all requests through r_{j-1}^i. Let $u_i(t) = r_j^i$ denote thread p_i's first unserviced request at time t. We partition the threads into two sets, $P(t)$ and $\overline{P(t)}$ as follows. If r_j^i is in the HBM at the start of time step t, then $p_i \in P(t)$, and otherwise $p_i \in \overline{P(t)}$. At time 0, the HBM is empty, and hence for all i, $p_i \in \overline{P(0)}$.

Algorithm PRIORITY. We assign a fixed priority for each thread. Without loss of generality, say that thread p_i has priority i, where priority 1 is the highest.

In each time step t and for each thread p_i:

(1) If $p_i \in P(t)$, then block $u_i(t)$ is transferred to p_i's core.
(2) Otherwise, $p_i \in \overline{P(t)}$.
   (a) If p_i is the highest priority thread among the threads in $\overline{P(t)}$, then $u_i(t)$ is transferred from DRAM to HBM. If the HBM is full, then the least-recently-used block among all the cores (breaking ties arbitrarily) in HBM is replaced.
   (b) Otherwise, p_i ***stalls*** (i.e., waits, since only one block can be fetched from DRAM to HBM in each step).

It takes one time step to transfer a block from HBM to a core or from DRAM to HBM. Thus, if $p_i \in P(t)$ (conditional in Step 1 holds), then after Step 1 in PRIORITY, $u_i(t+1) = r_{j+1}^i$. Otherwise, after PRIORITY executes Step 2, $u_i(t+1) = r_j^i$.

Analysis of PRIORITY. For the analysis (only), we divide PRIORITY's execution on instance $\mathcal{I}$, denoted PRIORITY($\mathcal{I}$), into phases of length k. Phase ϕ (for $\phi \geq 1$) begins at the start of time step $(\phi - 1)k + 1$ and finishes at the end of step ϕk. We say thread p_i is ***productive in Phase ϕ*** if and only if p_i serves at least $k/4$ requests in Phase ϕ or finishes the thread's execution. Otherwise, p_i is ***unproductive in Phase ϕ.***

Definition 2. *Let $\mathcal{B}(\phi)$ denote the set of distinct blocks that* PRIORITY($\mathcal{I}$) *serves in Phase ϕ. There are two types of phases:*

Contested: $k/2 \leq |\mathcal{B}(\phi)| \leq k$ *or*
Uncontested: $|\mathcal{B}(\phi)| < k/2$.

The following lemma shows uncontested phases are productive.

Lemma 8. *For any uncontested phase in* PRIORITY($\mathcal{I}$)*, all unfinished threads serve at least $k/4$ requests.*

PROOF. Since there are fewer than $k/2$ distinct blocks accessed in Phase ϕ, then there are at most $k/2$ steps when any block is brought in from DRAM. Thus, since a phase has k time steps, there are at least $k/2$ steps when no thread needs a block that is not in HBM. During each of these steps, any active thread serves a page request from its sequence or finishes. □

The next two lemmas show that for every Phase ϕ, PRIORITY($\mathcal{I}$) satisfies the following two conditions:

- Each phase in PRIORITY($\mathcal{I}$) has at least one productive thread.
- In every contested phase, the productive threads alone serve at least $k/2$ distinct blocks.

Lemma 9. *Every phase of* PRIORITY($\mathcal{I}$) *has at least one productive thread.*

PROOF. Let thread p_i have the highest priority among the threads that run in Phase ϕ. If p_i finishes its execution in Phase ϕ, then by definition, it is a productive thread. Otherwise, in each time step, a block of p_i is transferred from DRAM to HBM or from HBM to p_i's core. Being the highest priority thread, p_i never stalls. Since the phase has length k, thread p_i services at least $k/2$ block requests or finishes, making it a productive thread. □

Lemma 10. *For every contested phase ϕ, the productive threads access $g(\phi)$ distinct blocks, where $|\mathcal{B}(\phi)| \geq g(\phi) \geq k/2$.*

PROOF. For ease of presentation, rename the threads that have not completed executing, so that p_1 is the highest priority thread, p_2 is the next highest priority thread, and so on.

We say that a thread p_i is ***active in step t*** if it is not stalled during the step. Thus, a block for p_i is transferred either from DRAM to HBM or from HBM to p_i's core. Say that thread p_i accesses h_i distinct blocks in Phase ϕ.

Thread p_1 is never stalled. Thus, it is active for k steps unless it finishes executing before the phase ends.

Thread p_2 can be stalled for at most h_1 time steps. This is because p_1 grabs the DRAM-to-HBM channel at most h_1 times during the phase. (Thread p_1 might grab the channel *fewer* than h_1 times, since the requested blocks might already be in HBM from a previous phase.) Thus, p_2 is active for at least $k - h_1$ steps unless it finishes executing before the phase ends.

Similarly, p_3 can be stalled for at most $h_1 + h_2$ time steps. This is because p_3 is only stalled when either p_1 or p_2 grabs the DRAM-to-HBM channel. In general, p_i can be stalled for at most $\sum_{j=1}^{i-1} h_j$ steps, and thus is active for $k - \sum_{j=1}^{i-1} h_j$ steps unless it finishes before the phase ends.

Let ℓ be the lowest priority thread in Phase ϕ such that $\sum_{j=1}^{\ell} h_j \geq k/2$.

We show that for all $i = 1 \ldots \ell$, thread p_i is productive. For $i = 1 \ldots \ell$, thread p_i is active for at least $k - \sum_{j=1}^{\ell-1} h_j \geq k/2$ steps unless it finishes earlier. If a thread finishes earlier, then by definition, it is productive. Otherwise, if a thread is active for $k/2$ steps, then it must serve at least $k/4$ block requests in its sequence, since it takes two steps to bring a block from DRAM to a thread's core. Hence, all ℓ threads are productive threads.

Thus, together all the productive threads access $\sum_{j=1}^{\ell} h_j \geq k/2$ distinct blocks, establishing the lemma. ◻

For a phase ϕ, let $\mathcal{B}^*(\phi)$ denote the set of distinct blocks requested by all the productive threads in phase ϕ. Let Φ_1 denote the set of contested phases. Similar to our previous notation, let $\eta^*_{\text{PRIORITY}}(\Phi_1) \equiv \sum_{\phi \in \Phi_1} |\mathcal{B}^*(\phi)|$.

Corollary 1. *Let* PRIORITY *have* Z_1 *contested phases. Then* $\eta^*_{\text{PRIORITY}}(\Phi_1) \geq (k/2)Z_1$.

Analysis of OPT**.** We divide OPT's execution on instance $\mathcal{I}$ into ***phases***. Each phase has a fixed length of exactly $k/4$ time steps (except possibly the last phase).

We now compare the number of phases in OPT versus PRIORITY. Our proof of the following lemma is similar to that of Lemma 5. However, since PRIORITY can not pack and execute chunks the way the offline algorithm k-PACKING does in a phase, we base the proof on the productive threads in a fixed-length phase. Later we show that considering only productive threads is enough to establish the constant-competitiveness of PRIORITY.

Lemma 11. $\eta^*_{\text{PRIORITY}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$.

PROOF. Consider an arbitrary phase ϕ of OPT. Let B be a block that is requested by thread p at least once during phase ϕ. Let the first and last requests of block B in phase ϕ be the fth and jth reference to B respectively in thread p's request sequence. All $j - f + 1$ references to Block B in phase ϕ together contribute exactly 1 to $\eta(\Phi_{OPT})$. Let r'_f and r'_j be the references in thread p's request stream corresponding to the fth and jth reference to block B respectively. Because each phase of OPT has $k/4$ time steps, there are at most $k/4$ requests between r'_f and r'_j in thread p. Let ϕ_f be the phase in PRIORITY that serves reference r'_f. Let ϕ^*_f be phase ϕ_f if thread p is productive in ϕ_f. Otherwise, let ϕ^*_f be the first phase after ϕ_f in PRIORITY where thread p is productive and it serves a reference to block B in $[r'_f, \ldots, r'_j]$. Let ϕ_g be the next phase of PRIORITY after phase ϕ^*_f that serves a reference to block B where thread p is productive. Both phase ϕ^*_f and phase ϕ_g execute $k/4$ requests for thread p (unless phase ϕ_g is the last phase for thread p.). Thus all the fth through jth request to block B are served by PRIORITY in phase ϕ^*_f or in phase ϕ_g, or in phases where thread p is not productive. Only phases ϕ^*_f and ϕ_g might contribute counts to $\eta^*_{\text{PRIORITY}}(\Phi_1)$, and together they contribute at most 2 to $\eta^*_{\text{PRIORITY}}(\Phi_1)$. Summing over all phases in Φ_{OPT} proves the lemma. ◻

Lemma 12. *Suppose* PRIORITY *has* Z_1 *contested phases and* OPT *has* Y *phases. Then* $Z_1 \leq 5Y$.

PROOF. From Lemma 11, $\eta^*_{\text{PRIORITY}}(\Phi_1) \leq 2\eta(\Phi_{OPT})$. Also, from Corollary 1, $\eta^*_{\text{PRIORITY}}(\Phi_1) \geq (k/2)Z_1$ and from Lemma 6, $\eta(\Phi_{OPT}) \leq (5k/4)Y$. Combining these, we get $Z_1 \leq 5Y$. ◻

Lemma 13. *Suppose* PRIORITY *has* Z_2 *uncontested phases and* OPT *has* Y *phases. Then* $Z_2 \leq Y$.

PROOF. The proof is essentially the same as that for Lemma 4. ◻

PROOF OF THEOREM 1: Suppose that PRIORITY has Z_1 contested and Z_2 uncontested phases. Then PRIORITY's makespan $T(\text{PRIORITY})$ satisfies the following:

$$T(\text{PRIORITY}) \leq k(Z_1 + Z_2).$$

Suppose OPT has Y phases. Using the results from Lemma 12 and Lemma 13, we get the following:

$$T(\text{PRIORITY}) \leq 6kY.$$

However, from Observation 2, $T(\text{OPT}) \geq (k/4)(Y - 1)$. The theorem follows, since

$$T(\text{PRIORITY}) \leq 24T(\text{OPT}) + 6k = O(T(\text{OPT})).$$

◻

Why do we use LRU for block replacement? Many arguments in proofs above assume that all references to a block within the same phase cost at most one access to DRAM. This assumes that once a block is brought in during a phase, it will stay in HBM for the rest of the phase. Thus future accesses to that block in the same phase are HBM hits. LRU is one way to ensure that new blocks coming in do not knock out very-recently-fetched blocks. Our phases have at most k distinct blocks. Thus the blocks brought in during the phase can all fit into HBM without evicting each other when using LRU.

5 FCFS WITH LRU IS NOT A GOOD POLICY IN THE HBM MODEL

In this section we consider a very natural contention-resolution policy FCFS for the DRAM-HBM channel with a widely used natural block replacement policy LRU. We show that the contention-resolution policy FCFS with the block replacement policy LRU (let us call it FCFS+LRU) works very poorly in the HBM model. We prove the following theorem.

THEOREM 2. *There exists 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.*

We give FCFS+LRU d-memory augmentation, that is, FCFS+LRU has an HBM of size k, whereas OPT has an HBM of size k/d. We assume that d divides k. Given the memory augmentation d, we set $k = 2pd$ where p is the number of cores. We could have chosen any larger value of k. Larger values of k capture reasonable HBM sizes and we would obtain the same lower bound. The request sequences would need to be updated accordingly.

PROOF.

$$R^i = \Big(\underbrace{\overbrace{x^i_1, x^i_1, \ldots, x^i_1}^{\ell+i}, \ldots, \overbrace{x^i_{p-1}, x^i_{p-1}, \ldots, x^i_{p-1}}^{\ell+i}}_{2d-1 \text{ light phases}}, \underbrace{\overbrace{x^i_1, x^i_2, \ldots, x^i_{k/p+1}, x^i_1, x^i_2, \ldots, x^i_{k/p+1}, \cdots}^{\ell+i}}_{1 \text{ heavy phase}}\Big)^{\lambda}.$$

The request sequence of core p_i is divided into phases of length $\ell + i$. There are two types of phases: light phases and heavy phases. In a light phase a single block is requested for $\ell + i$ times. In a heavy phase $k/p + 1$ blocks are requested in round-robin fashion until length ℓ and then the last requested block is repeated for i times. The additive term i in the length of a phase ensures synchronization among the cores in the execution of FCFS+LRU. If all p cores start executing their corresponding light phases at the same time, then

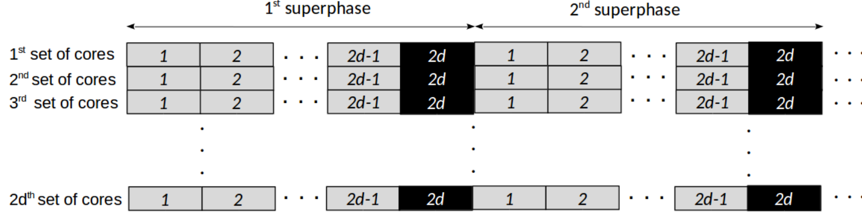

Figure 2: The execution of FCFS+LRU. Boxes with grey color represent light phases and boxes with black color represents heavy phases.

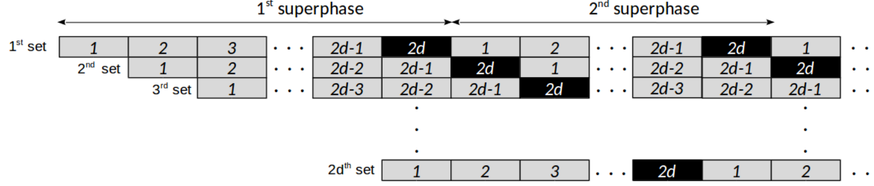

Figure 3: The execution of OPT. The sequences are shifted to align such that at most one set of cores run their heavy phases simultaneously.

all finish at the same time. Similarly, if all p threads start executing their corresponding heavy phases, then all finish at the same time. There are $2d-1$ light phases followed by a heavy phase. Together these $2d$ phases are called a superphase. Request sequence R^i is the concatenation of such a super phase for λ times.

We divide the cores into $2d$ sets each containing $\frac{p}{2d}$ cores. We assume that $2d$ divides p. Let $\mathcal{P}_1 = \{p_j | 1 \leq j \leq \frac{p}{2d}\}$ denote the first set of $\frac{p}{2d}$ cores, and $\mathcal{P}_i = \{P_j | i\frac{p}{2d} \leq j \leq (i+1)\frac{p}{2d}\}$ denote the i-th set of $\frac{p}{2d}$ cores.

OPT has enough space in its HBM to hold a block of a light phase from each core in $(2d-1)$ sets and $k/p+1$ blocks of a heavy phase from each core in one set. Recall that each set has $p/2d$ cores.

$$\left(\frac{k}{p}+1\right)\frac{p}{2d} + (2d-1)\frac{p}{2d} = \frac{k}{2d} + p.$$
$$= k/d \text{ putting } k = 2pd.$$

The execution of FCFS+LRU. As FCFS+LRU cannot see the future, it starts executing all the cores simultaneously. All p cores start and finish their corresponding light phases at the same time. Similarly, all the p cores start and finish their corresponding heavy phases at the same time. See Figure 2. As the total number of blocks of p heavy phases is larger than than size of HBM, every request in the heavy phase incurs an HBM miss. Hence, all the heavy phases run serially, thus taking at least $p\ell$ time steps. As there are λ superphases and each superphase has one heavy phase, FCFS+LRU takes at least $T(\text{FCFS+LRU}) = p\ell\lambda$ time steps to execute the whole program.

The execution of OPT. The optimal policy OPT aligns the request sequences such that no two cores run their heavy phases simultaneously. OPT does so by shifting the starting time of cores. It starts executing cores in set $\mathcal{P}_2$ when cores in set $\mathcal{P}_1$ start their 2-nd phase. Similarly, OPT starts executing cores in set $\mathcal{P}_3$ when cores in set $\mathcal{P}_2$ start their 2-nd phase. Note that cores in set $\mathcal{P}_1$ start their 3-rd phase at the same time. In general, cores in set $\mathcal{P}_{i+1}$ start when cores in set $\mathcal{P}_i$ start their 2-nd phase for $1 \leq i \leq 2d-1$. After this initial alignment, only $p/2d$ cores from one set run their heavy phase and all other cores run their corresponding light phases. As the HBM of OPT has enough space to hold all the blocks of a heavy phases from $p/2d$ cores each and a light phase from each of the rest of the cores, all the cores can progress simultaneously in every phase. See Figure 3.

OPT runs heavy phases from at most one set of $p/2d$ cores and rest of the cores runs light phases. Together, k/d blocks are fetched in k/d time steps and once all the blocks are in HBM, all p cores can run their phases in parallel, taking $l+p$ time at most. Hence, OPT finishes a phase in $(k/d+\ell+p) = \ell+3p$ time steps as $k = 2pd$. Recall that a superphase has $2d$ phases and there are λ such superphases in each request sequence. As OPT shifts the starting time of the request sequences, OPT effectively runs $\lambda+1$ superphases. OPT finishes a superphase in $(\ell+3p)2d$ time steps. Choosing an appropriate value of ℓ with respect to p, we get $(\ell+3p)2d \leq 3\ell d$. The whole program finishes in $T(\text{OPT}) = 3\ell d(\lambda+1) \leq 4\ell d\lambda$ time steps.

Competitive ratio of FCFS+LRU. Applying the makespan of FCFS+LRU and OPT, we get the following competitive ratio of FCFS+LRU.

$$\frac{T(\text{FCFS+LRU})}{T(\text{OPT})} \geq \frac{p\ell\lambda}{4\ell d\lambda} = \frac{p}{4d}$$

If FCFS+LRU gets s bandwidth augmentation, then its running time is reduced by at most a factor of s as between HBM and DRAM s blocks can be transferred in one time step. Hence, the competitive ratio of FCFS+LRU becomes $\Theta(\frac{p}{sd})$ and the theorem is proved. $\square$

6 NP-HARDNESS OF THE MAKESPAN-MINIMIZATION PROBLEM

In this section we show that the offline makespan-minimization problem is strongly NP-hard. Our proof is based on a polynomial-time reduction from the strongly NP-hard problem 3-partition.

3-partition. Given a set $A = \{a_1, a_2, \cdots, a_{3n}\}$ of $3n$ integers such that $\sum_{i=1}^{3n} a_i = nB$ and $B/4 < a_i < B/2$ for each $1 \le i \le 3n$, can $I = \{1, 2, \cdots, 3n\}$ be partitioned into disjoint sets $I_1, I_2, \cdots, I_n$, such that $\sum_{i \in I_j} a_i = B$ for each $1 \le j \le n$?

Reduction. Given an instance of the 3-partition problem with $3n$ integers $\{a_1, a_2, \cdots, a_{3n}\}$ and target sum B for each subset, we create an instance of the makespan-minimization problem as follows. For each integer a_i, we create a request sequence $R^i = \left(r_1^i, r_2^i, \ldots, r_{a_i}^i\right)^{\lfloor 4B/a_i \rfloor}$, that is R^i is formed by repeating $\left(r_1^i, r_2^i, \ldots, r_{a_i}^i\right)$ for $\lfloor 4B/a_i \rfloor$ times. Recall that r_j^i denotes the j-th request in core p_i's request sequence. Therefore, all the $3n$ request sequences together have nB distinct blocks. We also create two auxiliary request sequences T_1 and T_2 as follows.

Sequence T_1 has length $(3nB + n + 1)$ where the first $3nB + n$ requests are all distinct, but then the last request is a repeat of the penultimate request.

$$T_1 = b_1, b_2, b_3, \ldots, b_{3nB+n-1}, b_{3nB+n}, b_{3nB+n}.$$

Sequence T_2 is a concatenation of n rounds. Each round consists of two consecutive phases. In the first phase of a round, T_2 requests the same block for $(2B + 1)$ times. In second phase, it requests $2B$ distinct blocks. Thus, each round is of length $(4B + 1)$ and requests $(2B + 1)$ distinct blocks. Let $x_i = (2B + 1)i$ where $0 \le i < n$. Then,

$$T_2 = \underbrace{\overbrace{d_1, d_1, \ldots, d_1}^{2B+1}, \overbrace{d_2, d_3, \ldots d_{2B+1}}^{2B}}_{\text{Round 1}}, \ldots,$$

$$\underbrace{\overbrace{d_{x_i+1}, d_{x_i+1}, \ldots, d_{x_i+1}}^{2B+1}, \overbrace{d_{x_i+2}, d_{x_i+3}, \ldots, d_{x_i+2B+1}}^{2B}}_{\text{Round } i}, \ldots$$

There are $(3n + 2)$ cores and the HBM size is $(B + 2)$. Sequence T_1 and T_2 have $(3nB + n)$ and $(2nB + n)$ distinct block requests respectively. Sequence $R^1, R^2, \ldots, R^{3n}$ together have nB distinct block requests. The total number of distinct blocks is $(6nB + 2n)$.

Theorem 3. *An instance of 3-partition has a solution if and only if the derived makespan-minimization problem has a makespan of $(6nB + 2n + 1)$.*

Before proving Theorem 3, we first show some properties of any schedule of the derived makespan-minimization problem instance that has a makespan of $6nB + 2n + 1$.

Lemma 14. *Suppose that there is a schedule $\mathcal{S}$ with makespan $6nB + 2n + 1$. Then $\mathcal{S}$ fetches a block from DRAM to HBM in every time step except the last one.*

Proof. There are a total of $(6nB + 2n)$ distinct blocks. Hence, there must be at least $(6nB + 2n)$ HBM misses by any cache-replacement policy. If the target makespan is $(6nB + 2n + 1)$, the channel between the HBM and DRAM must always be busy except for the last time step (used for transferring the last block from HBM to a core). □

Observation 3. *Suppose that there is a schedule $\mathcal{S}$ of the derived makespan-minimization problem instance with makespan $(6nB + 2n + 1)$. Then $\mathcal{S}$ cannot evict a block from HBM unless the block is not requested in the future.*

Lemma 15. *The schedules of the auxiliary cores T_1 and T_2 are fixed in every solution of the derived makespan-minimization problem instance that has a makespan of $(6nB + 2n + 1)$.*

Proof. The target makespan is $(6nB + 2n + 1)$ and the last time step is used to serve a block from HBM. Hence, the first $(6nB + 2n)$ time steps can be used to fetch blocks from DRAM to HBM. Since T_1 requests $(3nB + n)$ distinct blocks, at every alternative time slot, T_1 must use the HBM-DRAM channel. In particular, T_1 must fetch a block in the first time step. Otherwise, it can not finish within the target makespan while fetching $(3nB + n)$ distinct blocks and serving the last distinct block twice. Hence, the schedule of T_1 is fixed: it fetches every odd timestep.

Similarly, the schedule of T_2 is fixed. In the first time step, thread T_1 brings in a block from DRAM. We show that in each of the remaining $(6nB + 2n)$ time steps, thread T_2 either brings in a block from DRAM or serves a block from HBM to its core. Recall that T_2 has n rounds. The first phase of each round takes $(2B + 2)$ steps because in the first step it brings a block from DRAM and serves the same block for $(2B + 1)$ steps. In the second phase, T_2 fetches $2B$ distinct blocks in every alternating step of $4B$ time steps. Hence, each round takes $6B + 2$ steps. All n rounds are finished in $6nB + 2n$ steps. Since T_2 starts executing at the second time step, it finishes execution in time $6nB + 2n + 1$ (Figure 4). □

Let $\mathcal{S}$ be a schedule that has makespan $(6nB + 2n + 1)$. The schedule of T_1 and T_2 are fixed. Except for the first time step, core T_2 is either fetching a block from DRAM or serving a block from HBM to the core. We say that when T_2 finishes round i, schedule $\mathcal{S}$ finishes round i. When T_2 finishes the first phase of round i, schedule $\mathcal{S}$ finishes the first phase of round i and similarly for phase 2 of round i.

Observation 4. *The auxiliary cores T_1 and T_2 always occupy one block each in HBM.*

Observation 5. *Let $\mathcal{S}$ be a schedule that has makespan $(6nB+2n+1)$. Then the first phase of each round of $\mathcal{S}$ has B time steps when the DRAM-HBM channel is used by neither T_1 nor T_2.*

Lemma 16. *Let $\mathcal{S}$ be a schedule that has makespan $(6nB + 2n + 1)$. Then no core besides T_1 and T_2 runs in two rounds of $\mathcal{S}$.*

Proof. We prove this by contradiction. Let core p run in rounds i and j. Core p can fetch blocks only during the first phase of round i. Since it continues to round j, it could not fetch all its blocks in the first phase. Otherwise, it could have finished its round-robin phase in the second phase of round i. That means some of the blocks that are fetched in round i, will be used in round j in the round-robin

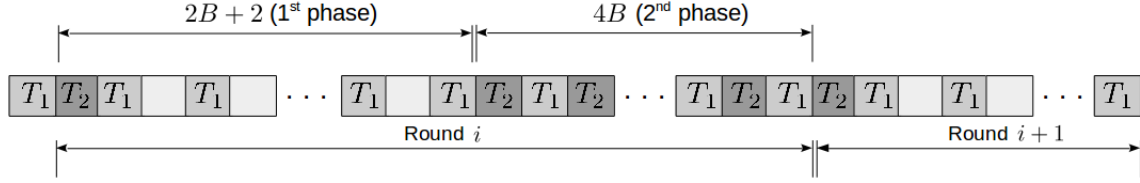

Figure 4: Each box represents a time step. Boxes with labels T_1 and T_2 denote that cores T_1 and T_2 fetch a block from DRAM respectively. A box without any label denotes that one of the $3n$ cores fetches a block from DRAM. The first phase has B boxes without labels. Each box in the second phase is labelled by either T_1 or T_2.

phase. From Observation 3, we know that once a block is fetched, it can not be kicked out unless it is not requested later. This implies that some blocks in HBM must be held for core p in round j.

There are B free time slots in round j and the HBM has B blocks left for cores other than T_1 and T_2. If some block is already occupied by some core at round j, then at least one time slot cannot bring in a block during the first phase of round j. This contradicts Lemma 14. □

Lemma 17. *Let $\mathcal{S}$ be a schedule that has makespan $(6nB + 2n + 1)$. Then exactly three cores besides T_1 and T_2 can run in a round of $\mathcal{S}$.*

Proof. No core besides T_1 and T_2 runs in two rounds. The core associated with integer a_i accesses a_i distinct blocks, so during the round where that core runs, it must read in a_i blocks during the B time slots when neither thread T_1 nor thread T_2 are accessing the DRAM. Since $B/4 < a_i < B/2$ for all a_i, at most three cores can share a round. Since there are $3n$ cores associated with integers a_i and only n blocks, then at least three blocks must run in any round. □

Proof of Theorem 3. Suppose that there is a solution to the 3-partition instance. The solution is a partition of $3n$ integers into n disjoint sets such that the sum of the integers in each set is B. We create a schedule of the derived makespan-minimization problem instance using the solution of the 3-partition instance. We schedule T_1 and T_2 as shown in Figure 4. There are n rounds in the schedule. The first phase in each round has B time-steps when neither T_1 nor T_2 use the HBM-DRAM channel. If the first set in the solution of 3-partition has integers a_i, a_j and a_k, we schedule core p_i, p_j and p_k in the first round. Recall that p_i has a_i distinct blocks. These three cores fetch a total of B blocks in the first phase of the first round and they execute the remaining round-robin accesses to these blocks by the end of the second phase of the round. The second phase has length $4B$ and the length of the request sequence of each core representing a_i is at most $4B$. Hence, p_i, p_j and p_k finish their execution in the second phase. Similarly, for each set in the 3-partition solution, we schedule the cores accordingly. As the n rounds finish in time $6nB + 2n + 1$, we have the target makespan.

Now suppose that there is a makespan of $(6nB + 2n + 1)$. Then from lemma 17, exactly three cores can run in a round. This gives a mapping from $3n$ cores to n rounds. As three cores are running in a round and total free slots (also total number of distinct blocks in these three cores) are B, this gives a 3-partition solution. Hence, *minimize-makespan* is strongly NP-hard. □

7 PERFORMANCE METRIC IN HBM MODEL

In this section we show that minimizing a traditional scheduling performance metric like makespan is better for the HBM model than minimizing the number of HBM misses. In particular we show there exist request sequences where any policy that minimizes the number of HBM misses can have arbitrarily bad running time. We prove the following theorem.

Theorem 4. *There exist p request sequences such that any policy that minimizes the number of HBM misses when serving the sequences has a makespan that is a $\Theta(p)$ factor larger than the optimal makespan.*

Proof. There are p cores and the HBM has size k. The request sequence R^i for core p_i has length $n + 2k$ and uses k distinct blocks. The first n requests are np/k round-robin repetitions of k/p blocks. Then there are two round-robin repetitions of all k blocks. We call the last two length-k subsequences ***large passes***.

$$R^i = \overbrace{\underbrace{x_1^i, x_2^i, \dots x_{k/p}^i}_{k/p}, \dots \underbrace{x_1^i, x_2^i, \dots x_{k/p}^i}_{k/p}}^{n}, \overbrace{\underbrace{x_1^i, x_2^i, \dots x_k^i}_{k}, \underbrace{x_1^i, x_2^i, \dots x_k^i}_{k}}^{2k}.$$

Because each core requests exactly k blocks in total, which exactly fills the size-k HBM, the fewest possible HBM misses is kp. This is achievable, for example, by running the request sequence for each core serially. Thus any policy that minimizes the HBM misses cannot evict a block once it is fetched from DRAM to HBM until its last reference is executed. Otherwise, there are at least $kp + 1$ HBM misses, which is not optimal.

The makespan of a single thread (i.e. when $p = 1$) is $n + 3k$. Serving the first k/p requests requires two time steps per element: one to bring the block in from DRAM and another to serve it from HBM to the core. The next n requests are HBM hits, so require one step each. These are the last $n - k/p$ requests in the first part of the sequence and the first k/p requests in the first large pass. Serving the remaining $k - k/p$ blocks in the first large pass requires two ticks each, and the final large pass requires k time steps. In all that is $2k/p + n + 2(k - k/p) + k = n + 3k$.

The last time each block is accessed by its core is during the last large pass. For a minimum-miss execution, if the threads execute in order, cores p_i and p_{i+1} can overlap for at most k time steps. Core p_{i+1} can bring in its first block when x_1^i is accessed for the last time at the start of the last large pass for R^i. In general the execution of

R^i can overlap at most k with both R^{i-1} and R^{i+1}. So each sequence overlaps for $2k$ timeslots, except the first and last, which overlap for only k time slots. Let MinimumMisses represent any policy that minimizes the number of HBM misses. Then we have.

$$T(\text{MinimizeMisses}) \geq p(n+k) + 2k.$$

There is a feasible policy that runs faster for sufficiently large n. It brings the first k/p blocks for each thread into the HBM. Because threads can interleave, this requires time at most k. Then it can execute all p threads in parallel for their first n block requests with no HBM misses. The last two rounds for each core are almost serialized. Each core requires $2k$ time to bring in its k blocks, interleaved with reading the blocks the first time. There is one more time step to reread the first block before the next core can start bringing in its blocks. Thus each core controls the DRAM channel for $2k + 1$ time steps, and there are k steps at the end for the last sequence to finish. This is a loose analysis, since threads can start executing the first round robins as soon as their blocks are in and the first core can start bringing in the rest of its first large pass as other threads are finishing their first round robins. Still we have an upper bound on the time to execute this strategy:

$$T(\text{OPT}) \leq 2k + n + p(2k+1).$$

Setting $n = p(2k+1)$ suffices to show

$$T(\text{MinimizeMisses}) \geq \frac{p}{2} T(\text{OPT}).$$

Hence, the makespan of MinimizeMisses is a $\Theta(p)$-factor larger than the optimal makespan.

□

Acknowledgments

This research was supported in part by the Laboratory Directed Research and Development program at Sandia National Laboratories, 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.

This research was also supported by NSF grants CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, CCF-1439084, CCF-1733873, CCF-1527692,CCF-1824303, CCF-1845146, and CMMI-1938909, as well as by a Google Research Award, an Infor Research Award, and a Carnegie Bosch Junior Faculty Chair.

We thank Si Hammond and Gwen Voskuilen from Sandia National Laboratories for sharing their computer architecture expertise.

8 DISCUSSION/FUTURE WORK

In this paper, we present the first theoretical discussion of automatic algorithms to manage high-bandwidth memory by (1) presenting a simple theoretical model for such memories; (2) arguing that the obvious cost metric of minimizing HBM misses is misleading; (3) showing that the far-channel arbitration policy is the key piece of using HBM efficiently and that the obvious FCFS policy does not work; and (4) designing simple but counterintuitive constant-competitive algorithms for HBM management.

Our model of HBM necessarily makes some simplifying assumptions — in particular, we assume that the *HBM channel ratio* — the ratio of the number of near channels to the number of far channels — is p, the number of cores. This gives a clean first model for analyzing the effect of uneven bandwidth. The actual HBM channel ratio, on today's architectures [26, 46, 47], is smaller (e.g. roughly 5 in the Intel Knights Landing). We argue that our model is still asymptotically realistic since the true cache hierarchy bundled with HBM increases the effective channel ratio. In real computations such as the sorting algorithms of [20], the ratio of the number of accesses to bundled HBM (true cache plus HBM), is indeed roughly p times the number of DRAM accesses. Furthermore, HBM channel ratio is likely to increase in the near future [43].

There are a number of relevant theoretical and practical questions we leave for future work. For instance, we would like to understand the practical impact of this work. Motivated by the discussion of the work by Butcher et al. [20] in Section 1.2, suppose that an HBM manufacturer like Intel had known about prioritized far-channel arbitration when they were creating the relevant system software. Might a differently-designed cache mode have given (for example) GNU::parallel a speedup of 1.5X over what it gets now? From the algorithm design perspective, we can ask the following questions: (1) Is there a solution to the makespan problem when the request sequences are not disjoint. (2) Relatedly, we currently consider the p cores to be running their own independent sequential jobs; what if the cores were running one (or more) parallel job(s)? (3) What kind of far-channel arbitration policy works with other block replacement policies, for instance, direct-mapped cache? (4) What if we made our model more general where the far channel bandwidth was asymptotically larger than 1, but still asymptotically smaller than p?

REFERENCES

- [1] High-performance on-package memory, January 2015. http://www.micron.com/products/hybrid-memory-cube/high-performance-on-package-memory.
- [2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 305–314, May 1987.
- [3] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, pages 3–28, March 1990.
- [4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.
- [5] K. Agrawal, M. Bender, R. Das, W. Kuszmaul, E. Peserico, and M. Scquizzato. Brief announcement: Green paging and parallel paging. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*, 2020.
- [6] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *Proc. 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 580–589, 1996.
- [7] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. *Algorithmica*, 32(2):277–301, February 2002.
- [8] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 197–206, 2008.
- [9] R. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory allocations. Technical report, Duke University, 1998.
- [10] M. Bender, R. Chowdhury, R. Das, R. Johnson, W. Kuszmaul, A. Lincoln, Q. Liu, J. Lynch, and H. Xu. Closing the gap between cache-oblivious and cache-adaptive analysis. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*, 2020.
- [11] M. A. Bender, J. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues. 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, May 2015.
- [12] M. A. Bender, J. W. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. *Journal of Parallel and Distributed Computing*, 102:213–228, 2017.
- [13] M. A. Bender, J. W. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. S. Resnick, and A. Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. *Journal of Parallel and Distributed Computing*, 102:213–228, 2017.
- [14] M. A. Bender, J. W. Berry, S. D. Hammond, B. Moore, B. Moseley, and C. A. Phillips. k-means clustering on two-level memory systems. In B. Jacob, editor, *Proc. 2015 International Symposium on Memory Systems, (MEMSYS)*, pages 197–205, Washington DC, USA, October 2015.
- [15] M. A. Bender, A. Conway, M. Farach-Colton, W. Jannen, Y. Jiao, R. Johnson, E. Knorr, S. McAllister, N. Mukherjee, P. Pandey, D. E. Porter, J. Yuan, and Y. Zhan. 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)*, pages 265–274, Phoenix, AZ, June 2019.
- [16] M. A. Bender, E. D. Demaine, R. Ebrahimi, J. T. Fineman, R. Johnson, A. Lincoln, J. Lynch, and S. McCauley. Cache-adaptive analysis. In *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, July 2016.
- [17] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, Portland, OR, USA, January 2014.
- [18] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510. Society for Industrial and Applied Mathematics, 2008.
- [19] A. Borodin, P. Raghavan, S. Irani, and B. Schieber. Competitive paging with locality of reference. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 249–259. Citeseer, 1991.
- [20] N. Butcher, S. L. Olivier, J. Berry, S. D. Hammond, and P. M. Kogge. Optimizing for knl usage modes when data doesn't fit in mcdram. In *Proceedings of the 47th International Conference on Parallel Processing*, page 37. ACM, 2018.
- [21] C. Byun, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, et al. Benchmarking data analysis and machine learning applications on the intel knl many-core processor. *arXiv preprint arXiv:1707.03515*, 2017.
- [22] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, et al. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115. ACM, 2007.
- [23] R. Cole and V. Ramachandran. Bounding cache miss costs of multithreaded computations under general schedulers. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 351–362, 2017.
- [24] R. Das, S.-Y. Tsai, S. Duppala, J. Lynch, E. M. Arkin, R. Chowdhury, J. S. Mitchell, and S. Skiena. 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*, pages 359–368, 2019.
- [25] A. S. de Loma. New results on fair multi-threaded paging. *Electronic Journal of SADIO*, 1(1):21–36, 1998.
- [26] D. W. Doerfler. Trinity: Next-generation supercomputer for the asc program. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2014.
- [27] E. Feuerstein and A. S. de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.
- [28] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.
- [29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual ACM Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.
- [30] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, Jan. 2012.
- [31] A. Hassidim. Cache replacement policies for multicore processors. In A. C. Yao, editor, *Proc. Innovations in Computer Science (ICS)*, pages 501–509, 2010.
- [32] M. M. Javanmard, P. Ganapathi, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury. Toward efficient architecture-independent algorithms for dynamic programs. In *International Conference on High Performance Computing*, pages 143–164. Springer, 2019.
- [33] A. K. Katti and V. Ramachandran. Competitive cache replacement strategies for shared cache environments. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 215–226. IEEE, 2012.
- [34] http://www.hpcwire.com/2014/06/24/micron-intel-reveal-memory-slice-knights-landing/.
- [35] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the" new normal"'for computer architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.
- [36] A. Li, W. Liu, M. R. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song. 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*, page 26, 2017.
- [37] A. Lincoln, Q. C. Liu, J. Lynch, and H. Xu. Cache-adaptive exploration: Experimental results and scan-hiding for adaptivity. In *Proc. 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 213–222, 2018.
- [38] A. López-Ortiz and A. Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 113–127. ACM, 2012.
- [39] http://nnsa.energy.gov/mediaroom/pressreleases/trinity.
- [40] E. Peserico. Paging with dynamic memory capacity. *CoRR*, abs/1304.6007, 2013.
- [41] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.
- [42] S. S. Seiden. Randomized online multi-threaded paging. *Nordic Journal of Computing*, 6(2):148–161, 1999.
- [43] Semiconductor Engineering. What's next for high bandwidth memory? https://semiengineering.com/whats-next-for-high-bandwidth-memory/, December 2019.
- [44] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985.
- [45] G. M. Slota and S. Rajamanickam. Experimental design of work chunking for graph algorithms on high bandwidth memory architectures. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 875–884. IEEE, 2018.
- [46] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.
- [47] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, and S. Parete-Koon. Announcing supercomputer summit. Technical report, ORNL (Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States)), 2016.