Managing High-Bandwidth Memory is a Parallel Scheduling Problem (full paper only)

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

Michael A. Bender
Stony Brook University
Stony Brook, New York, USA
bender@cs.stonybrook.edu

Kirk Pruhs
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
kirk@cs.pitt.edu

Benjamin Moseley
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
moseleyb@andrew.cmu.edu

Clifford Stein
Columbia University
New York City, New York, USA
cliff@ieor.columbia.edu

ABSTRACT

High-Bandwidth Memory (HBM) is a decade-old memory technology that is increasingly commonly being used in highly-parallel machines such as GPUs and multicores. Comparatively, HBM has higher bandwidth, smaller capacity, and similar latency to other DRAM technologies. Many systems use both HBM and other DRAM technologies, where HBM is naturally closer to the processor in the conceptual memory hierarchy. Thus, a natural resulting question is how one should best manage a collection of processes running on a HBM/DRAM memory hierarchy. Prior work introduced a theoretical model for addressing this question, and gave a competitive policy for the objective of minimizing makespan. Our main technical contribution is to give a competitive policy for the more commonly appropriate total/average response/completion time objective. However, we believe the broader, and more important contribution, is to make explicit the case (hinted at in the prior literature) that managing an HBM/DRAM hierarchy should be thought of as a parallel scheduling problem. To that end, we introduce a new online scheduling model that we call the *semi-normal* model. We then show how to use a competitive algorithm for scheduling in the semi-normal model as a black box to obtain a competitive algorithm for managing a HBM/DRAM memory hierarchy. Thus, as a result of this black-box conversion, competitiveness results in the semi-normal model translate (essentially) automatically into competitiveness results in the HBM/DRAM management model. Our main technical result is then an application of such a translation. That is, we show that a natural variant of the Round Robin (processor sharing) algorithm, naturally adapted for the seminormal model, is competitive for the objective of average/total completion time. Thus, we obtain an algorithm for managing a HBM/DRAM hierarchy that is competitive for the objective of average/total completion time, using this black-box reduction.

SPAA '25, July 28–August 1, 2025, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/25/07.
https://doi.org/10.1145/3694906.3743336

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:
Kunal Agrawal, Michael A. Bender, Kirk Pruhs, Benjamin Moseley, and Clifford Stein. 2025. Managing High-Bandwidth Memory is a Parallel Scheduling Problem (full paper only). In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3694906.3743336

1 INTRODUCTION

High Bandwidth Memory (HBM) is a decade-old computer memory interface technology for 3D-stacked synchronous dynamic random-access memory (SDRAM). HBM is commonly used with high-performance graphics accelerators, network devices, high-performance datacenter AI Application-specific integrated circuits, and in some supercomputers (such as the NEC SX-Aurora TSUBASA and Fujitsu A64FX) [33]. Comparatively, HBM has higher bandwidth, smaller capacity, and similar latency to other DRAM technologies. Many systems use both HBM and other DRAM technologies, where HBM is naturally closer to the processor in the conceptual memory hierarchy. Thus, a natural research question is how to best manage an HBM/DRAM memory hierarchy.

1.1 Prior Foundational Algorithmic Work

Das et al. [17] initiated the first foundational algorithmic investigation into managing a HBM/DRAM hierarchy. Their model is depicted in Figure 1. It has p processing cores, each with a unit-speed channel to HBM allowing up to p memory requests to be fulfilled by HBM on each time step. The HBM can store k blocks/pages. There is also a single unit-speed channel, called the *far channel*, between HBM and DRM. Therefore, if multiple cores try to access data which

doesn't reside in HBM, these requests have to be serialized on the far channel.

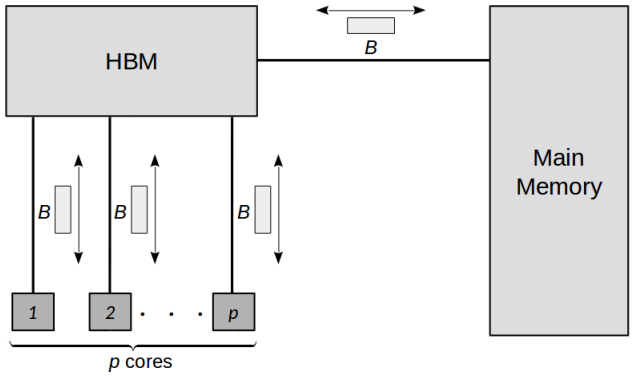

Figure 1: The HBM model with p cores and two levels of memory.

Das et al. [17] then observed that the natural way to manage a HBM/DRAM hierarchy is to have a *page replacement policy* and *far channel arbitration policy*. The page replacement policy determines the page that should be evicted from a full HBM when a new page is brought into HBM. The far channel arbitration policy determines which memory request is fulfilled first if there are multiple outstanding requests for memory blocks/pages that are only stored in DRAM. Managing traditional latency-heterogeneous memory hierarchies only requires a page replacement policy (and not a far channel arbitrary policy). There is extensive literature examining research page replacement policies for latency-heterogeneous memory hierarchies, but generally speaking the Least Recently Used (LRU) policy has emerged as the best policy [25].

Das et al. [17] then introduced the following model for generating memory accesses: Each of p cores executes an independent process, where each process consists of a sequence of requests to pages/blocks that are disjoint from the pages/blocks requested by the other processes.

Initially it might be natural to intuit that managing a HBM/DRAM hierarchy should be similar to managing a traditional latency-heterogeneous memory hierarchy, and thus the "right" policies should be similar. Das et al. [17] partially adopted and partially rejected this intuition. They partially rejected this intuition by observing that the traditional objective for latency-heterogeneous memories, namely minimizing the number of accesses to the far memory, is not appropriate for a HBM/DRAM hierarchy. Instead, they adopted a standard scheduling objective, namely minimizing makespan, which is the time that the last request is fulfilled [1], designed a HBM/DRAM management policy, and showed that this policy is (asymptotically) competitive for the makespan objective using techniques from the parallel scheduling literature. They seemingly partially adopted the intuition when they chose the page replacement policy to be Least Recently Used (LRU). The use of LRU page replacement policy seems to have been inherited by default from the prior literature on managing latency heterogeneous memory hierarchies, and was seemingly not the result of principled algorithm design.

1.2 Our Contributions

The starting point for our research is the observation that in many use cases for HBM/DRAM memories, makespan is clearly not an appropriate objective. For example, in a setting where processes are spawned by different users, average response time is a more appropriate objective. Further, the HBM/DRAM management policy from [17], which uses a far channel arbitration policy based on process ID, will perform poorly on this objective — in the worst case, the competitive ratio obtained obtained by these policies can be $\Omega(p)$ where p is the number of channels to HBM. Thus our original narrow goal was to develop and analyze competitive HBM/DRAM management policies for the average response time objective. However, we also have a broader goal, which is to try to make explicit the case that managing an HBM/DRAM hierarchy should be thought of as a parallel scheduling problem (which is only implicitly hinted at in [17]). And thus, the design and analysis of the management policies should be built upon algorithmic parallel scheduling techniques/results.

In a traditional parallel scheduling problem, one considers jobs which must be processed using processors. Any good parallel scheduling policy for the total completion time objective should prioritize shorter jobs and jobs that are more parallelizable (have smaller width). In this paper, we will relate this insight by mapping processes in the HBM/DRAM management algorithm to jobs in a parallel scheduling problem. Since HBM/DRAM management algorithm does not know typically lnow the lengths and parallelizability of the processes a priori, in order to work well, it is going to have to rotate jobs through the HBM to search for short and/or parallelizable processes. The far channel could be a bottleneck for moving the working sets of new processes into HBM, and thus it is natural for an HBM/DRAM management algorithm to conceptually process a job in *chunks*, where a chunk is large enough so that the cost/delay incurred by preemption can be amortized away.

In Section 3, we start by showing that there is always a competitive near-optimal HBM/DRAM policy that processes blocks in chunks. [2] Thus after this our algorithm analysis need only compare to the optimal chunked schedule.

In Section 4, we introduce a new online scheduling model, that we call the *semi-normal* model. In this model, each of p jobs consists of a sequence of chunks. Let x_j be the number of chunks in job j, and we use $C_{i,j}$ to denote chunk $j \in [1, x_j]$ of job j. Each chunk $C_{i,j}$ has an integer width $w_{i,j} \in [1, k]$. These chunks must be processed sequentially, so chunk i of job j can not be processed until after chunk $i-1$ of job j is processed. We say a chunk $C_{i,j}$ becomes *ready* after $C_{i-1,j}$ has been processed. A feasible schedule can process any collection of ready chunks with total width is at most k. In this way, the model is like scheduling on a multi-processor system. An online algorithm A does not a priori know anything about the number of chunks in a job, or the widths of the chunks. At

[1] Minimizing makespan and minimizing the number of accesses to far memory are essentially equivalent objectives in a latency heterogeneous memory hierarchy.

[2] [17] also showed such a result for makespan, but that construction was specific for makespan, and doesn't work for the sum of completion time objective.

each time t the online algorithm A can specify a permutation π_t of the chunks that are ready to be processed. Then a maximum prefix with aggregate width at most k is processed. The online algorithm A then learns the widths of the processed chunks, and for each chunk whether it was the last chunk in the job or not. We show that any online algorithm A in the semi-normal model can be converted in a black-box manner into an algorithm B to manage a HBM/DRAM hierarchy where for every job/process j the response time for j in A's schedule is essentially the same as the response time for j in B's schedule. Thus, as a result of black-box conversion, competitiveness results in the semi-normal model translate (essentially) automatically into competitiveness results in the HBM/DRAM management model. This is the first part of our case that HBM/DRAM managment should be thought of as a parallel scheduling problem.

The second part of our case is that the "semi-normal" scheduling model is sufficiently close to standard parallel scheduling models that we believe that one should be able to adopt many results over from standard parallel scheduling models to the semi-normal model. As one piece of supporting evidence for this second part of our case, in Section 5 we show that a variant of the Round Robin (processor sharing) algorithm, naturally adapted for the seminormal model, is competitive for the objective of average/total completion time, with the modest assumption that the aggregate width of the chunks in every job is at least k. [3] Further, the analysis is quite similar to the analysis of Round Robin in the standard "speedup curves" parallel scheduling model [30].

2 DEFINITIONS

Here we give some definitions, that the reader may refer to as needed.

Recall that the HBM model (formulated by Das et. al [17]) comprises a multi-core machine with p cores. Each core has a dedicated channel to HBM (these channels are called ***near channels***) while there is only one channel between HBM and DRAM (this channel is called the ***far channel***). Data is transferred along all these channels at the granularity of a block. It takes one unit time to transfer a block along any of these channels (from HBM to a core or from DRAM to HBM). HBM can hold k blocks at any time while DRAM (main memory) has no space limitation. This model captures three important characteristics of HBM. (1) Higher bandwidth between HBM and processors due to p channels between HBM and the cores vs 1 channel from DRAM to HBM. (2) Limited capacity (k blocks) of HBM. (3) Comparable latency between HBM and DRAM since the cost of each channel is 1.

We assume that each core runs its own program (process). The process running on core p_i is denoted by T_i. The instructions of the process generate a sequence of block requests. We denote $T^i = r_0^i, r_1^i, r_2^i, \ldots, r_{\rho(i)}^i$ as the sequence of the blocks requested by core p_i on its dedicated channel to the HBM. Therefore, process T_i has a total of $\rho(i)$ requests. As in Das et. al [17], we assume that each core is running a different process; therefore, the blocks accessed by each core are disjoint from all other cores. That is, if $i \neq j$, then $\forall q, s\ r_q^i \neq r_s^j$. Further, as in Das et. al [17], for simplicity we assume that processes are "long enough" — that is, the number of block requests in each request sequence is at least k, the size of HBM.

At any time t, if core p_i requests block r_q^i and this block is present in HBM, then this page is transferred to the core on its dedicated near channel from HBM and this request is *satisfied or served at time* t. The core will then move on to the next request in its sequence on time step $t+1$. If, on the other hand, the block r_q^i is not in HBM at time t, then the block must be retrieved from DRAM via the far channel. Therefore, satisfying this request may take several additional time units since many other requests may be contending for access to the far channel. How much time it takes to access this page depends both on how much contention there is for access to the far channel and the arbitration policy for the far channel. When a new block is brought into the HBM, some existing block will need to be evicted if HBM was full. The identify of the page that is evicted us determined by the block replacement policy.

We assume that all processes are available at time 0. The completion time of each job depends on the particular HBM policy. We define F_i^S to be the response/completion time of job T_i for policy S, which is the time that the last request $r_{\rho(i)}^i$ is processed. When it is clear from context, we will omit the superscript S.

The makespan is then the maximum completion time, that is, $\max_i F_i^S$. The total completion time is $\sum_{i=1}^{p} F_i^S$.

We will use *competitive analysis* to characterize the performance of HBM algorithms. There are two related notions of competitive analysis. Consider a metric (such as sum of completion times of all jobs), and a particular HBM policy S. Say that $M_S(\sigma)$ is the performance of S on an input σ — here the input is the set of requests generated by the processes and $M_O(\sigma)$ is the performance of OPT on the same input. We say that S is α-competitive (in the *absolute sense*) for the metric if for all valid inputs σ, we have

$$M_S(\sigma) \leq \alpha M_O(\sigma)$$

We say that S is α-competitive (in the *asymptotic sense*) on the metric if we for all valid inputs,

$$M_S(\sigma) \leq \alpha M_O(\sigma) + \gamma$$

where γ is a parameter that does not depend on the input. Note that γ may depend on the parameters of the machine — in this case, it may depend, for instance on the HBM size k or the number of processes p.

3 NEAR OPTIMAL CHUNKED SCHEDULES

In this section we show that there is always an approximately optimal chunked schedule. We start with some definitions, some of which formalize concepts used in Section 1.

Definition 1.

- *Process T_i is a sequence $T_i = r_1, r_2, r_3, \ldots, r_{\rho(i)}$ of requests, where each request specifies a block.*
- *A request r_j in process T_i is processed at a time t if, requests $r_1, \ldots, r_{j-1}$ have been previously processed, and block r_j is in HBM.*

[3] Some assumption like this is unavoidable as there is no way for a nonclairvoyant algorithm to find short narrow jobs quickly. Consider an instance with k jobs of length and width 1, and $\sqrt{k}$ jobs of length 1 and width k.

- *A schedule S specifies for each time, the blocks that are processed, the blocks that are evicted from HBM, and possibly up to one block that is copied from DRAM to HBM.*
- *F_i^S is the response/completion time of process T_i in schedule S, which is the time that the last request $r_{\rho(i)}$ is processed.*
- *We partition each process T_i into chunks, where each chunk (except possibly the last) contains exactly $k/4$ consecutive requests.*
- *Let C_{ij} be the j^{th} chunk in process T_i. So more graphically,*

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

  (This is overloaded to also represent the the jth chunk of the ith job in the semi-normal model).
- *A chunk C_{ij} is ready to be processed as soon as $C_{i,j-1}$ has been processed, and a chunk C_{ij} is processed when all the blocks in that chunk have been processed.*
- *Let $\mathcal{U}(C_{ij}, S, t)$ be the number of distinct unfinished blocks in chunk C_{ij} in schedule S at time t.*
- *Define the working set of a collection $\mathcal{C}$ of chunks, denoted $\mathcal{W}(\mathcal{C})$, to be the set of blocks in $\bigcup \mathcal{C}$. So $|\mathcal{W}(\mathcal{C})|$ is the number of distinct blocks in the chunks in $\mathcal{C}$.*

3.1 The Optimal Schedule

Let *OPT* denote the optimal schedule. We partition *OPT* into stages. The durations of the stages partition time into intervals of length of exactly $k/4$ time steps. We now observe some properties of *OPT*.

Observation 1. *For every process T_i, and for every stage σ, the number of blocks processed by OPT during stage σ is at most $k/4$.*

Proof. This follows because a stage lasts at most $k/4$ time units, and for each process at most one request can be processed in one time unit. □

Observation 2. *For every process T_i, and for every stage σ, there are at most two chunks in T_i that contain a block that is processed during stage σ.*

Proof. This follows because a stage lasts for $k/4$ time units, processing a block takes unit time, and each chunk (except maybe the last one) contains $k/4$ requests. □

Observation 3. *The number distinct blocks processed during a stage of OPT (by all processes) is at most $5k/4$.*

Proof. To be processed during a stage, a block either has to be in HBM at the start of the stage, or be copied from DRAM to HBM at some point during the stage. Then observe that at the start of the stage there are at most k distinct blocks already present in HBM. Then observe that, as a stage lasts at most $k/4$ units of time, and as at most one block can be copied from DRAM to HBM each unit of time, it must be the case that at most $k/4$ blocks can be copied from DRAM to HBM during a stage. □

Observation 4. *During a stage σ of OPT, it must be the case that the sum over the chunks $C_{i,j}$ of the number of distinct unprocessed blocks in $C_{i,j}$ decreases by at most $10k/4$. That is,*

$$\sum_{i=1}^{n} \sum_{C_{i,j}} \left(\mathcal{U}(C_{i,j}, OPT, (k-1)\sigma/4) - \mathcal{U}(C_{i,j}, OPT, k\sigma/4) \right) \leq 10k/4$$

Proof. This is an immediate consequence of Observation 2 and Observation 3. □

3.2 An Approximately Optimal Chunked Schedule

In this subsection we establish that there is a chunked schedule *AOPT*, whose cost is at most an $O(1)$ factor more than the cost of $OPT + kn$. A chunked schedule consists of a sequence of phases, where the durations of the phases partition time into intervals. Further, during each phase ϕ, a collection C_ϕ of chunks are processed, where these processed chunks have the following properties:

- Each chunk in C_ϕ is ready to run at the start of the phase.
- No process has more than one chunk in C_ϕ.
- The size of the working set of the chunks in C_ϕ is at most k, that is $|\mathcal{W}(C_\phi)| \leq k$.

Construction of *AOPT*: We now explain how to construct a chunked schedule *AOPT* from the optimal schedule *OPT*. Assume without loss of generality that the processes are numbered in nondecreasing order of completion time in *OPT*. We create *AOPT* one phase at a time, starting from the first phase. To create a phase let h be maximal such that the working set size of the chunks in $T_1, \ldots, T_h$ that are ready to run at the start of the phase is at most k. Then in the next phase, these ready to run chunks in $T_1, \ldots, T_h$ in are processed.

Let E_ϕ be the time when phase ϕ ends, and phase $\phi + 1$ begins in *AOPT*. We now make a couple of observations about properties of *AOPT*.

Observation 5. *The length of each phase ϕ in AOPT is at most $5k/4$, that is $E_\phi - E_{\phi-1} \leq 5k/4$.*

Proof. Because $\mathcal{W}(C_\phi) \leq k$, there can be at most k units of time where a block is copied from DRAM to HBM. If no block is transferred from DRAM to HBM at some unit of time, then it must be the case that the next request in each chunk in C_ϕ is to a block currently in HBM. Thus as each chunk contains at most $k/4$ requests, there can be at most $k/4$ time periods where no new block is transferred from DRAM to HBM. □

Lemma 1. *The cost of the schedule AOPT is bounded as follows:*

$$\sum_{i=1}^{n} F_i^{AOPT} \leq \frac{65}{3} \left(\sum_{i=1}^{n} F_i^{OPT} + k/4 \right).$$

Proof. Consider an arbitrary process T_h. We will show that $F_h^{AOPT} \leq \frac{65}{3} F_h^{OPT}$. Toward this end, define a phase ϕ of *AOPT* to be under utilized if a chunk of each process T_i, $i \in [h]$, is processed during ϕ, and fully utilized otherwise.

The number of under utilized phases in *AOPT* is at most the number of chunks in T_h, which in turn is at most the number of stages until T_h is completed in *OPT*.

Now let us consider a fully utilized phase ϕ in *AOPT*. Since ϕ was fully utilized, the chunks in C_ϕ that are processed during this

phase come from some processes $T_1, \ldots, T_j$ where $j < h$. Since the chunk in T_{j+1} that was ready to run at the start of ϕ has at most $k/4$ distinct blocks, the number of distinct blocks in the chunks in $T_1, \ldots, T_j$, that were ready to run at the start of ϕ, is at least $3k/4$. That is, $|\mathcal{W}(C_\phi)| \geq 3k/4$. Thus during the phase ϕ it must be the case that the sum over the chunks $C_{i,j}$, $i \in [h]$, of the number of distinct unprocessed requests in $C_{i,j}$ decreases by at least $3k/4$. That is,

$$\sum_{i=1}^{h} \sum_{C_{i,j}} \Big(\mathcal{U}(C_{i,j}, AOPT, E_{\phi-1}) - \mathcal{U}(C_{i,j}, AOPT, E_\phi) \Big) \geq 3k/4 \, .$$

T_h completes in OPT during the first stage σ where

$$\sum_{i=1}^{h} \sum_{C_{i,j}} \mathcal{U}(C_{i,j}, OPT, k\sigma/4) = 0 \, . \tag{1}$$

By Observation 4 the value of the expression in equation (1) decreases by a rate of at most $10k/4$ per stage. Thus the rate that $AOPT$ is reducing

$$\sum_{i=1}^{h} \sum_{C_{i,j}} \mathcal{U}(C_{i,j}, OPT, \cdot) \tag{2}$$

on a fully utilized phase, is at least $3/10$ of the rate that OPT is reducing this term per stage. Thus the number of fully utilized phases in $AOPT$ until T_h completes at most 10/3 times the number of stages until T_h completes in OPT.

Thus combining the analysis of under utilized and fully utilized phases in $AOPT$, we can conclude that the number of phases in $AOPT$ until T_h completes is at most $1 + 10/3 = 13/3$ times the number of stages until T_h completes in OPT. As we know from Observation 5 that the duration of each phase in $AOPT$ is of length at most 5 times the duration of a stage in OPT. Thus taking into account that OPT may complete T_h right at the start of a stage, we can conclude that

$$F_h^{AOPT} \leq \frac{65}{3}(F_h^{OPT} + k/4) \, .$$

□

4 REDUCTION TO THE SEMINORMAL MODEL

We start by designing an HBM/DRAM management algorithm B that uses an online seminormal scheduling algorithm A as as black box. Algorithm B produces a chunked schedule where phase ϕ of this chunked schedule matches the schedule produced by A at time ϕ.

Definition of Algorithm B: The algorithm B creates one job in the seminormal model for each of the p processes that it has. The algorithm B creates a chunked schedule. One phase of algorithm B's chunked schedule will correspond to a unit time step for algorithm A. We how explain how B operates during an arbitrary phase ϕ.

The algorithm B then simulates the algorithm A to determine the permutation π that B uses for time ϕ. All processes start phase ϕ classified as qualified, and their status changes to disqualified once a chunk of the process has been processed in this phase. During phase ϕ, algorithm B's far channel arbitration policy is to always give priority to the earliest qualified process in the π order, and algorithm B's page replacement policy is first evict an arbitrary block from an arbitrary disqualified process, and if that is not possible, evict an arbitrary block from the qualified process that is latest in the π order. So both policies are priority based, where the priorities are derived from π. Algorithm B ends a phase after $5k/4$ time steps.

Let ℓ be maximum such that, the chunks for processes $T_{\pi(1)}, \ldots, T_{\pi(\ell)}$ that were ready at the start of the phase, have aggregate working set size at most k. Let C be the collection of those chunks. As will be shown in Lemma 2, algorithm B will process all of the chunks in C. Algorithm A is then informed that during step ϕ it completed the ready chunks from jobs $\pi(1), \ldots, \pi(\ell)$, and informed that the width of the ready chunk from job $\pi(j)$ is the working set size of the chunk of $\pi(j) \in C$ that B processed during this phase.

During this phase algorithm B may have processed some blocks not in C. So that algorithm B can continue to perfectly mimic algorithm A, before the phase ends, algorithm B "forgets" that it processed any blocks not in C. So algorithm B starts phase $\phi + 1$ as if the only blocks processed during phase ϕ were those in C.

We can now state some properties of the HBM algorithm constructed in this manner.

Lemma 2. *During phase ϕ, algorithm B processes the chunks in C.*

Proof. Since the aggregate working set size of the chunks in C is at most k, there are are most k time units when a block in C is transferred from DRAM to HBM. Since $\pi(1) \ldots \pi(\ell)$ are the highest priority processes, they do not wait on any other processes on the far channel. Therefore, each of the processes $T_{\pi(1)}, \ldots, T_{\pi(\ell)}$ has at least $k/4$ time units after its ready chunk is fully in HBM to transfer this chunk to the appropriate core. Since a chunk has exactly $k/4$ accesses, this is sufficient to process the chunk. □

Now before analyzing the competitiveness of algorithm B we need some definitions.

Definition 2.

- *Let I be an arbitrary instance of the HBM/DRAM management problem.*
- *Let I_{sn} be the instance of the semi-normal scheduling problem created by algorithm B.*
- *Let $B(I)$ be the schedule produced by algorithm B on instance I, and its objective value.*
- *Let $A(I_{sn})$ the schedule produced by algorithm A on instance I_{sn}, and its objective value.*
- *Let $OPT(I)$ be the optimal schedule on instance I, and its objective value.*
- *Let $OPT(I_{sn})$ be the optimal schedule on instance I_{sn}, and its objective value.*

Lemma 3. $B(I) \leq 5k \; A(I_{sn})/4$.

Proof. Note that if the response time for a job j in the schedule produced by algorithm A is f_j, then the response time for process T_j in B's schedule is at most $5kf_j/4$. This follows from the fact that B mimics A, and the fact that a phase for B lasts at most $5k/4$ time units. □

Lemma 4. $k\ OPT(I_{sn}) \leq OPT(I) + 5kn/4$

Proof. Let S be an arbitrary chunked schedule for an arbitrary instance I of the HBM/DRAM memory management problem. Let f_j be the phase that process T_j completes in S. Then there is a schedule S' for I_{sn} where job j completes at time f_j. If a chunk C_{ij} of a particular process T_i executes in a phase ϕ of S, then the corresponding chunk of job J_i is scheduled at time ϕ in S'. Since the total working set of the chunks that are scheduled in a particular phase by HBM's chunked schedule is at most k, all the corresponding chunks in the semi-normal can be feasibly schedule. The k factor in front of $OPT(I_{sn})$ derives from the fact that a phase of S takes at least k time units (other than the last phase). The additive $5kn/4$ derives from the fact that $OPT(I)$ may complete a process right at the start of a phase, so may not incur additional waiting time for its last phase, which can last up to $5k/4$ time units. □

Theorem 1. *Assume algorithm A is α-competitive for total completion time then algorithm B is $(\frac{5}{4}\alpha + \frac{5^2}{4^2}\ \alpha)$-competitive for HBM/DRAM management for total completion time.*

Proof. Note that

$$B(I) \leq 5k\ A(I_{sn})/4 \tag{3}$$
$$\leq 5\alpha k\ OPT(I_{sn})/4 \tag{4}$$
$$\leq 5\alpha\ OPT(I)/4 + \frac{5^2}{4^2}\ \alpha kn \tag{5}$$
$$\leq (\frac{5}{4}\alpha + \frac{5^2}{4^2}\ \alpha)\ OPT(I) \tag{6}$$

Line (3) follows from Lemma 3. Line (4) follows from the fact that A is α-competitive. Line (3) follows from Lemma 4. Line (6) follows from our assumption that each process has at least k blocks, and thus $OPT(I)$ is at least kn. □

5 ROUND ROBIN IN THE SEMI-NORMAL MODEL

Here we consider scheduling in the seminormal model a collection of n jobs that arrive at time 0 with the objective to minimize the total completion time. The goal of this section is to provide evidence that, while the seminormal model is not identical to any known parallel model, algorithms and analysis techniques from other models are likely to be applicable to the seminormal model with relevant modifications. Towards this goal, we will analyze the simple round-robin algorithm which is known to be constant competitive for other parallel scheduling models for minimizing total completion time.

Because of the nonclairvoyant nature of the seminormal model, we need to make some assumptions about all jobs being not too small in order to obtain a reasonable competitiveness result. To see this consider an instance where there are k skinny jobs, each consisting of one chunk with width 1, and $\sqrt{k}$ fat jobs with width k. Then the optimal total completion times is to run the skinny jobs first, resulting in a total completion time of $\Theta(k)$. But any seminormal scheduling algorithm which doesn't know the width in advance will only be able to discover about $\sqrt{k}$ skinny jobs per unit of time, resulting in total completion time of $\Omega(k^{3/2})$. Drawing motivation from this example, we will assume that the aggregate width of every job is at least k.

The underlying principle of Round Robin scheduling is to share the processing capabilities evenly between the jobs waiting to be scheduled. As it is not possible in the semi-normal for all jobs to be processed at the same time, the sharing of the processor has to use time-multiplexing. In some sense the main intuitive insight is that in the setting of seminormal scheduling, where the jobs can have different widths, the "right" implementation of Round Robin is to maintain the invariant that the aggregate widths processed are the same for every job. So for example, if job i has constant width w and job j has constant width $2w$, then the "right" strategy for the online algorithm is to process i twice as often as j. Unfortunately, strictly maintaining this invariant would necessarily result in a noncompetitive algorithm since it could force most of the processors to be idle most of the time while they wait for low width job to catch up. Thus our implementation of Round Robin, which we call Semi-Normal Round Robin (SNRR) will attempt to balance the aggregate widths processed on the jobs as much as possible by favoring the jobs that have been processed the least to date.

We initially assume speed augmentation augmentation. That is, we assume SNRR has a speed $s = 2$ processor, meaning it can process up to aggregate width sk during a unit time step, and then compare the total completion time for SNRR to the optimal completion time for a speed one processor that process up to aggregate width k in a unit time step. The assumption of speed augmentation aids in keeping both the definition of the algorithm, and the analysis, relatively clean.

Semi-Normal Round Robin (SNRR) Algorithm Description: Let $W_j(t)$ be the aggregate width that SNRR has processed on job j before time t. At each time step t, the priority ordering π_t is by decreasing order of $W_j(t)$.

Before beginning our analysis we need some definitions.

Definition 3.

- *Let $U_R(t)$ be the collection of jobs unfinished by SNRR, with a speed $s = 2$ processor, at time t.*
- *Let let $n_R(t) = |U_R(t)|$.*
- *Let $U_O(t)$ be the collection of jobs unfinished in the optimal schedule for a unit speed processor at time t.*
- *Let $n_O(t) = |U_O(t)|$.*

Then our main technical lemma is the following.

Lemma 5. *Let $c = 2$ and $\gamma = 9$. Then for all t, $n_R(t) \leq cn_O(\gamma t)$ for $c = 2$.*

Proof. Assume to reach a contradiction that for some t it is the case that $n_R(t) > cn_O(\gamma t)$. We break the proof into cases.

Case 1: Assume $t \leq n/2$. Then by the assumption that the aggregate width of each job is at least k, $n_O(t) \geq n/2$. So the result

holds (as long as $c \geq 2$) as clearly $n_R(\gamma t) \leq n$. So from now on we assume $t \geq n/2$.

Case 2: We define a time t to be uncongested if the aggregate width processed by SNRR at time t is at most $(s-1)k$. Case 2 is when the number of uncongested times before time γt is at least t. Since SNRR processes a chunk from every unfinished job at each uncongested time, in this case, SNRR has processed every chunk that the optimal schedule processed by time t. Thus $n_R(\gamma t) \leq n_O(t)$.

Case 3: This is the interesting case where the number of uncongested times before time γt is at most t. Thus the aggregate width processed by SNRR by time γt is at least $(\gamma-1)(s-1)kt$. The aggregate width of the jobs the optimal schedule completed by time t (the complement set of $U_O(t)$) can be at most kt. Thus by time γt it must be the case that SNRR processed aggregate width of at least $((\gamma-1)(s-1)-1)kt$ on jobs in $U_O(t)$.

Define another constant $\delta = 2$ and define time u to be the latest time when it was the case that the aggregate width processed by SNRR on jobs in $U_O(u)$ is less than δkt. So between time u and time γt the aggregate width that SNRR processed from jobs in $U_O(t)$ is at least

$$((\gamma-1)(s-1)-1)kt - \delta kt$$

We now want to argue that most of the processing that SNRR did during the time interval $[u, t]$ was on low priority jobs, where a job j is low priority at time τ if $W_j(\tau) \geq \delta kt/n_O$. Note that, due to this definition, a job j can do at most $\delta kt/n_O + k$ work before it becomes low priority. Once a job is a low-priority job, it remains low priority for the purposes of this definition. Therefore, in aggregate, over all jobs in U_O, the total aggregate width processed by these jobs while they are high priority is at most $n_O \cdot (\frac{\delta kt}{n_O} + k)$.

For job $j \in U_O(t)$, let $\beta(j)$ be the aggregate width SNRR processes on this job while it was low priority. Let β be the sum of $\beta(j)$'s for all jobs j in $U_O(t)$. Therefore

$$\beta \geq ((\gamma-1)(s-1)-1)kt - n_O \cdot \left(\frac{\delta kt}{n_O} + k\right)$$

We now claim that there is at least one job $j^* \in U_R(\gamma t) - U_O(t)$ with aggregate width at most $\frac{\delta kt}{n_O}$. If this were not the case then the optimal schedule would have had to process $(cn_O - n_O)\frac{\delta kt}{n_O}$ aggregate width by time t, which would be a contradiction since $\delta > 1$ and optimal schedule can only process kt aggregate width by time t.

We now want to argue that SNRR processed j^* for at least t time units before time γt, which contradicts $j^* \in U_R(\gamma t) - U_O(t)$. Since j^*'s aggregate width is smaller than the threshold for becoming low priority, j^* was high priority at all times between time 0 and time γt. Therefore j^* had a chunk processed at each time τ where a low priority job was processed since low proiority jobs are only processed if there is processing capacity left after all high priority jobs have been allocated. As SNRR has an s speed processor, it can process at most aggregate width sk in a time step. Thus SNRR ran j^* for at least $\beta/(sk)$ time steps. Now plugging in our prior lower bound for β, and doing some simple algebra, we obtain:

$$\begin{aligned}
\beta/(sk) &\geq \frac{((\gamma-1)(s-1)-1)kt - n_O \cdot \left(\frac{\delta kt}{n_O} + k\right)}{sk} \\
&= \frac{((\gamma-1)(s-1)-1-\delta)t - n_O}{s} \\
&\geq \frac{((\gamma-1)(s-1)-1-\delta)t - 2t}{s} \\
&> t
\end{aligned}$$

The second inequality comes from our assumption that $t \geq n/2$ this case, and and the obvious fact that $n_O \leq n$. The last inequality follows from the fact that $s = 2$, $\delta = 2$, and $\gamma = 9$. This contradicts the fact that j^* was finished by time t in the optimal schedule. □

Lemma 6. *Let $\gamma = 9$. SNRR with a speed 2 processor is an 8γ-competitive algorithm for the objective of total completion time.*

Proof. Let $\gamma = 9$. Let t_i be time 2^i. Let n_i be the number of jobs incomplete in the optimal solution at time t_i. Let $t_i^R = \gamma 2^i$ and let n_i^R be the number of jobs alive at this time in SNRR's schedule. In this proof, we assume that jobs completed by SNRR between t_{i-1}^R and t_i^R are completed at t_i^R, only increasing SNRR's total completion time. Therefore, $n_{i-1}^R - n_i^R$ jobs are alive for time t_i^R for all i. Similarly, we assume that jobs completed by the optimal solution during $(t_{i-2}, t_{i-1}]$ are completed at time t_{i-2} as this only reduces the cost of the optimal solution.

Therefore, the total completion time of SNRR is at most $\sum_{i=1}^{\infty}(n_{i-1}^R - n_i^R)t_i^R$ and the total completion time for the optimal solution is at least $\sum_{i=1}^{\infty}(n_{i-2} - n_{i-1})t_{i-2}$ We will now show that the former is at worst an 8γ factor larger than the latter.

$$\begin{aligned}
\sum_{i=1}^{\infty}(n_{i-1}^R - n_i^R)t_i^R &\leq \sum_{i=1}^{\infty}(n_{i-1}^R - n_i^R)\gamma t_i \\
&\leq \gamma \sum_{i=1}^{\infty} n_{i-1}^R t_i \\
&\leq 2\gamma \sum_{i=1}^{\infty} n_{i-1} t_i \qquad \text{[Lemma 5]} \\
&\leq 4\gamma \sum_{i=1}^{\infty} n_{i-1} t_{i-2} \\
&= 4\gamma \sum_{i=1}^{\infty}(n_{i-2} - n_{i-1}) \sum_{j=1}^{i-2} t_j \\
&\leq 8\gamma \sum_{i=1}^{\infty}(n_{i-2} - n_{i-1}) t_{i-2}
\end{aligned}$$

In the equality line we use that $\sum_{i=1}^{\infty} n_{i-1}t_{i-2} = \sum_{i=1}^{\infty}(n_{i-2} - n_{i-1})\sum_{j=1}^{i-2} t_j$. This follows because on the left hand side, a job contributes t_{i-2} if the job is alive at time t_{i-1} (i.e. is counted in n_{i-1}) for all i. The second summation says that a job completed between times t_{i-2} and t_{i-1} contribute t_j for all $j \in [i-2]$, which is equivalent. □

We now explain how to remove the speed s augmentation assumption. The speed 1 version SNRR(1) could simulate the speed s

version of SNRR(s) in the following way. SNRR(1) will use $2s$ time steps for each step of SNRR(s). Let π_t be the priority order that SNRR(s) uses for a time step t. Then SNRR(1) will use the same priority order π_t for each of these $2s$ steps, only removing jobs that had a chunk processed on earlier one of these $2s$ time steps. Then SNRR(1) will complete all the chunks that SNRR(s) did because First-Fit is 2-competitive for bin packing [21]. If SNRR(1) happens to complete some chunks that SNRR(s) didn't during these time steps, then will forget that it completed them going forward. Admittedly this definition of SNRR(1) is clunky. To see why one can't keep the same natural definition of SNRR without speed augmentation, consider a situation where the first job in the priority order had width 1 and the second job had width k. Then even though SNRR has width at least k it can process, it will only process width 1. And if this happened for a long period of time, SNRR would be wasting most of its processing capability.

6 RELATED WORK

Most prior work on HBM focuses on empirical analysis of benefits of using HBM by showing improved performance for individual algorithms [9–11, 14, 15, 28, 32].

Das et. al [17] provide the theoretical model for HBM that we use in this paper and also provide a constant competitive algorithm for minimizing makespan as mentioned in Section 1. In later work [20], this model was experimentally validated using simulations indicating that the theoretical results hold in experimental settings under appropriate assumptions.

HBM management problem can be said to be related to paging since HBMs can be thought of as a form of cache. Paging on sequential machines has been studied for decades. On parallel machines, there are many models for caches, both private and shared [1–3, 6, 7, 12, 13, 16, 18, 26]. These include models where there are private caches associated with each core [8]. Most relevant to the HBM model is how to do paging on shared caches where multiple threads or processes share a single cache. There is significant work in this area as well [4, 5, 19, 23, 24, 27, 29, 31].

Multi-core paging problems on shared caches are related to HBM management in that both have to decide what fraction of cache is allocated to each thread over time. However, paging problems do not have the far-channel arbitration problem. Interestingly, even for traditional caches, when we consider parallelism, some researchers have noticed that it makes sense to consider traditional scheduling objective such as makespan or sum of completion timees rather than paging objectives such as number of cache misses [4, 5, 24]. In particular, some of the recent results on managing a shared cache also have a distinct flavor of parallel scheduling, albeit this point is not explicitly made in the papers [4, 5].

In this paper, we have argued that the theory of HBM management is closely related to scheduling parallel jobs on a parallel machine — there is an enormous body of work in this area for various models of parallelism and for various optimization metrics. The model of parallelism plays a big role here and the algorithmic results from some modeles translate to others, but most do not directly translate. Perhaps the most relevant model is the speedup curves model [22] where the job's parallelism and its ability to use processors changes over time. In particular, on every time step, the amount of work completed by a job depends on the number of processors allocated to the job. However, there is a caveat. In this model, the marginal benefit of each processor is monotonically non-decreasing as the number of processors allocated to each job increases. This is distinct from the semi-normal model since the work done by the job is 0 unless it receives exactly the number of processors dictated by the width of the next step. However, we believe that the results on the speedup curves model are likely to be closely related to the results of the semi-normal model; whether one must prove this one algorithm at a time or if there is a black-box conversion is left to future work.

7 CONCLUSIONS AND FUTURE WORK

Our main contribution is to make an explicit case that managing an HBM/DRAM memory hierarchy should be thought of as a parallel scheduling problem by showing how to translate results in a "semi-normal" scheduling model to the HBM/DRAM management model. There are several natural directions for future investigation. While, in this paper, we assumed that all jobs are processes/jobs arrive at time 0, we observe that there appears to be no great difficulty in extending the results in this paper to other standard scheduling objectives and/or allowing processes to arrive over time.

A stronger case could be for thinking of HBM management as a parallel scheduling problem by showing how to translate results from a "normal" parallel scheduling model. A challenge with this approach is that most of the literature on the seemingly best candidates for the "normal" parallel scheduling model assumes continuous time and an infinitely divisible processor, but reasonably modeling HBM/DRAM management using continuous time and an infinitely divisible HBM seems problematic. The use of continuous time and an infinitely divisible processor in the normal models significantly eases the task of algorithm analysis, but it is generally not known how to translate positive results in the normal models, under the assumption of continuous time and an infinitely divisible processor, to a discrete time variant of these models, where processes have to share via time multiplexing. So this fundamental scheduling issue needs to be resolved before one can hope to apply scheduling results in these "normal" models to HBM/DRAM management.

Presumably the model proposed in Das et al. [17] assumed only one process per core to avoid considering a policy to also manage the core to HBM channel. However, the model should ideally be extended to allow multiple processes per core, which is a common use case. Similarly, another natural direction is to consider threads, that share an address space, instead of processes, which do not share an address space. It seems like managing an HBM/DRAM memory hierarchy with threads is significantly more challenging as now the parallelism of a chunk will, in some sense, depend on which other chunks are in HBM (as the parallelism for a collection of chunks depends on the aggregate number of distinct pages in the chunks).

Rather than assuming that each process has at least k requests, one could have shown asymptotic competitiveness instead of absolute competitiveness with an additive factor that depends on k. So a natural research direction is to strengthen the result for makespan in Das et al. [17], and our result for total completion time, to remove

the assumption about the processes being long, or equivalently to give absolute competitiveness results even for short sequences.

Acknowledgments

Kunal Agrawal was supported by National Science Foundation grants CCF-2106699, CCF-2107280, PPoSS-2216971. Michael Bender was supported by National Science Foundation grants CCF 2247577, CCF 2106827, and by the John L. Hennessy Chair in Computer Science. Kirk Pruhs was supported by National Science Foundation grant CCF-2209654. Benjamin Moseley supported in part by a Google Research Award, an Infor Research Award, a Carnegie Bosch Junior Faculty Chair, NSF grant CCF-2121744 and ONR Grant N000142212702. Clifford Stein is supported by the NSF grant CCF-2218677, ONR grant ONR-13533312, and by the Wai T. Chang Chair in Industrial Engineering and Operations Research.

REFERENCES

[1] 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.

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

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

[4] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. [n.d.]. *Tight Bounds for Parallel Paging and Green Paging*. 3022–3041. https://doi.org/10.1137/1.9781611976465.180 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976465.180

[5] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2022. Online Parallel Paging with Optimal Makespan. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures* (Philadelphia, PA, USA) *(SPAA '22)*. Association for Computing Machinery, New York, NY, USA, 205–216. https://doi.org/10.1145/3490148.3538577

[6] 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.

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

[8] 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.

[9] 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.

[10] 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

[11] Michael A. Bender, Jonathan W. Berry, Simon D. Hammond, Branden Moore, Benjamin Moseley, and Cynthia A. Phillips. 2015. k-Means Clustering on Two-Level Memory Systems. In *Proc. 2015 International Symposium on Memory Systems, (MEMSYS)*, Bruce Jacob (Ed.). Washington DC, USA, 197–205.

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] 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*. ACM, 105–115.

[17] Rathish Das, Kunal Agrawal, Michael Bender, Jonathan Berry, Benjamin Moseley, and Cynthia Phillips. 2020. How to Manage High-Bandwidth Memory Automatically. In *Proc. 32st ACM on Symposium on Parallelism in Algorithms and Architectures*.

[18] 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.

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

[20] 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* (Philadelphia, PA, USA) *(SPAA '22)*. Association for Computing Machinery, New York, NY, USA, 147–159. https://doi.org/10.1145/3490148.3538570

[21] György Dósa and Jirí Sgall. 2013. First Fit bin packing: A tight analysis. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany (LIPIcs, Vol. 20)*, Natacha Portier and Thomas Wilke (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 538–549.

[22] Jeff Edmonds and Kirk Pruhs. 2012. Scalably scheduling processes with arbitrary speedup curves. 8, 3 (2012). https://doi.org/10.1145/2229163.2229172

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

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

[25] Sandy Irani. 1996. Competitive Analysis of Paging. In *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996) (Lecture Notes in Computer Science, Vol. 1442)*, Amos Fiat and Gerhard J. Woeginger (Eds.). Springer, 52–73.

[26] 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.

[27] 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.

[28] 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.

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

[30] Kirk Pruhs, Jirí Sgall, and Eric Torng. 2004. Online Scheduling. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*, Joseph Y.-T. Leung (Ed.). Chapman and Hall/CRC.

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

[32] 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.

[33] Wikipedia contributors. 2025. High Bandwidth Memory — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=High_Bandwidth_Memory&oldid=1272614625 [Online; accessed 15-February-2025].