Enabling GPU Memory Oversubscription via Transparent Paging to an NVMe SSD

Joshua Bakita and James H. Anderson

Department of Computer Science
University of North Carolina, Chapel Hill

How can we do more, with less?

How can we do more, with less?

Processor Cores

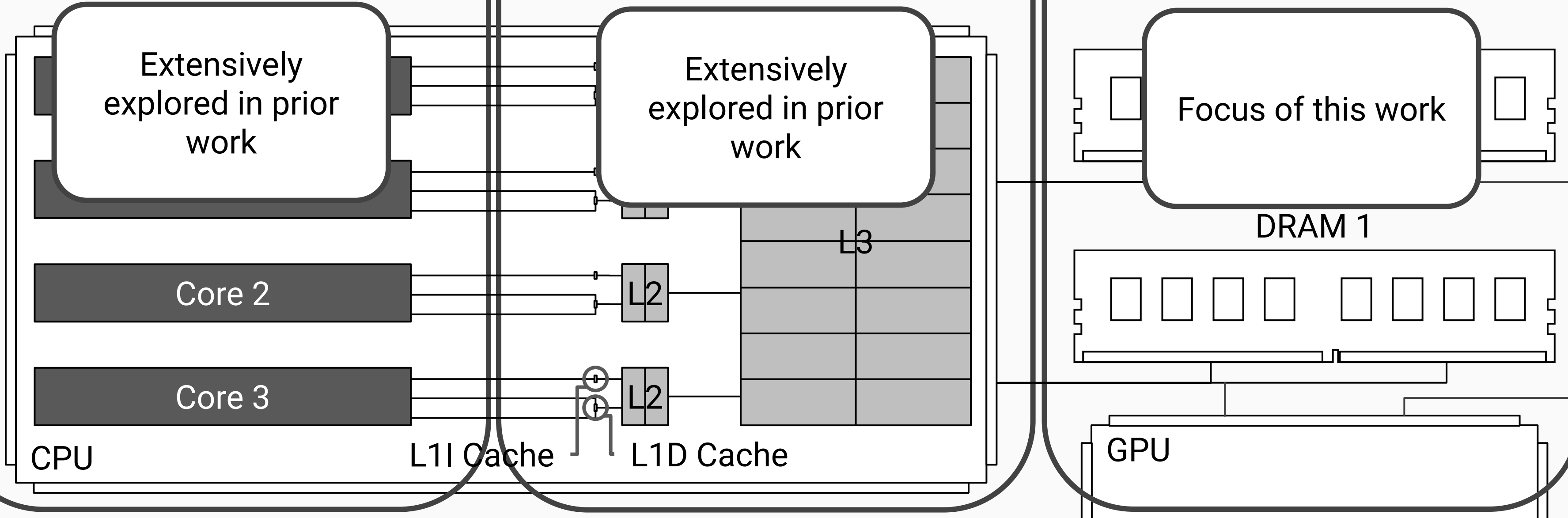
- ➔ More efficient scheduling and provisioning

Memory Caches and Interconnects

- ➔ More efficient scheduling and provisioning

Main Memory (DRAM)

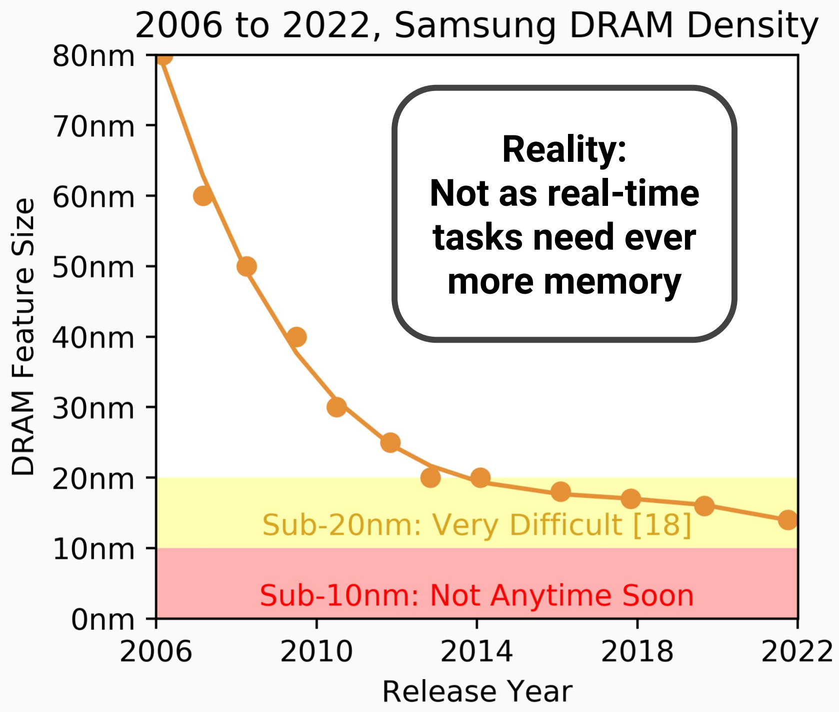
- ➔ Memory oversubscription via paging to permanent storage



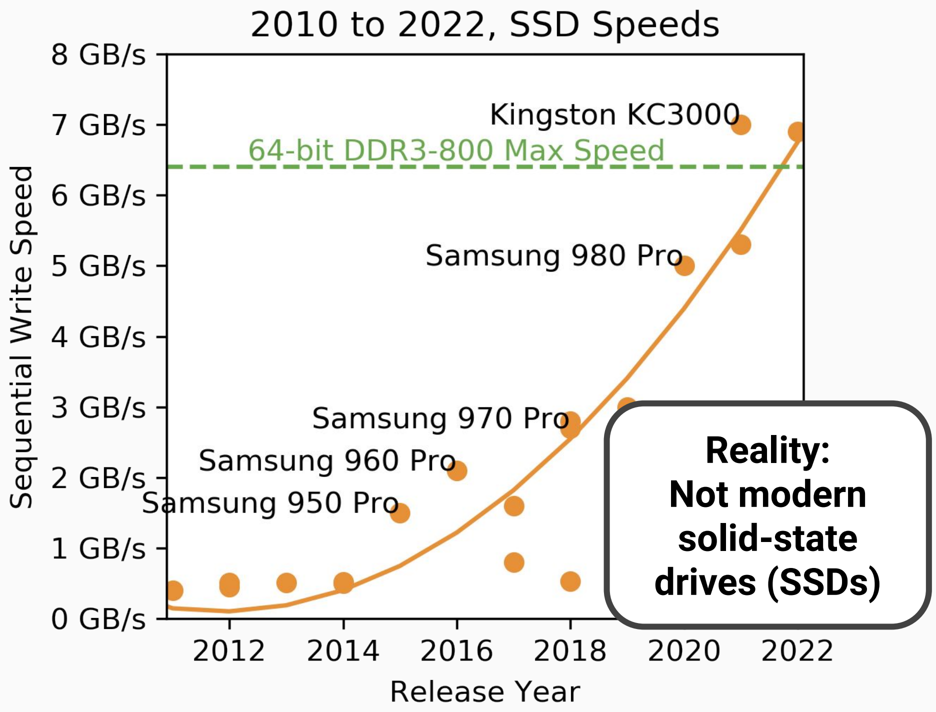
Why memory oversubscription?

Some assumptions worth revisiting…

Assumption: DRAM is plentiful



Assumption: Storage is too slow



Key Goals

Prior work [15-17, 44-45] limited to this scope

Memory oversubscription for a real-time system that is:

Predictable | **Fast** | **Easily Applicable**

With key insights drawn from **technology trends**, **real-time scheduling**, and **GPU architecture**, we achieve all three for real-time CPU+GPU+SSD systems.

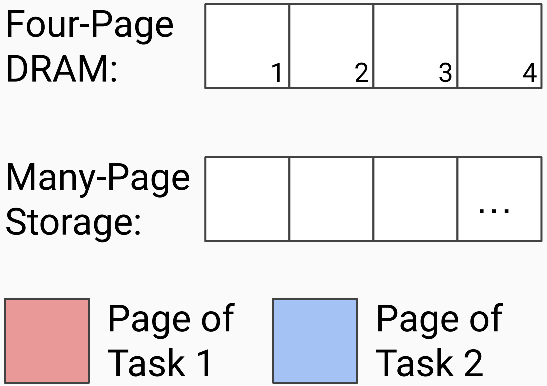
Enabling Predictable Memory Oversubscription

Goal 1 of 3

<u>Predictable</u> Oversubscription

Dangers to avoid

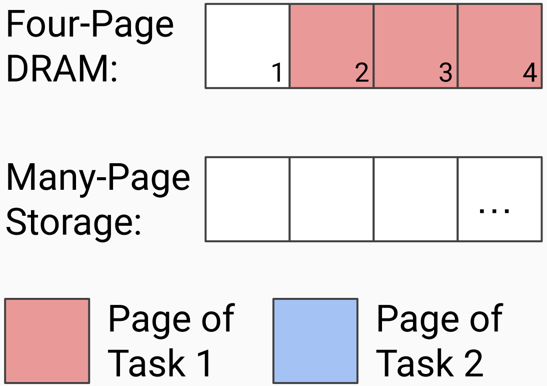
Demand paging combined with least-recently-used (LRU) eviction



Predictable Oversubscription

Dangers to avoid

Demand paging combined with least-recently-used (LRU) eviction

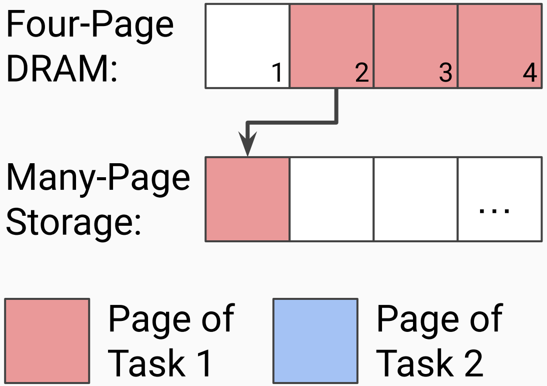


1. Task 1 runs a job, accessing pages 3 -> 4 -> 2
2. Task 2 runs a job, needing two pages of DRAM. The OS selects the LRU page of Task 1 and moves it to storage.

Predictable Oversubscription

Dangers to avoid

Demand paging combined with least-recently-used (LRU) eviction



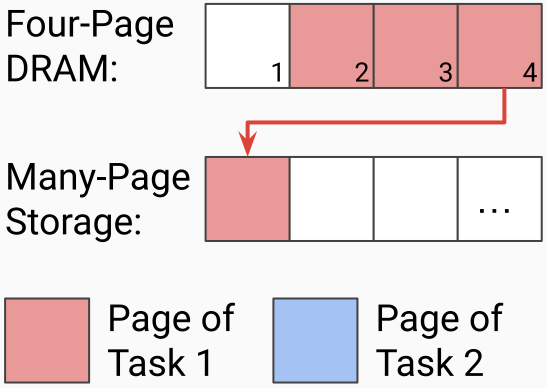
1. Task 1 runs a job, accessing pages 3 -> 4 -> 2
2. Task 2 runs a job, needing two pages of DRAM. The OS selects the LRU page of Task 1 and moves it to storage.

Case 1:
Page 2

Predictable Oversubscription

Dangers to avoid

Demand paging combined with least-recently-used (LRU) eviction



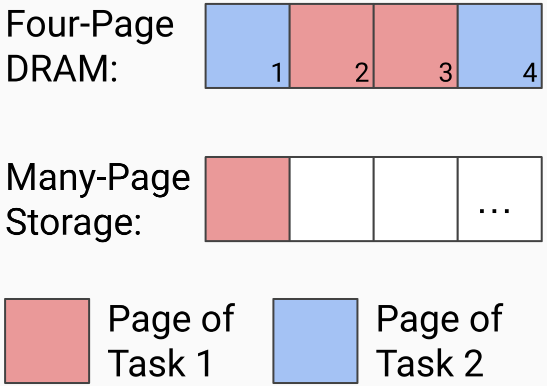
1. Task 1 runs a job, accessing pages 3 -> 4 -> 2
2. Task 2 runs a job, needing two pages of DRAM. The OS selects the LRU page of Task 1 and moves it to storage.

Case 1:
Page 2

Case 2:
Page 4

Dangers to avoid

Demand paging combined with least-recently-used (LRU) eviction



1. Task 1 runs a job, accessing pages 3 -> 4 -> 2
2. Task 2 runs a job, needing two pages of DRAM. The OS selects the LRU page of Task 1 and moves it to storage.

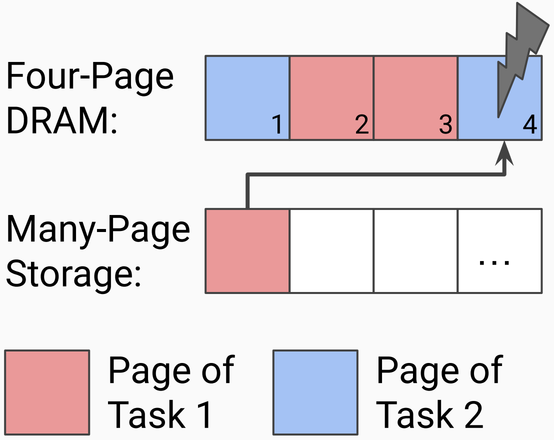
   Case 1: **Page 2**

   Case 2: **Page 4**

3. Task 2 runs its job to completion.

Dangers to avoid

Demand paging combined with least-recently-used (LRU) eviction



1. Task 1 runs a job, accessing pages 3 -> 4 -> 2
2. Task 2 runs a job, needing two pages of DRAM. The OS selects the LRU page of Task 1 and moves it to storage.

   Case 1: **Page 2**

   Case 2: **Page 4**

3. Task 2 runs its job to completion.
4. Task 1 runs its next job, accessing pages 3 and 4. It's execution time will vary greatly depending on what was moved to storage.

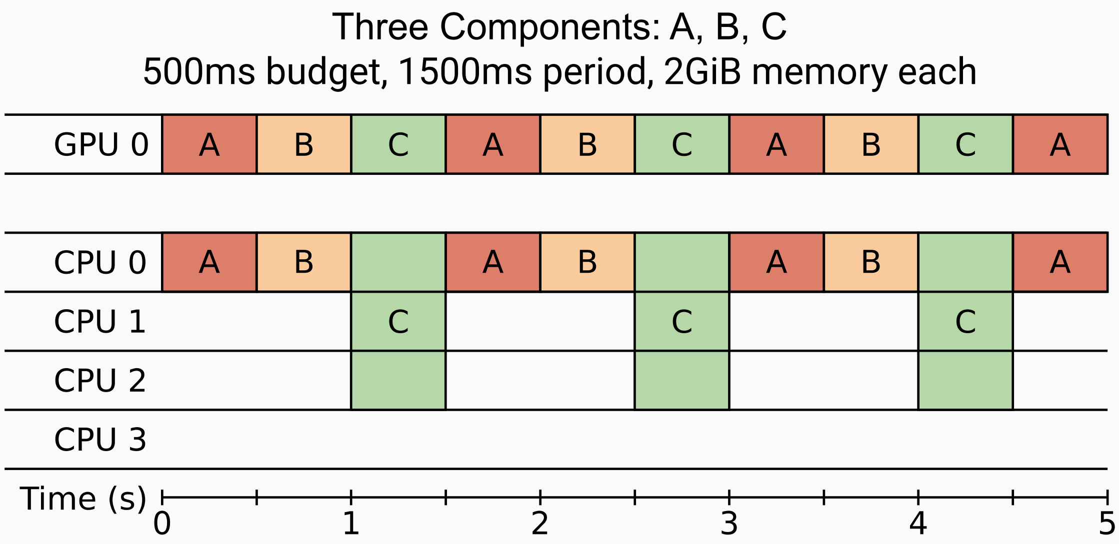
Predictable Oversubscript.

Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…



Works fine, given 6 GiB of DRAM.

What if we only have 4 GiB?

Predictable Oversubscript.

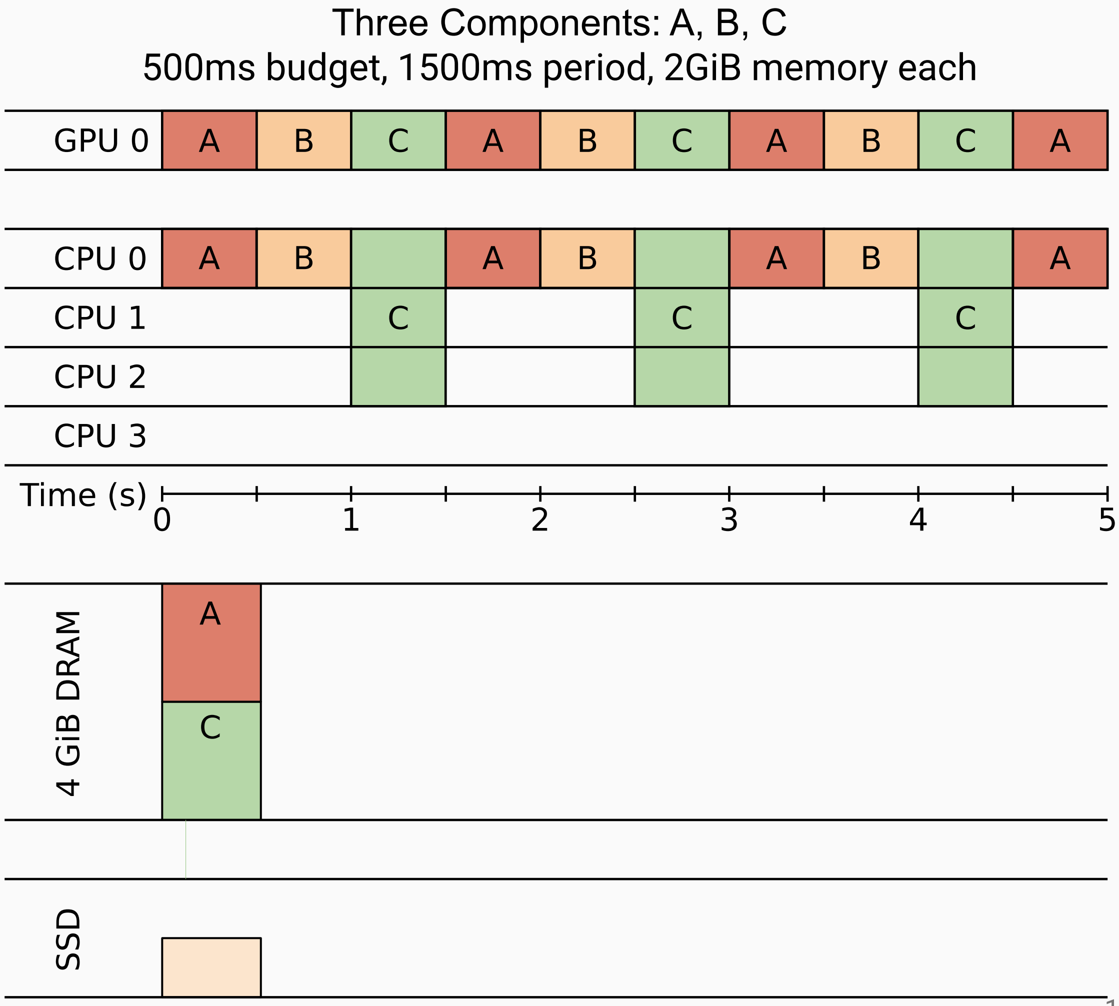
Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.



Predictable Oversubscript.

Using schedule foreknowledge

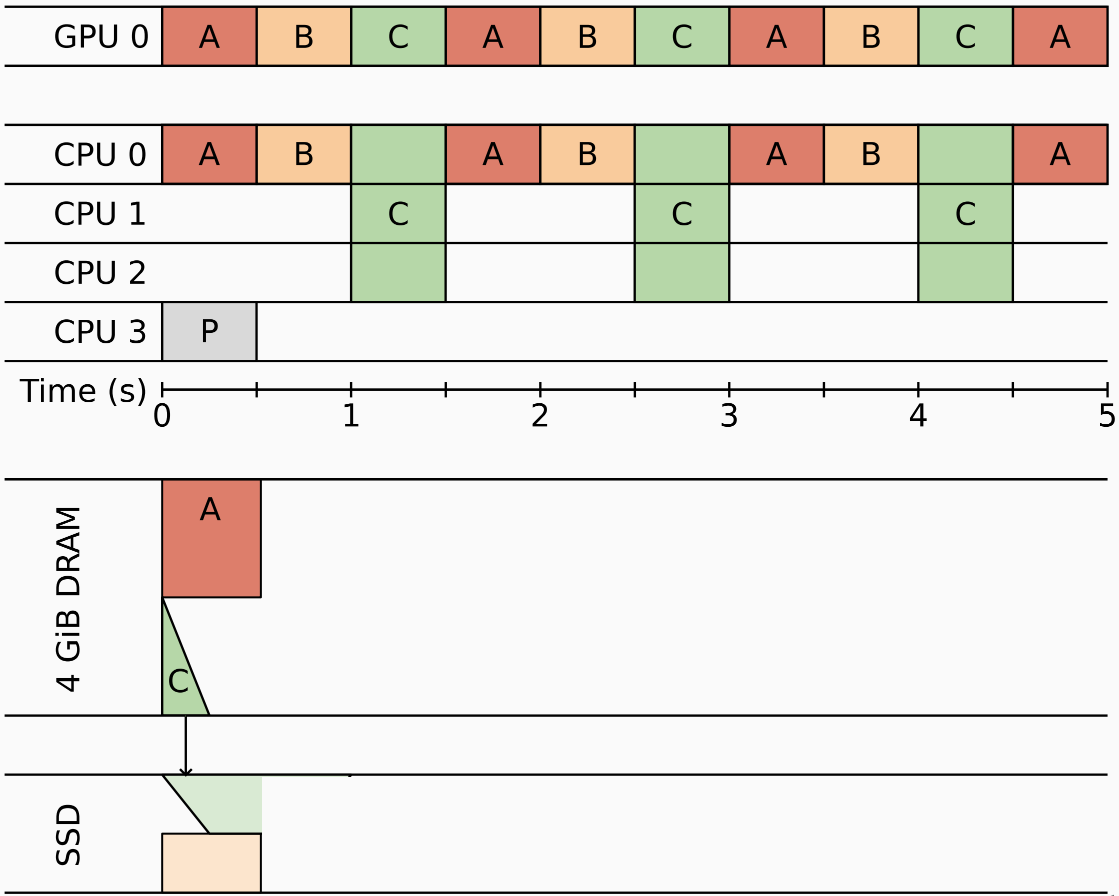
After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.

Three Components: A, B, C
500ms budget, 1500ms period, 2GiB memory each



Predictable Oversubscript.

Using schedule foreknowledge

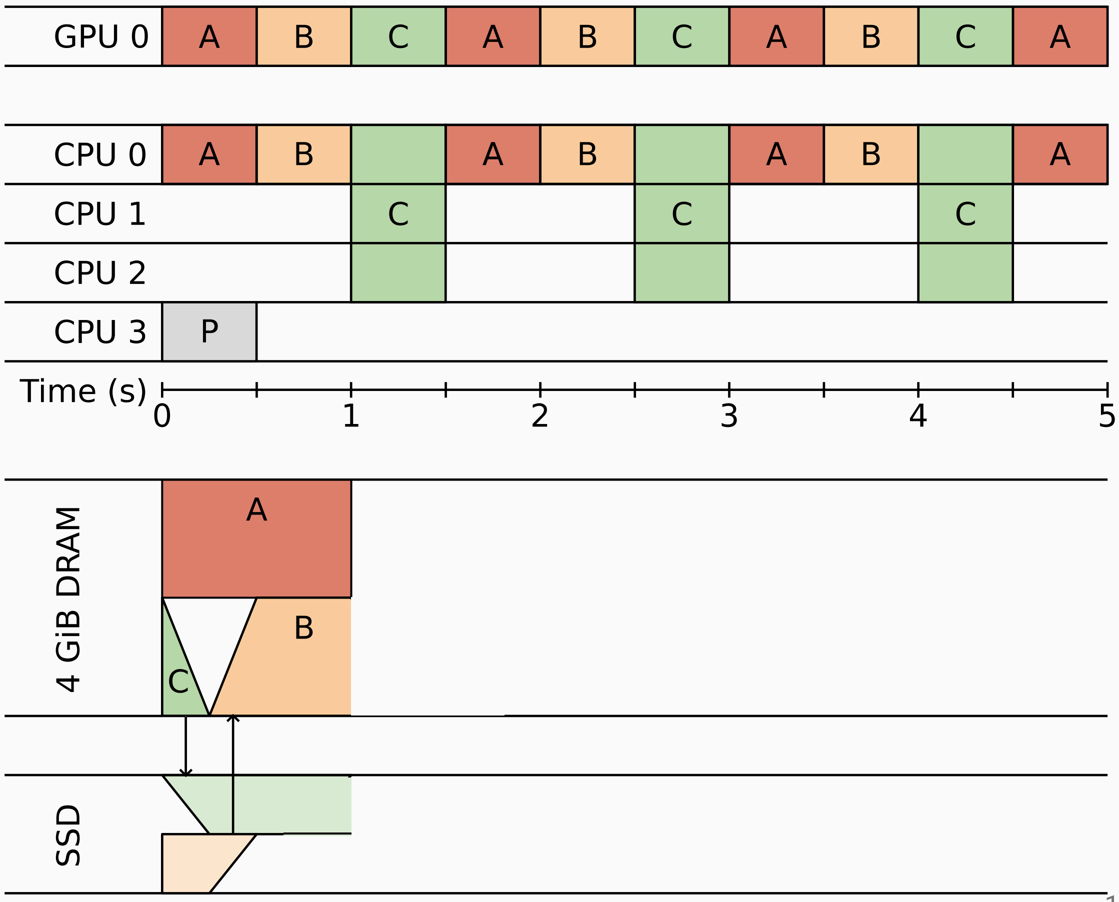
After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example...

Creating a memory schedule.

Three Components: A, B, C
500ms budget, 1500ms period, 2GiB memory each



Predictable Oversubscript.

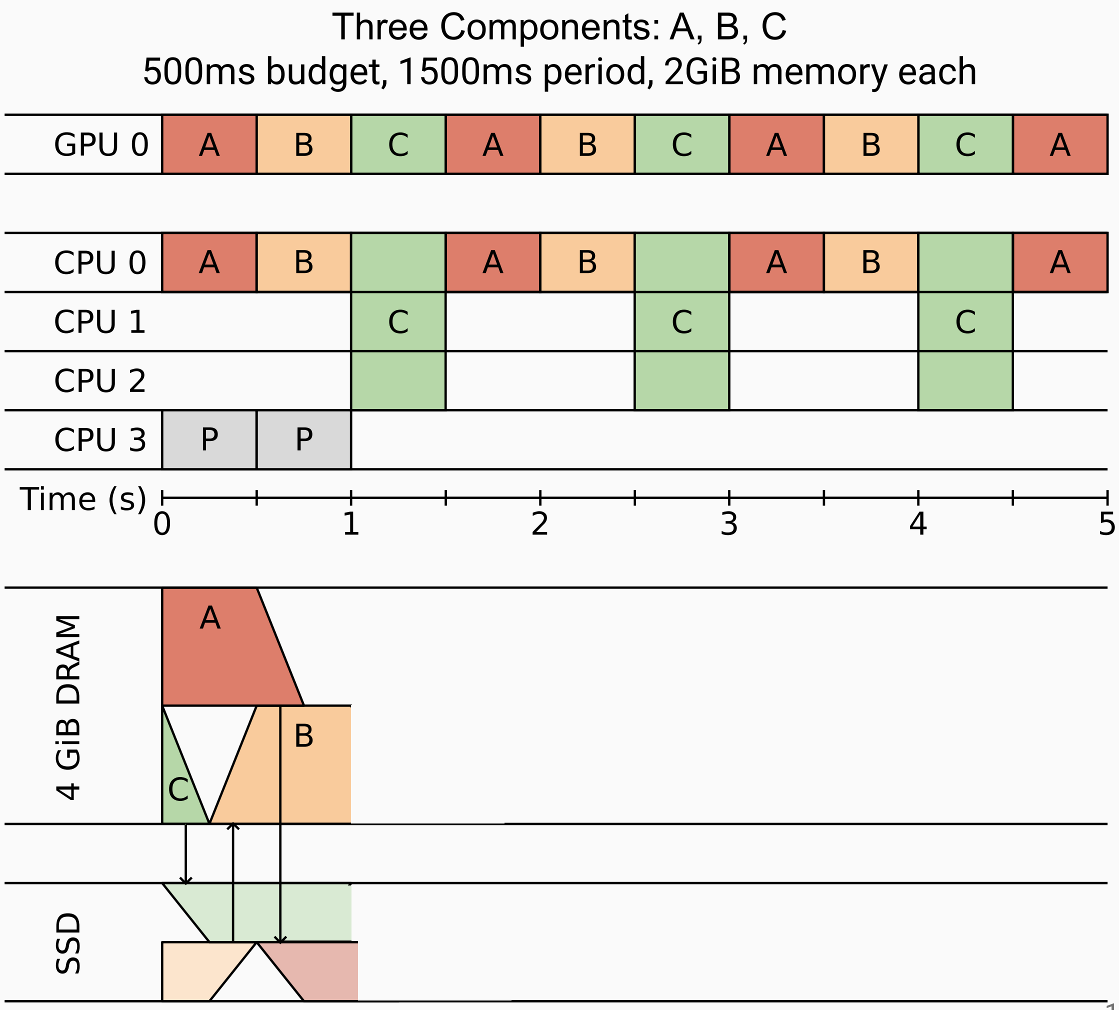
Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.



<u>Predictable</u> Oversubscript.

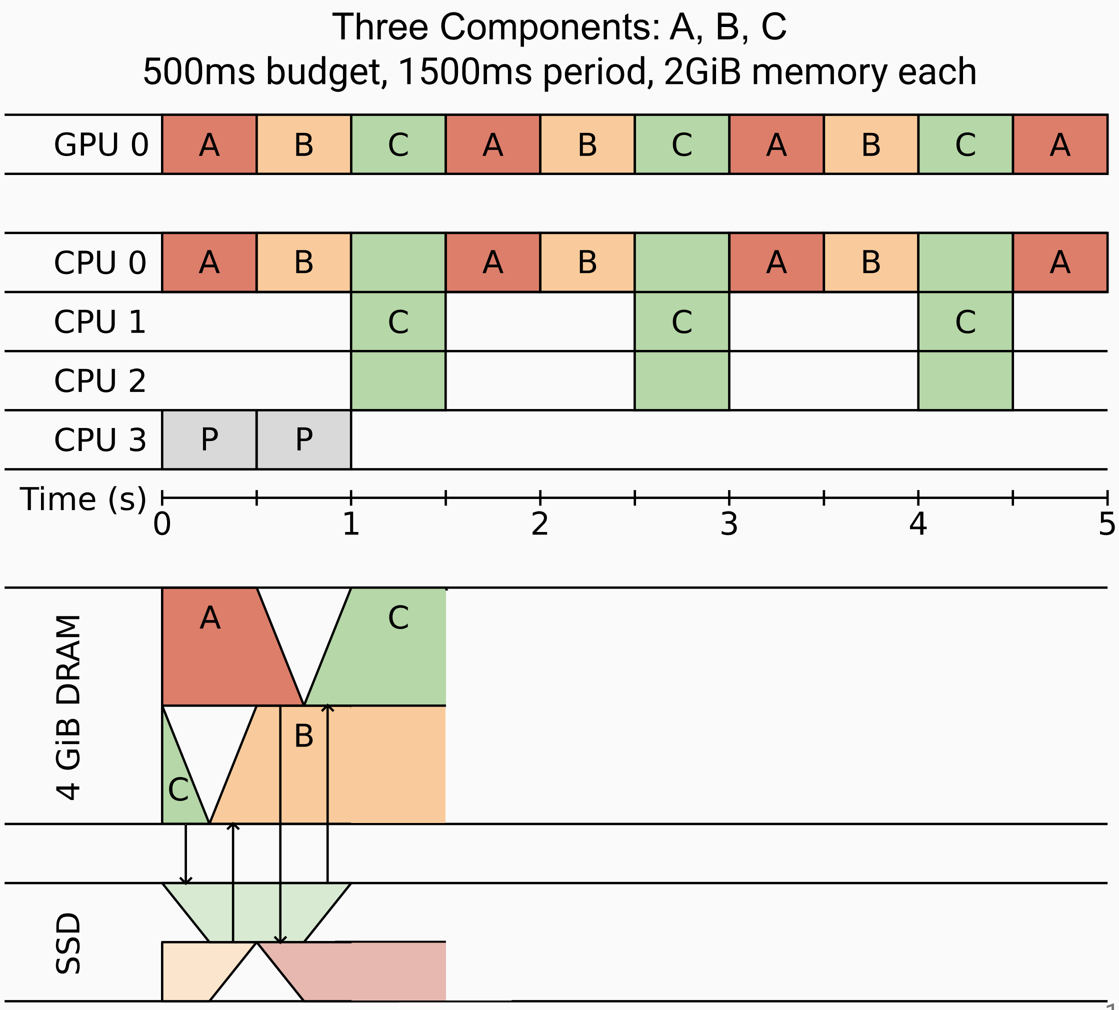
Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.



Predictable Oversubscript.

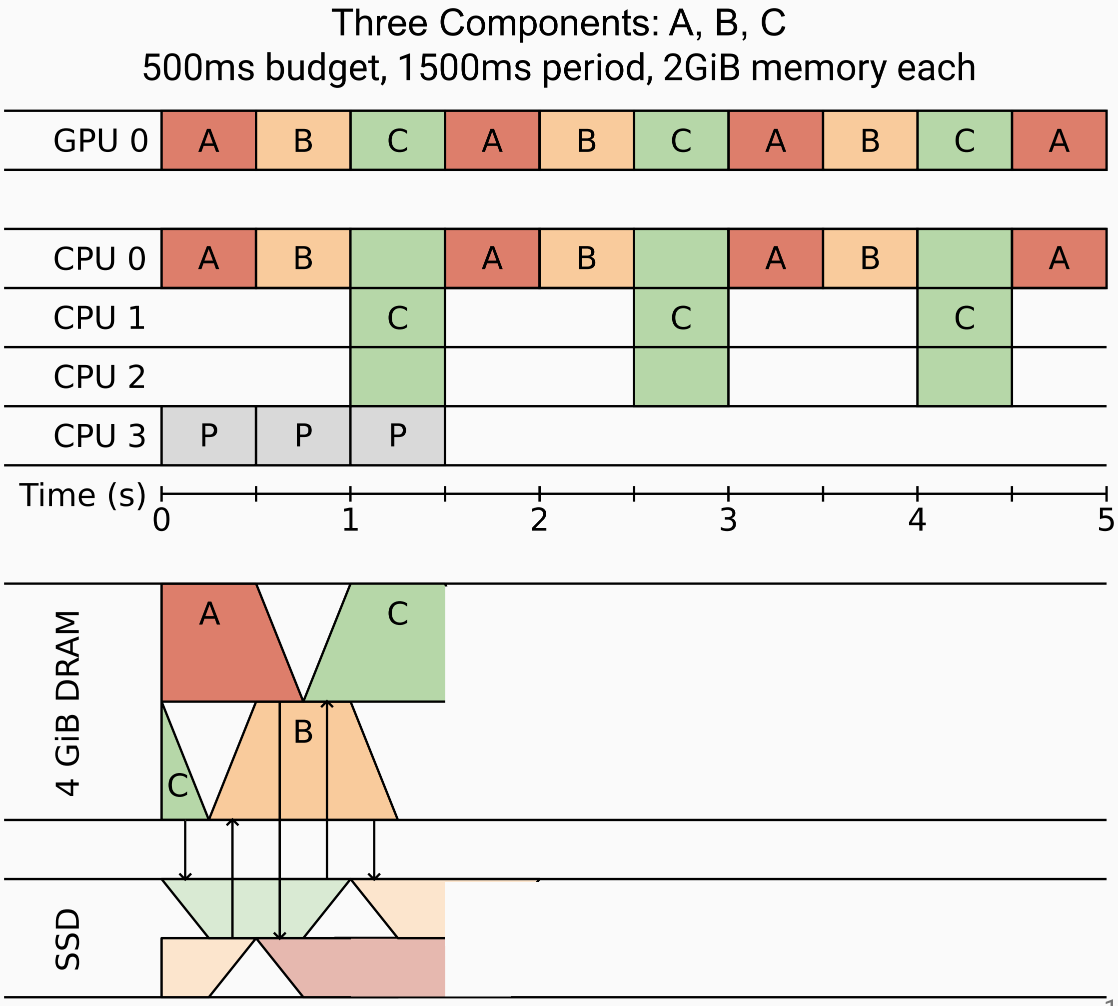
Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.



Predictable Oversubscript.

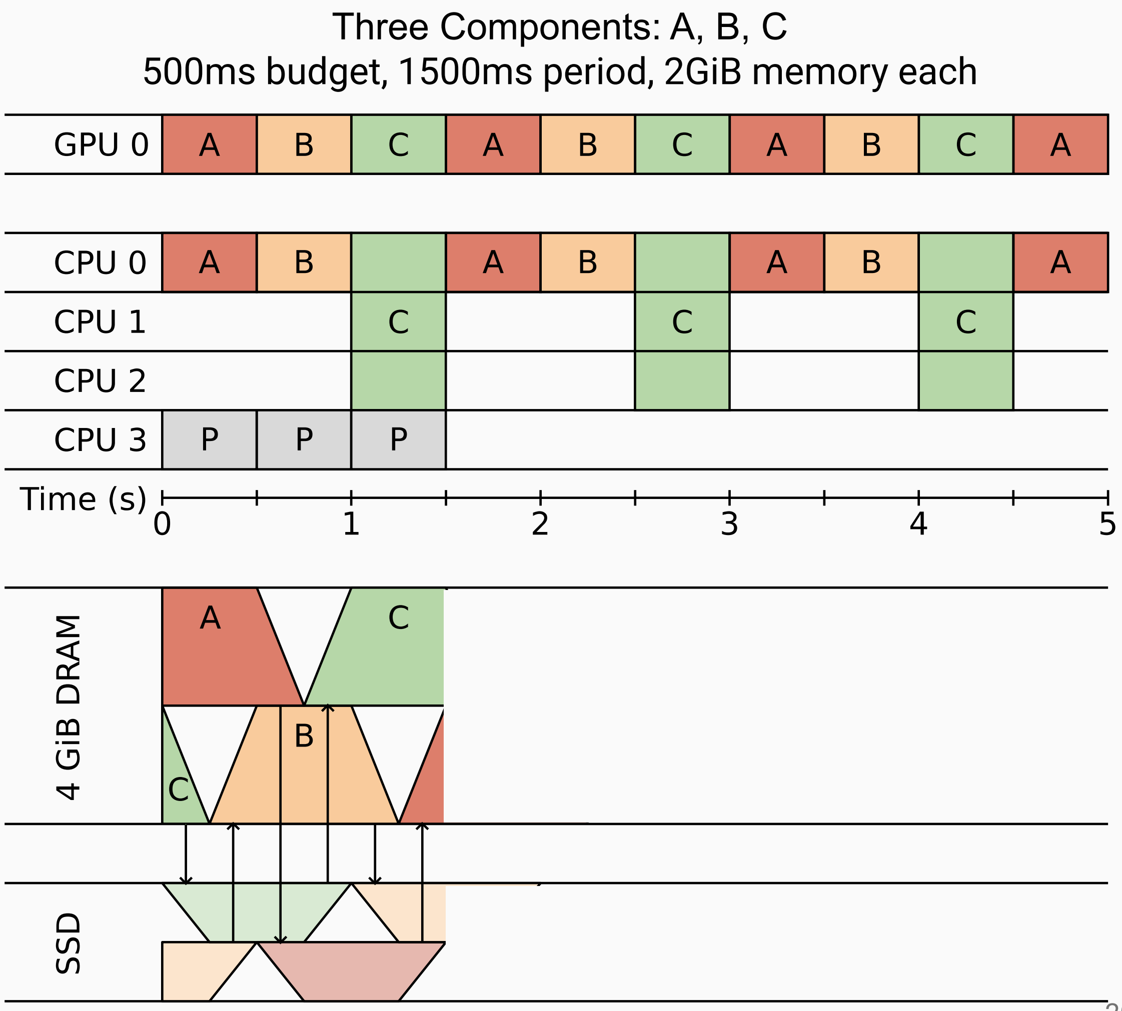
Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.



Predictable **Oversubscript.**

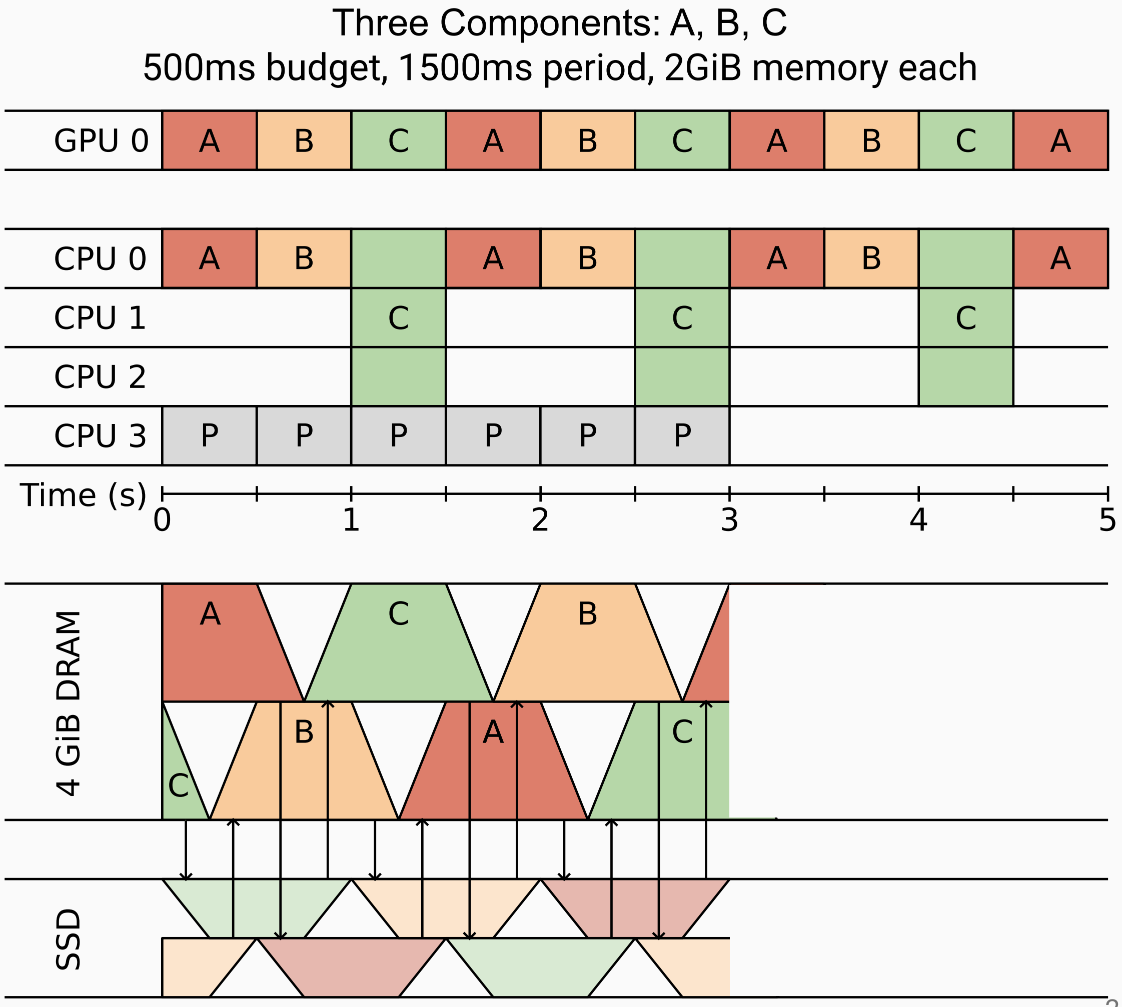
Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the next job arrives.

With a table-driven scheduler, we know exactly.

Consider an example…

Creating a memory schedule.



Predictable Oversubscript.

Using schedule foreknowledge

After a job completes in a real-time system, we know the minimum amount of time before the

W e

kr

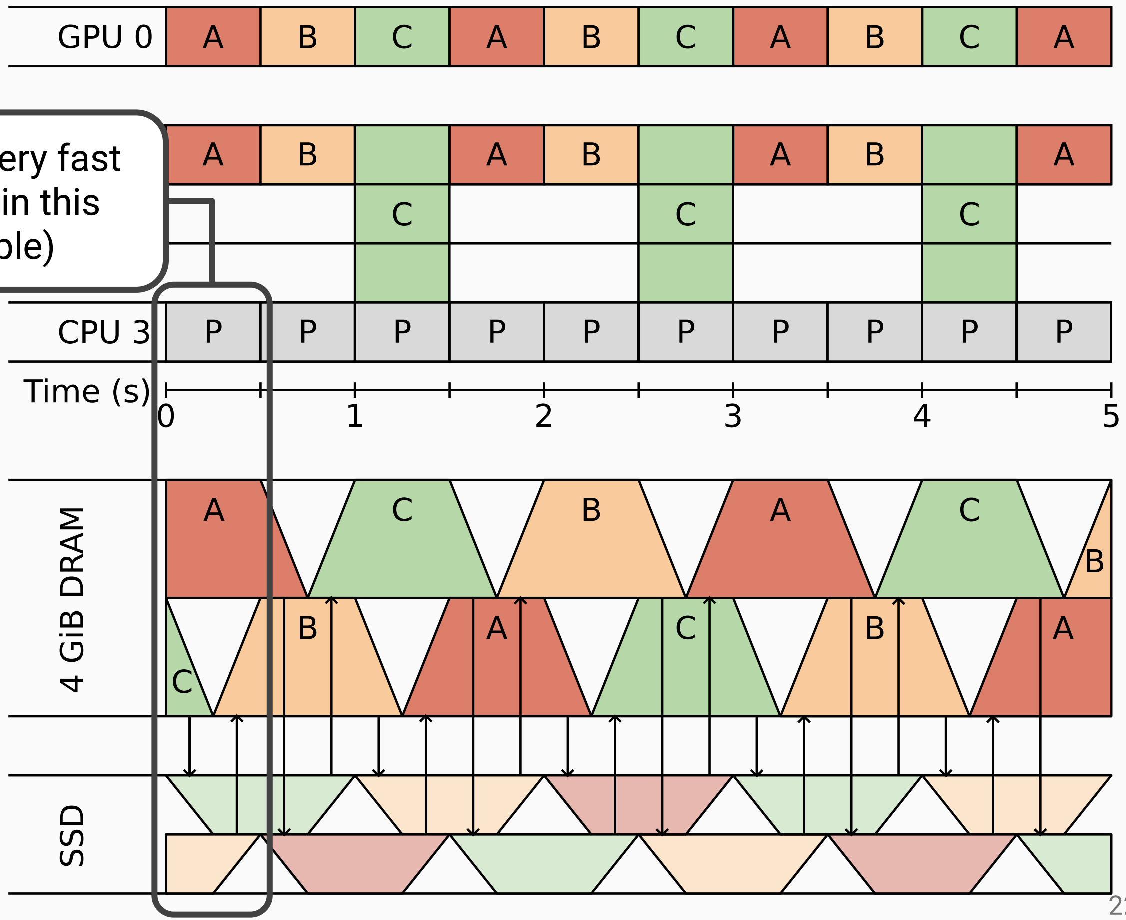
Co

Creating a memory schedule.

Must be very fast (8 GiB/s in this example)

Key Insight:
By using period information to schedule paging operations, we can make them predictable.

Three Components: A, B, C
500ms budget, 1500ms period, 2GiB memory each



Enabling <u>Fast</u> Memory Oversubscription

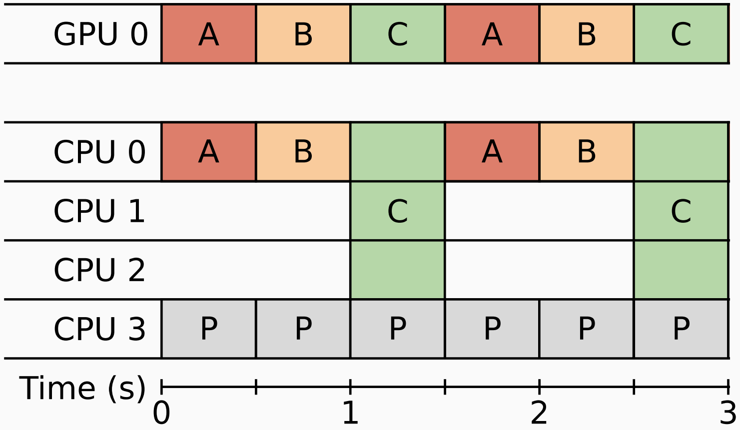
Goal 2 of 3

<u>Fast</u> **Oversubscription**

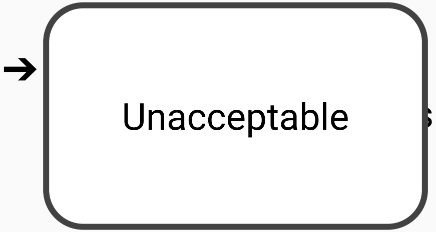
What's already available?

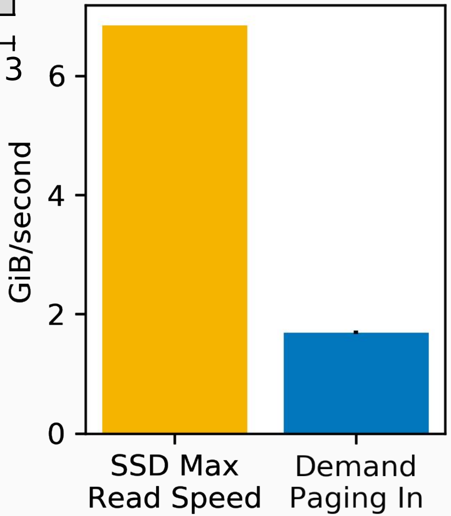
Demand paging for loading data from storage.

Three Components: A, B, C
500ms budget, 1500ms period, 2GiB memory each



➔ Need 8 GiB/s to meet needs of example





Platform: NVIDIA Jetson AGX Xavier

Fast Oversubscription

What slows demand paging?

Key Insight:
Synchronization costs make paging multicore CPU memory unacceptable in a real-time Linux system

Overheads? Profiling results

14%	19%	27%	40%
Page allocation	Page mapping	Bookkeeping and actual I/O	Locking and retry commit

On Jetson Xavier AGX running Linux 4.9 with our Sabrenet Rocket 4 Plus SSD.

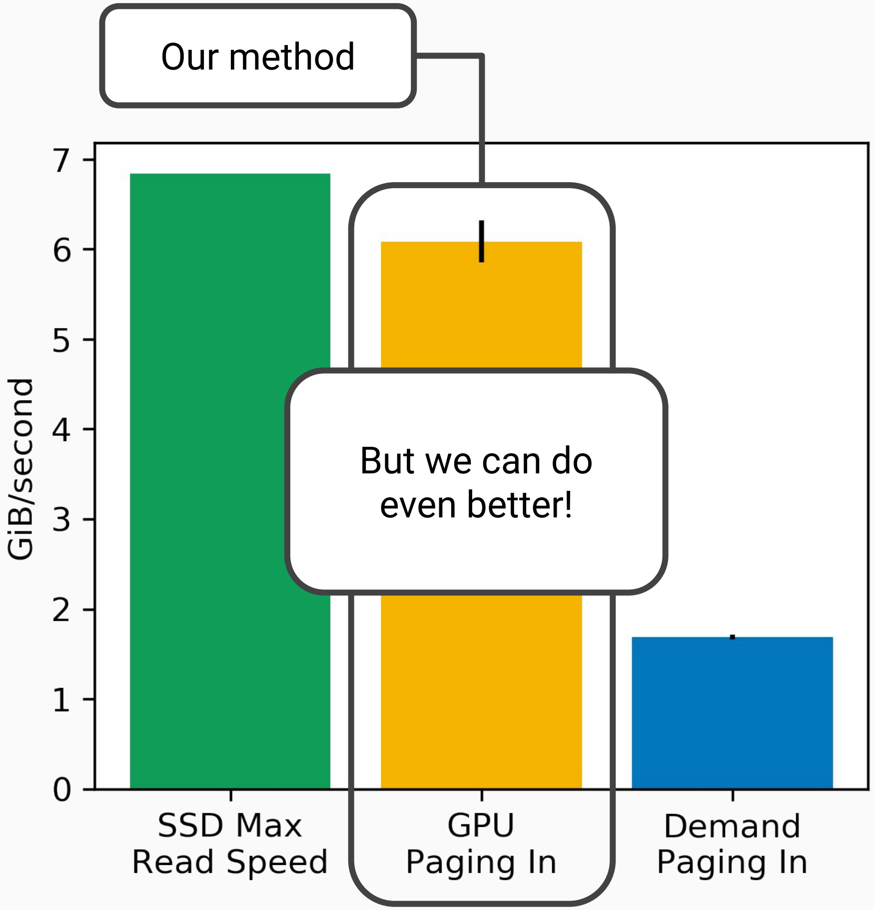
Fast Oversubscription

Paging GPU memory

Key Insight:
GPU APIs do not allow for cross-application shared pages

Key Insight:
Pages can be moved directly from GPU virtual memory to and from an SSD without a mapping on the CPU

Key Insight:
GPU memory is most commonly used to store *read-only* weights



Our GPU paging method is 3 times faster than demand paging

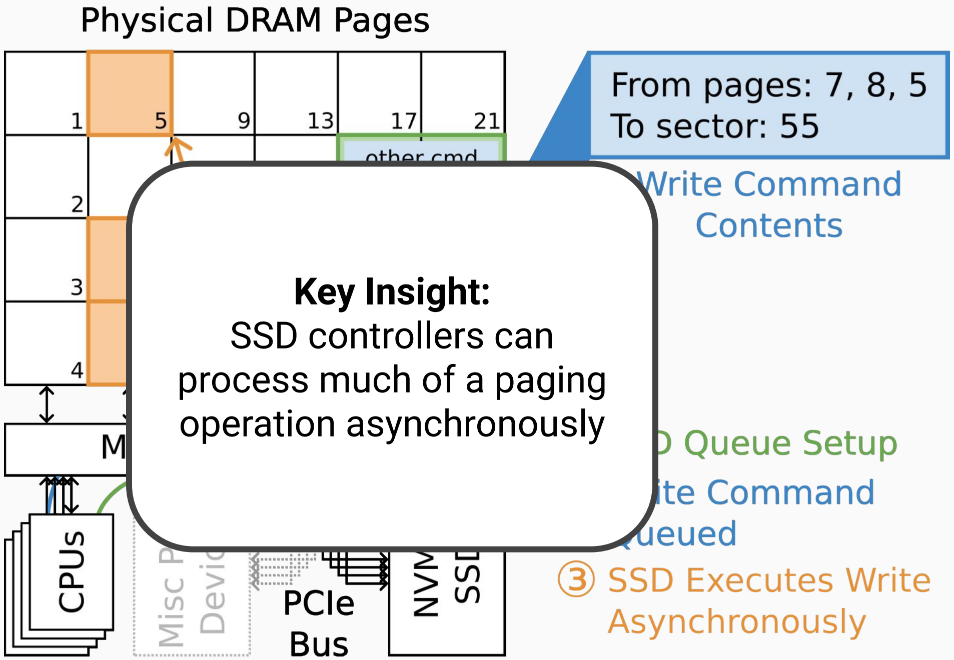
Fast Oversubscription

Utilizing the SSD controller

SSDs support command offloading. Can we use this instead of the CPU?

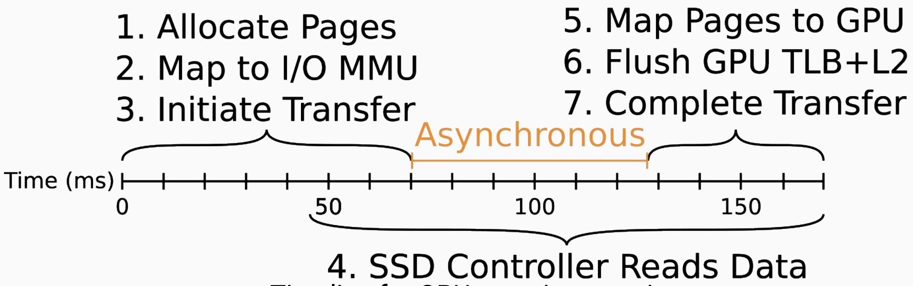
Allows for:

- ➔ 30% asynchronous page-in
- ➔ 80% asynchronous page-out



Key Insight:
SSD controllers can process much of a paging operation asynchronously

Scatter-gather write operation on an NVMe SSD



Timeline for GPU page-in operation

Enabling Easily Applicable Memory Oversubscription

Goal 3 of 3

Easily Applicable Oversubscription

A simple API

On Linux:

0. Use `strace` to identify address space (*AS_ID*) and allocation ID (*BUF_ID*)
1. `ioctl(`*`AS_ID`*`, NVGPU_AS_IOCTL_WRITE_SWAP_BUFFER, {`*`BUF_ID`*`});`
2. `ioctl(`*`AS_ID`*`, NVGPU_AS_IOCTL_READ_SWAP_BUFFER, {`*`BUF_ID`*`});`

Asynchronous variants available.

Code is open source and documented. See https://www.cs.unc.edu/~jbakita/rtss22-ae.html to get started.

Conclusions

We can make memory oversubscription in a real-time system:

Predictable

How can we know what and when to page?

Use foreknowledge present in schedule

Fast

How can synchronization overheads be avoided?

Page GPU mem

How fast can we go?

>6GB/s read, >5GB/s write

Easily Applicable

Can we make GPU paging easy to use?

Yes, via our 2-line Linux API

What you have to read the paper for…

Evaluation:

- Comparison to direct I/O, and how we manage to be faster
- Benchmarks which demonstrate our minimal impact on memory bandwidth
- Exact distributions for all benchmark results
- Full details on our supported API calls

Regarding SSDs:

- Details on how we offload paging operations onto the SSD controller
- How we ensure SSD caches don't bottleneck
- How we utilize real-time GPU scheduling invariants to speed up page-out operations

Regarding GPUs:

- Details on how memory allocations work on NVIDIA's embedded boards
- Details on NVIDIA GPU virtual memory capabilities
- History of NVIDIA page table formats
- How to determine GPU address space and buffer IDs
- A version of strace supporting detailed tracing of all NVIDIA driver syscalls on Jetson boards

\+ More details and background on everything covered in this presentation

Thanks!
Questions?

Future work:

- ➔ OS scheduler integration
- ➔ Application to mode changes, DNN layers, etc.
- ➔ Increase performance, portability, and SSD space allocation algorithm

Contact:
Email: jbakita@cs.unc.edu
Twitter: @JJBakita
Web: https://cs.unc.edu/~jbakita

Old Well, University of North Carolina at Chapel Hill, Winter 2017