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 at Chapel Hill
Email: {jbakita, anderson}@cs.unc.edu

***Abstract*—Safety-critical embedded systems are experiencing increasing computational and memory demands as edge-computing and autonomous systems gain adoption. Main memory (DRAM) is often scarce, and existing mechanisms to support DRAM oversubscription, such as demand paging or compile-time transformations, either imply serious CPU capacity loss, or put unacceptable constraints on program structure. This work proposes an alternative: paging *GPU* rather than *CPU* memory buffers directly to permanent storage to enable efficient and predictable memory oversubscription. This paper focuses on *why* GPU paging is useful and *how* it can be efficiently implemented. Specifically, a GPU paging implementation is proposed as an extension to NVIDIA's embedded Linux GPU drivers. In experiments reported herein, this implementation was seen to be three times faster end-to-end than demand paging, with 81% lower overheads. It also achieved speeds above the fastest prexisting Linux userspace I/O APIs with low DRAM and bus interference to CPU tasks—at most a 17% slowdown.**

I. Introduction

One solution to the increasing complexity of embedded and safety-critical systems is to make the systems composable via the use of independently certifiable components. Robust time or space partitioning of shared hardware between these components is key to enabling composability. Such partitioning has been studied in several settings including accelerators [1], [2], shared caches [3]–[5], buses [6]–[11], and DRAMs [12]–[14]. Spatial partitioning is often used in these works to subdivide shared caches and DRAMs.

Unfortunately, a system may have insufficient resources to provide static partitions of caches and DRAM to all components. Trends in hardware and application design, alongside size, weight, power, and cost (SWAP-C) constraints particularly underscore the issue of insufficient DRAM. In the context of autonomous vehicles, industry researchers have noted "we need to minimize the required RAM size to reduce the unit cost of production" [15]. In other contexts, real-time researchers have highlighted the scarcity of DRAM [16], [17] and hardware researchers have repeatedly published on the fundamental limits of DRAM density [18]–[20]. These constraints collide with next-generation system needs, as machine-learning-based processing tasks continue to require ever more DRAM [21].

One way to address this challenge is via DRAM oversubscription: allowing more memory to be simultaneously allocated than is physically present in the system. In non-real-time systems, paging is often utilized to support this by moving ("paging out") the least-recently-used (LRU) pages to permanent storage until they are again needed. Unfortunately, the real-time systems community often discounts such an approach due to overheads commonly associated with accessing storage hardware, unpredictable page selection due to imprecise LRU tracking, and analytical costs stemming from operating system (OS) state synchronization. While other issues exist, these specific issues *need not apply*.

In this work, we demonstrate how the counter-intuitive *predictability* of GPUs in embedded systems from NVIDIA, combined with advances in solid state drives (SSD), can be applied to construct a predictable, high-throughput, and low-overhead DRAM oversubscription scheme for real-time systems via transparent paging of GPU buffers to SSD storage.

A) Related Work: Support for DRAM oversubscription of any sort in the real-time community has focused on compile-time transformations [16], [17] and small-scale systems [15]. Beyond the real-time systems community, work to support oversubscription of GPU DRAM [22]–[26] has focused on paging GPU memory to CPU memory—an intractable approach on embedded systems where CPU and GPU share the same DRAM. This paper builds most closely on the concept of "Scheduler-Assisted Prefetching" [27], which operates on the principle that pages can be predictably loaded into memory based on scheduler foreknowledge. Our work also draws inspiration from the PREM memory-management model [28].

B) Contributions: In this first work supporting GPU DRAM oversubscription via transparent paging to storage, we:

1) Demonstrate how unique properties of table-driven embedded real-time systems can negate downsides typically associated with paging DRAM.
2) Show how GPU design and SSD trends further negate downsides often associated with paging DRAM.
3) Benchmark and profile demand paging in Linux, identify shortfalls, and address these in our GPU paging system.
4) Integrate our GPU paging technique into a real-time scheduling and locking model for component-driven real-time systems with accelerators called TimeWall [1].
5) Extend the NVIDIA driver to support oversubscription of GPU DRAM via transparent paging to SSD on the NVIDIA Jetson Xavier, a commodity embedded system.
6) Benchmark our GPU paging implementation against alternative approaches.
7) Experimentally measure overheads and DRAM interference caused by our GPU paging implementation.

*Work was supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

TABLE I
EXPERIMENTAL SYSTEM SPECIFICATIONS

Platform	NVIDIA Jetson AGX Xavier
CPU	NVIDIA 8-core ARMv8 @ 2.26GHz
GPU	NVIDIA Volta Integrated GPU, 512 CUDA Cores @ 1.37GHz
DRAM	1x 16 GiB 256-bit LPDDR4x @ 2133MHz
SSD	Sabrent Rocket 4.0 Plus 1TB, on x4 PCIe 4.0 bus
Operating System	NVIDIA L4T 32.7.2 (Linux 4.9.253)
CUDA Version	10.2

In all, our work enables flexible, transparent, and fast oversubscription of GPU memory on NVIDIA's embedded platforms in a way that can be applied to real-time or best-effort systems.

The remainder of this paper covers our contributions in order. We provide background, assumptions, models, and justifications in Sec. II. Sec. III applies this information, in cohort with data from microbenchmarks, to design a GPU DRAM oversubscription system. Sec. IV presents details on our study of GPU function, and details our implementation of GPU DRAM oversubscription. Sec. V benchmarks this implementation against demand paging and direct I/O, and measures any DRAM or bus interference. Sec. VI touches on additional related work before we conclude in Sec. VII.

II. BACKGROUND

Here we detail notation, demand paging, our real-time system model, our GPU platform, and SSD function. For specificity and reproducibility, Table I lists our system setup.

A. Unit Notation

Throughout this work, we use ISO standard notation [29] such that kB/MB/GB refers to $1000/1000^2/1000^3$ bytes, and KiB/MiB/GiB refers to $1024/1024^2/1024^3$ bytes. Permanent storage speeds and capacities generally use SI units, while DRAM and software generally use power-of-two units.

B. Demand Paging

A common feature in today's general-purpose operating systems is demand paging, which allows for main memory (DRAM) oversubscription. These implementations generally rely on the assumption that the least-recently-used (LRU) memory is least likely to be used next. As available memory declines, such systems "page out" the least-recently-used pages by copying them to a dedicated area of permanent storage (such as a disk or SSD), marking any virtual memory mappings non-present, and freeing the copied pages for the use of others. Such *paged-out* memory generally remains on disk until an application attempts to use it, generating a *page fault*. Page faults, a hardware interrupt, occur when software attempts to access virtual memory addresses that are marked non-present. Upon receipt of a page fault, the OS first checks if the associated page is paged out, and if so, executes a "page in" operation. Paging in is the reverse of paging out, and involves copying that page from storage into memory, remapping the virtual address, and deleting the now-unused copy from storage. This process of paging in data on receipt of a page fault is called *demand paging*.

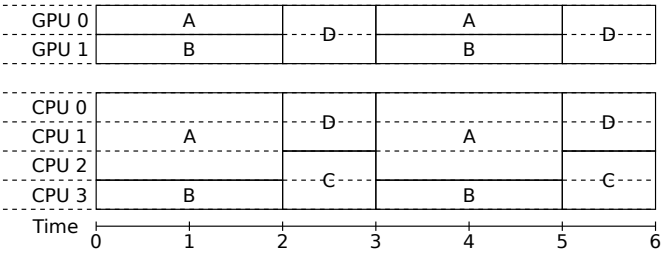

Fig. 1. Example component schedule with TimeWall. Boxes are components.

Demand paging in, and LRU paging out, can be rife with overheads and unpredictabilities. Most processor designs, including x86-64 and ARM64, allow for only highly approximate forms of LRU page tracking, which results in unpredictable decisions as to which page is least-recently used. This non-determinism makes paging-out decisions, and the associated eventual page faults, hard to predict. Some work [16], [17] has addressed this page-selection problem by using intra-application static analysis rather than LRU, but the shortcomings of static analysis limit generalizability. We will experimentally return to the issue of overheads in Sec. III-A.

C. Real-Time Model

As covered in Sec. 1, composable systems are crucial to enabling the certifiability of increasingly complex embedded systems. The composable system model considered in this work is called TimeWall [1], and was developed to enable composability in multicore platforms with accelerators. The system is composed of components, where each component can be independently certified and co-run with an arbitrary set of competing components. The framework ensures *temporal isolation* between components on CPUs and GPUs, is designed to use *spatial partitioning* of caches and DRAM between components, and assumes that all non-isolated hardware resources are put under maximum contention during the certification process. This allows for components to internally use arbitrary schedulers and run arbitrary tasks without affecting other components in the system. Each component is defined by a number of cores, number of GPUs, computation budget, and period. We extend this model to include the amount of DRAM required by each component on each computing resource.

A diagram of such a system is shown in Fig. 1. This figure illustrates the schedule of a quad core system with two GPUs shared across four components (numbered A to D). Component A, for example, has a budget of 2 time units, a period of 3 time units, and requires three CPUs and one GPU. An important property of this system is that the component schedule is fully deterministic, and can be executed by a table scheduler. See Amert et al. [1] for further discussion.

D. GPU Platform

This work uses NVIDIA's Jetson AGX Xavier development board as an exemplar of an embedded platform containing an advanced GPU. This system is based on NVIDIA's Xavier system-on-a-chip (SoC), and is designed for applications in

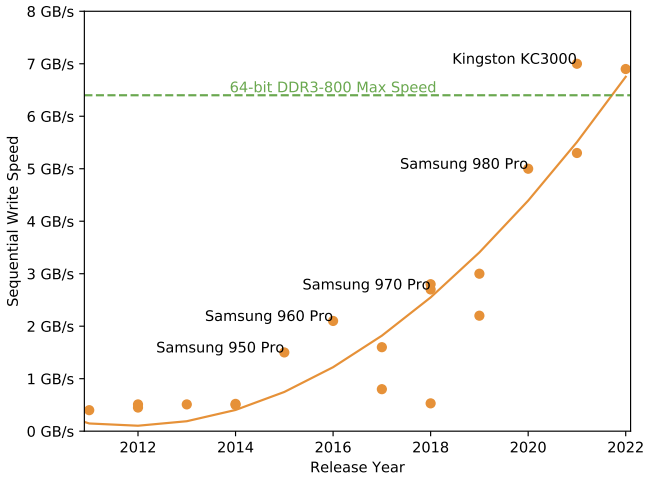

Fig. 2. History of write speeds of high-end SSD available from Samsung, Kingston, and Western Digital/SanDisk over the past 10 years.

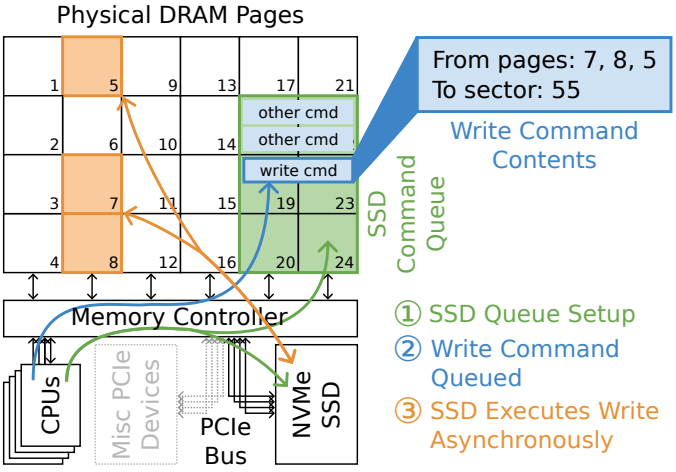

Fig. 3. Illustration of the major steps of I/O on an NVMe drive.

safety-critical real-time systems including robotics and autonomous vehicles. This SoC includes eight ARMv8 cores and 512 CUDA cores in its CPU and GPU respectively. NVIDIA's GPUs are often preferred by autonomous system developers not only for their industry-leading performance and capabilities, but also due to NVIDIA's easy-to-use and capable CUDA API. We consider such a system in this work for the above reasons, but also because (contrary to popular belief) NVIDIA's GPU drivers for their embedded SoCs are fully open sourced under the MIT license [30], [31].

The GPU included in the Xavier SoC is based on NVIDIA's Volta GPU architecture (2017), which includes several memory-management features relevant to this work. All NVIDIA GPUs since at least their Tesla GPU architecture (2006) support per-application GPU virtual address spaces and multiple page sizes. (Note that context switching between these address spaces is costly on GPUs due to the high amount of state present across the hundreds of parallel processors.) Inside each address space, up until at least the Hopper GPU architecture (2022), the default page size is 4 KiB with an option for 64 KiB or larger pages on newer architectures. On NVIDIA's embedded SoC's, these pages are allocated from the same pool used by the CPU, rather than from a dedicated DRAM carveout as in some other SoCs. This implies that using less GPU memory frees up pages for the CPU and vice-versa. We further investigate this mechanism in Sec. IV-A1.

E. Developments in Solid State Drives

SSD speeds have increased geometrically over the past ten years. To illustrate this, we conducted a survey of the advertised read and write speeds for high-end commodity consumer SSDs from three top manufacturers for the past ten years. The surveyed write speeds are plotted in Fig. 2. (SSD read speeds are always in excess of write speeds, so read speeds are omitted for clarity.) Commodity SSD write speeds have gone from 400MB/s to 7GB/s, a 17.5x improvement—putting SSD speeds above the DRAM speeds of just a few years ago (64-bit DDR3-800, shown as green dashed line in Fig. 2). Yet costs have declined, from over $1/GB to less than 20¢/GB for the high-end SSDs plotted in Fig. 2.

For the purposes of this work, it is important to understand how SSDs interface with the OS to execute I/O operations. The steps and components involved in a single sequential write are shown in Fig. 3. This figure assumes a standard NVMe SSD. The first step ①, typically occurs during system startup. In this step, the OS allocates protected memory in DRAM for an SSD control queue and configures the SSD with the location of this queue via a register write. In step ②, the OS creates and queues a sequential write command into the queue from ①. This write command essentially consists of a list of physical pages to write, and the starting sector of the write. Note that the listed pages need not be contiguous, even though they will be written to the SSD sequentially as a contiguous block. When the pages are discontiguous (as in Fig. 3), the write is called a *gather* operation. (*Scatter-gather* support is required of NVMe SSDs.) In ③, the SSD directly reads the queued command from DRAM, and executes it by copying page 7 into sector 55, page 8 into sector 56, and page 5 into sector 57. This concludes the write operation.

Support for scatter-gather is crucial to achieving high-throughput. Pages needing I/O are generally scattered across DRAM, and the time to create, queue, and start an I/O command can exceed the time to copy a single page ($< 1\mu s$).

SSDs include microcontrollers for command processing and flash management. Not all designs have desirable real-time worst-case performance, but a rich body of literature seeks to address this [32]–[35], and so we do not consider it further.

To enable high write speeds, some SSDs will temporarily treat their TLC (Triple-Level Cell) NAND as pseudo-SLC (Single-Level Cell) NAND. This is referred to as Dynamic SLC Caching, and can result in substantially reduced I/O performance when more than one-third of a TLC SDD's capacity is in use. Our platform includes an SSD with this feature, so we avoid filling more than one-third of the drive.

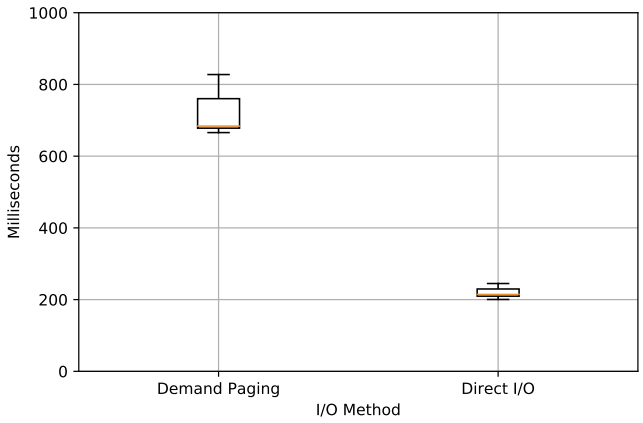

Fig. 4. Box plot of time to read and walk 1 GiB from our SSD.

III. Designing GPU Paging

Given the constraints of real-time systems, GPUs, and SSDs, how can we enable useful OS-level memory oversubscription support on our platform? What carries over from traditional demand paging? What aspects of real-time systems can we leverage? How can we design a system that's applicable to as many task systems as possible?

A. To Demand Page, or Not to Demand Page?

In Sec. II-B, we commented on the unsuitability of LRU page selection for paging-out decisions in a real-time system, but what of demand paging in?

To evaluate demand paging, we created and ran a microbenchmark test on our system to measure how quickly it can load 1 GiB of data from an NVMe SSD. To emulate demand paging, we memory-mapped (via `mmap`) our SSD block device, and sequentially walked the buffer at a 4 KiB stride to page fault in all 1 GiB. For overhead comparison, we run the same test with direct I/O (via `read()` on the device opened with `O_DIRECT`) substituted for demand paging.[1] Both sets of measurements include allocation and sequential walk times. We ran each experiment 1,000 times, clearing Linux's page, dentry, and inode caches between each experiment, and plot the results in Fig. 4 as a box plot.

Fig. 4 shows that demand paging is, on average, 3.2x slower than direct I/O, with a standard deviation 3.8x higher. This clearly indicates that, from a throughput perspective, for moving program data to or from storage, bulk, direct I/O is preferable over a demand paging system similar to Linux's.

Given the unsuitability of LRU eviction and demand paging, what can we substitute in a real-time system?

B. Leveraging Real-Time Foreknowledge

Real-time systems both require and exhibit determinism. This determinism is often present as a *foreknown schedule* (such as with our model), which can be leveraged to enable paging without the need for inaccurate LRU tracking or high-overhead demand paging. We illustrate this in Figs. 5-8.

[1]Find our experimental code at http://cs.unc.edu/%7Ejbakita/rtss22-ae.html.

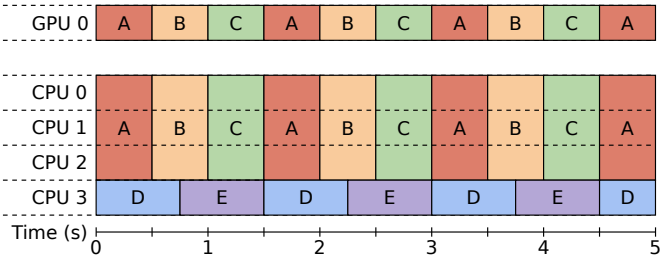

Fig. 5. Ideal schedule

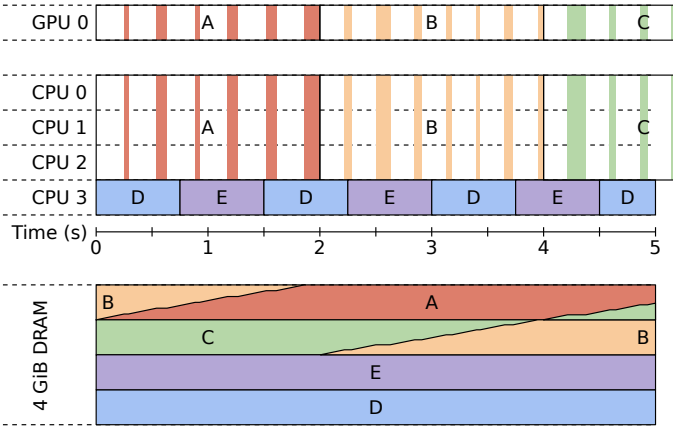

Fig. 6. Schedule using demand paging

Our example system contains five components, A, B, C, D, and E, all with a target period of 1500 ms. Components A, B, and C require 500 ms of computation, while D and E need 750 ms. All components require 1 GiB of DRAM. The ideal schedule is shown in Fig. 5, and is easily achievable for systems with at least 5 GiB of DRAM—but what if the system only has 4 GiB? For the following discussion, we assume SSD read or write rates of 1.33 GiB/s when demand paging, and 4 GiB/s when using direct I/O. These rates match those found in Fig. 4. The schedule with demand paging and LRU eviction is shown in Fig. 6. Observe the long gaps between shaded

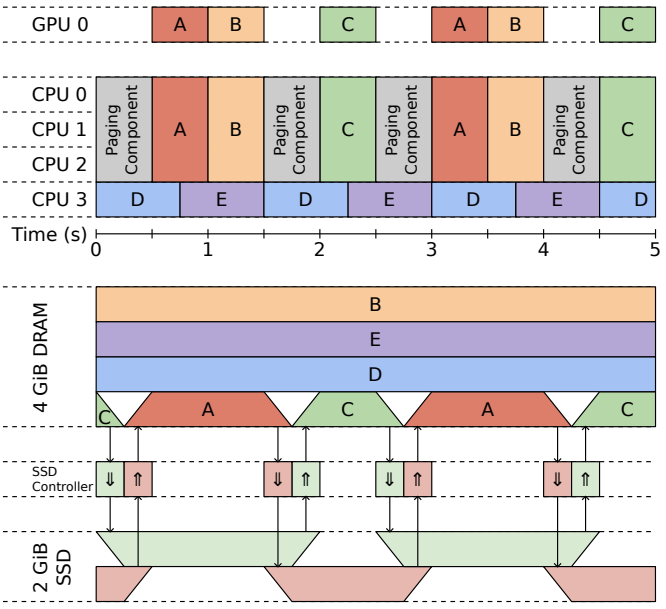

Fig. 7. Scheduling using a paging component

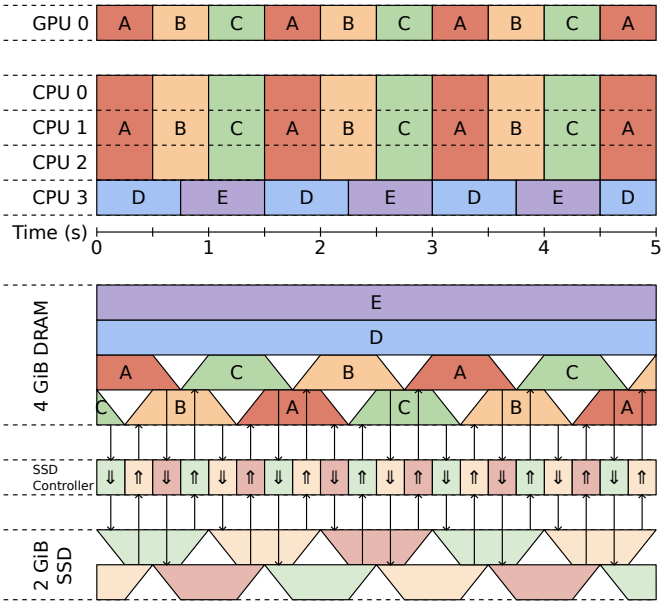

Fig. 8. Schedule using asynchronous paging (our approach)

areas—those represent times when components A, B, or C are suspended to process page fault-in operations. As LRU eviction ensures that each of these components has no resident pages at the start of their executions, they must save 1 GiB and load 1 GiB each time at 1.33 GiB/s, adding 1500 ms. Such a system requires 6000 ms periods on A, B, and C—4× longer than the 1500 ms target.

One way to reduce this capacity loss is shown in Fig. 7. In this diagram, demand paging and LRU eviction are disabled. A "paging component" is added before component A and C that does direct I/O to swap out A's memory for C's memory and vice-versa.[2] Starting with this figure, we include timelines of SSD controller and storage status. Note how most of the read and write work is done by the SSD controller. The paging component allows A, B, and C to support a period of 2500 ms—better, but still nearly 2× the target.

But we can do better! Consider how NVMe SSD I/O works (as detailed in Sec. II-E): I/O commands are put in a queue that is asynchronously executed by the SSD. The CPU need only create and queue commands—a highly optimizable process. This leads to the approach proposed by this paper and shown in Fig. 8. Our table-driven component schedule gives us full information about what will run next, so we queue up asynchronous paging I/O operations for the SSD controller to execute concurrently with component executions. This allows all needed pages for each component to be paged-in by the start of their time slice, resulting in support for 25% DRAM oversubscription and the target 1500 ms period.

For this approach to widely apply, paging must be as fast and as asynchronous as possible to allow for paging small-period tasks. In Fig. 8, we assume minimal blocking or other CPU overheads. How can we design our system to approach no overheads?

[2]This variety of synchronous paging is very similar to the swapping of time-sliced systems from the early days of computing [36].

C. Managing Overheads

Unfortunately, merely discarding demand paging and LRU eviction does not save us from all CPU overheads. State must still by synchronized across cores, pages must still be allocated and freed, and virtual memory mappings must still be redone. None of this can be offloaded to the SSD controller.

One way to avoid these unpredictabilities and overheads is to page GPU memory rather than CPU memory. At first glance, this may sound counterintuitive—GPUs are generally treated as less predictable than CPUs. However, from a memory-management perspective, GPUs are *more predictable*.[3] Additionally, as mentioned in Sec. II-D, GPU and CPU often share DRAM, so saving GPU memory can indirectly free memory for CPU tasks.

This section first justifies our emphasis on overheads with a case study considering demand paging on Linux before addressing how synchronization, page mapping, and allocation overheads can be minimized through paging GPU memory.

1) Demand Paging: A Case Study in Runaway Overheads: Overheads can dominate the I/O cost in a poorly designed paging system, and we use demand paging in Linux as a case study of this. Fig. 9 is a performance profile[4] of the page fault handler as triggered by our microbenchmark from Sec. III-A.

In this flame graph [37], each box represents a single function, the box width represents the frequency of function occurrence in the profiling trace, and the boxes are arranged top down according to callee/caller relationships; *i.e.*, the graph is hierarchically ordered such that the box below each function is its immediate caller. For example, `do_page_fault` is called[5] by `seq_walk`. x-axis ordering is meaningless.

Note that *uncontended* lock acquisition, lock release, and retry-based commit functions (blue) collectively consume 40% of the profile, page mapping (green) consumes 19% of the profile, and page allocation (gray) consumes 14% of the profile. In our *best-case* and *uncontended* system, I/O time is not even visibly discernible due to the runaway overheads.

2) Avoiding Multicore Synchronization: The significant synchronization overheads of the prior section stem largely from Linux's need to share state across cores to support dynamic workloads on multicore systems.

GPU paging can avoid most multicore synchronization overheads, as only a single application and virtual address space pair execute on the GPU at any given time.[6] CPU tasks may queue work or request changes to GPU state while another task is running on the GPU, but these operations only take effect after a GPU context switch and can be deferred.

3) Accelerating Page Mapping: On most platforms, the cost of updating virtual address spaces—page mapping—is largely a function of the number of page table walks required. Theoretically, only one full page-table walk is required per

[3]Note that we are referring to the fundamentals of GPU hardware, not CUDA. Some aspects of CUDA's memory management are unpredictable.

[4]Using Linux's `perf` user- and kernel-space profiler, sampling at 1kHz.

[5]Implicitly, as this is the page-fault handler.

[6]Except with NVIDIA's Multi-Process Service (MPS) or Multi-Instance GPU (MIG), but these are unavailable on NVIDIA's embedded platforms.

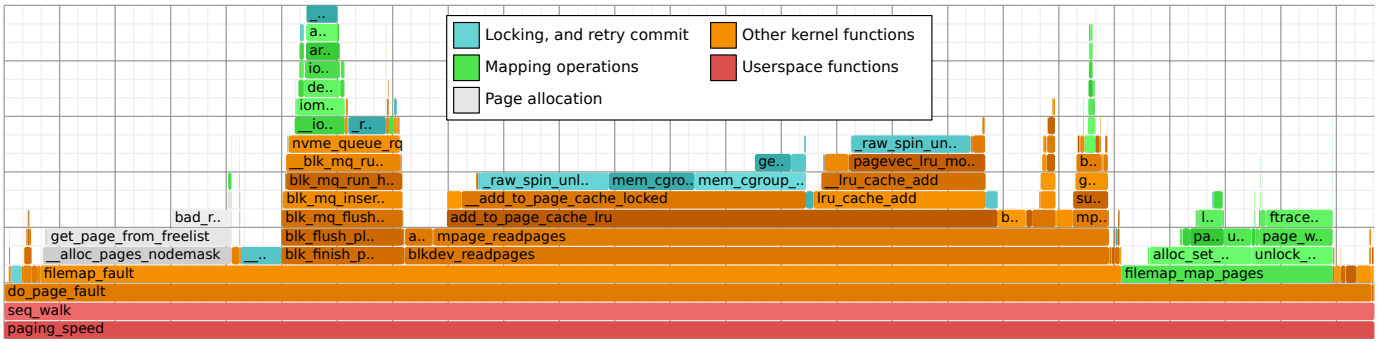

Fig. 9. Profiling results for demand paging, plotted as a flame graph.

remapping of a virtually contiguous address range. On CPU, this cost can become inflated if pages are shared across applications (requiring per-application page-table walks), or if paged buffers are small and virtually discontiguous. Both situations are common in modern CPU applications. On a GPU, shared pages are mostly unsupported, and large, virtually contiguous buffers are common (such as with machine-learning model weights).

4) Addressing Allocation: On a CPU, the widespread expectation that commodity operating systems support memory overcommit has made it extremely difficult to predetermine which application will need which pages at which times. This dynamism requires the use of complicated allocators (such as Linux's buddy allocator), and precludes the use of a "memory schedule." GPUs do not generally support memory overcommit, and memory preallocation is common. This allows for the use of GPU pages to be simply scheduled out, and for interactions with the complex Linux allocators to be avoided.

D. Additional Design Considerations

As with conventional CPU paging systems, we build our system to be completely transparent to userspace applications. From CUDA to YOLO, applications should require no changes to be usable with GPU paging. During nominal operation, the fact that an application's GPU memory was paged out and back in between periods should be *logically* undetectable.

Unfortunately, nearly any operation in a modern system imputes some interfering side-effect on tasks which is *temporally* detectable. Reading and writing from DRAM to the SSD adds bus and memory controller interference for all components in our system. We address this by assuming that during the component certification process, "evil tasks" are run on other cores and on the SSD to emulate maximum interference. We estimate the amount of this interference in Sec. V-C.

The remainder of this work implements and evaluates the transparent GPU paging system that we have now described.

IV. Implementing GPU Paging

We have now discussed how paging GPU memory with real-time foreknowledge can obviate many of the issues traditionally associated with paging in a real-time system. In this section, we investigate how to implement it.

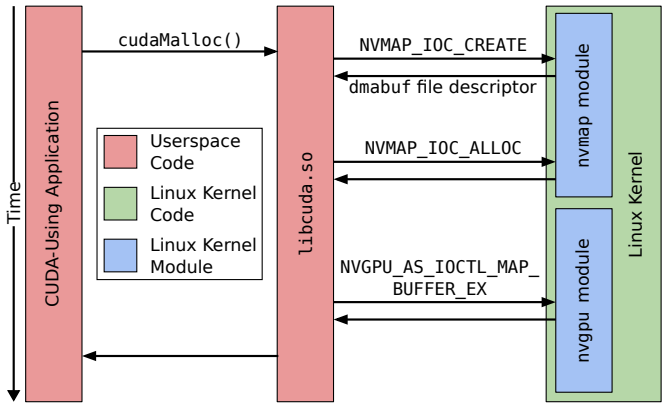

Fig. 10. Steps for GPU memory allocation with NVIDIA's embedded drivers.

A. Existing GPU Capabilities and Design

NVIDIA GPUs have been much lambasted for their poorly documented and often closed-source software, with much work dedicated to merely understanding the fundamentals of GPU memory [38], [39] and scheduling [40]–[42]. In accord with those works, we investigate how GPU page allocation and mapping work for NVIDIA's embedded, integrated GPUs in order to enable paging of GPU memory.

1) Creating GPU Memory Buffers: In Sec. II-D, we highlighted that both GPU and CPU pages are allocated from the same pool on NVIDIA's embedded platforms. NVIDIA's documentation notes this, but gives no details on *how* it works.

To discern such details, we created a simple application using NVIDIA's CUDA API that allocates and accesses GPU memory. We ran and profiled its syscalls with Linux's `strace` tool, finding several IOCTL syscalls closely associated with memory allocation via `cudaMalloc`.

To understand the nature of these syscalls, we obtained the headers for NVIDIA's `nvmap` and `nvgpu` drivers, and extended `strace`[7] to decode them. The result is illustrated in Fig. 10, and the following explanation is informed by our close reading of the source codes [30], [31].

GPU memory allocation begins with the IOCTL syscall `NVMAP_IOC_CREATE`. This syscall takes a single `size` parameter and returns a newly created `dmabuf` FD (file descriptor) handle, but does not allocate any backing pages. The following `NVMAP_IOC_ALLOC` syscall allocates the pages,

[7]Code linked to from http://cs.unc.edu/%7Ejbakita/rtss22-ae.html.

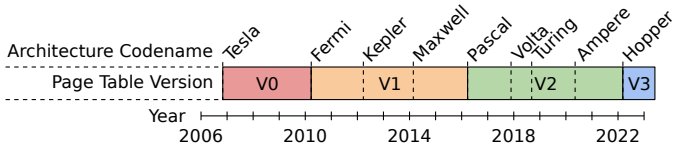

Fig. 11. Timeline of NVIDIA GPU page table versions.

as parameterized by the FD handle, a heap mask, flags, and alignment. The heap mask controls the source of the pages, by default Linux's buddy allocator. The resulting allocation is not physically contiguous, and is internally tracked as a scatterlist. At this point, the pages are allocated, but not mapped into GPU virtual memory.

Mapping is more complicated, and first requires address-space creation with the `NVGPU_GPU_IOCTL_ALLOC_AS` syscall (not shown; normally done during CUDA library initialization). The `NVGPU_AS_IOCTL_MAP_BUFFER_EX` syscall can then be called on the address-space handle to map pages into the GPU virtual address space (shown). This syscall is parameterized by the type of page mapping, the offset in virtual memory, and the `dmabuf` FD obtained from the earlier `nvmap` call. This syscall will be covered further in Sec. IV-A2. Once allocated and mapped, buffers persist until unmapped via an `NVGPU_AS_IOCTL_UNMAP_BUFFER` syscall and freed with an `NVMAP_IOC_FREE` syscall.

To our understanding, NVIDIA's embedded GPUs have all handled page allocations this way since the Tegra K1 in 2014, and NVIDIA's QNX and Horizon OS[8] drivers work similarly.

2) Inside GPU Virtual Memory Management: In Sec. II-D, we mentioned that NVIDIA GPUs have long supported per-application virtual address spaces. However, little public information is available regarding how they map pages or are activated—necessary background for GPU paging. We provide historical framing first, then dive into the relevant specifics.

NVIDIA's GPU page-table layouts (and consequent mappings) have changed over the years, and only one generation—Pascal (2016)—is somewhat documented [43]. We disambiguated this history by cross-referencing the `nvgpu` and `nouveau` driver sources, alongside NVIDIA's limited documentation [43]. Our findings are shown in Fig. 11.

Fig. 11 shows that the Volta-based GPU in our platform supports the second generation of NVIDIA's page tables. These five-level page tables support 4 KiB, 64 KiB, or 2 MiB pages. Each page table is bound to a "channel," and context switches between channels are controlled by NVIDIA's time slice group (TSG) runlist scheduler [41]. Details on the runlist scheduler are a topic for another work, but it suffices to say that no channel is active unless it has pending work. As the real-time system model we use (TimeWall [1]) strictly enforces that no GPU work is pending at the conclusion of a component's time slice, we know that the page table for a component's GPU work is only active during its time slice. This allows for an important simplification in Sec. IV-B3. Additionally, GPU context switches can be very expensive [41], so longer GPU time slices are preferred for performance.

[8]Also known as the Nintendo Switch System Software.

Algorithm 1 GPU Page Mapping

```
1:  procedure MapContiguous(pages, offset, flags)
2:      pte ← FindPTE(offset)                    ▷ Walk page table
3:      for all page ∈ pages do
4:          WritePTE(pte, page, flags)           ▷ Overwrite PTE
5:          pte ← NextPTE(pte)                   ▷ Increment
6:      end for
7:  end procedure
    Map a set of discontiguous page sets.
8:  procedure MapScatterlist(sgl, offset, flags)
9:      SetupIOMMU(sgl)    ▷ Grant GPU physical access
10:     for all page_set ∈ sgl do
11:         MapContiguous(page_set, offset, flags)
12:         offset += Length(entry)
13:     end for
14: end procedure
```

The `nvgpu` driver creates page-table mappings when requested by internal functions or userspace syscalls. Alg. 1 shows the steps to create a new mapping in the `nvgpu` driver.

Problems with this algorithm to improve upon include that, to map n pages contiguously, it may perform up to n page-table walks. This problem stems from Lines 10 and 11. Each scatterlist entry points to one or more pages, with the number being inversely correlated to the memory fragmentation of the system. MapContiguous walks the page tables on each call, and is called for each scatterlist entry. This results in the number of page-table walks, and overall mapping speed, implicitly becoming a function of system memory fragmentation—a highly unpredictable design.

An additional relevant aspect of Alg. 1 is the SetupIOMMU call at Line 9. Devices such as the GPU and SSD on our platform are prevented from freely accessing physical memory by an I/O MMU. This protects main memory from malfunctioning or malicious devices, but adds non-negligible overheads to paging (visible in Fig. 9 as the left green stack).

The development of our implementation involved understanding many more platform and GPU mechanisms, but we omit the non-essential digression.

B. Assembling Transparent GPU Paging

Given our high-level design goals for GPU paging and an understanding of the technical limitations of NVIDIA's hardware and drivers, we now present our assemblage of these components into a system supporting GPU paging.

GPU paging in (resp., out) involves three major steps: allocating (resp. freeing) backing pages, reading (resp., writing) saved state from (resp. to) the SSD, and updating (resp. invalidating) virtual memory mappings. We cover these steps in order, and then present our control API for GPU paging.

1) Handling Buffer Allocations: To allow for freeing and reallocating backing pages as part of a page out/in cycle, we extend the `nvmap` module. By default, this module provides no means to free a buffer's backing pages without also freeing the FD handle. As these handles are held by userspace, and

we aim for a transparent paging system, the FD handles must be preserved. Recreating an `nvmap` buffer with an identical FD is somewhat difficult, so we instead add an API to roll back an `nvmap` buffer to a pre-`NVMAP_IOC_ALLOC` state. We call this API `NVMAP_IOC_DEALLOC` and pair it with an `NVMAP_IOC_REALLOC` API which does the reverse.

To select the pages freed or allocated by these APIs, it is possible to use a memory schedule as mentioned in Sec. III-C4. However, for simplicity in our initial implementation, we instead rely on `nvmap`'s default allocator.

2) I/O: We rely on Linux's preexisting in-kernel framework for asynchronous block I/O. In particular, to reduce overheads, we chain our block I/O commands to join the buffer scatterlists into a single, large scatter-gather I/O command. This command encompasses the full region to be paged in/out and can be passed as a single command to the NVMe SSD controller.

Prior to page-out writes, we flush the GPU Level-2 (L2) cache, as DRAM accesses from PCIe devices (such as our NVMe SSD) are not coherent with the GPU cache.

3) Handling Page Mappings: As part of paging, normally virtual memory mapping invalidation is part of paging out, and mapping recreation is part of paging in. The invalidation step prevents applications from mistakenly or maliciously attempting to access pages no longer allocated to them upon page-out completion. We forgo this step, as our model ensures that memory is paged in prior to the start of a GPU time slice, and that GPU work runs strictly within its allocated time slice (as discussed in Sec. IV-A2).

Conversely, remapping is necessary, as during buffer reallocation we may receive a different set of backing pages (see Fig. 8 for example). This is performed similarly to NVIDIA's page mapping code (outlined in Alg. 1), but with three key differences. First, we skip logic for PDE (Page Directory Entry) or PTE (Page Table Entry) creation (not shown in Alg. 1). Our transparent GPU paging approach maintains the same GPU virtual addresses before and after paging, so we can safely assume that all needed PDEs and PTEs already exist.

Second, rather than fully overwriting a PTE as in Line 4 in Alg. 1, we update only the pointer to the DRAM page. This avoids duplicate tracking of metadata already present in the page table, as necessary with NVIDIA's algorithm.

Third, to aid in predictability, we perform a page-table walk for every page mapping. This matches the worst-case algorithmic efficiency of NVIDIA's algorithm, but avoids linking execution time with memory fragmentation. A single-page-walk approach in the worst case is possible and greatly desirable. Unfortunately, it requires immensely complex code to handle the mix of page sizes allowed for by NVIDIA's page tables, and so we leave this for future work.

On our platform, CPU DRAM accesses are not always cache coherent with the GPU. To work around this, we flush the GPU L2 and Translation Lookaside Buffer (TLB) caches after GPU page table mapping updates.

4) Kernel and Userspace API: We extend the `nvgpu` driver with an API allowing either a specific buffer, or all buffers in a specified GPU virtual address space to be paged.[9] Our API supports either synchronous or asynchronous modes (as in Fig. 7 and Fig. 8 respectively). This API can be accessed from user- or kernel-space via the following new IOCTL syscalls on a GPU-virtual-address-space handle:

1) `NVGPU_AS_IOCTL_WRITE_SWAP_BUFFER`
2) `NVGPU_AS_IOCTL_READ_SWAP_BUFFER`
3) `NVGPU_AS_IOCTL_WRITE_SWAP_BUFFER_ASYNC`
4) `NVGPU_AS_IOCTL_WRITE_SWAP_BUFFER_ASYNC_FINISH`
5) `NVGPU_AS_IOCTL_READ_SWAP_BUFFER_ASYNC`
6) `NVGPU_AS_IOCTL_READ_SWAP_BUFFER_ASYNC_FINISH`

These syscalls all take one integer parameter, the FD handle of the buffer to be paged, or `NVGPU_SWAP_ALL` to indicate a paging of all buffers.

The `*_ASYNC` calls initiate a paging operation, up through the dispatching of an I/O command to the SSD. These calls must be followed by a call to the respective `*_ASYNC_FINISH` call, which checks that the I/O is completed (or waits if it is not) before finishing the paging operation with a free or mapping. The non-postfixed calls combine the two `*_ASYNC_*` calls into a single blocking operation to save a lock acquisition and some internal lookups.

Our API is thread safe, and uses a per-GPU-virtual-address-space lock internally to prevent race conditions. This lock is fully private to an application and cannot be obtained otherwise. For global state such as SSD sector assignments for paged data, we use atomic variables.

Not all GPU buffers are pageable. Specifically, any pinned host ("zero-copy") memory is unpagable. This is because zero-copy memory buffers are simultaneously mapped into GPU and CPU virtual address spaces. Updating CPU virtual address mappings raises a bevy of predictability issues, as covered in Sec. III-C, so we do not attempt to support it. Our API includes robust internal error checking, and will return a descriptive error code upon this, or any other, misuse.

When an entire buffer is marked as read-only in GPU memory, we can execute the copy step of paging out only once. This allows the same, saved buffer to be reused during each subsequent page-in operation. As SSDs have a limited number of write cycles,[10] this not only increases performance, but extends the lifetime of the SSD. Machine learning algorithms often fill the majority of GPU memory with read-only weight files, so this is commonly useful.

In totality, our API allows for any or all GPU buffers to be paged upon request, synchronously or asynchronously. This enables both our TimeWall-extending model, and any other, yet to be devised applications. For additional details on any aspect of our implementation, we refer readers to our well-documented source code.[11]

[9] An application may simultaneously manage multiple GPU address spaces.
[10] Around 1,000 TLC writes per cell.
[11] Get our kernel code at http://cs.unc.edu/%7Ejbakita/rtss22-ae.html.

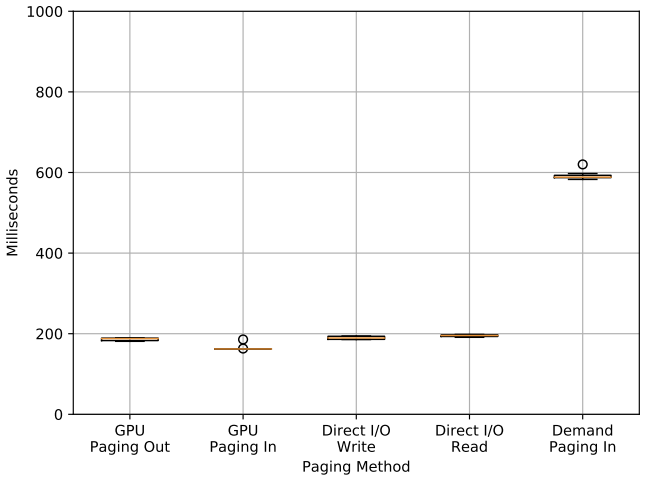

Fig. 12. Box plot of time to page in or out (read or write) 1 GiB.

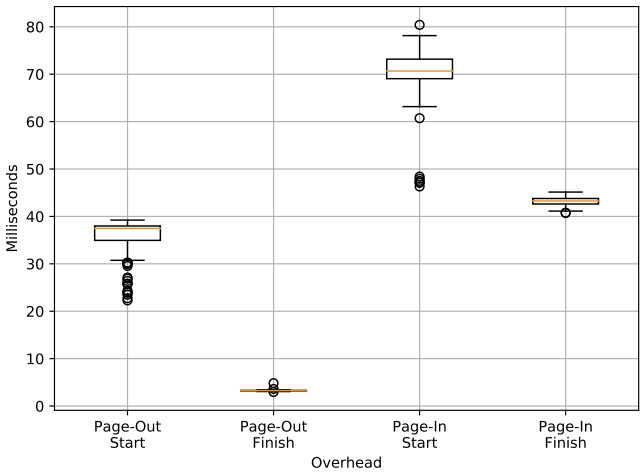

Fig. 13. Box plot of time to initiate or finish a 1 GiB GPU paging operation.

V. Evaluation

We now evaluate the performance and utility of our paging system via various performance and overhead benchmarks.

A. Throughput

We benchmark the throughput of our GPU paging system against both direct I/O and demand paging. We test the time to write out and free, or allocate, read in, and map, a buffer. With demand paging, where page-out operations are implicit, we cannot test the time to write out a buffer. All operations are performed on a randomly filled, 1 GiB buffer with a one-second pause between a read-in and write-out operation. We use `O_SYNC` with direct I/O to insure that writes are flushed to disk. We emulate demand paging as in Sec. III-A, but subtract out sequential walk times.[12] For GPU paging, we use our synchronous APIs in this experiment.

The experimental results are shown in Fig. 12 as a box plot. All benchmarks are run uncontended and for 1,000 samples.

Obs. 1. *GPU paging is three times faster than demand paging.*

GPU paging is faster than demand paging, but its speed even slightly exceeds direct I/O—a laudable reduction of overheads. Loading data on average takes 161 ms, 192 ms, and 615 ms for GPU paging, direct I/O, and demand paging respectively. Writing data on average takes 186 ms and 189 ms for GPU paging and direct I/O respectively.

Our choice of paging GPU buffers to avoid the overheads of CPU paging appears to have worked well. We aimed to approach the speeds of direct I/O, but in fact exceeded them! We suspect this is the case because direct I/O is controlled and parameterized by userspace—a design that requires more internal checks and memsets to prevent malicious use.

An implication of these results is that our approach is faster than any application could implement on its own. One may have thought that good design would be to have an application reduce its memory consumption internally by moving unused buffers to and from storage as necessary. However, as there are no general-purpose userspace I/O APIs faster than direct I/O, this appears to not be the case.

Obs. 2. *GPU paging is more predictable than direct I/O or demand paging.*

Even in the uncontended environment of Fig. 12, which is favorable for direct I/O and demand paging, our GPU paging system exhibits slightly more timing determinism. While difficult to see in Fig. 12, GPU paging times have a slightly slower standard deviation than direct I/O or demand paging. The standard deviations for loading data are 0.97, 6.39, and 6.48 for GPU paging, direct I/O, and demand paging, respectively. The write standard deviations are 3.70 and 3.77 for GPU paging and direct I/O, respectively.

B. Synchronous Overheads

As discussed in Sec. III-C, a major goal of our system design is to avoid the levels of overhead that afflict demand paging in Linux. Our throughput results show that this goal was largely achieved, but how much room is left for improvement? To test this, we created a benchmark using our asynchronous GPU paging APIs. This benchmark calls each API on a 1 GiB GPU buffer, waiting at least one second before making the `*_FINISH` calls to allow for all I/O to complete asynchronously. How long each API call takes is measured, and we repeat this cycle for 1,000 iterations. Fig. 13 shows the results as a box plot.

Obs. 3. *GPU page-in operations achieve overheads 81% lower than demand paging.*

Fig. 13 shows that our approach requires 70 ms to start and 43 ms to finish a GPU-page-in operation on average. This sums to synchronous overheads of 113 ms—81% less than the fully-synchronous 592 ms required by demand paging. Our method could be better, as 113 ms is still most of the total 161 ms GPU-page-in time, and omits interrupt processing time.

[12]Emulating demand paging requires a sequential walk, so we separately measure the time of this and subtract it from the demand paging times.

As mentioned in Sec. IV-B2, we use Linux's built-in mechanisms to dispatch our block I/O commands. While Linux can theoretically pass our entire command at once to the SSD controller, we found this not to be the case. Instead, the command is split, copied and remarshaled at least once by other parts of the kernel. This meaningfully inflates both page-in and page-out operation start overheads. Further, while our implementation does not require interrupts to register completion, Linux forces them by default. Future work, or a production implementation, could directly dispatch commands to a dedicated SSD control queue and disable interrupts to avoid these overheads.

Additionally, we incur meaningful overheads from our simplifying choices to re-purpose `nvmap`'s backing page allocator and to use an inefficent page-mapping algorithm. See Sec. IV-B1 and Sec. IV-B3 for further discussion.

Obs. 4. *GPU page-out operations are 80% asynchronous.*

Fig. 13 shows that our approach requires 35 ms to start and 3 ms to finish a GPU-page-out operation on average. This sums to synchronous overheads of 38 ms—only 20% of the total 186 ms GPU-page-out time.

When paging out, we avoid `nvmap`'s allocator and our inefficient page mapping logic, but we still incur some penalties for Linux's non-optimal block I/O dispatch logic.

In all, we successfully reduce overheads to a fraction of those typical for paging, and see opportunities to further cut overheads by half or more using the improvements we outline.

C. Bus and DRAM Interference

As mentioned in Sec. III-D, our GPU paging system is expected to cause DRAM and bus interference when the SSD performs I/O operations. This will be observable as a slowdown in CPU tasks.[13] The relationship is symmetric, in that heavy CPU loads should also slow our GPU paging system. This section measures the magnitudes of these slowdowns.

In order to test worst-case interference, we create DRAM-thrashing "evil tasks." Our DRAM-thrashing tasks work by allocating a buffer several times larger than the CPU caches, then walking each cache line in these buffers in either sequential, or random-dependent, order. In both modes, every cache line is touched in the buffer before the process repeats. This ensures that every access must be fetched from DRAM.

We configure between one and seven (cores minus one) sequential-DRAM-thrashing tasks, and run them both with and without interference from a tightly looped GPU paging task. We plot the minimum bandwidth observed within these thrashing task sets in Fig. 14 (left axis, higher is better) alongside the mean time taken by the co-running GPU paging task (right axis, lower is better).

Obs. 5. *GPU paging cuts CPU DRAM bandwidth by* $< 5\%$.

The gap between the two thrashing-task-bandwidth lines in Fig. 14 shows how much our thrashing CPU tasks are slowed by the addition of GPU paging. This gap is *at most* 5% (0.8 GiB/s). The impact to the speed of GPU paging (dotted line in Fig. 14) is greater, particularly against six or seven DRAM-thrashing tasks. In these cases it is 5% and 19% slower respectively. These mild results are not particularly surprising, as our GPU paging system has a maximum throughput of approximately 5 GiB/s—minor compared to the advertised throughput of 137 GiB/s in the DRAM and bus subsystem.

[13] SSD reads and writes are not coherent with CPU caches, so we do not analyze cache interference.

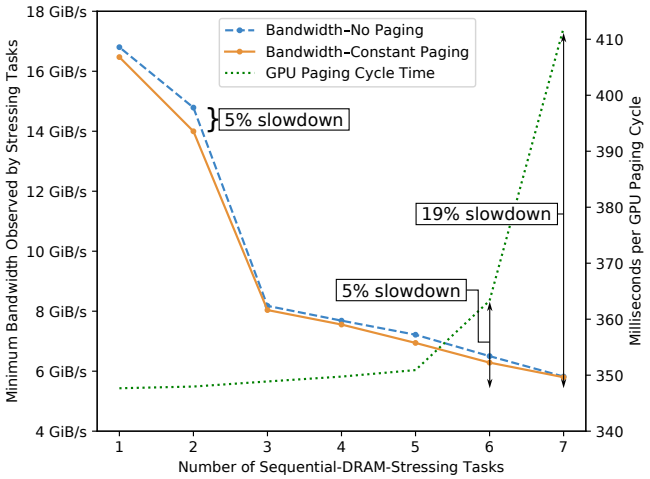

Fig. 14. DRAM interference effects with between one and seven competing sequential-DRAM-thrashing CPU tasks, and our GPU paging.

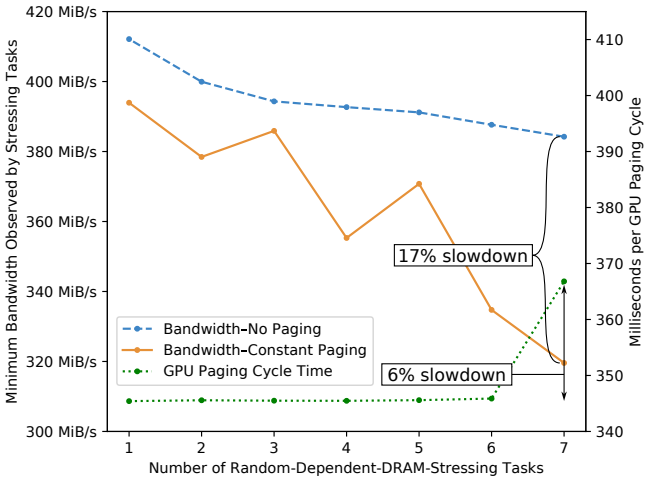

Fig. 15. DRAM interference effects with between one and seven competing random-dependent-DRAM-thrashing CPU tasks, and our GPU paging.

But what of the DRAM latency impact? To measure this, we repeat the same experiment, but with our random-dependent-DRAM-thrashing tasks. These tasks defeat the latency-hiding and prefetching capabilities of modern out-of-order CPUs and caches by making the address of each load unavailable until the prior load fully completes. This is done via a randomly shuffled linked list. The results are shown in Fig. 15.

Obs. 6. *Random-dependent-DRAM-thrashing tasks slow GPU paging by* $< 6\%$.

Conversely to our sequential-DRAM-thrashing experiments,

random-dependent-DRAM-thrashing tasks barely slow GPU paging (dotted line in Fig. 15), but not the reverse. Adding GPU paging slows our random-dependent-DRAM-thrashing tasks by as much as 17% (65 MiB/s). This may indicate that sequential-DRAM-thrashing tasks bottleneck on internal bus limits before saturating DRAM cycles, whereas DRAM-cycle limits bottleneck random-dependent-DRAM-thrashing tasks.

In all, our paging approach typically has negligible DRAM interference impact, slowing other applications at most 17%, while being slowed at most 19%.

D. Maintainability and Robustness

Through careful engineering effort, our implementation adds only 586 lines to the `nvgpu` driver, and only 118 lines to the `nvmap` driver. Many of these lines are comments. This compactness allows for easier analysis, improvement, and maintenance, resulting in less-buggy code. This benefit was evident throughout our experiments, with no crashes, data corruption, or other errors arising from our implementation.

VI. Related Work

Some early papers addressing paging in real-time systems include the work of Puaut and Hardy [16], [17]. These works use compile-time static analysis to determine when to evict or load specific pages to or from storage for a specific program. This general approach has more recently been advanced by the RT-PLRU family of work [15], [44], [45] to not just reduce, but minimize the DRAM necessary in an embedded system. These works are particularly relevant, as they page to and from NAND flash (which SSDs are composed of).

Other works dealing with SSDs in real-time systems generally focus on how to meet real-time guarantees, or merely performance goals, in the SSD controller's flash translation layer (FTL) [32]–[35]. We assume that a production version of our system would apply such a real-time SSD controller.

As far as we are aware, prior work on memory oversubscription with GPUs has dealt with SSDs in a purely theoretical sense [46], or entirely focused on paging from GPU DRAM to CPU DRAM [22]–[25]. NVIDIA supports a variant of this latter sort of GPU memory oversubscription via "CUDA Unified Memory," [26], [47] but does not provide any means for paging to permanent storage such as an SSD.

Prior efforts to understand the behavior of NVIDIA GPUs and CUDA from a real-time systems perspective include a broad set of works directed at queuing [40], [48], [49], synchronization [50], and memory [38] behaviors, as well as further GPU scheduling details [41], [42].

Finally, the body of work considering the impacts of memory interference in real-time embedded systems is vast, so we focus our discussion on works merely considering memory interference in NVIDIA's embedded platforms. The comprehensive performance analysis of Capodieci *et al.* [39] is particularly notable, but other works have studied interference [51] and proposed mitigations [52] for these platforms.

VII. Conclusion

In this work, we studied the capabilities and constraints of modern, real-time embedded systems with GPUs and SDDs, finding a novel opportunity to implement DRAM oversubscription via GPU paging. We design our system to work symbiotically with preexisting real-time component-driven scheduling systems, and explain how GPU paging can be, counter-intuitively, *more predictable* than CPU paging.

We implement our GPU paging system in NVIDIA's embedded Linux drivers, extending our design to permit for use by best-effort tasks as well. We benchmark this implementation, finding it to be three times faster than demand paging, and faster than even direct I/O. We benchmark our overheads, finding GPU paging to reduce synchronous overheads 81% from demand paging—all while causing no more than a 17% slowdown from interference.

In future work, we hope to implement our GPU paging system in a real-time, table driven scheduler, as the modified nature of NVIDIA's kernel made it infeasible to do so here. In the future, we also hope to reduce GPU paging overheads, support paging memory from discrete GPUs, and test the applicability of our technique to graphical applications. Further, we aim to explore how our method could enable fast mode changes or state snapshots for GPU-using applications.

References

[1] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "TimeWall: Enabling time partitioning for real-time multi-core+accelerator platforms," in *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, Dec 2021, pp. 455–468.

[2] B. Forsberg, L. Benini, and A. Marongiu, "HePREM: A predictable execution model for GPU-based heterogeneous SoCs," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 17–29, 2021.

[3] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in Intel processors," in *Proceedings of the 14th European Conference on Computer Systems*, Mar 2019, pp. 1–17.

[4] J. Herter, P. Backes, F. Haupenthal, and J. Reineke, "CAMA: A predictable cache-aware memory allocator," in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, July 2011, pp. 23–32.

[5] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr 2020, pp. 296–309.

[6] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2014, pp. 1–10.

[7] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro, "A tdma-based arbitration scheme for mixed-criticality multicore platforms," in *Proceedings of the 1st IEEE International Conference on Event-based Control, Communication, and Signal Processing*, June 2015, pp. 1–6.

[8] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time SDRAM controller for mixed time-criticality systems," in *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Sep 2013, pp. 1–10.

[9] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based DRAM controller for mixed-criticality systems," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr 2015, pp. 317–326.

[10] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013, pp. 55–64.

[11] G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha, "Schedulability analysis for memory bandwidth regulated multicore real-time systems," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 601–614, 2015.

[12] M. Hillenbrand, M. Gottschlag, J. Kehne, and F. Bellosa, "Multiple physical mappings: Dynamic DRAM channel sharing and partitioning," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, Sep 2017, pp. 1–9.

[13] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[14] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr 2011, pp. 269–279.

[15] K.-S. We, C.-G. Lee, K. Yi, K.-J. Lin, and Y. S. Lee, "HRT-PLRU: A new paging scheme for executing hard real-time programs on NAND flash memory," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 927–940, 2014.

[16] I. Puaut and D. Hardy, "Predictable paging in real-time systems: A compiler approach," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, Jul 2007, pp. 169–178.

[17] D. Hardy and I. Puaut, "Predictable code and data paging for real time systems," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, Jul 2008, pp. 266–275.

[18] B. R. Childers, J. Yang, and Y. Zhang, "Achieving yield, density and performance effective DRAM at extreme technology sizes," in *Proceedings of the 1st International Symposium on Memory Systems*. Association for Computing Machinery, Oct 2015, p. 78–84.

[19] W. P. Noble and W. W. Walker, "Fundamental limitations on DRAM storage capacitors," *IEEE Circuits and Devices Magazine*, vol. 1, no. 1, pp. 45–52, 1985.

[20] K. Itoh, Y. Nakagome, S. Kimura, and T. Watanabe, "Limitations and challenges of multigigabit dram chip design," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 5, pp. 624–634, 1997.

[21] I. Stoica, D. Song, R. A. Popa, D. A. Patterson, M. W. Mahoney, R. H. Katz, A. D. Joseph, M. Jordan, J. M. Hellerstein, J. Gonzalez, K. Goldberg, A. Ghodsi, D. E. Culler, and P. Abbeel, "A Berkeley view of systems challenges for AI," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-159, Oct 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-159.html

[22] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for GPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 345–357.

[23] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in CPU-GPU unified memory," in *2021 Design, Automation, and Test in Europe Conference and Exhibition*, Feb 2021, pp. 1212–1217.

[24] S. Zhang, Y. Yang, L. Shen, and Z. Wang, "Efficient data communication between CPU and GPU through transparent partial-page migration," in *Proceedings of the 20th International Conference on High Performance Computing and Communications; Proceedings of the 16th International Conference on Smart City; Proceedings of the 4th International Conference on Data Science and Systems*, 2018, pp. 618–625.

[25] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Mar. 2015, pp. 65–77.

[26] Chirayu Garg, Nikolay Sakharnykh, "Improving GPU memory oversubscription performance," NVIDIA, Tech. Rep., Oct 2021. [Online]. Available: https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/

[27] S. A. Belogolov, J. Park, J. Park, and S. Hong, "Scheduler-assisted prefetching: Efficient demand paging for embedded systems," in *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2008, pp. 111–119.

[28] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, "Memory-aware scheduling of multicore task sets for real-time systems," in *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2012, pp. 300–309.

[29] ISO Central Secretary, "Quantities and units -— Part 13: Information science and technology," International Organization for Standardization, Standard ISO/IEC 80000-13:2008, 2008.

[30] NVIDIA, "nvgpu git repository." [Online]. Available: git://nv-tegra.nvidia.com/linux-nvgpu.git

[31] ——, "Tegra modules git repository." [Online]. Available: git://nv-tegra.nvidia.com/linux-nvidia.git

[32] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-time flash translation layer for NAND flash memory storage systems," in *Proceedings of the 18th IEEE Real Time and Embedded Technology and Applications Symposium*, Apr 2012, pp. 35–44.

[33] H. Cho, D. Shin, and Y. I. Eom, "KAST: K-associative sector translation for NAND flash memory in real-time systems," in *2009 Design, Automation, and Test in Europe Conference and Exhibition*, Apr 2009, pp. 507–512.

[34] K. Missimer and R. West, "Partitioned real-time nand flash storage," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 185–195.

[35] K. Missimer, M. Athanassoulis, and R. West, "Telomere: Real-time NAND flash storage," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 1, Jan 2022.

[36] P. J. Denning, "Virtual memory," *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.

[37] B. Gregg, "The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging," *Queue*, vol. 14, no. 2, p. 91–110, Mar 2016.

[38] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr 2017, pp. 353–364.

[39] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, "Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms," in *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2020, pp. 1–10.

[40] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proceedings of the 38th IEEE Real-Time Systems Symposium*, Dec 2017, pp. 104–115.

[41] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *Proceedings of the 39th IEEE Real-Time Systems Symposium*, Dec 2018, pp. 119–130.

[42] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr 2020, pp. 213–225.

[43] NVIDIA, "Open GPU documentation." [Online]. Available: https://github.com/NVIDIA/open-gpu-doc

[44] D. Lee, J.-C. Kim, C.-G. Lee, and K. Kim, "mRT-PLRU: A general framework for real-time multitask executions on NAND flash memory," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 758–771, 2013.

[45] J.-C. Kim, D. Lee, C.-G. Lee, and K. Kim, "RT-PLRU: A new paging scheme for real–time execution of program codes on NAND flash memory for portable media players," *IEEE Transactions on Computers*, vol. 60, no. 8, pp. 1126–1141, 2011.

[46] T.-Y. Wang, C.-F. Wu, C.-W. Tsao, Y.-H. Chang, and T.-W. Kuo, "Scheduling-aware prefetching: Enabling the PCIe SSD to extend the global memory of GPU device," in *Proceedings of the 10th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, Aug. 2021, pp. 1–6.

[47] T. Allen and R. Ge, "Demystifying GPU UVM cost with deep runtime and workload analysis," in *2021 IEEE International Parallel and Distributed Processing Symposium*, 2021, pp. 141–150.

[48] N. Otterness, M. Yang, T. Amert, J. Anderson, and F. D. Smith, "Inferring the scheduling policies of an embedded CUDA gpu," in *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real Time Applications*, July 2017.

[49] J. Bakita, N. Otterness, J. H. Anderson, and F. D. Smith, "Scaling up: The validation of empirically derived scheduling rules on NVIDIA GPUs," in *Proceedings of 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2018.

[50] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems," in *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, July 2018, pp. 20:1–20:21.

[51] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation*, Sep 2017, pp. 1–10.

[52] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni, "Dynamic memory bandwidth allocation for real-time GPU-based SoC platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3348–3360, 2020.